# riptable

rtosholdings

Sep 13, 2023

# CONTENTS


1 Intro to Riptable 1

2 riptable 167

Python Module Index 717

Index 719

CHAPTER

ONE

# INTRO TO RIPTABLE

Welcome to Riptable. This intro guide is intended to help get you familiar with Riptable's basic functionality and syntax.

Questions or suggestions? Email RiptableDocumentation@sig.com.

## 1.1 Introduction

### 1.1.1 What Is Riptable?

Riptable is an open source library built for high-performance data analysis. It's similar to Pandas by design, but it's been optimized to meet the needs of Riptable's core users: quantitative analysts interacting live with large volumes of trading data.

Riptable is based on NumPy, so it shares many core NumPy methods for array-based operations. Riptable has also implemented its own Pandas-like functions for grouping and aggregation. For users who work with large datasets, Riptable improves on NumPy and Pandas by using multi-threading and efficient memory management, much of it implemented at the C++ level.

Riptable's APIs are designed to be more feature-rich and easier to work with than those provided by Pandas and other existing libraries.

NumPy and Pandas users will find it easy to convert their data to Riptable (and back again if need be). It's also possible to convert data from CSV or SQL files. Similarly, h5 files can be converted to Riptable's format. Matlab users, who will generally find similar syntax and functionality in Riptable, can use special keyword arguments to convert Matlab data to Riptable's format. See *Work with Riptable Files and Other File Formats* for details.

For data visualization, any of the standard plotting tools (for example, matplotlib.pyplot) will work out of the box. To see a few basic examples, check out the *Visualize Data* section.

### 1.1.2 Who This Tutorial Is For

If you're new to Riptable, this tutorial is for you. It's intended to help get you familiar with Riptable's basic functionality and syntax.

Some experience with Python will be helpful, especially familiarity with dictionary syntax, sequences (lists, tuples, etc.), and basic functions and arguments.

### 1.1.3 A Note to Pandas Users

If you've used Pandas, you'll notice many similarities in Riptable – though be aware that Riptable has some not-always-immediately-obvious differences. This tutorial doesn't call out those differences specifically; see the API Reference for details of differences in specific methods, functions, attributes, etc.

### 1.1.4 Install and Import Riptable

To install Riptable on Windows or Linux, create a Conda environment and type:

```
conda install riptable
```

To access Riptable and its functions in your Python code, add these lines to your code:

```
import riptable as rt
import numpy as np
```

### 1.1.5 Display Options

You can modify Riptable's default display options using the attributes offered in rt.Display.options. Here are a few you might find useful.

#### General Display Options

Some general options you can set for a session:

```
# Display all Dataset columns -- the default max is 9.
rt.Display.options.COL_ALL = True

# Render up to 100MM before showing in scientific notation.
rt.Display.options.E_MAX = 100_000_000

# Truncate small decimals, rather than showing infinitesimal scientific notation.
rt.Display.options.P_THRESHOLD = 0

# Put commas in numbers.
rt.Display.options.NUMBER_SEPARATOR = True

# Turn on Riptable autocomplete (start typing, then press Tab to see options).
rt.autocomplete()
```

### Contextual Help

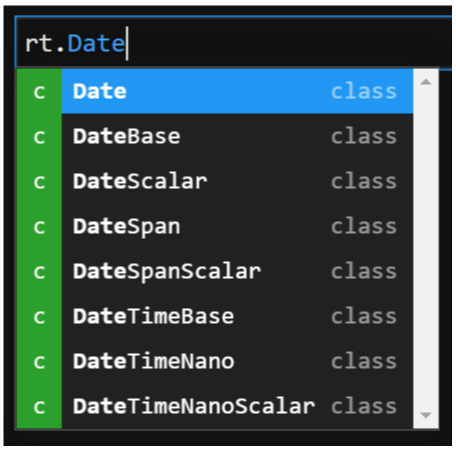
The `rt.autocomplete()` option listed above can be used as an alternative to Python's built-in `dir()` function, which shows various attributes and methods associated with an object.

For example, to see the attributes and methods of Riptable's Date object, you can use `dir()`:

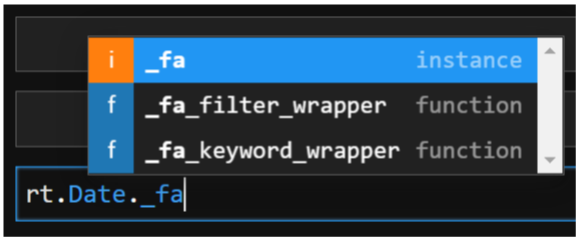
```
>>> # Limit and format the output.
>>> dir_date = dir(rt.Date)
>>> print("Some of the attributes and methods include...\n")
>>> print(", ".join(list(dir_date)[::10]))
Some of the attributes and methods include...

CompressPickle, T, _LDUMP, _TON, __array_function__, __class__, __doc__, __hash__, __
↪init_subclass__, __le__, __new__, __rfloordiv__, __rsub__, __truediv__, _check_mathops,
↪ _fa_keyword_wrapper, _max, _nanstd, _reduce_op_identity_value, _yearday_splits_leap,␣
↪argpartition, clip_upper, cummin, differs, ema_decay, format_date_num, is_leapyear,␣
↪isnormal, map_old, move_mean, nanmean, nonzero, push, reshape, round, sign, strides,␣
↪tolist, year
```

Note: The resulting list may not be complete. For details, see Python's documentation for `dir()` in the section on built-in functions.

Alternatively, you can use Riptable's autocomplete interface. With `rt.autocomplete()` turned on, type `rt.Date.<TAB>` where <TAB> is the Tab key. You'll see a pop-up list of attributes and methods. Keep typing to narrow down the list.




Note that private/internal attributes and methods (those whose names are preceded by an underscore) are omitted by default, but you can access them by typing the underscore. For example: `rt.Date._fa<TAB>`.




You can access the doc string on any (documented) function or object with the following syntax:

- IPython prompt: `my_func?`
- Python prompt: `help(my_obj)`

For example:

```
>>> rt.sum?
Signature: rt.sum(*args, filter=None, dtype=None, **kwargs)
Docstring:
Compute the sum of the values in the first argument.

When possible, ``rt.sum(x, *args)`` calls ``x.sum(*args)``; look there for
documentation. In particular, note whether the called function accepts the
keyword arguments listed below. For example, `Dataset.sum()` does not accept
the `filter` or `dtype` keyword arguments.

For ``FastArray.sum``, see `numpy.sum` for documentation but note the following:

* The `dtype` keyword argument may not work as expected:

    * Riptable data types (for example, `rt.float64`) are ignored.
    * NumPy integer data types (for example, `numpy.int32`) are also ignored.
    * NumPy floating point data types are applied as `numpy.float64`.

* If you include another NumPy parameter (for example, ``axis=0``), the NumPy
implementation of ``sum`` will be used and the ``dtype`` will be used to
compute the sum.

Parameters
----------
filter : array of bool, default None
    Specifies which elements to include in the sum calculation.
dtype : rt.dtype or numpy.dtype, optional
    The data type of the result. By default, for integer input the result `dtype` is
    ``int64`` and for floating point input the result `dtype` is ``float64``. See
    the notes above about using this keyword argument with `FastArray` objects
    as input.

See Also
--------
numpy.sum
nansum : Sums the values, ignoring NaNs.
FastArray.sum : Sums the values of a `FastArray`.
Dataset.sum : Sums the values of numerical `Dataset` columns.
GroupByOps.sum : Sums the values of each group. Used by `Categorical` objects.

Examples
--------
>>> a = rt.FastArray([1, 3, 5, 7])
>>> rt.sum(a)
16

>>> a = rt.FastArray([1.0, 3.0, 5.0, 7.0])
>>> rt.sum(a)
16.0
File:      c:\\riptable\\rt_numpy.py
Type:      function
```

You can access the source code with ??:

```
>>> rt.sum??
Signature: rt.sum(*args, filter=None, dtype=None, **kwargs)
Docstring:
Compute the sum of the values in the first argument.

When possible, ``rt.sum(x, *args)`` calls ``x.sum(*args)``; look there for
documentation. In particular, note whether the called function accepts the
keyword arguments listed below. For example, `Dataset.sum()` does not accept
the `filter` or `dtype` keyword arguments.

For ``FastArray.sum``, see `numpy.sum` for documentation but note the following:

* The `dtype` keyword argument may not work as expected:

    * Riptable data types (for example, `rt.float64`) are ignored.
    * NumPy integer data types (for example, `numpy.int32`) are also ignored.
    * NumPy floating point data types are applied as `numpy.float64`.

* If you include another NumPy parameter (for example, ``axis=0``), the NumPy
implementation of ``sum`` will be used and the ``dtype`` will be used to
compute the sum.

Parameters
----------
filter : array of bool, default None
    Specifies which elements to include in the sum calculation.
dtype : rt.dtype or numpy.dtype, optional
    The data type of the result. By default, for integer input the result `dtype` is
    ``int64`` and for floating point input the result `dtype` is ``float64``. See
    the notes above about using this keyword argument with `FastArray` objects
    as input.

See Also
--------
numpy.sum
nansum : Sums the values, ignoring NaNs.
FastArray.sum : Sums the values of a `FastArray`.
Dataset.sum : Sums the values of numerical `Dataset` columns.
GroupByOps.sum : Sums the values of each group. Used by `Categorical` objects.

Examples
--------
>>> a = rt.FastArray([1, 3, 5, 7])
>>> rt.sum(a)
16

>>> a = rt.FastArray([1.0, 3.0, 5.0, 7.0])
>>> rt.sum(a)
16.0
Source:
def sum(*args,filter = None, dtype = None,**kwargs):
```

(continues on next page)

(continued from previous page)

```
    '''
    Compute the sum of the values in the first argument.

    When possible, ``rt.sum(x, *args)`` calls ``x.sum(*args)``; look there for
    documentation. In particular, note whether the called function accepts the
    keyword arguments listed below. For example, `Dataset.sum()` does not accept
    the `filter` or `dtype` keyword arguments.

    For ``FastArray.sum``, see `numpy.sum` for documentation but note the following:

    * The `dtype` keyword argument may not work as expected:

        * Riptable data types (for example, `rt.float64`) are ignored.
        * NumPy integer data types (for example, `numpy.int32`) are also ignored.
        * NumPy floating point data types are applied as `numpy.float64`.

    * If you include another NumPy parameter (for example, ``axis=0``), the NumPy
    implementation of ``sum`` will be used and the ``dtype`` will be used to
    compute the sum.

    Parameters
    ----------
    filter : array of bool, default None
        Specifies which elements to include in the sum calculation.
    dtype : rt.dtype or numpy.dtype, optional
        The data type of the result. By default, for integer input the result `dtype` is
        ``int64`` and for floating point input the result `dtype` is ``float64``. See
        the notes above about using this keyword argument with `FastArray` objects
        as input.

    See Also
    --------
    numpy.sum
    nansum : Sums the values, ignoring NaNs.
    FastArray.sum : Sums the values of a `FastArray`.
    Dataset.sum : Sums the values of numerical `Dataset` columns.
    GroupByOps.sum : Sums the values of each group. Used by `Categorical` objects.

    Examples
    --------
    >>> a = rt.FastArray([1, 3, 5, 7])
    >>> rt.sum(a)
    16

    >>> a = rt.FastArray([1.0, 3.0, 5.0, 7.0])
    >>> rt.sum(a)
    16.0
    '''
    kwargs = _np_keyword_wrapper(filter=filter, dtype=dtype, **kwargs)
    args = _convert_cat_args(args)
    if hasattr(args[0], 'sum'):
        return args[0].sum(*args[1:], **kwargs)
```

(continues on next page)

(continued from previous page)

```
    return builtins.sum(*args,**kwargs)
File:      c:\\riptable\\rt_numpy.py
Type:      function
```

### Dataset Display Options

When you view a Dataset, some data might be elided or truncated. By default:

- Up to 9 columns are shown. If the Dataset has more than 9 columns, the middle columns are elided (with a "…" column displayed).
- Up to 30 rows are shown. If the Dataset has more than 30 rows, the middle rows are elided (with a "…" row displayed).
- Strings are displayed up to 15 characters, with additional characters truncated.

The following internal/private methods override the defaults on a per-display basis:

- Show all columns and rows (up to 10,000 rows), as well as long strings: `ds._A`
- Show all columns and long strings: `ds._H`
- Show all columns with wrapping, and long strings: `ds._G`
- Show all rows (up to 10,000): `ds._V`
- Transpose columns and rows: `ds._T`

Now that we're all set up, we're ready to look at Riptable's foundational data structures: *Riptable Datasets, FastArrays, and Structs*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.2 Riptable Datasets, FastArrays, and Structs

### 1.2.1 What Is a Dataset?

A Dataset is a table of data that consists of a sequence of columns of the same length. It's similar to a spreadsheet, a SQL table, a Pandas DataFrame or the data.frame in R. The Dataset is the workhorse of Riptable.

Each column in a Dataset consists of a key (also referred to as the column label, header, or name) and a series of values stored in a Riptable FastArray. A FastArray is a 1-dimensional array of values that are all the same data type, or dtype.

Though each Dataset column has a single dtype, the Dataset overall can hold columns of various dtypes.

Dataset rows are implicitly indexed by integer. You can select rows using their indices, but you can't reindex rows or give them arbitrary labels. This restriction helps Riptable perform Dataset operations more efficiently.

### 1.2.2 Create a Dataset

Generally speaking, there are a few ways to create Riptable Datasets. You can convert a Python dictionary or use Riptable's dictionary-style syntax, or create an empty Dataset and add arrays as columns.

#### Convert a Python Dictionary to a Riptable Dataset

If you have a Python dictionary, it's easy to convert it to a Riptable Dataset:

```
>>> my_dict = {'Column1': ['A', 'B', 'C', 'D'], 'Column2': [0, 1, 2, 3]}  # Create a␣
↪Python dictionary
>>> ds = rt.Dataset(my_dict)  # Convert it to a Riptable Dataset
>>> ds
#   Column1   Column2
-   -------   -------
0   A               0
1   B               1
2   C               2
3   D               3
```

Another way to think of a Dataset is as a dictionary of same-length FastArrays, where each key is a column name that's mapped to a FastArray of values that all have the same dtype.

For Python dictionary details, see Python's documentation.

#### Use the Dataset Constructor with Dictionary-Style Input

`rt.Dataset()` uses dictionary-style syntax:

```
>>> ds = rt.Dataset({'Column1': ['A', 'B', 'C', 'D'], 'Column2': [0.0, 1.0, 2.0, 3.0]})
>>> ds
#   Column1   Column2
-   -------   -------
0   A            0.00
1   B            1.00
2   C            2.00
3   D            3.00
```

#### Create an Empty Dataset and Add Columns to It

You can also create an empty dataset by using `rt.Dataset()` without any dictionary input ...

```
>>> ds = rt.Dataset()
```

... and then add columns to it.

### Add Dataset Columns (FastArrays)

The first column you add to the Dataset can be any length, but all future columns must match that length.

The columns you add to the Dataset become aligned, meaning that they share the same row index.

You can add a column to a Dataset using attribute assignment or dictionary-style syntax. Here, we use attribute assignment to create a column named 'Column1' that holds a list of values:

```
>>> ds.Column1 = [0.0, 1.0, 2.0, 3.0, 4.0]
>>> ds
#   Column1
-   -------
0      0.00
1      1.00
2      2.00
3      3.00
4      4.00
```

The list becomes a FastArray. You can use attribute access to get the column's data:

```
>>> ds.Column1
FastArray([0., 1., 2., 3., 4.])
```

Here, we use dictionary-style syntax to add a column of integers:

```
>>> ds['Ints'] = [1, 2, 3, 4, 5]
>>> ds
#   Column1   Ints
-   -------   ----
0      0.00      1
1      1.00      2
2      2.00      3
3      3.00      4
4      4.00      5
```

And we can use dictionary-style syntax to access column data:

```
>>> ds['Ints']
FastArray([1, 2, 3, 4, 5])
```

### A Note About Column Names

Column names should meet Python's rules for well-formed variable names. If a column name doesn't meet these rules (for example, if it's a procedurally generated name that starts with a symbol), you can't refer to it or get its data using attribute access.

For example, trying to access a column called #%&ColumnName with `ds.#%&ColumnName` will give you a syntax error. To access the column, you'll need to use dictionary-style syntax: `ds['#%&ColumnName']`.

Python keywords and Riptable class methods are also restricted. If you're not sure whether a column name is valid, you can use the Dataset method `is_valid_colname()`.

For example, `for` is invalid because it's a Python keyword:

```
>>> ds.is_valid_colname('for')
False
```

And `col_move` is invalid because it's a Dataset class method:

```
>>> ds.is_valid_colname('col_move')
False
```

You can see all restricted names with `get_restricted_names`:

```
>>> # Limit and format the output.
>>> print("Some of the restricted names include...\n")
>>> print(", ".join(list(ds.get_restricted_names())[::10]))
Some of the restricted names include...

mask_or_isinf, __reduce_ex__, imatrix_xy, __weakref__, dtypes, _get_columns, from_arrow,␣
↪elif, __imul__, _deleteitem, __rsub__, _index_from_row_labels, as_matrix, putmask, _as_
↪meta_data, shape, cat, __invert__, try, _init_columns_as_dict, label_as_dict, col_str_
↪replace, _replaceitem, label_set_names, __contains__, __floordiv__, _row_numbers,␣
↪filter, __init__, sorts_on, flatten_undo, col_str_match, __dict__, size, __rand__,␣
↪info, col_remove, as, or
```

### Add a NumPy Array as a Column

If you have a 1-dimensional NumPy array, you can add that as a column – it also will be converted to a FastArray:

```
>>> my_np_array = np.array([5.0, 6.0, 7.5, 8.5, 9.0])
>>> ds.NPArr = my_np_array
>>> ds
#   Column1   Ints   NPArr
-   -------   ----   -----
0      0.00      1    5.00
1      1.00      2    6.00
2      2.00      3    7.50
3      3.00      4    8.50
4      4.00      5    9.00
```

Warning: Although you can technically convert a 2-dimensional (or higher) NumPy array to a multi-dimensional FastArray, multi-dimensional FastArrays aren't supported and you could get unexpected results when you try to work with one:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a_fa = rt.FastArray(a)
C:\\riptable\\rt_fastarray.py:561: UserWarning: FastArray contains two or more␣
↪dimensions greater than one - shape:(3, 4).  Problems may occur.
  warnings.warn(warning_string)
```

If you don't specify the dtype, Riptable makes its best guess:

```
>>> ds.Ints.dtype
dtype('int32')
```

If you want to specify the dtype, create a FastArray directly with the `dtype` parameter:

```
>>> ds.Floats = rt.FastArray([0, 1, 2, 3, 4], dtype=float)
>>> ds
#   Column1   Ints   NPArr   Floats
-   -------   ----   -----   ------
0      0.00      1    5.00     0.00
1      1.00      2    6.00     1.00
2      2.00      3    7.50     2.00
3      3.00      4    8.50     3.00
4      4.00      5    9.00     4.00
```

Tip: You can also create a FastArray using the shortcut `rt.FA()`.

If you add a column with a single value, the value is duplicated to fill every existing row:

```
>>> ds.Ones = 1
>>> ds
#   Column1   Ints   NPArr   Floats   Ones
-   -------   ----   -----   ------   ----
0      0.00      1    5.00     0.00      1
1      1.00      2    6.00     1.00      1
2      2.00      3    7.50     2.00      1
3      3.00      4    8.50     3.00      1
4      4.00      5    9.00     4.00      1
```

Instantiating a column with ones or zeros as placeholder data can be useful – see some options in the *Instantiate with Placeholder Values and Generate Sample Data* section.

### 1.2.3 Get Basic Info About a Dataset

Datasets have attributes (sometimes also called properties) that give you information about them.

To better see how they work, let's create a slightly larger Dataset:

```
  >>> rng = np.random.default_rng(seed=42)  # Construct a random number generator
  >>> ds2 = rt.Dataset()
  >>> N = 50
  >>> ds2.Symbol = rt.FA(np.random.choice(['AAPL', 'AMZN', 'TSLA', 'SPY', 'GME'], N))
  >>> ds2.Size = rt.FA(np.random.choice([100, 200, 300, 400, 500], N))
  >>> ds2.Value = rng.random(N)
  >>> ds2
  #   Symbol   Size   Value
---   ------   ----   -----
  0   SPY       500    0.77
  1   AMZN      500    0.44
  2   AAPL      400    0.86
  3   SPY       300    0.70
  4   TSLA      300    0.09
  5   SPY       400    0.98
  6   GME       300    0.76
  7   TSLA      500    0.79
  8   AAPL      400    0.13
  9   GME       500    0.45
 10   SPY       300    0.37
```

(continues on next page)

(continued from previous page)

```
 11   SPY       400    0.93
 12   TSLA      100    0.64
 13   AMZN      100    0.82
 14   SPY       400    0.44
...   ...       ...     ...
 35   AMZN      400    0.19
 36   GME       200    0.13
 37   AMZN      400    0.48
 38   SPY       500    0.23
 39   TSLA      500    0.67
 40   AMZN      100    0.44
 41   AAPL      300    0.83
 42   AAPL      400    0.70
 43   AAPL      200    0.31
 44   AAPL      300    0.83
 45   TSLA      100    0.80
 46   GME       500    0.39
 47   AAPL      300    0.29
 48   AAPL      200    0.68
 49   GME       400    0.14
```

Use `shape` to get the Dataset's dimensions returned as a tuple (rows, cols):

```
>>> ds2.shape
(50, 3)
```

See the dtypes of a Dataset (note the plural `.dtypes` vs. the singular `.dtype` for FastArrays):

```
>>> ds2.dtypes
{'Symbol': dtype('S4'), 'Size': dtype('int32'), 'Value': dtype('float64')}
```

Datasets also have methods that give you a feel for the data they contain. Useful methods for seeing quick subsets of your Dataset are `head()`, `tail()`, and `sample()`. By default, `head()` and `tail()` show you the first or last 20 rows, respectively, while `sample()` shows you 10 rows randomly selected from the Dataset. For each, you can pass an argument to show a custom number of rows.

The first 5 rows:

```
>>> ds2.head(5)
#   Symbol   Size   Value
-   ------   ----   -----
0   SPY       500    0.77
1   AMZN      500    0.44
2   AAPL      400    0.86
3   SPY       300    0.70
4   TSLA      300    0.09
```

The last 10 rows:

```
>>> ds2.tail(10)
#   Symbol   Size   Value
-   ------   ----   -----
0   AMZN      100    0.44
```

(continues on next page)

(continued from previous page)

```
1   AAPL      300    0.83
2   AAPL      400    0.70
3   AAPL      200    0.31
4   AAPL      300    0.83
5   TSLA      100    0.80
6   GME       500    0.39
7   AAPL      300    0.29
8   AAPL      200    0.68
9   GME       400    0.14
```

If the first or last rows aren't representative of your data, it can be preferable to use `sample`:

```
>>> ds2.sample()
#   Symbol   Size   Value
-   ------   ----   -----
0   GME       300    0.76
1   SPY       400    0.44
2   AMZN      100    0.83
3   TSLA      400    0.76
4   SPY       200    0.97
5   GME       100    0.15
6   SPY       400    0.97
7   AMZN      500    0.37
8   AMZN      400    0.19
9   AAPL      200    0.68
```

For numerical data, `describe()` gives you summary statistics. Non-numerical columns are ignored:

```
>>> ds2.describe()
*Stats     Size   Value
------   ------   -----
Count     50.00   50.00
Valid     50.00   50.00
Nans       0.00    0.00
Mean     302.00    0.54
Std      142.13    0.28
Min      100.00    0.04
P10      100.00    0.14
P25      200.00    0.32
P50      300.00    0.52
P75      400.00    0.78
P90      500.00    0.86
Max      500.00    0.98
MeanM    302.38    0.54
```

For each numerical column, `describe()` provides these summary statistics:

| Calculation | Description |
|---|---|
| Count | Total number of items |
| Valid | Total number of valid values |
| Nans | Total number of NaN values* |
| Mean | Mean |
| Std | Standard deviation |
| Min | Minimum value |
| P10 | 10th percentile |
| P25 | 25th percentile |
| P50 | 50th percentile |
| P75 | 75th percentile |
| P90 | 90th percentile |
| Max | Maximum value |
| MeanM | Mean without top or bottom 10% |

*NaN stands for Not a Number, and is commonly used to represent missing data. For details, see *Working with Missing Data*.

You can also use `describe()` on a single column:

```
>>> ds2.Value.describe()
*Stats   Value
------   -----
Count    50.00
Valid    50.00
Nans      0.00
Mean      0.54
Std       0.28
Min       0.04
P10       0.14
P25       0.32
P50       0.52
P75       0.78
P90       0.86
Max       0.98
MeanM     0.54
```

If your Dataset is very large, you can get column statistics with `statx()`, which you can import from `riptable.rt_stats`. `statx()` provides rapid sampling and gives you a few more percentiles than `describe()` does, but it works only on one column at a time:

```
>>> from riptable.rt_stats import statx
>>> statx(ds2.Value)
        Stat      Value
0        min   0.043804
1       0.1%   0.044784
2         1%   0.053610
3        10%   0.138769
4        25%   0.315731
5        50%   0.515145
6        75%   0.777277
7        90%   0.862050
```

(continues on next page)

(continued from previous page)

```
8         99%   0.973209
9       99.9%   0.975381
10        max   0.975622
11       Mean   0.535233
12     StdDev   0.277838
13      Count  50.000000
14  NaN_Count   0.000000
```

### Other Useful Dataset Methods

See a column's unique values:

```
>>> ds2.Symbol.unique()
FastArray([b'AAPL', b'AMZN', b'GME', b'SPY', b'TSLA'], dtype='|S4')
```

A note about strings in FastArrays: When you view a FastArray of strings, you'll see a 'b' next to each string. These b's indicate that the strings are encoded to byte strings, which saves memory compared to saving strings as ASCII.

Count the number of unique values in a column:

```
>>> ds2.Symbol.count()
*Symbol   Count
-------   -----
AAPL         12
AMZN         12
GME           7
SPY           8
TSLA         11
```

Note that `count()` displays aggregated results. We'll look more at Riptable's structures and functions for aggregations later, when we cover Categoricals and Accums.

View the Dataset as a dictionary:

```
>>> ds2.asdict()
{'Symbol': FastArray([b'TSLA', b'SPY', b'GME', b'SPY', b'SPY', b'AAPL', b'AAPL',
        b'SPY', b'TSLA', b'AMZN', b'SPY', b'AMZN', b'AMZN', b'TSLA',
        b'GME', b'SPY', b'SPY', b'SPY', b'SPY', b'GME', b'AAPL',
        b'AAPL', b'TSLA', b'SPY', b'AMZN', b'TSLA', b'TSLA', b'AAPL',
        b'TSLA', b'SPY', b'GME', b'AAPL', b'SPY', b'AMZN', b'AAPL',
        b'AAPL', b'AMZN', b'TSLA', b'GME', b'AMZN', b'GME', b'AMZN',
        b'AAPL', b'AMZN', b'AAPL', b'AAPL', b'AMZN', b'GME', b'AAPL',
        b'AMZN'], dtype='|S4'),
 'Size': FastArray([400, 100, 100, 300, 300, 400, 300, 300, 300, 200, 500, 500,
        500, 400, 400, 100, 500, 400, 500, 200, 400, 500, 300, 200,
        200, 500, 400, 100, 500, 500, 300, 300, 200, 300, 500, 200,
        200, 500, 200, 300, 400, 200, 100, 500, 100, 400, 400, 200,
        200, 400]),
 'Value': FastArray([0.77395605, 0.43887844, 0.85859792, 0.69736803, 0.09417735,
        0.97562235, 0.7611397 , 0.78606431, 0.12811363, 0.45038594,
        0.37079802, 0.92676499, 0.64386512, 0.82276161, 0.4434142 ,
        0.22723872, 0.55458479, 0.06381726, 0.82763117, 0.6316644 ,
```

(continues on next page)

(continued from previous page)

```
        0.75808774, 0.35452597, 0.97069802, 0.89312112, 0.7783835 ,
        0.19463871, 0.466721  , 0.04380377, 0.15428949, 0.68304895,
        0.74476216, 0.96750973, 0.32582536, 0.37045971, 0.46955581,
        0.18947136, 0.12992151, 0.47570493, 0.22690935, 0.66981399,
        0.43715192, 0.8326782 , 0.7002651 , 0.31236664, 0.8322598 ,
        0.80476436, 0.38747838, 0.2883281 , 0.6824955 , 0.13975248])}
```

### 1.2.4 Select Dataset Columns

As mentioned above, you can access a Dataset column using attribute access (`ds.Column1`) or using dictionary-style syntax (`ds['Column1']`).

To select multiple columns of a Dataset, pass a list of column names to `col_filter()`:

```
>>> ds.col_filter(['Floats', 'Ones'])
#   Floats   Ones
-   ------   ----
0     0.00      1
1     1.00      1
2     2.00      1
3     3.00      1
4     4.00      1
```

`col_filter()` also accepts regular expressions:

```
>>> ds.col_filter(regex='Col*')
#   Column1
-   -------
0      0.00
1      1.00
2      2.00
3      3.00
4      4.00
```

For selecting subsets of columns, Riptable supports all of the indexing, slicing, and "fancy indexing" operations supported by NumPy arrays.

Select a single value at index 0:

```
>>> ds.Column1[0]
0.0
```

Get a slice of contiguous values from index 1 (included) to index 4 (excluded):

```
>>> ds.Column1[1:4]
FastArray([1., 2., 3.])
```

To use fancy indexing, pass an array that specifies noncontiguous indices and your desired ordering:

```
>>> ds.Floats[[1, 3, 0]]
FastArray([1., 3., 0.])
```

You can also set values using indexing and slicing:

```
>>> ds.Column1[0] = 5.0
>>> ds.Ints[1:3] = 4
>>> ds.Floats[2:4] = 10.0, 20.0
>>> ds.Ones[[1, 3, 0]] = 2_000_000, 4_000_000, 5_000_000  # Underscores are nice for␣
↪code readability!
>>> ds
#   Column1   Ints   NPArr   Floats      Ones
-   -------   ----   -----   ------   -------
0      5.00      1    5.00     0.00   5000000
1      1.00      4    6.00     1.00   2000000
2      2.00      4    7.50    10.00         1
3      3.00      4    8.50    20.00   4000000
4      4.00      5    9.00     4.00         1
```

Warning: Trying to insert a floating-point value into a column/FastArray of integers will cause the floating-point value to be silently truncated:

```
>>> ds.Ones[0] = 1.5
>>> ds
#   Column1   Ints   NPArr   Floats      Ones
-   -------   ----   -----   ------   -------
0      5.00      1    5.00     0.00         1
1      1.00      4    6.00     1.00   2000000
2      2.00      4    7.50    10.00         1
3      3.00      4    8.50    20.00   4000000
4      4.00      5    9.00     4.00         1
```

To learn more about accessing data using indexing and slicing, see examples for 1-dimensional NumPy ndarrays in NumPy's documentation.

### 1.2.5 Select Dataset Rows

To select Dataset rows, you need to also specify which columns you want.

First row, Column1:

```
>>> ds[0, 'Column1']
5.0
```

You can also refer to columns by number:

```
>>> ds[0, 0]
5.0
```

The `:` specifies all columns:

```
>>> ds[0:3, :]
#   Column1   Ints   NPArr   Floats      Ones
-   -------   ----   -----   ------   -------
0      5.00      1    5.00     0.00         1
1      1.00      4    6.00     1.00   2000000
2      2.00      4    7.50    10.00         1
```

Or you can pass a list of multiple columns:

```
>>> ds[0:2, ['Ints', 'Ones']]
#   Ints      Ones
-   ----   -------
0      1         1
1      4   2000000
```

More often, you'll probably use filters to get subsets of your data. That's covered in more detail in *Get and Operate on Subsets of Data Using Filters*.

### 1.2.6 Perform Operations on Dataset Columns

FastArrays are a subclass of NumPy's ndarray. Thanks to this, you can do anything with FastArrays that you can do with NumPy arrays.

In particular, NumPy's universal functions (ufuncs) are supported, allowing for fast, vectorized operations. (Vectorized functions operate element-wise on arrays without using Python loops, which are slow.) See the NumPy API Reference for a complete list and documentation for all NumPy methods.

Note, though, that Riptable has implemented its own optimized version of many NumPy methods. If you call a NumPy method that's been optimized by Riptable, the Riptable method is called. We encourage you to call the Riptable method directly to avoid any confusion about what method is being called. See *NumPy Methods Optimized by Riptable* for details.

If a method hasn't been optimized by Riptable, the NumPy method is called.

#### Arithmetic on Column Values

You can do various arithmetic operations on any numerical column (or standalone FastArray) and optionally put the results into a new column.

Binary operations on two columns are performed on an element-by-element basis. The columns must be the same length:

```
>>> ds3 = rt.Dataset()
>>> ds3.A = [0, 1, 2]
>>> ds3.B = [5, 5, 5]
>>> ds3.C = ds3.A + ds3.B
>>> ds3
#   A   B   C
-   -   -   -
0   0   5   5
1   1   5   6
2   2   5   7
```

FastArrays also support broadcasting, which allows you to perform a binary operation on a FastArray and a scalar. For example, you can add a scalar to an array.

Riptable will upcast data types as necessary to preserve information:

```
>>> ds3.D = ds3.A + 5.1
>>> ds3
#   A   B   C      D
-   -   -   -   ----
0   0   5   5   5.10
```

(continues on next page)

(continued from previous page)

```
1   1   5   6   6.10
2   2   5   7   7.10
```

Note that the standard order of operations is respected:

```
>>> ds3.E = -(0.5*ds3.A + 1) ** 2
>>> ds3
#   A   B   C      D       E
-   -   -   -   ----   -----
0   0   5   5   5.10   -1.00
1   1   5   6   6.10   -2.25
2   2   5   7   7.10   -4.00
```

You can populate a Dataset column with the results of an operation on a column of another Dataset, as long as the resulting FastArray is the right length for the Dataset you want to add it to:

```
>>> ds4 = rt.Dataset({'A': [10, 11, 12], 'B': [21, 22, 23]})
>>> ds3.F = ds4.A * 2
>>> ds3
#   A   B   C      D       E    F
-   -   -   -   ----   -----   --
0   0   5   5   5.10   -1.00   20
1   1   5   6   6.10   -2.25   22
2   2   5   7   7.10   -4.00   24
```

### Delete a Column from a Dataset

To delete a column from a Dataset, use `del ds.ColumnName`.

### Reducing Operations vs. Non-Reducing Operations

The operations we've performed so far have been *non-reducing* operations. A non-reducing operation takes in multiple input values and returns one output value for each input value. That is, the resulting FastArray is the same length as the FastArray you operated on, and it can be added to the same Dataset.

A *reducing* operation, on the other hand, takes in multiple inputs and returns one value. `sum()` and `mean()` are examples of reducing operations. This distinction will be more important when we talk about Categoricals and operations on grouped data. For now, we'll get the results of two reducing operations without adding them to a Dataset.

The total of the Size column:

```
>>> ds2.Size.sum()
15700
```

The average of the Value column:

```
>>> ds2.Value.mean()
0.5352327331104895
```

Tip: Many column operations can be called in two ways: as a method called on a FastArray (`ds2.Size.sum()`) or as a Riptable function with the column as the argument (`rt.sum(ds2.Size)`).

### Watch Out for Missing Values

When you're working with real data, there will often be missing values. Take care when performing operations! In Riptable, missing floating-point values are represented by `nan`. In a regular arithmetic operation with a floating-point `nan`, the result is `nan`:

```
>>> y = rt.FA([1.0, 2.0, 3.0, rt.nan])
>>> y.sum()
nan
```

Fortunately, many functions have "nan" versions that ignore `nan` values:

```
>>> y.nansum()
6.0
```

Useful NaN functions:

| **Function** | **Description** (all functions ignore NaN values) |
|---|---|
| nanmin(), nanmax() | Minimum and maximum |
| nanvar() | Variance |
| nanmean() | Mean |
| nanstd() | Standard deviation |
| nansum() | Total of all items |
| nanargmin(), nanargmax() | Index of the minimum or maximum value |
| rollingnansum(), rollingnanmean() | Rolling sum, rolling mean |

Another way to deal with NaN values is to replace them with other values. For details, see *Working with Missing Data*.

### Sort Column Values

Sorting a column is straightforward. Use `sort_copy()` to return a sorted version of the array without modifying the original input, or `sort_inplace()` if you're OK with modifying the original data:

```
>>> ds4 = rt.Dataset()
>>> ds4.A = rng.choice(['AAPL', 'AMZN', 'TSLA', 'SPY', 'GME'], 10)
>>> ds4.B = rng.integers(low=0, high=5, size=10)
>>> ds4.C = rng.random(10)
>>> ds4
#   A      B      C
-   ----   -   ----
0   GME    1   0.67
1   AAPL   3   0.47
2   GME    2   0.57
3   AAPL   2   0.76
4   SPY    2   0.63
5   SPY    2   0.55
6   SPY    0   0.56
7   SPY    0   0.30
8   TSLA   1   0.03
9   SPY    0   0.44
```

You can sort by one column:

```
>>> ds4.sort_copy('A')
#   A      B      C
-   ----   -   ----
0   AAPL   2   0.76
1   AAPL   3   0.47
2   GME    1   0.67
3   GME    2   0.57
4   SPY    0   0.56
5   SPY    0   0.30
6   SPY    0   0.44
7   SPY    2   0.63
8   SPY    2   0.55
9   TSLA   1   0.03
```

Or by more than one column by passing an ordered list:

```
>>> ds4.sort_copy(['A', 'B'])
#   A      B      C
-   ----   -   ----
0   AAPL   2   0.76
1   AAPL   3   0.47
2   GME    1   0.67
3   GME    2   0.57
4   SPY    0   0.56
5   SPY    0   0.30
6   SPY    0   0.44
7   SPY    2   0.63
8   SPY    2   0.55
9   TSLA   1   0.03
```

With `sort_copy()`, the original Dataset is not modified:

```
>>> ds4
#   A      B      C
-   ----   -   ----
0   SPY    0   0.56
1   SPY    0   0.30
2   SPY    0   0.44
3   GME    1   0.67
4   TSLA   1   0.03
5   GME    2   0.57
6   AAPL   2   0.76
7   SPY    2   0.63
8   SPY    2   0.55
9   AAPL   3   0.47
```

Use `sort_inplace()` if you want to modify the original input (for example, if your data needs to be sorted by time, but isn't):

```
>>> ds4.sort_inplace('B')
#   A      B      C
-   ----   -   ----
0   SPY    0   0.56
```

(continues on next page)

(continued from previous page)

```
1   SPY    0   0.30
2   SPY    0   0.44
3   GME    1   0.67
4   TSLA   1   0.03
5   GME    2   0.57
6   AAPL   2   0.76
7   SPY    2   0.63
8   SPY    2   0.55
9   AAPL   3   0.47
```

Change the sort order by passing `ascending=False`:

```
>>> ds4.sort_copy('A', ascending=False)
#   A      B      C
-   ----   -   ----
0   TSLA   1   0.03
1   SPY    2   0.55
2   SPY    2   0.63
3   SPY    0   0.44
4   SPY    0   0.30
5   SPY    0   0.56
6   GME    2   0.57
7   GME    1   0.67
8   AAPL   3   0.47
9   AAPL   2   0.76
```

### Split Data into New Columns Using String Operations

Sometimes related pieces of data come bundled together in a single string, and you want to break up the data into separate columns.

For example, take a look at the OSI Symbol field commonly found in trading-related data. OSIs are the official name for a tradable option. They contain several pieces of information that are separated by colons.

For example, in **AAPL:191018:260:0:C**:

- AAPL is the underlying symbol
- 191018 represents an expiration date of 2019-10-18
- 260 is the strike price dollar amount
- The 0 is the strike price penny amount
  - Other possibilities: :0: for 0.00, :5: for 0.50, :3: for 0.30, :25: for 0.25, :15: for 0.15
- "C" indicates a call ("P" indicates a put)

Here's what OSI Symbols might look like in a Dataset. We'll use `str.extract()` to break them into separate columns:

```
>>> ds5 = rt.Dataset(
...     {'OSISymbol':['SPY:191003:187:0:C','SPY:191003:193:0:C','TLT:191003:135:5:P',
...                   'AAPL:191018:260:0:C', 'AAPL:191018:265:0:P'],
...                   'Delta':[.93, .71, -.72, .45, -.81],
...                   'PnL':[1.03, 0.61, 0.52, -0.14, .68]
...     })
```

(continues on next page)

(continued from previous page)

```
>>> ds5
#   OSISymbol          Delta     PnL
-   ---------------    -----    -----
0   SPY:191003:187:     0.93     1.03
1   SPY:191003:193:     0.71     0.61
2   TLT:191003:135:    -0.72     0.52
3   AAPL:191018:260     0.45    -0.14
4   AAPL:191018:265    -0.81     0.68
```

`str.extract()` uses regular expressions to match patterns and capture/extract the subpatterns that are surrounded by parentheses. Each captured subpattern is returned in a separate column.

Below, we define five capture groups that correspond to five returned columns of data. Inside the capture groups, we match any letters or numbers:

```
>>> ds5[['Symbol', 'Expiration', 'StrikeDollar', 'StrikePenny',
...      'PutCall']] = ds5.OSISymbol.str.extract('(.*):(.*):(.*):(.*):(.*)',
...      names=['Symbol', 'Expiration', 'StrikeDollar', 'StrikePenny', 'PutCall'])
>>> ds5
#   OSISymbol          Delta     PnL   Symbol   Expiration   StrikeDollar   StrikePenny  ␣
↪PutCall
-   ---------------    -----    -----   ------   ----------   ------------   -----------  ␣
↪-------
0   SPY:191003:187:     0.93     1.03   SPY      191003       187            0            ␣
↪C
1   SPY:191003:193:     0.71     0.61   SPY      191003       193            0            ␣
↪C
2   TLT:191003:135:    -0.72     0.52   TLT      191003       135            5            ␣
↪P
3   AAPL:191018:260     0.45    -0.14   AAPL     191018       260            0            ␣
↪C
4   AAPL:191018:265    -0.81     0.68   AAPL     191018       265            0            ␣
↪P
```

It's not ideal to have the strike dollar and strike penny amounts in separate columns, so we'll add a fix:

```
>>> ds5.Strike = (ds5.StrikeDollar + '.' + ds5.StrikePenny).astype('float')
>>> del ds5.StrikeDollar
>>> del ds5.StrikePenny
>>> ds5
#   OSISymbol          Delta     PnL   Symbol   Expiration   PutCall   Strike
-   ---------------    -----    -----   ------   ----------   -------   ------
0   SPY:191003:187:     0.93     1.03   SPY      191003       C         187.00
1   SPY:191003:193:     0.71     0.61   SPY      191003       C         193.00
2   TLT:191003:135:    -0.72     0.52   TLT      191003       P         135.50
3   AAPL:191018:260     0.45    -0.14   AAPL     191018       C         260.00
4   AAPL:191018:265    -0.81     0.68   AAPL     191018       P         265.00
```

### 1.2.7 Hold Two or More Datasets in a Struct

When you're working with multiple Datasets, it can be helpful to keep them together in a Riptable Struct. Structs were created as a base class for Datasets. They also replicate Matlab structs.

You can think of a Struct as a Python dictionary, but with attribute access allowed for keys.

Data structures stored together in a Struct don't need to be aligned:

```
>>> s = rt.Struct()
>>> s.ds = ds
>>> s.ds2 = ds2
```

You can access each data structure using attribute-style access. For example:

```
>>> s.ds2
  #   Symbol   Size   Value
---   ------   ----   -----
  0   AAPL      300    0.77
  1   AMZN      100    0.44
  2   AAPL      300    0.86
  3   GME       500    0.70
  4   SPY       100    0.09
  5   AMZN      300    0.98
  6   TSLA      200    0.76
  7   SPY       300    0.79
  8   TSLA      300    0.13
  9   TSLA      300    0.45
 10   AAPL      400    0.37
 11   AAPL      400    0.93
 12   AAPL      400    0.64
 13   GME       100    0.82
 14   AMZN      100    0.44
...   ...       ...     ...
 35   GME       200    0.19
 36   TSLA      400    0.13
 37   SPY       200    0.48
 38   AMZN      500    0.23
 39   GME       400    0.67
 40   AAPL      300    0.44
 41   SPY       100    0.83
 42   TSLA      500    0.70
 43   AAPL      500    0.31
 44   AAPL      100    0.83
 45   AAPL      200    0.80
 46   AMZN      400    0.39
 47   AMZN      500    0.29
 48   AMZN      300    0.68
 49   AMZN      400    0.14
```

Riptable has a few other methods for operating on strings. We'll use them as the basis for filtering data in the next section, *Get and Operate on Subsets of Data Using Filters*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.3 Get and Operate on Subsets of Data Using Filters

Earlier, we used indexing and slicing to select data. You can also use filters to get data that meets a certain condition.

Datasets and FastArrays have a `filter()` method that returns the subset of data that meets a given condition. But to operate on that data, it's often better to pass a `filter` keyword argument to the method you're using.

This section covers:

- How to create conditions for filtering – specifically, how comparison operators create mask arrays that can be used to filter
- What to expect when you filter a FastArray or a Dataset
- How string operations can be used to create filters
- How to create more complex filters using logic operators
- How to replace values using filters and `rt.where`
- How to operate on filtered Datasets in a memory-efficient way

### 1.3.1 Comparison Operators and Mask Arrays

When used to compare scalar values, comparison operators (>, <, >=, <=, ==, !=) return True or False.

```
>>> x = 10
>>> x == 10  # equal
True
>>> x != 12  # not equal
True
>>> x > 12  # greater than
False
```

In NumPy and Riptable, comparison operators are ufuncs, which means they can be used to compare arrays.

When an array is compared element-wise with a scalar value or a same-length array, the result is an array of Booleans.

```
>>> a = rt.FastArray([1, 2, 3, 4, 5])
>>> b = rt.FastArray([0, 5, 2, 4, 8])
>>> a > 3
FastArray([False, False, False,  True,  True])
>>> a <= b
FastArray([False,  True, False,  True,  True])
```

These Boolean arrays can be used to filter data. In this context, they're often called Boolean mask arrays.

For the FastArray and Dataset `filter()` methods, you can pass a Boolean mask array directly or pass a comparison (or other operation) that results in a mask array. Here, we'll focus on the various ways to generate mask arrays based on comparisons and other conditions.

### 1.3.2 Filter a FastArray with a Comparison

Above, we compared two FastArrays, `a` and `b`, using the condition `a <= b` to create a Boolean mask array.

To filter `a` based on that condition (that is, to show only the values of `a` for which `a <= b` is True), use the FastArray `filter()` method with the condition.

```
>>> f = a <= b
>>> a.filter(f)
FastArray([2, 4, 5])
```

Note that the returned FastArray is a copy; the original is unchanged:

```
>>> a
FastArray([1, 2, 3, 4, 5])
```

### 1.3.3 Filter a Dataset with a Comparison

Datasets also have a `filter()` method. It returns a copy of the Dataset with only the rows that meet the desired condition.

We'll work with this Dataset:

```
>>> ds = rt.Dataset({
...     'OSISymbol': ['VIX:200520:35:0:P', 'AAPL:200417:255:0:P', 'LITE:200619:82:5:P',
...                   'SPY:200406:265:0:C', 'MFA:200515:2:0:C', 'XOM:220121:60:0:C',
...                   'CCL:200717:12:5:C', 'AXSM:200515:85:0:C', 'UBER:200515:33:0:C',
...                   'TLT:200529:165:0:P'],
...     'UnderlyingSymbol': ['VIX', 'AAPL', 'LITE', 'SPY', 'MFA', 'XOM', 'CCL', 'AXSM',
...                          'UBER', 'TLT'],
...     'TradeDate': rt.Date(['2020-03-03', '2020-03-19', '2020-03-24', '2020-04-06',
...                           '2020-04-20', '2020-04-23', '2020-04-27', '2020-05-01',
...                           '2020-05-13', '2020-05-26']),
...     'TradeSize': [3., 1., 5., 50., 10., 5., 1., 6., 3., 1.],
...     'TradePrice': [13.4, 27.5, 14.8,  0.14, 0.29,  3.75,  2.55,  7.79,  0.77,  1.78␣
↪],
...     'OptionType': ['P', 'P', 'P', 'C', 'C', 'C', 'C', 'C', 'C', 'P'],
...     'Traded': [False, False, True, False, True, True, False, True, True, False]
...   })
>>> ds
#   OSISymbol         UnderlyingSymbol    TradeDate   TradeSize   TradePrice  ␣
↪OptionType   Traded
-   ---------------   ----------------   ----------   ---------   ----------   ----------
↪   ------
0   VIX:200520:35:0   VIX                2020-03-03        3.00        13.40   P        ␣
↪    False
1   AAPL:200417:255   AAPL               2020-03-19        1.00        27.50   P        ␣
↪    False
2   LITE:200619:82:   LITE               2020-03-24        5.00        14.80   P        ␣
↪     True
3   SPY:200406:265:   SPY                2020-04-06       50.00         0.14   C        ␣
↪    False
4   MFA:200515:2:0:   MFA                2020-04-20       10.00         0.29   C        ␣
```

(continues on next page)

(continued from previous page)

```
↪     True
5   XOM:220121:60:0   XOM                2020-04-23         5.00          3.75   C          ␣
↪     True
6   CCL:200717:12:5   CCL                2020-04-27         1.00          2.55   C          ␣
↪    False
7   AXSM:200515:85:   AXSM               2020-05-01         6.00          7.79   C          ␣
↪     True
8   UBER:200515:33:   UBER               2020-05-13         3.00          0.77   C          ␣
↪     True
9   TLT:200529:165:   TLT                2020-05-26         1.00          1.78   P          ␣
↪    False
```

Say we want to see only the rows with options that are puts.

The syntax is the same as for FastArrays:

```
>>> f = ds.OptionType == 'P'
>>> ds.filter(f)
#   OSISymbol          UnderlyingSymbol    TradeDate   TradeSize   TradePrice  ␣
↪OptionType   Traded
-   ---------------    ----------------    ----------  ---------   ----------   ----------
↪   ------
0   VIX:200520:35:0    VIX                 2020-03-03         3.00        13.40   P          ␣
↪    False
1   AAPL:200417:255    AAPL                2020-03-19         1.00        27.50   P          ␣
↪    False
2   LITE:200619:82:    LITE                2020-03-24         5.00        14.80   P          ␣
↪     True
3   TLT:200529:165:    TLT                 2020-05-26         1.00         1.78   P          ␣
↪    False
```

By default all columns are returned. If you want to return only certain columns, you can combine the mask array with column selection:

```
>>> ds.filter(f).col_filter(['OSISymbol', 'TradeSize'])
```

Alternatively, you can use the syntax we used to select Dataset rows to select rows based on the filter, along with the columns you want:

```
>>> ds[f, [0, 3]]
#   OSISymbol          TradeSize
-   ---------------    ---------
0   VIX:200520:35:0         3.00
1   AAPL:200417:255         1.00
2   LITE:200619:82:         5.00
3   TLT:200529:165:         1.00
```

Here it could also make sense to pass the Traded column directly as a mask array:

```
>>> ds.filter(ds.Traded)
#   OSISymbol          UnderlyingSymbol    TradeDate   TradeSize   TradePrice  ␣
↪OptionType   Traded
-   ---------------    ----------------    ----------  ---------   ----------   ----------
```

(continues on next page)

(continued from previous page)

```
↪   ------
0   LITE:200619:82:   LITE              2020-03-24        5.00         14.80   P         ␣
↪      True
1   MFA:200515:2:0:   MFA               2020-04-20       10.00          0.29   C         ␣
↪      True
2   XOM:220121:60:0   XOM               2020-04-23        5.00          3.75   C         ␣
↪      True
3   AXSM:200515:85:   AXSM              2020-05-01        6.00          7.79   C         ␣
↪      True
4   UBER:200515:33:   UBER              2020-05-13        3.00          0.77   C         ␣
↪      True
```

Note: Keep in mind that every time you use `filter()`, it makes a copy of the Dataset that takes up memory. We cover a couple of strategies for minimizing memory use below, when we talk about operations on filtered data.

### 1.3.4 Use FastArray String Methods to Create Filters

FastArray string methods are useful for creating conditions you can use to filter.

Create a filter for OSISymbol strings that start with 'A':

```
>>> f = ds.OSISymbol.str.startswith('A')
>>> f
FastArray([False,  True, False, False, False, False, False,  True, False, False])
```

For OSISymbol strings that contain the substring '2005':

```
>>> f = ds.OSISymbol.str.contains('2005')
>>> f
FastArray([ True, False, False, False,  True, False, False,  True,  True, True])
```

For UnderlyingSymbol strings that end with 'L':

```
>>> f = ds.UnderlyingSymbol.str.regex_match('L$')
>>> f
FastArray([False,  True, False, False, False, False,  True, False, False, False])
```

### 1.3.5 Create More Complex Boolean Mask Filters with Bitwise Logic Operators (&, |, ~)

You can build more complex filters using Python's bitwise logic operators, & (bitwise and), | (bitwise or), and ~ (bitwise not).

Let's say you want to construct a filter that returns True for calls over $2.00. You can use & to ensure that both of those conditions are met:

```
>>> callsover2 = (ds.OptionType == 'C') & (ds.TradePrice > 2.00)
>>> callsover2
FastArray([False, False, False, False, False,  True,  True,  True, False, False])
```

Warning: When you use bitwise logic operators, always wrap the expressions on either side in parentheses (as above) to make sure they're evaluated in the right order. Without the parentheses, operator precedence rules would cause the

expression above to be evaluated as `ds.OptionType == ('C' & ds.TradePrice) > 2.00`, which would result in an extremely slow call into native Python, followed by a crash. Also note that the Python keywords `AND`, `OR`, and `NOT` do not work with Boolean arrays. Use `&`, `|`, or `~` instead.

More examples of filter combinations:

```
>>> # Define two filters
>>> f1 = (ds.TradeSize <= 3.00)
>>> f2 = (ds.TradePrice > 3.00)
```

True if both are True:

```
>>> f1 & f2
FastArray([ True,  True, False, False, False, False, False, False, False, False])
```

True if either one is True:

```
>>> f1 | f2
FastArray([ True,  True,  True, False, False,  True,  True,  True,  True, True])
```

The negation of the `f1` filter:

```
>>> ~f1
FastArray([False, False,  True,  True,  True,  True, False,  True, False, False])
```

If you have complex filter criteria you want to reuse, assigning variable names to your filters can make things easier. You can also store your filters in a Riptable Struct:

```
>>> s = rt.Struct()
>>> s.ds = ds
>>> s.callsover2 = callsover2
>>> s
#   Name         Type      Size               0       1       2
-   ----------   -------   ----------------   -----   -----   -----
0   ds           Dataset   10 rows x 7 cols
1   callsover2   bool      10                 False   False   False
```

### 1.3.6 Set Values in Columns with Masks and `rt.where()`

You can also use mask arrays to update values that meet the filter condition.

Note, though, that the values are updated in place, not copied!

Suppose you want to update all the puts to be marked as traded. The FastArray `filter()` method doesn't let you set new values, but you can use the following syntax:

```
>>> f = ds.OptionType == 'P'
>>> ds.Traded[f] = True
>>> ds
#   OSISymbol          UnderlyingSymbol    TradeDate   TradeSize   TradePrice  ␣
↪OptionType   Traded
-   ---------------   ----------------   ----------   ---------   ----------   ----------
↪   ------
0   VIX:200520:35:0   VIX                2020-03-03         3.00        13.40   P        ␣
```

(continues on next page)

(continued from previous page)

```
↪     True
1   AAPL:200417:255   AAPL              2020-03-19        1.00         27.50   P         ␣
↪     True
2   LITE:200619:82:   LITE              2020-03-24        5.00         14.80   P         ␣
↪     True
3   SPY:200406:265:   SPY               2020-04-06       50.00          0.14   C         ␣
↪    False
4   MFA:200515:2:0:   MFA               2020-04-20       10.00          0.29   C         ␣
↪     True
5   XOM:220121:60:0   XOM               2020-04-23        5.00          3.75   C         ␣
↪     True
6   CCL:200717:12:5   CCL               2020-04-27        1.00          2.55   C         ␣
↪    False
7   AXSM:200515:85:   AXSM              2020-05-01        6.00          7.79   C         ␣
↪     True
8   UBER:200515:33:   UBER              2020-05-13        3.00          0.77   C         ␣
↪     True
9   TLT:200529:165:   TLT               2020-05-26        1.00          1.78   P         ␣
↪     True
```

What if you want to provide one value where the mask is True and a different value where the mask is False?

`rt.where()` is a function that works as an if-then-else procedure.

It takes three arguments:

- `condition`
- `x`
- `y`

Where the condition is met, it returns `x`; otherwise, it returns `y`. (If `x` or `y` is an array, the value that corresponds to the True or False is used.)

Here, for instance, `rt.where` returns `a` where `a < 5`; otherwise it returns `10 * a`:

```
>>> a = rt.FA([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> rt.where(a < 5, a, 10 * a)
FastArray([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

In the Dataset above, we can have `rt.where()` mark puts as traded and calls as not traded. Note that `rt.where()` returns a FastArray, so the result needs to be assigned as a Dataset column.

```
>>> ds.Traded = rt.where(ds.OptionType == 'P', True, False)
>>> ds[['OptionType', 'Traded']]
#   OptionType   Traded
-   ----------   ------
0   P              True
1   P              True
2   P              True
3   C             False
4   C             False
5   C             False
6   C             False
7   C             False
```

(continues on next page)

(continued from previous page)

```
8   C             False
9   P              True
```

### 1.3.7 Operate on Filtered Data in a Dataset

Looking at filtered data can provide some useful insights. But often, you want to operate on it.

Say you want to compute the total size of options that were traded. Given that we just covered `filter()`, you might be tempted to do this:

```
>>> ds.filter(ds.Traded).TradeSize.nansum()
303.0
```

However, remember that `filter()` returns a copy of the Dataset, filtered by the mask array. This is unnecessary here – we're only interested in the subset of one column of data. Fortunately, there are a couple of ways to work only on the data we need.

We can pass a filter argument to `nansum()` with the Boolean array contained in `ds.Traded`:

```
>>> ds.TradeSize.nansum(filter=ds.Traded)
303.0
```

This gets the sum of only the values in the TradeSize column that meet the filter criteria.

Note that the `filter=` is needed here – if you just pass the Boolean array by itself, the array will be silently ignored:

```
>>> ds.TradeSize.nansum(ds.Traded)
384.0
```

Alternatively, we can use the FastArray `filter()` method to get only the sizes for the options that were traded. Then we get the sum:

```
>>> ds.TradeSize.filter(ds.Traded).nansum()
303.0
```

Both of these methods are much more memory-friendly and computationally efficient than filtering (and making a copy of) the entire Dataset.

Next, we'll check out Riptable's datetime objects: *Work with Dates and Times*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.4 Work with Dates and Times

In Riptable, there are three fundamental date and time classes:

- `rt.Date`, used for date information with no time attached to it.
- `rt.DateTimeNano`, used for data with both date and time information (including time zone), to nanosecond precision.
- `rt.TimeSpan`, used for "time since midnight data," with no date information attached.

Here, we'll cover how to create date and time objects, how to extract data from these objects, how to use date and time arithmetic to build useful date and time representations, and how to reformat date and time information for display.

### 1.4.1 `Date` Objects

A Date object stores an array of dates with no time data attached. You can create Date arrays from strings, integer date values, or Matlab ordinal dates. For Matlab details, see Matlab Dates and Times.

Creating Date arrays from strings is fairly common. If your string dates are in YYYYMMDD format, you can simply pass the list of strings to `rt.Date()`:

```
>>> rt.Date(['20210101', '20210519', '20220308'])
Date(['2021-01-01', '2021-05-19', '2022-03-08'])
```

If your string dates are in another format, you can tell `rt.Date()` what to expect using Python `strptime` format code:

```
>>> rt.Date(['12/31/19', '6/30/19', '02/21/19'], format='%m/%d/%y')
Date(['2019-12-31', '2019-06-30', '2019-02-21'])
```

For a list of format codes and `strptime` implementation details, see Python's 'strftime' cheatsheet. The formatting codes are the same for `strftime` and `strptime`.

Note: Under the hood, dates are stored as integers – specifically, as the number of days since the Unix epoch, 01-01-1970:

```
>>> date_arr = rt.Date(['19700102', '19700103', '19700212'])
>>> date_arr._fa
FastArray([ 1,  2, 42])
```

Dates have various properties (a.k.a. attributes) that give you information about a Date.

Let's create a Dataset with a column of Dates, then use Date properties to extract information into new columns:

```
>>> ds = rt.Dataset()
>>> # Generate a range of dates, spaced 15 days apart
>>> ds.Dates = rt.Date.range('2019-01-01', '2019-02-30', step=15)
>>> # Some useful Date properties
>>> ds.Year = ds.Dates.year
>>> ds.Month = ds.Dates.month  # 1=Jan, 12=Dec
>>> ds.Day_of_Month = ds.Dates.day_of_month
>>> ds.Day_of_Week = ds.Dates.day_of_week  # 0=Mon, 6=Sun
>>> ds.Day_of_Year = ds.Dates.day_of_year
>>> ds
#        Dates    Year   Month   Day_of_Month   Day_of_Week   Day_of_Year
-   ----------   -----   -----   ------------   -----------   -----------
0   2019-01-01   2,019       1              1             1             1
1   2019-01-16   2,019       1             16             2            16
2   2019-01-31   2,019       1             31             3            31
3   2019-02-15   2,019       2             15             4            46
```

The following two properties are particularly useful when you want to group data by month or week. We'll see some examples when we talk about Categoricals and Accums:

```
>>> ds.Start_of_Month = ds.Dates.start_of_month
>>> ds.Start_of_Week = ds.Dates.start_of_week  # Returns the date of the previous Monday
```

(continues on next page)

(continued from previous page)

```
>>> ds
#        Dates    Year   Month   Day_of_Month   Day_of_Week   Day_of_Year   Start_of_
↪Month   Start_of_Week
-   ----------   -----   -----   ------------   -----------   -----------   ----------
↪-   -------------
0   2019-01-01   2,019       1              1             1             1       2019-01-
↪01      2018-12-31
1   2019-01-16   2,019       1             16             2            16       2019-01-
↪01      2019-01-14
2   2019-01-31   2,019       1             31             3            31       2019-01-
↪01      2019-01-28
3   2019-02-15   2,019       2             15             4            46       2019-02-
↪01      2019-02-11
```

We used Python's `strptime` format code above to tell `rt.Date()` how to parse our data. Riptable date and time objects can also use the `strftime()` method to format data for display:

```
>>> ds.MonthYear = ds.Dates.strftime('%b%y')
>>> ds.col_filter(['Dates', 'MonthYear'])
#        Dates   MonthYear
-   ----------   ---------
0   2019-01-01   Jan19
1   2019-01-16   Jan19
2   2019-01-31   Jan19
3   2019-02-15   Feb19
```

You can do some arithmetic with date and time objects. For example, we can get the number of days between two dates by subtracting one date from another:

```
>>> date_span = ds.Dates.max() - ds.Dates.min()
>>> date_span
DateSpan(['45 days'])
```

This returns a DateSpan object, which is a way to represent the delta, or duration, between two dates. You can convert it to an integer if you prefer:

```
>>> date_span.astype(int)
FastArray([45])
```

If you add a DateSpan to a Date, you get a Date:

```
>>> ds.Dates.min() + date_span
Date(['2019-02-15'])
```

Subtracting an array of dates from an array of dates gives you an array of DateSpans. The two Date arrays must be the same length:

```
>>> ds.DateDiff = ds.Dates - ds.Start_of_Month
>>> ds.col_filter(['Dates', 'Start_of_Month', 'DateDiff'])
#        Dates   Start_of_Month   DateDiff
-   ----------   --------------   --------
0   2019-01-01       2019-01-01     0 days
1   2019-01-16       2019-01-01    15 days
```

(continues on next page)

(continued from previous page)

```
2   2019-01-31       2019-01-01    30 days
3   2019-02-15       2019-02-01    14 days
```

Or you can subtract one Date from every record in a Date array:

```
>>> ds.Dates2 = ds.Dates - rt.Date('20190102')
>>> ds.col_filter(['Dates', 'Dates2'])
#        Dates    Dates2
-   ----------   -------
0   2019-01-01   -1 days
1   2019-01-16   14 days
2   2019-01-31   29 days
3   2019-02-15   44 days
```

### 1.4.2 DateTimeNano Objects

A DateTimeNano object stores data that has both date and time information, with the time specified to nanosecond precision.

Like Date objects, DateTimeNano objects can be created from strings. Strings are common when the data is from, say, a CSV file.

Unlike Date objects, DateTimeNanos are time-zone-aware. When you create a DateTimeNano, you need to specify the time zone of origin with the from_tz argument. Since Riptable is mainly used for financial market data, its time zone options are limited to NYC, DUBLIN, and (as of Riptable 1.3.6) Australia/Sydney, plus GMT and UTC (which is an alias for GMT).

(If you're wondering why 'Australia/Sydney' isn't abbreviated, it's because Riptable uses the standard time zone name from the tz database. In the future, Riptable will support only the standard names in the tz database.)

```
>>> rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17'], from_tz='GMT')
DateTimeNano(['20210101 04:31:15.000000000', '20210519 01:21:17.000000000'], to_tz='NYC')
```

Notice that the DateTimeNano is returned with to_tz='NYC'. This is the time zone the data is displayed in; NYC is the default. You can change the display time zone when you create the DateTimeNano by using to_tz:

```
>>> time_arr = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17'],
...                            from_tz='GMT', to_tz='GMT')
>>> time_arr
DateTimeNano(['20210101 09:31:15.000000000', '20210519 05:21:17.000000000'], to_tz='GMT')
```

And as with Dates, you can specify the format of your string data:

```
>>> rt.DateTimeNano(['12/31/19', '6/30/19'], format='%m/%d/%y', from_tz='NYC')
DateTimeNano(['20191231 00:00:00.000000000', '20190630 00:00:00.000000000'], to_tz='NYC')
```

When you're dealing with large amounts of data, it's more typical to get dates and times that are represented as nanoseconds since the Unix epoch (01-01-1970). In fact, that is how DateTimeNano objects are stored (it's much more efficient to store numbers than strings):

```
>>> time_arr._fa
FastArray([1609493475000000000, 1621401677000000000], dtype=int64)
```

If your data comes in this way, rt.DateTimeNano() can convert it easily. Just supply the time zone:

```
>>> rt.DateTimeNano([1609511475000000000, 1621416077000000000], from_tz='NYC')
DateTimeNano(['20210101 14:31:15.000000000', '20210519 09:21:17.000000000'], to_tz='NYC')
```

To split the date off a DateTimeNano, use `rt.Date()`:

```
>>> rt.Date(time_arr)
Date(['2021-01-01', '2021-05-19'])
```

To get the time, use `time_since_midnight()`:

```
>>> time_arr.time_since_midnight()
TimeSpan(['09:31:15.000000000', '05:21:17.000000000'])
```

Note that the result is a TimeSpan. We'll look at these more in the next section.

You can also get the time in nanoseconds since midnight:

```
>>> time_arr.nanos_since_midnight()
FastArray([34275000000000, 19277000000000], dtype=int64)
```

DateTimeNanos can be reformatted for display using `strftime()`:

```
>>> time_arr.strftime('%m/%d/%y %H:%M:%S')  # Date and time
array(['01/01/21 09:31:15', '05/19/21 05:21:17'], dtype=object)
```

Just the time:

```
>>> time_arr.strftime('%H:%M:%S')
array(['09:31:15', '05:21:17'], dtype=object)
```

Some arithmetic:

```
>>> # Create two DateTimeNano arrays
>>> time_arr1 = rt.DateTimeNano(['20220101 12:00:00', '20220301 13:00:00'], from_tz='NYC
↪', to_tz='NYC')
>>> time_arr2 = rt.DateTimeNano(['20190101 11:00:00', '20190301 11:30:00'], from_tz='NYC
↪', to_tz='NYC')
```

DateTimeNano - DateTimeNano = TimeSpan

```
>>> timespan1 = time_arr1 - time_arr2
>>> timespan1
TimeSpan(['1096d 01:00:00.000000000', '1096d 01:30:00.000000000'])
```

DateTimeNano + TimeSpan = DateTimeNano

```
>>> dtn1 = time_arr1 + timespan1
>>> dtn1
DateTimeNano(['20250101 13:00:00.000000000', '20250301 14:30:00.000000000'], to_tz='NYC')
```

DateTimeNano - TimeSpan = DateTimeNano

```
>>> dtn2 = dtn1 - timespan1
>>> dtn2
DateTimeNano(['20220101 12:00:00.000000000', '20220301 13:00:00.000000000'], to_tz='NYC')
```

### 1.4.3 TimeSpan Objects

You saw above how a TimeSpan represents a duration of time between two DateTimeNanos. You can also think of it as a representation of a time of day.

Recall that you can split a TimeSpan off a DateTimeNano using time_since_midnight(). Just keep in mind that a TimeSpan by itself has no absolute reference to Midnight of any day in particular.

As an example, let's say you want to find out which trades were made before a certain time of day (on any day). If your data has DateTimeNanos, you can split off the TimeSpan, then filter for the times you're interested in:

```
>>> rng = np.random.default_rng(seed=42)
>>> ds = rt.Dataset()
>>> N = 100  # Length of the Dataset
>>> ds.Symbol = rt.FA(rng.choice(['AAPL', 'AMZN', 'TSLA', 'SPY', 'GME'], N))
>>> ds.Size = rng.random(N) * 100
>>> # Create a column of randomly generated DateTimeNanos
>>> ds.TradeDateTime = rt.DateTimeNano.random(N)
>>> ds.TradeTime = ds.TradeDateTime.time_since_midnight()
>>> ds
  #   Symbol    Size                 TradeDateTime            TradeTime
---   ------   -----   ---------------------------   ------------------
  0   AAPL     19.99   20190614 13:07:21.352420597   13:07:21.352420597
  1   SPY       0.74   19970809 19:34:40.178693393   19:34:40.178693393
  2   SPY      78.69   19861130 20:06:31.775222495   20:06:31.775222495
  3   TSLA     66.49   20081111 04:15:24.079385833   04:15:24.079385833
  4   TSLA     70.52   20190419 06:21:31.197889103   06:21:31.197889103
  5   GME      78.07   19861112 05:20:14.239289462   05:20:14.239289462
  6   AAPL     45.89   20110329 20:55:07.198530171   20:55:07.198530171
  7   SPY      56.87   19780303 03:19:32.676920289   03:19:32.676920289
  8   AMZN     13.98   19930305 22:34:02.767331408   22:34:02.767331408
  9   AAPL     11.45   19840723 04:08:10.118105881   04:08:10.118105881
 10   TSLA     66.84   19940814 03:08:03.730164619   03:08:03.730164619
 11   GME      47.11   19730612 22:33:46.871406555   22:33:46.871406555
 12   SPY      56.52   19840118 14:01:10.111423986   14:01:10.111423986
 13   SPY      76.50   19740813 15:26:44.457459450   15:26:44.457459450
 14   SPY      63.47   20050106 18:13:57.982489010   18:13:57.982489010
...   ...        ...                           ...                   ...
 85   SPY       2.28   19930706 00:24:05.337093375   00:24:05.337093375
 86   AAPL     95.86   20140823 11:35:14.816318096   11:35:14.816318096
 87   AMZN     48.23   20070929 22:49:10.456157805   22:49:10.456157805
 88   SPY      78.27   19930616 20:30:27.490477141   20:30:27.490477141
 89   GME       8.27   19860626 07:48:16.756213658   07:48:16.756213658
 90   TSLA     48.67   20060824 19:29:19.583638324   19:29:19.583638324
 91   GME      49.07   19751026 20:29:32.616225869   20:29:32.616225869
 92   GME      93.78   19911222 14:53:30.879285646   14:53:30.879285646
 93   AMZN     57.17   19970715 20:26:36.179803660   20:26:36.179803660
 94   GME      47.35   19961214 10:26:16.609357094   10:26:16.609357094
 95   AMZN     26.70   19830606 14:02:30.699183111   14:02:30.699183111
 96   AMZN     33.16   19821114 05:56:13.504071773   05:56:13.504071773
 97   SPY      52.07   19740606 03:47:03.798827481   03:47:03.798827481
 98   SPY      43.89   19881226 22:19:55.209671459   22:19:55.209671459
 99   AAPL      2.16   19840720 11:51:26.734190049   11:51:26.734190049
```

If we want to find the trades that happened before 10:00 a.m., we need a TimeSpan that represents 10:00 a.m. Then we

can can compare our TradeTimes against it.

To construct a TimeSpan from scratch, you can pass time strings in `%H:%M:%S` format:

```
>>> rt.TimeSpan(['09:00', '10:45', '02:30', '15:00', '23:10'])
TimeSpan(['09:00:00.000000000', '10:45:00.000000000', '02:30:00.000000000', '15:00:00.
↪000000000', '23:10:00.000000000'])
```

Or from an array of numerics, along with a unit, like hours:

```
>>> rt.TimeSpan([9, 10, 12, 14, 18], unit='h')
TimeSpan(['09:00:00.000000000', '10:00:00.000000000', '12:00:00.000000000', '14:00:00.
↪000000000', '18:00:00.000000000'])
```

For our purposes, this will do:

```
>>> tenAM = rt.TimeSpan(10, unit='h')
>>> tenAM
TimeSpan(['10:00:00.000000000'])
```

Now we can compare the TradeTime values against it. We'll put the results of the comparison into a column so we can spot check them:

```
>>> ds.TradesBefore10am = (ds.TradeTime < tenAM)
>>> ds
  #   Symbol    Size                TradeDateTime            TradeTime  ␣
↪TradesBefore10am
---   ------   -----   --------------------------   ------------------   ---------------
↪-
  0   AAPL     19.99   20190614 13:07:21.352420597   13:07:21.352420597             ␣
↪False
  1   SPY       0.74   19970809 19:34:40.178693393   19:34:40.178693393             ␣
↪False
  2   SPY      78.69   19861130 20:06:31.775222495   20:06:31.775222495             ␣
↪False
  3   TSLA     66.49   20081111 04:15:24.079385833   04:15:24.079385833              ␣
↪True
  4   TSLA     70.52   20190419 06:21:31.197889103   06:21:31.197889103              ␣
↪True
  5   GME      78.07   19861112 05:20:14.239289462   05:20:14.239289462              ␣
↪True
  6   AAPL     45.89   20110329 20:55:07.198530171   20:55:07.198530171             ␣
↪False
  7   SPY      56.87   19780303 03:19:32.676920289   03:19:32.676920289              ␣
↪True
  8   AMZN     13.98   19930305 22:34:02.767331408   22:34:02.767331408             ␣
↪False
  9   AAPL     11.45   19840723 04:08:10.118105881   04:08:10.118105881              ␣
↪True
 10   TSLA     66.84   19940814 03:08:03.730164619   03:08:03.730164619              ␣
↪True
 11   GME      47.11   19730612 22:33:46.871406555   22:33:46.871406555             ␣
↪False
 12   SPY      56.52   19840118 14:01:10.111423986   14:01:10.111423986             ␣
↪False
```

(continues on next page)

(continued from previous page)

```
 13   SPY      76.50   19740813 15:26:44.457459450   15:26:44.457459450             ␣
↪False
 14   SPY      63.47   20050106 18:13:57.982489010   18:13:57.982489010             ␣
↪False
...   ...       ...                           ...                   ...              ..
↪.
 85   SPY       2.28   19930706 00:24:05.337093375   00:24:05.337093375              ␣
↪True
 86   AAPL     95.86   20140823 11:35:14.816318096   11:35:14.816318096             ␣
↪False
 87   AMZN     48.23   20070929 22:49:10.456157805   22:49:10.456157805             ␣
↪False
 88   SPY      78.27   19930616 20:30:27.490477141   20:30:27.490477141             ␣
↪False
 89   GME       8.27   19860626 07:48:16.756213658   07:48:16.756213658              ␣
↪True
 90   TSLA     48.67   20060824 19:29:19.583638324   19:29:19.583638324             ␣
↪False
 91   GME      49.07   19751026 20:29:32.616225869   20:29:32.616225869             ␣
↪False
 92   GME      93.78   19911222 14:53:30.879285646   14:53:30.879285646             ␣
↪False
 93   AMZN     57.17   19970715 20:26:36.179803660   20:26:36.179803660             ␣
↪False
 94   GME      47.35   19961214 10:26:16.609357094   10:26:16.609357094             ␣
↪False
 95   AMZN     26.70   19830606 14:02:30.699183111   14:02:30.699183111             ␣
↪False
 96   AMZN     33.16   19821114 05:56:13.504071773   05:56:13.504071773              ␣
↪True
 97   SPY      52.07   19740606 03:47:03.798827481   03:47:03.798827481              ␣
↪True
 98   SPY      43.89   19881226 22:19:55.209671459   22:19:55.209671459             ␣
↪False
 99   AAPL      2.16   19840720 11:51:26.734190049   11:51:26.734190049             ␣
↪False
```

And of course, we can use the Boolean array to filter the Dataset:

```
>>> ds.filter(ds.TradesBefore10am)
  #   Symbol    Size                TradeDateTime            TradeTime  ␣
↪TradesBefore10am
---   ------   -----   -------------------------   -----------------   ---------------
↪-
  0   TSLA     66.49   20081111 04:15:24.079385833   04:15:24.079385833              ␣
↪True
  1   TSLA     70.52   20190419 06:21:31.197889103   06:21:31.197889103              ␣
↪True
  2   GME      78.07   19861112 05:20:14.239289462   05:20:14.239289462              ␣
↪True
  3   SPY      56.87   19780303 03:19:32.676920289   03:19:32.676920289              ␣
↪True
```

(continues on next page)

(continued from previous page)

```
  4   AAPL     11.45   19840723 04:08:10.118105881   04:08:10.118105881              ␣
↪True
  5   TSLA     66.84   19940814 03:08:03.730164619   03:08:03.730164619              ␣
↪True
  6   SPY      55.36   20010615 00:14:45.718385740   00:14:45.718385740              ␣
↪True
  7   GME      23.39   19751116 06:06:50.777397710   06:06:50.777397710              ␣
↪True
  8   TSLA     29.36   19920606 01:44:12.762930709   01:44:12.762930709              ␣
↪True
  9   GME      66.19   20150907 07:56:58.291001076   07:56:58.291001076              ␣
↪True
 10   GME      46.19   19771105 07:18:54.592658284   07:18:54.592658284              ␣
↪True
 11   SPY      50.10   19980211 08:39:58.366644251   08:39:58.366644251              ␣
↪True
 12   AAPL     15.23   19840811 03:03:32.341618015   03:03:32.341618015              ␣
↪True
 13   AMZN     38.10   19730321 08:49:53.629495873   08:49:53.629495873              ␣
↪True
 14   AAPL     30.15   20091103 04:56:46.941815206   04:56:46.941815206              ␣
↪True
...   ...        ...                           ...                   ...               ..
↪.
 19   GME      75.85   19870605 00:16:50.617990376   00:16:50.617990376              ␣
↪True
 20   AAPL     43.21   19880730 01:20:25.325405869   01:20:25.325405869              ␣
↪True
 21   AAPL     64.98   19750705 03:28:57.626851689   03:28:57.626851689              ␣
↪True
 22   AAPL     41.58   19900712 07:39:20.866244793   07:39:20.866244793              ␣
↪True
 23   SPY       4.16   20090512 03:17:20.112309966   03:17:20.112309966              ␣
↪True
 24   AMZN     32.99   20010910 02:18:44.384567415   02:18:44.384567415              ␣
↪True
 25   AMZN     14.45   19901004 00:53:54.407173923   00:53:54.407173923              ␣
↪True
 26   TSLA     10.34   19961220 04:54:14.777983172   04:54:14.777983172              ␣
↪True
 27   SPY      58.76   20070922 04:55:14.156355503   04:55:14.156355503              ␣
↪True
 28   TSLA     92.51   19851209 01:52:03.199471749   01:52:03.199471749              ␣
↪True
 29   GME      34.69   20160202 09:57:41.083925341   09:57:41.083925341              ␣
↪True
 30   SPY       2.28   19930706 00:24:05.337093375   00:24:05.337093375              ␣
↪True
 31   GME       8.27   19860626 07:48:16.756213658   07:48:16.756213658              ␣
↪True
 32   AMZN     33.16   19821114 05:56:13.504071773   05:56:13.504071773              ␣
↪True
```

(continues on next page)

(continued from previous page)

```
 33   SPY      52.07   19740606 03:47:03.798827481   03:47:03.798827481              ␣
↪True
```

If we only want to see certain columns of the Dataset, we can combine the filter with slicing:

```
>>> ds[ds.TradesBefore10am, ['Symbol', 'Size']]
  #   Symbol    Size
---   ------   -----
  0   TSLA     66.49
  1   TSLA     70.52
  2   GME      78.07
  3   SPY      56.87
  4   AAPL     11.45
  5   TSLA     66.84
  6   SPY      55.36
  7   GME      23.39
  8   TSLA     29.36
  9   GME      66.19
 10   GME      46.19
 11   SPY      50.10
 12   AAPL     15.23
 13   AMZN     38.10
 14   AAPL     30.15
...   ...        ...
 19   GME      75.85
 20   AAPL     43.21
 21   AAPL     64.98
 22   AAPL     41.58
 23   SPY       4.16
 24   AMZN     32.99
 25   AMZN     14.45
 26   TSLA     10.34
 27   SPY      58.76
 28   TSLA     92.51
 29   GME      34.69
 30   SPY       2.28
 31   GME       8.27
 32   AMZN     33.16
 33   SPY      52.07
```

Or if we just want the total size of AAPL trades before 10am:

```
>>> aapl10 = (ds.Symbol == 'AAPL') & (ds.TradesBefore10am)
>>> ds.Size.nansum(filter = aapl10)
274.92741837733035
```

### Other Useful things to Do with TimeSpans

We can compare two `DateTimeNano` columns to find times that are close together – for example, those less than 10ms apart.

To illustrate this, we'll create some randomly generated small `TimeSpan`s to add to our column of `DateTimeNano`s:

```
>>> # Create TimeSpans from 1 millisecond to 19 milliseconds
>>> some_ms = rt.TimeSpan(rng.integers(low=1, high=20, size=N), 'ms')
>>> # Offset the TimeSpans in our original DateTimeNano
>>> ds.TradeDateTime2 = ds.TradeDateTime + some_ms
>>> ds.col_filter(['Symbol', 'TradeDateTime', 'TradeDateTime2']).head()
 #   Symbol   TradeDateTime                TradeDateTime2
--   ------   --------------------------   --------------------------
 0   AAPL     20100614 01:47:46.306210225   20100614 01:47:46.313210225
 1   SPY      20131004 12:02:28.251037257   20131004 12:02:28.267037257
 2   SPY      19721212 00:54:12.641763127   19721212 00:54:12.642763127
 3   TSLA     19720118 19:33:36.911790260   19720118 19:33:36.929790260
 4   TSLA     19750331 15:04:15.847968984   19750331 15:04:15.858968984
 5   GME      19740912 18:18:46.660464416   19740912 18:18:46.663464416
 6   AAPL     19820906 09:31:02.911852383   19820906 09:31:02.917852383
 7   SPY      19900810 10:42:02.603793160   19900810 10:42:02.614793160
 8   AMZN     19870318 06:54:30.389382275   19870318 06:54:30.395382275
 9   AAPL     20031029 09:53:06.898676308   20031029 09:53:06.901676308
10   TSLA     20160319 00:33:40.035581577   20160319 00:33:40.048581577
11   GME      19801024 01:38:46.310440408   19801024 01:38:46.323440408
12   SPY      19791105 17:08:46.460502123   19791105 17:08:46.463502123
13   SPY      20110304 07:11:03.437823831   20110304 07:11:03.443823831
14   SPY      20140303 01:58:10.917868743   20140303 01:58:10.922868743
15   SPY      19990514 19:33:06.261903491   19990514 19:33:06.274903491
16   TSLA     19840808 16:34:56.776803922   19840808 16:34:56.790803922
17   AAPL     19711222 11:39:46.898769893   19711222 11:39:46.912769893
18   GME      20090605 13:23:02.120390523   20090605 13:23:02.138390523
19   TSLA     19900227 19:36:40.067192555   19900227 19:36:40.082192555
```

Now we can find the trades that occurred within 10ms of each other, and again put the results into a new column for a spot check.

```
>>> ds.Within10ms = (abs(ds.TradeDateTime.time_since_midnight()
...                   - ds.TradeDateTime2.time_since_midnight())) < rt.TimeSpan(10, 'ms')
>>> ds.col_filter(['Symbol', 'TradeDateTime', 'TradeDateTime2', 'Within10ms']).head()
 #   Symbol   TradeDateTime                TradeDateTime2               Within10ms
--   ------   --------------------------   --------------------------   ----------
 0   AAPL     19771006 11:46:39.512132962   19771006 11:46:39.519132962         True
 1   SPY      20000321 15:00:25.630646023   20000321 15:00:25.646646023        False
 2   SPY      19720130 05:36:37.195744004   19720130 05:36:37.196744004         True
 3   TSLA     19960902 00:45:11.619930786   19960902 00:45:11.637930786        False
 4   TSLA     19901216 15:52:53.935112408   19901216 15:52:53.946112408        False
 5   GME      19900910 22:20:09.846455444   19900910 22:20:09.849455444         True
 6   AAPL     20000825 20:59:19.248822244   20000825 20:59:19.254822244         True
 7   SPY      19740216 18:32:16.051989951   19740216 18:32:16.062989951        False
 8   AMZN     19951222 07:27:43.668483372   19951222 07:27:43.674483372         True
 9   AAPL     20180708 11:19:48.016609690   20180708 11:19:48.019609690         True
10   TSLA     20110429 21:11:34.789939106   20110429 21:11:34.802939106        False
```

(continues on next page)

(continued from previous page)

```
11   GME      19921202 20:27:45.957970537   19921202 20:27:45.970970537        False
12   SPY      19980801 10:04:29.793513895   19980801 10:04:29.796513895         True
13   SPY      19970217 08:00:06.615346852   19970217 08:00:06.621346852         True
14   SPY      20060915 20:18:28.369763536   20060915 20:18:28.374763536         True
15   SPY      19991220 16:10:56.841720714   19991220 16:10:56.854720714        False
16   TSLA     19730131 01:08:43.413049524   19730131 01:08:43.427049524        False
17   AAPL     20040518 15:53:50.561136824   20040518 15:53:50.575136824        False
18   GME      19710809 14:51:55.347200052   19710809 14:51:55.365200052        False
19   TSLA     19980613 01:40:56.278221632   19980613 01:40:56.293221632        False
```

And again we can use the result as a mask array:

```
>>> ds[ds.Within10ms, ['Symbol', 'Size']]
  #   Symbol    Size
---   ------   -----
  0   AAPL     19.99
  1   SPY      78.69
  2   GME      78.07
  3   AAPL     45.89
  4   AMZN     13.98
  5   AAPL     11.45
  6   SPY      56.52
  7   SPY      76.50
  8   SPY      63.47
  9   TSLA     21.46
 10   AMZN     40.85
 11   SPY      28.14
 12   TSLA     29.36
 13   GME      66.19
 14   TSLA     55.70
...   ...        ...
 37   TSLA     49.40
 38   TSLA     10.34
 39   SPY      58.76
 40   GME      17.06
 41   GME      34.69
 42   SPY      59.09
 43   SPY       2.28
 44   AAPL     95.86
 45   GME       8.27
 46   GME      49.07
 47   GME      93.78
 48   AMZN     33.16
 49   SPY      52.07
 50   SPY      43.89
 51   AAPL      2.16
```

A common situation is having dates as date strings and times in nanos since midnight. You can use some arithmetic to build a DateTimeNano: `Date` + `TimeSpan` = `DateTimeNano`:

```
>>> ds = rt.Dataset({
...     'Date': ['20111111', '20200202', '20220222'],
```

(continues on next page)

(continued from previous page)

```
...     'Time': [44_275_000_000_000, 39_287_000_000_000, 55_705_000_000_000]
...     })
>>> # Convert the date strings to rt.Date objects
>>> ds.Date = rt.Date(ds.Date)
>>> # Convert the times to rt.TimeSpan objects
>>> ds.Time = rt.TimeSpan(ds.Time)
>>> ds
#         Date                 Time
-   ----------   ------------------
0   2011-11-11   12:17:55.000000000
1   2020-02-02   10:54:47.000000000
2   2022-02-22   15:28:25.000000000
```

At this point, you might want to simply add `ds.Date` and `ds.Time` to get a `DateTimeNano`:

```
>>> ds.DateTime = ds.Date + ds.Time
>>> ds
#         Date                 Time                         DateTime
-   ----------   ------------------   ----------------------------
0   2011-11-11   12:17:55.000000000   20111111 12:17:55.000000000
1   2020-02-02   10:54:47.000000000   20200202 10:54:47.000000000
2   2022-02-22   15:28:25.000000000   20220222 15:28:25.000000000
```

And that seems to work. However, remember that `DateTimeNano`s need to have a time zone. Here, GMT was assumed:

```
>>> ds.DateTime
DateTimeNano(['20111111 12:17:55.000000000', '20200202 10:54:47.000000000', '20220222 15:
↪28:25.000000000'], to_tz='GMT')
```

Specify your desired time zone so you don't end up with unexpected results down the line:

```
>>> ds.DateTime2 = rt.DateTimeNano((ds.Date + ds.Time), from_tz='NYC')
>>> ds.DateTime2
DateTimeNano(['20111111 12:17:55.000000000', '20200202 10:54:47.000000000', '20220222 15:
↪28:25.000000000'], to_tz='NYC')
```

Warning: Given that `TimeSpan + Date = DateTimeNano`, and also that you can use `rt.Date(my_dtn)` to get a `Date` from a `DateTimeNano`, you might reasonably think you can get the `TimeSpan` from a `DateTimeNano` using `rt.TimeSpan(my_dtn)`.

However, that result includes the number of days since January 1, 1970. To get the `TimeSpan` from a `DateTimeNano`, use `time_since_midnight()` instead.

| Datetime Arithmetic |
|---|
| Date + Date = TypeError |
| Date + DateTimeNano = TypeError |
| Date + DateSpan = Date |
| Date + TimeSpan = DateTimeNano |
| |
| Date - Date = DateSpan |
| Date - DateSpan = Date |
| Date - DateTimeNano = TimeSpan |
| Date - TimeSpan = DateTimeNano |
| |
| DateTimeNano - DateTimeNano = TimeSpan |
| DateTimeNano - TimeSpan = DateTimeNano |
| DateTimeNano + TimeSpan = DateTimeNano |
| |
| TimeSpan - TimeSpan = TimeSpan |
| TimeSpan + TimeSpan = TimeSpan |

Next, we'll look at Riptable's vehicle for group operations: *Perform Group Operations with Categoricals*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.5 Perform Group Operations with Categoricals

Riptable Categoricals have two related uses:

- They efficiently store string (or other large dtype) arrays that have repeated values. The repeated values are partitioned into groups (a.k.a. categories), and each group is mapped to an integer. For example, in a Categorical that contains three 'AAPL' symbols and four 'MSFT' symbols, the data is partitioned into an 'AAPL' group that's mapped to 1 and a 'MSFT' group that's mapped to 2. This integer mapping allows the data to be stored and operated on more efficiently.
- They're Riptable's class for doing group operations. A method applied to a Categorical is applied to each group separately.

We'll talk about group operations first, then look at how Categoricals store data under the hood.

Here's a simple Dataset with repeated stock symbols and some values:

```
>>> ds = rt.Dataset()
>>> ds.Symbol = rt.FA(['AAPL', 'MSFT', 'AAPL', 'TSLA', 'MSFT', 'TSLA'])
>>> ds.Value = rt.FA([5, 10, 15, 20, 25, 30])
>>> ds
#   Symbol   Value
-   ------   -----
0   AAPL         5
1   MSFT        10
2   AAPL        15
3   TSLA        20
4   MSFT        25
5   TSLA        30
```

### 1.5.1 Categoricals for Group Operations

We know how to get the sum of the Value column:

```
>>> ds.Value.sum()
105
```

Categoricals make it just as easy to get the sum for each symbol.

Use the Categorical constructor to turn the Symbol column into a Categorical:

```
>>> ds.Symbol = rt.Categorical(ds.Symbol)  # Note: rt.Cat() also works.
```

Now we call the `sum()` method on the Categorical, passing it the data we want to sum for each group:

```
>>> ds.Symbol.sum(ds.Value)
*Symbol   Value
-------   -----
AAPL         20
MSFT         35
TSLA         50
```

A Dataset is returned containing the groups from the Categorical and the result of the operation we called on each group.

Note the prepended '*' in the Symbol column. This indicates that the column was used as the grouping variable in an operation.

### 1.5.2 Categoricals as Split, Apply, Combine Operations

Hadley Wickham, known for his work on Rstats, described the operation (also known as a "group by" operation) as *split, apply, combine*.

The illustration below shows how the groups are split based on the "keys" (or, in Riptable's case, the Categorical values). The sum method is then applied to each group separately, and the results are combined into an output array.

### 1.5.3 Operations Supported by Categoricals

Categoricals support most common reducing functions, which return one value per group. Some of the more common ones:

| Reducing Function | Description |
|---|---|
| `count()` | Total number of items |
| `first()`, `last()` | First item, last item |
| `mean()`, `median()` | Mean, median |
| `min()`, `max()` | Minimum, maximum |
| `std()`, `var()` | Standard deviation, variance |
| `prod()` | Product of all items |
| `sum()` | Sum of all items |

Here's the *complete list of Categorical reducing functions*.

Categoricals also support non-reducing functions. But because non-reducing functions return one value for each value in the original data, the results are a little different.

For example, take `cumsum()`, which is a running total.

When it's applied to a Categorical, the function does get applied to each group separately. However, the returned Dataset has one result per value of the original data:

```
>>> ds.Value2 = rt.FA([2, 10, 5, 25, 8, 20])
>>> ds.Symbol.cumsum(ds.Value2)
#   Value2
-   ------
0        2
1       10
2        7
3       25
4       18
5       45
```

The alignment of the result to the original data is easier to see if you add the results to the Dataset:

```
>>> ds.CumValue2 = ds.Symbol.cumsum(ds.Value2)
>>> # Sort to make the cumulative sum per group more clear, then display only the␣
↪relevant columns.
>>> ds.sort_copy('Symbol').col_filter(['Symbol', 'Value2', 'CumValue2'])
#   Symbol   Value2   CumValue2
-   ------   ------   ---------
0   AAPL          2           2
1   AAPL          5           7
2   MSFT         10          10
3   MSFT          8          18
4   TSLA         25          25
5   TSLA         20          45
```

A commonly used non-reducing function is `shift()`. You can use it to compare values with shifted versions of themselves – for example, today's price compared to yesterday's price, the volume compared to the volume an hour ago, etc.

Where a category has no previous value to shift forward, the missing value is filled with an invalid value (e.g., `Inv` for integers or `nan` for floats):

```
>>> ds.PrevValue = ds.Symbol.shift(ds.Value)
>>> ds.col_filter(['Symbol', 'Value', 'PrevValue'])
#   Symbol   Value   PrevValue
-   ------   -----   ---------
0   AAPL         5         Inv
1   MSFT        10         Inv
2   AAPL        15           5
3   TSLA        20         Inv
4   MSFT        25          10
5   TSLA        30          20
```

Other non-reducing functions include `rolling_sum()`, `rolling_mean()` and their nan-versions `rolling_nansum()` and `rolling_nanmean()`, and `cumsum()` and `cumprod()`.

Other functions not listed here can also be applied to Categoricals, including lambda functions and other user-defined functions, with the help of `apply()`. More on that below.

### 1.5.4 Expand the Results of Reducing Operations with `transform`

Notice that if we try to add the result of a *reducing* operation to a Dataset, Riptable complains that the result isn't the right length:

```
>>> try:
...     ds.Mean = ds.Symbol.mean(ds.Value)
... except TypeError as e:
...     print("TypeError:", e)
TypeError: ('Row mismatch in Dataset._check_addtype.  Tried to add Dataset of different␣
↪lengths', 6, 3)
```

You can expand the result of a reducing function so that it's aligned with the original data by passing `transform=True` to the function:

```
>>> ds.MaxValue = ds.Symbol.max(ds.Value, transform=True)
>>> ds.sort_copy(['Symbol', 'Value']).col_filter(['Symbol', 'Value', 'MaxValue'])
#   Symbol   Value   MaxValue
-   ------   -----   --------
0   AAPL         5         15
1   AAPL        15         15
2   MSFT        10         25
3   MSFT        25         25
4   TSLA        20         30
5   TSLA        30         30
```

The max value per symbol is repeated for every instance of the symbol.

### 1.5.5 Apply an Operation to Multiple Columns or a Dataset

You can apply a function to multiple columns by passing a list of column names. Here's a reducing function applied to two columns:

```
>>> ds.Value3 = ds.Value * 2  # Add another column of data.
>>> ds.Symbol.max([ds.Value, ds.Value3])
*Symbol   Value   Value3
-------   -----   ------
AAPL         15       30
MSFT         25       50
TSLA         30       60
```

Note the syntax for adding the results of an operation on two columns to a Dataset. To be the right length for the Dataset, the results have to be from a non-reducing function or a reducing function that has `transform=True`:

```
>>> ds[['MaxValue', 'MaxValue3']] = ds.Symbol.max([ds.Value, ds.Value3],
...                                               transform=True)[['Value', 'Value3']]
#   Symbol   Value   Value3   MaxValue   MaxValue3
-   ------   -----   ------   --------   ---------
0   AAPL         5       10         15          30
1   AAPL        15       30         15          30
2   MSFT        10       20         25          50
3   MSFT        25       50         25          50
4   TSLA        20       40         30          60
5   TSLA        30       60         30          60
```

You can also apply a function to a whole Dataset. Any column for which the function fails – for example, a numerical function on a string column – is not returned:

```
>>> ds.OptionType = list("PC")*3  # Add a string column.
>>> ds.Symbol.max(ds)
*Symbol   Value   CumValue   Value3   MaxValue   MaxValue3
-------   -----   --------   ------   --------   ---------
AAPL         15         20       30         15          30
MSFT         25         35       50         25          50
TSLA         30         50       60         30          60
```

### 1.5.6 Categoricals for Storing Strings

To get a better sense of how Categoricals store data, let's look at one under the hood:

```
>>> ds.Symbol
Categorical([AAPL, MSFT, AAPL, TSLA, MSFT, TSLA]) Length: 6
  FastArray([1, 2, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'AAPL', b'MSFT', b'TSLA'], dtype='|S4') Unique count: 3
```

The first line shows the 6 symbols. You can access the array with `expand_array`:

```
>>> ds.Symbol.expand_array
FastArray([b'AAPL', b'MSFT', b'AAPL', b'TSLA', b'MSFT', b'TSLA'],
          dtype='|S8')
```

The second line is a FastArray of integers, with one integer for each unique category of the Categorical. It's accessible with `_fa`:

```
>>> ds.Symbol._fa
FastArray([1, 2, 1, 3, 2, 3], dtype=int8)
```

The list of unique categories is shown in the third line. You can access the list with `category_array`:

```
>>> ds.Symbol.category_array
FastArray([b'AAPL', b'MSFT', b'TSLA'], dtype='|S4')
```

It's the same thing we get if we do:

```
>>> ds.Symbol.unique()
FastArray([b'AAPL', b'MSFT', b'TSLA'], dtype='|S4')
```

We can get a better picture of the mapping by putting the integer FastArray into the Dataset:

```
>>> ds.Mapping = ds.Symbol._fa
>>> ds.col_filter(['Symbol', 'Mapping'])
#   Symbol   Mapping
-   ------   -------
0   AAPL           1
1   MSFT           2
2   AAPL           1
3   TSLA           3
4   MSFT           2
5   TSLA           3
```

Because it's much more efficient to pass around integers than it is to pass around strings, it's common for string data with repeated values to be stored using integer mapping.

If you have data stored as integers (for example, datetime data), you can create a Categorical using the integer array and an array of unique categories:

```
>>> c = rt.Categorical([1, 3, 2, 2, 1, 3, 3, 1], categories=['a','b','c'])
>>> c
Categorical([a, c, b, b, a, c, c, a]) Length: 8
  FastArray([1, 3, 2, 2, 1, 3, 3, 1]) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Notice that in this Categorical and the one we created above, the base index is 1, not 0.

This brings us to an important note about Categoricals: By default, the base index is 1; 0 is reserved for holding any values of the Categorical that are filtered out of operations on the Categorical.

Values can be filtered out of all operations or specific ones.

### 1.5.7 Filter Values or Categories from All Categorical Operations

When you create a Categorical, you can filter certain values or entire categories from all operations on it.

We'll start with filtering values. Say we have a Dataset with symbols 'A' and 'B' that are associated with exchanges 'X', 'Y', and 'Z'.

```
>>> # Create the Dataset.
>>> rng = np.random.default_rng(seed=42)
>>> N = 25
>>> symbol_exchange = rt.Dataset()
>>> symbol_exchange.Symbol = rt.FA(rng.choice(['A', 'B'], N))
>>> symbol_exchange.Exchange = rt.FA(rng.choice(['X', 'Y', 'Z'], N))
>>> symbol_exchange
 #   Symbol   Exchange
--   ------   --------
 0   B        Y
 1   A        X
 2   B        Z
 3   A        Y
 4   B        Y
 5   B        Y
 6   B        Y
 7   B        X
 8   A        Y
 9   B        Z
10   A        X
11   B        Y
12   B        X
13   A        Y
14   B        Y
15   B        Z
16   A        X
17   A        X
18   A        Y
19   A        Z
```

(continues on next page)

(continued from previous page)

```
20   A        Z
21   B        X
22   B        X
23   A        Z
24   B        X
```

We want to make the Symbol column a Categorical, but we're interested in only the symbol values that are associated with the 'X' exchange.

When we create the Categorical, we can use the `filter` keyword argument with a Boolean mask array that's True for symbol values associated with the 'X' exchange:

```
>>> exchangeX = symbol_exchange.Exchange == 'X'  # Create a mask array.
>>> c_x = rt.Cat(symbol_exchange.Symbol, filter=exchangeX)
```

When we view the Categorical, we can see that symbol values associated with exchanges 'Y' and 'Z' are shown as 'Filtered', and the 'Filtered' values are mapped to the 0 index in the integer array:

```
>>> c_x
Categorical([Filtered, A, Filtered, Filtered, Filtered, ..., Filtered, B, B, Filtered,␣
↪B]) Length: 25
  FastArray([0, 1, 0, 0, 0, ..., 0, 2, 2, 0, 2], dtype=int8) Base Index: 1
  FastArray([b'A', b'B'], dtype='|S1') Unique count: 2
```

To better see what's filtered, we can add it to the Dataset:

```
>>> symbol_exchange.Filtered = c_x
 #   Symbol   Exchange   Filtered
--   ------   --------   --------
 0   B        Y          Filtered
 1   A        X          A
 2   B        Z          Filtered
 3   A        Y          Filtered
 4   B        Y          Filtered
 5   B        Y          Filtered
 6   B        Y          Filtered
 7   B        X          B
 8   A        Y          Filtered
 9   B        Z          Filtered
10   A        X          A
11   B        Y          Filtered
12   B        X          B
13   A        Y          Filtered
14   B        Y          Filtered
15   B        Z          Filtered
16   A        X          A
17   A        X          A
18   A        Y          Filtered
19   A        Z          Filtered
20   A        Z          Filtered
21   B        X          B
22   B        X          B
23   A        Z          Filtered
```

(continues on next page)

(continued from previous page)

```
24   B        X          B
```

Now, a group operation applied to the Categorical omits the filtered values:

```
>>> c_x.count()
*Symbol   Count
-------   -----
A             4
B             5
```

Filtering out an entire category (here, the 'A' symbol) is similar:

```
>>> f_A = symbol_exchange.Symbol != 'A'
>>> c_b = rt.Categorical(symbol_exchange.Symbol, filter=f_A)
>>> c_b
Categorical([B, Filtered, B, Filtered, B, ..., Filtered, B, B, Filtered, B]) Length: 25
  FastArray([1, 0, 1, 0, 1, ..., 0, 1, 1, 0, 1], dtype=int8) Base Index: 1
  FastArray([b'B'], dtype='|S1') Unique count: 1
```

The filtered category is entirely omitted from operations:

```
>>> c_b.count()
*Symbol   Count
-------   -----
B            14
```

If you're creating a Categorical from integers and provided categories, another way to filter a category is to map it to 0.

Because 0 is reserved for the Filtered category, here 'a' is mapped to 1 and 'b' is mapped to 2. And because there's no 3 to map to 'c', 'c' becomes Filtered:

```
>>> c1 = rt.Categorical([0, 2, 1, 1, 0, 2, 2, 0], categories=['a','b','c'])
>>> c1
Categorical([Filtered, b, a, a, Filtered, b, b, Filtered]) Length: 8
  FastArray([0, 2, 1, 1, 0, 2, 2, 0]) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

In this case, the filtered category appears in the result, but it's still omitted from calculations on the Categorical:

```
>>> c1.count()
*key_0   Count
------   -----
a            2
b            3
c            0
```

Note that the first column in the output is labeled 'key_0'. This was code-generated because there was no explicit column name declaration. You can use the *FastArray.set_name()* method to assign a column name to the Categorical before doing any grouping operations. The Count column was created by the `count()` method.

### 1.5.8 Filter Values or Categories from Certain Categorical Operations

It's also possible to filter values for only a certain operation.

In *Get and Operate on Subsets of Data Using Filters*, we saw that many operations called on FastArrays / Dataset columns take a `filter` keyword argument that limits the data operated on:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a.mean(filter=a > 2)
4.0
```

It's similar with Categoricals:

```
>>> Symbol = rt.Cat(rt.FA(['AAPL', 'MSFT', 'AAPL', 'TSLA', 'MSFT', 'TSLA']))
>>> Value = rt.FA([5, 10, 15, 20, 25, 30])
>>> Symbol.mean(Value, filter=Value > 20.0)
*key_0   col_0
------   -----
AAPL       nan
MSFT     25.00
TSLA     30.00
```

The data that doesn't meet the condition is omitted from the computation for only that operation.

To filter out an entire category:

```
>>> ds.Symbol.mean(ds.Value, filter=ds.Symbol != 'MSFT')
*Symbol   Value
-------   -----
AAPL      10.00
MSFT        nan
TSLA      25.00
```

In this case, the filtered category is shown, but the result of the operation on its values is NaN.

If you want to make sure your filter is doing what you intend before you apply a function to the filtered data, you can call `set_valid()` on the Categorical.

Calling `set_valid()` on a Categorical returns a Categorical of the same length in which everywhere the filter result is False, the category gets set to 'Filtered' and the associated index value is 0. This is in contrast to filtered Datasets, where `filter()` returns a smaller Dataset, reduced to only the rows where the filter result is True (where the filter condition is met).

```
>>> Symbol.set_valid(ds.Value > 20.0)
Categorical([Filtered, Filtered, Filtered, Filtered, MSFT, TSLA]) Length: 6
  FastArray([0, 0, 0, 0, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'MSFT', b'TSLA'], dtype='|S4') Unique count: 2
```

To more closely spot-check, put the filtered values in a Dataset:

```
>>> ds_test = rt.Dataset()
>>> ds_test.SymbolTest = ds.Symbol.set_valid(ds.Value > 20.0)
>>> ds_test.ValueTest = ds.Value
>>> ds_test
#   SymbolTest   ValueTest
-   ----------   ---------
```

(continues on next page)

(continued from previous page)

```
0   Filtered             5
1   Filtered            10
2   Filtered            15
3   Filtered            20
4   MSFT                25
5   TSLA                30
```

The advice to avoid making unnecessary copies of large amounts of data using `set_valid()` also applies to Categoricals.

### 1.5.9 Multi-Key Categoricals

Multi-key Categoricals let you create and operate on groupings based on two related categories.

An example is a symbol-month pair, which you could use to get the average value of a stock for each month in your data:

```
>>> ds_mk = rt.Dataset()
>>> N = 25
>>> ds_mk.Symbol = rt.FA(rng.choice(['AAPL', 'AMZN', 'MSFT'], N))
>>> ds_mk.Value = rt.FA(rng.random(N))
>>> ds_mk.Date = rt.Date.range('20210101', '20211231', step=15)
>>> ds_mk
 #   Symbol   Value         Date
--   ------   -----   ----------
 0   AAPL      0.59   2021-01-01
 1   MSFT      0.78   2021-01-16
 2   AAPL      0.80   2021-01-31
 3   AAPL      0.95   2021-02-15
 4   AMZN      0.25   2021-03-02
 5   MSFT      0.59   2021-03-17
 6   AMZN      0.10   2021-04-01
 7   MSFT      0.62   2021-04-16
 8   MSFT      0.17   2021-05-01
 9   AAPL      0.56   2021-05-16
10   MSFT      0.57   2021-05-31
11   AMZN      0.47   2021-06-15
12   AMZN      0.52   2021-06-30
13   AAPL      0.76   2021-07-15
14   AMZN      0.80   2021-07-30
15   MSFT      0.49   2021-08-14
16   AMZN      0.60   2021-08-29
17   AAPL      0.93   2021-09-13
18   AMZN      0.12   2021-09-28
19   MSFT      0.12   2021-10-13
20   MSFT      0.09   2021-10-28
21   AAPL      0.66   2021-11-12
22   MSFT      0.42   2021-11-27
23   MSFT      0.77   2021-12-12
24   AAPL      0.67   2021-12-27
```

We want to group the dates by month. An easy way to do this is by using `start_of_month`:

```
>>> ds_mk.Month = ds_mk.Date.start_of_month
>>> ds_mk
#   Symbol   Value         Date        Month
--   ------   -----   ----------   ----------
 0   AAPL      0.59   2021-01-01   2021-01-01
 1   MSFT      0.78   2021-01-16   2021-01-01
 2   AAPL      0.80   2021-01-31   2021-01-01
 3   AAPL      0.95   2021-02-15   2021-02-01
 4   AMZN      0.25   2021-03-02   2021-03-01
 5   MSFT      0.59   2021-03-17   2021-03-01
 6   AMZN      0.10   2021-04-01   2021-04-01
 7   MSFT      0.62   2021-04-16   2021-04-01
 8   MSFT      0.17   2021-05-01   2021-05-01
 9   AAPL      0.56   2021-05-16   2021-05-01
10   MSFT      0.57   2021-05-31   2021-05-01
11   AMZN      0.47   2021-06-15   2021-06-01
12   AMZN      0.52   2021-06-30   2021-06-01
13   AAPL      0.76   2021-07-15   2021-07-01
14   AMZN      0.80   2021-07-30   2021-07-01
15   MSFT      0.49   2021-08-14   2021-08-01
16   AMZN      0.60   2021-08-29   2021-08-01
17   AAPL      0.93   2021-09-13   2021-09-01
18   AMZN      0.12   2021-09-28   2021-09-01
19   MSFT      0.12   2021-10-13   2021-10-01
20   MSFT      0.09   2021-10-28   2021-10-01
21   AAPL      0.66   2021-11-12   2021-11-01
22   MSFT      0.42   2021-11-27   2021-11-01
23   MSFT      0.77   2021-12-12   2021-12-01
24   AAPL      0.67   2021-12-27   2021-12-01
```

Now all Dates in January are associated to 2021-01-01, all Dates in February are associated to 2021-02-01, etc. These firsts of the month are our month groups.

We create a multi-key Categorical by passing `rt.Cat()` the Symbol and Month columns:

```
>>> ds_mk.Symbol_Month = rt.Cat([ds_mk.Symbol, ds_mk.Month])
>>> ds_mk.Symbol_Month
Categorical([(AAPL, 2021-01-01), (MSFT, 2021-01-01), (AAPL, 2021-01-01), (AAPL, 2021-02-
↪01), (AMZN, 2021-03-01), ..., (MSFT, 2021-10-01), (AAPL, 2021-11-01), (MSFT, 2021-11-
↪01), (MSFT, 2021-12-01), (AAPL, 2021-12-01)]) Length: 25
  FastArray([ 1,  2,  1,  3,  4, ..., 17, 18, 19, 20, 21], dtype=int8) Base Index: 1
  {'Symbol': FastArray([b'AAPL', b'MSFT', b'AAPL', b'AMZN', b'MSFT', ..., b'MSFT', b'AAPL
↪', b'MSFT', b'MSFT', b'AAPL'], dtype='|S4'), 'Month': Date(['2021-01-01', '2021-01-01',
↪ '2021-02-01', '2021-03-01', '2021-03-01', ..., '2021-10-01', '2021-11-01', '2021-11-01
↪', '2021-12-01', '2021-12-01'])} Unique count: 21
```

And now we can get the average value for each symbol-month pair:

```
>>> ds_mk.Symbol_Month.mean(ds_mk.Value)
*Symbol       *Month   Value
-------   ----------   -----
AAPL      2021-01-01    0.69
MSFT      2021-01-01    0.78
AAPL      2021-02-01    0.95
```

(continues on next page)

(continued from previous page)

```
AMZN      2021-03-01    0.25
MSFT      2021-03-01    0.59
AMZN      2021-04-01    0.10
MSFT      2021-04-01    0.62
.         2021-05-01    0.37
AAPL      2021-05-01    0.56
AMZN      2021-06-01    0.49
AAPL      2021-07-01    0.76
AMZN      2021-07-01    0.80
MSFT      2021-08-01    0.49
AMZN      2021-08-01    0.60
AAPL      2021-09-01    0.93
AMZN      2021-09-01    0.12
MSFT      2021-10-01    0.10
AAPL      2021-11-01    0.66
MSFT      2021-11-01    0.42
.         2021-12-01    0.77
AAPL      2021-12-01    0.67
```

The aggregated results are presented with the two group keys arranged hierarchically. The dot indicates that the category above is repeated.

All the functions supported by Categoricals can also be used for multi-key Categoricals.

You can also filter multi-key Categoricals by calling `set_valid()` on the Categorical, and operate on filtered data by passing the filter keyword argument to the function you use.

Later on we'll cover another Riptable function, `Accum2()`, that aggregates two groups similarly but provides summary data and a styled output.

### 1.5.10 Partition Numeric Data into Bins for Analysis

When you have a large array of numeric data, `rt.cut()` and `rt.qcut()` can help you partition the values into Categorical bins for analysis.

Use `cut()` to create equal-width bins or bins defined by specified endpoints. Use `qcut()` to create bins based on sample quantiles.

Let's use a moderately large Dataset:

```
>>> N = 1_000
>>> ds2 = rt.Dataset()
>>> ds2.Symbol = rt.FA(rng.choice(['AAPL', 'AMZN', 'MSFT'], N))
>>> base_price = 100 + rt.FA(np.linspace(0, 900, N))
>>> noise = rt.FA(rng.normal(0, 50, N))
>>> ds2.Price = base_price + noise
>>> ds2
  #   Symbol      Price
---   ------   --------
  0   AMZN        93.87
  1   AMZN       150.69
  2   AAPL       154.76
  3   MSFT       153.99
  4   AMZN       105.55
```

(continues on next page)

(continued from previous page)

```
  5   AMZN        62.25
  6   MSFT        51.22
  7   AMZN       123.54
  8   AAPL       126.17
  9   AAPL       172.47
 10   AAPL       164.01
 11   MSFT       103.30
 12   AAPL        48.60
 13   AAPL        95.76
 14   AMZN       123.47
...   ...           ...
985   AMZN     1,027.85
986   AAPL       993.06
987   AMZN       867.37
988   AAPL       940.92
989   AAPL     1,025.38
990   MSFT     1,052.54
991   AAPL     1,048.25
992   AMZN       914.09
993   AMZN     1,009.67
994   AAPL     1,046.27
995   AAPL       913.48
996   AMZN       996.90
997   AMZN     1,011.89
998   MSFT       984.06
999   MSFT       907.39
```

### Create equal-width bins with `rt.cut()`

To partition values into equal-width bins, use `cut()` and specify the number of bins:

```
>>> ds2.PriceBin = rt.cut(ds2.Price, bins=5)
>>> ds2
  #   Symbol      Price   PriceBin
---   ------   --------   -----------------
  0   AMZN        93.87   -3.011->221.182
  1   AMZN       150.69   -3.011->221.182
  2   AAPL       154.76   -3.011->221.182
  3   MSFT       153.99   -3.011->221.182
  4   AMZN       105.55   -3.011->221.182
  5   AMZN        62.25   -3.011->221.182
  6   MSFT        51.22   -3.011->221.182
  7   AMZN       123.54   -3.011->221.182
  8   AAPL       126.17   -3.011->221.182
  9   AAPL       172.47   -3.011->221.182
 10   AAPL       164.01   -3.011->221.182
 11   MSFT       103.30   -3.011->221.182
 12   AAPL        48.60   -3.011->221.182
 13   AAPL        95.76   -3.011->221.182
 14   AMZN       123.47   -3.011->221.182
...   ...           ...   ...
```

(continues on next page)

(continued from previous page)

```
 985  AMZN     1,027.85   893.763->1117.956
 986  AAPL       993.06   893.763->1117.956
 987  AMZN       867.37   669.569->893.763
 988  AAPL       940.92   893.763->1117.956
 989  AAPL     1,025.38   893.763->1117.956
 990  MSFT     1,052.54   893.763->1117.956
 991  AAPL     1,048.25   893.763->1117.956
 992  AMZN       914.09   893.763->1117.956
 993  AMZN     1,009.67   893.763->1117.956
 994  AAPL     1,046.27   893.763->1117.956
 995  AAPL       913.48   893.763->1117.956
 996  AMZN       996.90   893.763->1117.956
 997  AMZN     1,011.89   893.763->1117.956
 998  MSFT       984.06   893.763->1117.956
 999  MSFT       907.39   893.763->1117.956
```

Notice that the bins form the categories of a Categorical:

```
>>> ds2.PriceBin
Categorical([-3.011->221.182, -3.011->221.182, -3.011->221.182, -3.011->221.182, -3.011->
↪221.182, ..., 893.763->1117.956, 893.763->1117.956, 893.763->1117.956, 893.763->1117.
↪956, 893.763->1117.956]) Length: 1000
  FastArray([1, 1, 1, 1, 1, ..., 5, 5, 5, 5, 5], dtype=int8) Base Index: 1
  FastArray([b'-3.011->221.182', b'221.182->445.376', b'445.376->669.569', b'669.569->
↪893.763', b'893.763->1117.956'], dtype='|S17') Unique count: 5
```

To specify your own bin endpoints, provide an array. Here, we define two bins: one for prices from 0 to 600 (with both endpoints, 0 and 600, included), and one for prices from 600 to 1,200 (600 excluded, 1,200 included):

```
>>> bins = [0, 600, 1200]
>>> ds2.PriceBin2 = rt.cut(ds2.Price, bins)
>>> ds2
  #   Symbol      Price   PriceBin            PriceBin2
---   ------   --------   -----------------   -------------
  0   AMZN        93.87   -3.011->221.182     0.0->600.0
  1   AMZN       150.69   -3.011->221.182     0.0->600.0
  2   AAPL       154.76   -3.011->221.182     0.0->600.0
  3   MSFT       153.99   -3.011->221.182     0.0->600.0
  4   AMZN       105.55   -3.011->221.182     0.0->600.0
  5   AMZN        62.25   -3.011->221.182     0.0->600.0
  6   MSFT        51.22   -3.011->221.182     0.0->600.0
  7   AMZN       123.54   -3.011->221.182     0.0->600.0
  8   AAPL       126.17   -3.011->221.182     0.0->600.0
  9   AAPL       172.47   -3.011->221.182     0.0->600.0
 10   AAPL       164.01   -3.011->221.182     0.0->600.0
 11   MSFT       103.30   -3.011->221.182     0.0->600.0
 12   AAPL        48.60   -3.011->221.182     0.0->600.0
 13   AAPL        95.76   -3.011->221.182     0.0->600.0
 14   AMZN       123.47   -3.011->221.182     0.0->600.0
...   ...           ...   ...                 ...
985   AMZN     1,027.85   893.763->1117.956   600.0->1200.0
986   AAPL       993.06   893.763->1117.956   600.0->1200.0
```

(continues on next page)

(continued from previous page)

```
987   AMZN       867.37   669.569->893.763    600.0->1200.0
988   AAPL       940.92   893.763->1117.956   600.0->1200.0
989   AAPL     1,025.38   893.763->1117.956   600.0->1200.0
990   MSFT     1,052.54   893.763->1117.956   600.0->1200.0
991   AAPL     1,048.25   893.763->1117.956   600.0->1200.0
992   AMZN       914.09   893.763->1117.956   600.0->1200.0
993   AMZN     1,009.67   893.763->1117.956   600.0->1200.0
994   AAPL     1,046.27   893.763->1117.956   600.0->1200.0
995   AAPL       913.48   893.763->1117.956   600.0->1200.0
996   AMZN       996.90   893.763->1117.956   600.0->1200.0
997   AMZN     1,011.89   893.763->1117.956   600.0->1200.0
998   MSFT       984.06   893.763->1117.956   600.0->1200.0
999   MSFT       907.39   893.763->1117.956   600.0->1200.0
```

### Create bins based on sample quantiles with `rt.qcut()`

To partition values into bins based on sample quantiles, use `qcut()`.

We'll create another Dataset with symbol groups and contracts per day:

```
>>> N = 1_000
>>> ds3 = rt.Dataset()
>>> ds3.SymbolGroup = rt.FA(rng.choice(['spx', 'eqt_comp', 'eqt300', 'eqtrest'], N))
>>> ds3.ContractsPerDay = rng.integers(low=0, high=5_000, size=N)
>>> ds3.head()
 #   SymbolGroup   ContractsPerDay
--   -----------   ---------------
 0   eqt300                 1,624
 1   spx                      851
 2   spx                    3,487
 3   eqt300                   345
 4   eqtrest                2,584
 5   spx                    3,639
 6   spx                    4,741
 7   eqtrest                1,440
 8   eqtrest                   39
 9   spx                    3,618
10   eqt_comp                   7
11   eqt300                   331
12   spx                    4,952
13   eqt_comp               4,312
14   eqt_comp               3,537
15   eqt300                 4,177
16   eqt_comp                 376
17   eqt_comp                 444
18   eqt_comp               1,504
19   eqtrest                  118
```

Create three labeled, quantile-based bins for the volume:

```
>>> label_names = ['Low', 'Medium', 'High']
>>> ds3.Volume = rt.qcut(ds3.ContractsPerDay, q=3, labels=label_names)
```

(continues on next page)

(continued from previous page)

```
>>> ds3.head()
 #   SymbolGroup   ContractsPerDay   Volume
--   -----------   ---------------   ------
 0   eqt300                  1,624   Low
 1   spx                       851   Low
 2   spx                     3,487   High
 3   eqt300                    345   Low
 4   eqtrest                 2,584   Medium
 5   spx                     3,639   High
 6   spx                     4,741   High
 7   eqtrest                 1,440   Low
 8   eqtrest                    39   Low
 9   spx                     3,618   High
10   eqt_comp                    7   Low
11   eqt300                    331   Low
12   spx                     4,952   High
13   eqt_comp                4,312   High
14   eqt_comp                3,537   High
15   eqt300                  4,177   High
16   eqt_comp                  376   Low
17   eqt_comp                  444   Low
18   eqt_comp                1,504   Low
19   eqtrest                   118   Low
```

As with `cut()`, the bins form the categories of a Categorical:

```
>>> ds3.Volume
Categorical([High, High, Medium, High, Low, ..., Low, Medium, High, Low, Low]) Length:␣
↪1000
  FastArray([4, 4, 3, 4, 2, ..., 2, 3, 4, 2, 2], dtype=int8) Base Index: 1
  FastArray([b'Clipped', b'Low', b'Medium', b'High'], dtype='|S7') Unique count: 4
```

The 'Clipped' bin is created to hold any out-of-bounds values (though there are none in this case).

Alternatively, you can give `qcut()` a list of quantiles (numbers between 0 and 1, inclusive). Here, we create quartiles:

```
>>> quartiles = [0, .25, .5, .75, 1.]
>>> ds3.VolQuartiles = rt.qcut(ds3.ContractsPerDay, q=quartiles)
>>> ds3.head()
 #   SymbolGroup   ContractsPerDay   Volume   VolQuartiles
--   -----------   ---------------   ------   ---------------
 0   eqt300                  1,624   Low      1273.75->2601.0
 1   spx                       851   Low      0.0->1273.75
 2   spx                     3,487   High     2601.0->3793.0
 3   eqt300                    345   Low      0.0->1273.75
 4   eqtrest                 2,584   Medium   1273.75->2601.0
 5   spx                     3,639   High     2601.0->3793.0
 6   spx                     4,741   High     3793.0->4991.0
 7   eqtrest                 1,440   Low      1273.75->2601.0
 8   eqtrest                    39   Low      0.0->1273.75
 9   spx                     3,618   High     2601.0->3793.0
10   eqt_comp                    7   Low      0.0->1273.75
11   eqt300                    331   Low      0.0->1273.75
```

(continues on next page)

(continued from previous page)

```
12   spx                    4,952   High     3793.0->4991.0
13   eqt_comp               4,312   High     3793.0->4991.0
14   eqt_comp               3,537   High     2601.0->3793.0
15   eqt300                 4,177   High     3793.0->4991.0
16   eqt_comp                 376   Low      0.0->1273.75
17   eqt_comp                 444   Low      0.0->1273.75
18   eqt_comp               1,504   Low      1273.75->2601.0
19   eqtrest                  118   Low      0.0->1273.75
```

### 1.5.11 Per-Group Calculations with Other Functions

Categoricals support most common functions. For functions that aren't supported (for example, a function you've written), you can use `apply_reduce()` to apply a reducing function and `apply_nonreduce()` to apply a non-reducing function.

#### apply_reduce()

The function you use with `apply_reduce()` can take in one or multiple columns/FastArrays as input, but it must return a single value per group.

To illustrate, we'll use `apply_reduce()` with two simple lambda functions that each return one value. (A lambda function is an anonymous function that consists of a single statement and gives back a return value. When you have a function that takes a function as an argument, using a lambda function as the argument can sometimes be simpler and clearer than defining a function separately.)

First, we'll create a new Dataset:

```
>>> N = 50
>>> ds = rt.Dataset()
>>> ds.Symbol = rt.Cat(rng.choice(['AAPL', 'AMZN', 'TSLA', 'SPY', 'GME'], N))
>>> ds.Value = rng.random(N) * 100
>>> ds.Value2 = ds.Value * 2
>>> ds.sample()
#   Symbol   Value   Value2
-   ------   -----   ------
0   SPY      41.04    82.09
1   TSLA     93.07   186.14
2   AMZN      2.03     4.05
3   AAPL     16.19    32.37
4   AMZN      2.42     4.85
5   TSLA     98.13   196.26
6   SPY      98.67   197.34
7   SPY      62.31   124.61
8   TSLA     96.79   193.58
9   TSLA     67.35   134.70
```

The first lambda function takes one column as input:

```
>>> # ds.Value becomes the 'x' in our lambda function.
>>> ds.Symbol.apply_reduce(lambda x: x.min() + 2, ds.Value)
*Symbol   Value
```

(continues on next page)

(continued from previous page)

```
-------   -----
AAPL      11.36
AMZN       4.03
GME       16.65
SPY        7.76
TSLA       2.10
```

Our second lambda function takes two columns as input:

```
>>> ds.Symbol.apply_reduce(lambda x, y: x.sum() * y.mean(), (ds.Value, ds.Value2))
*Symbol       Value
-------   ---------
AAPL      26,904.13
AMZN      39,400.64
GME       26,857.53
SPY       32,560.75
TSLA      74,124.69
```

Also note that in this example, the first column listed in the tuple is the column name shown in the output.

If you like, you can use `transform=True` to expand the results and assign them to a column:

```
>>> ds.MyCalc1 = ds.Symbol.apply_reduce(lambda x: x.min() + 2, ds.Value, transform=True)
>>> ds.MyCalc2 = ds.Symbol.apply_reduce(lambda x, y: x.sum() * y.mean(),
...                                     (ds.Value, ds.Value2), transform=True)
>>> ds
  #   Symbol   Value   Value2   MyCalc1     MyCalc2
---   ------   -----   ------   -------   ---------
  0   AAPL     12.39    24.77     11.36   26,904.13
  1   SPY      41.04    82.09      7.76   32,560.75
  2   AMZN     55.69   111.39      4.03   39,400.64
  3   TSLA     93.07   186.14      2.10   74,124.69
  4   TSLA      3.62     7.24      2.10   74,124.69
  5   TSLA     62.15   124.29      2.10   74,124.69
  6   SPY      45.77    91.55      7.76   32,560.75
  7   AMZN      2.03     4.05      4.03   39,400.64
  8   SPY      24.95    49.91      7.76   32,560.75
  9   AMZN     11.85    23.70      4.03   39,400.64
 10   AMZN     21.68    43.36      4.03   39,400.64
 11   TSLA     27.46    54.91      2.10   74,124.69
 12   GME      40.13    80.26     16.65   26,857.53
 13   AMZN     52.90   105.81      4.03   39,400.64
 14   TSLA      0.10     0.20      2.10   74,124.69
...   ...        ...      ...       ...         ...
 35   TSLA     38.40    76.79      2.10   74,124.69
 36   AAPL     93.12   186.25     11.36   26,904.13
 37   SPY      14.92    29.85      7.76   32,560.75
 38   AAPL     99.71   199.41     11.36   26,904.13
 39   TSLA     37.91    75.83      2.10   74,124.69
 40   GME      64.88   129.75     16.65   26,857.53
 41   TSLA     96.79   193.58      2.10   74,124.69
 42   SPY       5.76    11.52      7.76   32,560.75
 43   TSLA     92.29   184.57      2.10   74,124.69
```

(continues on next page)

(continued from previous page)

```
 44   AMZN     56.78   113.56      4.03   39,400.64
 45   AMZN     70.44   140.88      4.03   39,400.64
 46   TSLA     14.92    29.84      2.10   74,124.69
 47   AAPL     53.34   106.68     11.36   26,904.13
 48   TSLA     67.35   134.70      2.10   74,124.69
 49   TSLA     45.62    91.25      2.10   74,124.69
```

As expected, every instance of a category gets the same value.

### apply_nonreduce()

For `apply_nonreduce()`, our lambda function computes a new value for every element of the original input:

```
>>> ds.MyCalc3 = ds.Symbol.apply_nonreduce(lambda x: x.cumsum() + 2, ds.Value)
>>> ds
  #   Symbol   Value   Value2   MyCalc1     MyCalc2   MyCalc3
---   ------   -----   ------   -------   ---------   -------
  0   AAPL     12.39    24.77     11.36   26,904.13     14.39
  1   SPY      41.04    82.09      7.76   32,560.75     43.04
  2   AMZN     55.69   111.39      4.03   39,400.64     57.69
  3   TSLA     93.07   186.14      2.10   74,124.69     95.07
  4   TSLA      3.62     7.24      2.10   74,124.69     98.69
  5   TSLA     62.15   124.29      2.10   74,124.69    160.84
  6   SPY      45.77    91.55      7.76   32,560.75     88.82
  7   AMZN      2.03     4.05      4.03   39,400.64     59.72
  8   SPY      24.95    49.91      7.76   32,560.75    113.77
  9   AMZN     11.85    23.70      4.03   39,400.64     71.57
 10   AMZN     21.68    43.36      4.03   39,400.64     93.25
 11   TSLA     27.46    54.91      2.10   74,124.69    188.30
 12   GME      40.13    80.26     16.65   26,857.53     42.13
 13   AMZN     52.90   105.81      4.03   39,400.64    146.15
 14   TSLA      0.10     0.20      2.10   74,124.69    188.40
...   ...        ...      ...       ...         ...       ...
 35   TSLA     38.40    76.79      2.10   74,124.69    417.18
 36   AAPL     93.12   186.25     11.36   26,904.13    133.05
 37   SPY      14.92    29.85      7.76   32,560.75    399.73
 38   AAPL     99.71   199.41     11.36   26,904.13    232.76
 39   TSLA     37.91    75.83      2.10   74,124.69    455.09
 40   GME      64.88   129.75     16.65   26,857.53    261.12
 41   TSLA     96.79   193.58      2.10   74,124.69    551.88
 42   SPY       5.76    11.52      7.76   32,560.75    405.49
 43   TSLA     92.29   184.57      2.10   74,124.69    644.17
 44   AMZN     56.78   113.56      4.03   39,400.64    437.63
 45   AMZN     70.44   140.88      4.03   39,400.64    508.07
 46   TSLA     14.92    29.84      2.10   74,124.69    659.09
 47   AAPL     53.34   106.68     11.36   26,904.13    286.10
 48   TSLA     67.35   134.70      2.10   74,124.69    726.44
 49   TSLA     45.62    91.25      2.10   74,124.69    772.06
```

Like `apply_reduce()`, `apply_nonreduce()` can take one or multiple columns as input:

```
>>> ds.MyCalc4 = ds.Symbol.apply_nonreduce(lambda x, y: x.cumsum() + y, (ds.Value, ds.
↪Value2))
>>> ds
  #   Symbol   Value   Value2   MyCalc1     MyCalc2   MyCalc3   MyCalc4
---   ------   -----   ------   -------   ---------   -------   -------
  0   AAPL     12.39    24.77     11.36   26,904.13     14.39     37.16
  1   SPY      41.04    82.09      7.76   32,560.75     43.04    123.13
  2   AMZN     55.69   111.39      4.03   39,400.64     57.69    167.08
  3   TSLA     93.07   186.14      2.10   74,124.69     95.07    279.21
  4   TSLA      3.62     7.24      2.10   74,124.69     98.69    103.94
  5   TSLA     62.15   124.29      2.10   74,124.69    160.84    283.13
  6   SPY      45.77    91.55      7.76   32,560.75     88.82    178.36
  7   AMZN      2.03     4.05      4.03   39,400.64     59.72     61.77
  8   SPY      24.95    49.91      7.76   32,560.75    113.77    161.68
  9   AMZN     11.85    23.70      4.03   39,400.64     71.57     93.27
 10   AMZN     21.68    43.36      4.03   39,400.64     93.25    134.61
 11   TSLA     27.46    54.91      2.10   74,124.69    188.30    241.21
 12   GME      40.13    80.26     16.65   26,857.53     42.13    120.39
 13   AMZN     52.90   105.81      4.03   39,400.64    146.15    249.96
 14   TSLA      0.10     0.20      2.10   74,124.69    188.40    186.59
...   ...        ...      ...       ...         ...       ...       ...
 35   TSLA     38.40    76.79      2.10   74,124.69    417.18    491.97
 36   AAPL     93.12   186.25     11.36   26,904.13    133.05    317.30
 37   SPY      14.92    29.85      7.76   32,560.75    399.73    427.57
 38   AAPL     99.71   199.41     11.36   26,904.13    232.76    430.17
 39   TSLA     37.91    75.83      2.10   74,124.69    455.09    528.92
 40   GME      64.88   129.75     16.65   26,857.53    261.12    388.88
 41   TSLA     96.79   193.58      2.10   74,124.69    551.88    743.46
 42   SPY       5.76    11.52      7.76   32,560.75    405.49    415.01
 43   TSLA     92.29   184.57      2.10   74,124.69    644.17    826.74
 44   AMZN     56.78   113.56      4.03   39,400.64    437.63    549.19
 45   AMZN     70.44   140.88      4.03   39,400.64    508.07    646.95
 46   TSLA     14.92    29.84      2.10   74,124.69    659.09    686.92
 47   AAPL     53.34   106.68     11.36   26,904.13    286.10    390.78
 48   TSLA     67.35   134.70      2.10   74,124.69    726.44    859.14
 49   TSLA     45.62    91.25      2.10   74,124.69    772.06    861.31
```

### apply()

If you want your custom function to return multiple aggregations – for example, you want to return both the mean value of one column and the minimum value of another column – use `apply()`.

Warning: Because `apply()` isn't a vectorized operation, it can be slow and use a lot of memory if you're using it on large amounts of data. Try to avoid it if you can.

To be used with `apply()`, your function must be able to take in a Dataset. It can return a Dataset, a single array, or a dictionary of column names and values.

Here's a function that performs two reducing operations and returns a Dataset:

```
>>> def my_apply_func(ds):
...     new_ds = rt.Dataset({
...     'Mean_Value': ds.Value.mean(),
```

(continues on next page)

(continued from previous page)

```
...     'Min_Value': ds.Value.min()
... })
...     return new_ds
```

Here it is applied:

```
>>> ds.Symbol.apply(my_apply_func, ds)
*Symbol   Mean_Value   Min_Value
-------   ----------   ---------
AAPL           47.35        9.36
AMZN           38.93        2.03
GME            51.82       14.65
SPY            40.35        5.76
TSLA           48.13        0.10
```

Our second function performs two non-reducing operations:

```
>>> def my_apply_func2(ds):
...     new_ds = rt.Dataset({
...         'Val1': ds.Value * 3,
...         'Val2': ds.Value * 4
...     })
...     return new_ds
>>> ds.Symbol.apply(my_apply_func2, ds)
*gb_key_0     Val1     Val2
---------   ------   ------
AAPL         37.16    49.54
SPY         123.13   164.18
AMZN        167.08   222.77
TSLA        279.21   372.28
TSLA         10.87    14.49
TSLA        186.44   248.58
SPY         137.32   183.09
AMZN          6.08     8.10
SPY          74.86    99.82
AMZN         35.55    47.39
AMZN         65.04    86.72
TSLA         82.37   109.83
GME         120.39   160.52
AMZN        158.71   211.62
TSLA          0.30     0.40
...            ...      ...
TSLA        115.19   153.58
AAPL        279.37   372.50
SPY          44.77    59.69
AAPL        299.12   398.83
TSLA        113.74   151.65
GME         194.63   259.51
TSLA        290.37   387.16
SPY          17.28    23.04
TSLA        276.86   369.15
AMZN        170.34   227.12
```

(continues on next page)

(continued from previous page)

```
AMZN        211.32   281.76
TSLA         44.76    59.67
AAPL        160.02   213.35
TSLA        202.05   269.41
TSLA        136.87   182.50
```

Because the operations in this function are non-reducing operations, the resulting Dataset is expanded.

Note that until a reported bug is fixed, column names might not persist through grouping operations.

For more in-depth information about Categoricals, see the `Categoricals User Guide`.

In the next section, *Accums*, we look at another way to do multi-key groupings with fancier output.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.6 Accums

Accums aggregate data similarly to Categoricals, but they distinguish themselves by providing a fancier output with overall aggregates in summary footers and columns.

### 1.6.1 `Accum2()`

`Accum2()` is very much like a multi-key Categorical: It computes aggregates values for pairs of groups. The difference is in the output – an `Accum2()` result looks more like a pivot table, with the first group passed to the function providing row labels and the second providing the column labels.

The function is also applied to each row and column, with results shown in a summary column and row, as well as to all columns and rows combined (with the result shown in the bottom right corner).

We'll use a Dataset that's similar to the one we used for multi-key Categoricals, so we can compare the output:

```
>>> rng = np.random.default_rng(seed=42)
>>> ds = rt.Dataset()
>>> N = 100
>>> ds.Symbol = rt.FA(rng.choice(['AAPL', 'AMZN', 'MSFT'], N))
>>> ds.Value = rt.FA(rng.random(N))
>>> ds.Date = rt.Date.range('20210101', days = 100)  # Dates from January to mid-April
>>> ds.Month = ds.Date.start_of_month
>>> # Accum2 can take Categoricals or FastArrays as input.
>>> # To use this ds for accum_ratio, we need Symbol and Month to be Categoricals.
>>> ds.Symbol = rt.Cat(ds.Symbol)
>>> ds.Month = rt.Cat(ds.Month)
>>> ds
  #   Symbol   Value         Date   Month
---   ------   -----   ----------   ----------
  0   AAPL      0.20   2021-01-01   2021-01-01
  1   MSFT      0.01   2021-01-02   2021-01-01
  2   AMZN      0.79   2021-01-03   2021-01-01
  3   AMZN      0.66   2021-01-04   2021-01-01
  4   AMZN      0.71   2021-01-05   2021-01-01
```

(continues on next page)

(continued from previous page)

```
  5   MSFT      0.78   2021-01-06   2021-01-01
  6   AAPL      0.46   2021-01-07   2021-01-01
  7   MSFT      0.57   2021-01-08   2021-01-01
  8   AAPL      0.14   2021-01-09   2021-01-01
  9   AAPL      0.11   2021-01-10   2021-01-01
 10   AMZN      0.67   2021-01-11   2021-01-01
 11   MSFT      0.47   2021-01-12   2021-01-01
 12   MSFT      0.57   2021-01-13   2021-01-01
 13   MSFT      0.76   2021-01-14   2021-01-01
 14   MSFT      0.63   2021-01-15   2021-01-01
...   ...        ...          ...   ...
 85   MSFT      0.02   2021-03-27   2021-03-01
 86   AAPL      0.96   2021-03-28   2021-03-01
 87   AAPL      0.48   2021-03-29   2021-03-01
 88   MSFT      0.78   2021-03-30   2021-03-01
 89   MSFT      0.08   2021-03-31   2021-03-01
 90   AMZN      0.49   2021-04-01   2021-04-01
 91   MSFT      0.49   2021-04-02   2021-04-01
 92   MSFT      0.94   2021-04-03   2021-04-01
 93   AMZN      0.57   2021-04-04   2021-04-01
 94   MSFT      0.47   2021-04-05   2021-04-01
 95   AAPL      0.27   2021-04-06   2021-04-01
 96   AAPL      0.33   2021-04-07   2021-04-01
 97   MSFT      0.52   2021-04-08   2021-04-01
 98   AMZN      0.44   2021-04-09   2021-04-01
 99   AAPL      0.02   2021-04-10   2021-04-01
```

Here's the `Accum2()` table before we apply an aggregation function. You can see how many values fall into each group pair:

```
>>> rt.Accum2(ds.Symbol, ds.Month)
Accum2 Keys
 X:Date(['2021-01-01', '2021-02-01', '2021-03-01', '2021-04-01'])
 Y:FastArray([b'AAPL', b'AMZN', b'MSFT'], dtype='|S4')
 Bins:20   Rows:100

*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01   Sum
-------   ----------   ----------   ----------   ----------   ---
AAPL               6            9            9            3    27
AMZN              13            8            9            3    33
MSFT              12           11           13            4    40
```

If we aggregate with `count()`, it has the same data, but we see the output formatting:

```
>>> rt.Accum2(ds.Symbol, ds.Month).count()
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01   Sum
-------   ----------   ----------   ----------   ----------   ---
AAPL               6            9            9            3    27
AMZN              13            8            9            3    33
MSFT              12           11           13            4    40
    Sum           31           28           31           10   100
```

The bottom row and rightmost column provide summary data.

Now we'll get the average value per symbol-month pair:

```
>>> rt.Accum2(ds.Symbol, ds.Month).mean(ds.Value)
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01   Mean
-------   ----------   ----------   ----------   ----------   ----
AAPL            0.35         0.40         0.54         0.21   0.41
AMZN            0.54         0.48         0.45         0.50   0.50
MSFT            0.44         0.47         0.42         0.61   0.46
   Mean         0.47         0.45         0.46         0.45   0.46
```

Note that the summary row and column show the mean values for all the input values for each group, not just the means of the displayed group means.

To illustrate: Here's the mean of the displayed group mean values for AAPL:

```
>>> (0.35 + 0.40 + 0.54 + 0.21) / 4
0.375
```

And here's the mean of all AAPL values:

```
>>> ds.Value.nanmean(filter=ds.Symbol == 'AAPL')
0.41317486824408933
```

For comparison, here's the multi-key Categorical version:

```
>>> ds.Symbol_Month = rt.Cat([ds.Symbol, ds.Month])
>>> ds.Symbol_Month.mean(ds.Value)
*Symbol   *Month       Value
-------   ----------   -----
AAPL      2021-01-01    0.35
MSFT      2021-01-01    0.44
AMZN      2021-01-01    0.54
AAPL      2021-02-01    0.40
AMZN      2021-02-01    0.48
MSFT      2021-02-01    0.47
.         2021-03-01    0.42
AMZN      2021-03-01    0.45
AAPL      2021-03-01    0.54
AMZN      2021-04-01    0.50
MSFT      2021-04-01    0.61
AAPL      2021-04-01    0.21
```

You can pass a filter keyword argument to the function you call on `Accum2()`:

```
>>> rt.Accum2(ds.Symbol, ds.Month).mean(ds.Value, filter=ds.Value > 0.5)
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01   Mean
-------   ----------   ----------   ----------   ----------   ----
AAPL            0.85         0.74         0.76          nan   0.77
AMZN            0.67         0.75         0.72         0.57   0.69
MSFT            0.65         0.78         0.70         0.73   0.71
   Mean         0.67         0.76         0.72         0.68   0.71
```

### 1.6.2 accum_ratio()

For each group pair, `accum_ratio()` computes a ratio of values you specify. The results are presented in an Accum table.

For our example we'll add PnL and Size (number of sales) columns, and we'll use `accum_ratio()` to get the PnL for each symbol-month bucket, weighted by size:

```
>>> ds.PnL = rng.normal(10, 20, 100)
>>> ds.Size = rng.random(100) * 100
```

Like `Accum2()`, `accum_ratio()` takes two Categoricals (a row Categorical and a column Categorical). You also specify the numerator values and denominator values. For each group pair, it sums the numerator values and denominator values and presents the ratios in a table:

```
>>> rt.accum_ratio(ds.Symbol, ds.Month, ds.PnL * ds.Size, ds.Size, include_numer=True)
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01   Ratio       Numer      ␣
↪Denom
-------   ----------   ----------   ----------   ----------   -----   ---------   -------
↪-
AAPL            3.13        11.93         1.95        28.86    8.81   12,363.71   1,404.
↪13
AMZN            5.54         2.36        23.34        -2.94   10.01   16,971.55   1,695.
↪67
MSFT           23.90        22.78        -1.40        -9.61   10.35   17,501.11   1,690.
↪46
  Ratio        10.13        13.17         7.31         8.25    9.78
  Numer    10,604.13    18,953.08    13,471.17     3,807.98           46,836.36
  Denom     1,047.18     1,438.84     1,842.65       461.59                      4,790.
↪26
```

The result is the ratio of the following two tables.

Numerator:

```
>>> rt.Accum2(ds.Symbol, ds.Month).nansum(ds.Size * ds.PnL)
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01      Nansum
-------   ----------   ----------   ----------   ----------   ---------
AAPL          699.07     5,075.98     1,100.76     5,487.90   12,363.71
AMZN        2,956.74     1,065.03    13,358.59      -408.81   16,971.55
MSFT        6,948.32    12,812.08      -988.18    -1,271.11   17,501.11
 Nansum    10,604.13    18,953.08    13,471.17     3,807.98   46,836.36
```

Denominator:

```
>>> rt.Accum2(ds.Symbol, ds.Month).nansum(ds.Size)
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01     Nansum
-------   ----------   ----------   ----------   ----------   --------
AAPL          223.12       425.49       565.38       190.13   1,404.13
AMZN          533.28       450.83       572.34       139.22   1,695.67
MSFT          290.78       562.52       704.92       132.24   1,690.46
 Nansum     1,047.18     1,438.84     1,842.65       461.59   4,790.26
```

When the numerator and denominator are the same, the result is as you might expect:

```
>>> rt.accum_ratio(ds.Symbol, ds.Month, ds.Size, ds.Size, include_numer=True)
*Symbol   2021-01-01   2021-02-01   2021-03-01   2021-04-01   Ratio      Numer      Denom
-------   ----------   ----------   ----------   ----------   -----   --------   --------
AAPL            1.00         1.00         1.00         1.00    1.00   1,404.13   1,404.13
AMZN            1.00         1.00         1.00         1.00    1.00   1,695.67   1,695.67
MSFT            1.00         1.00         1.00         1.00    1.00   1,690.46   1,690.46
  Ratio         1.00         1.00         1.00         1.00    1.00
  Numer     1,047.18     1,438.84     1,842.65       461.59            4,790.26
  Denom     1,047.18     1,438.84     1,842.65       461.59                      4,790.26
```

### 1.6.3 accum_ratiop()

`accum_ratiop()` takes one column of values as numerators and computes an internal ratio for each group pair, where the denominator is one of three sums:

- The row sum (`norm_by='R'`)
- The column sum (`norm_by='C'`)
- The total sum (`norm_by='T'`)

For example, this table shows that 30.30% of AAPL sales were in February:

```
>>> rt.accum_ratiop(ds.Symbol, ds.Month, ds.Size, norm_by='R')
*Symbol      2021-01-01   2021-02-01   2021-03-01   2021-04-01   TotalRatio      Total
----------   ----------   ----------   ----------   ----------   ----------   --------
AAPL              15.89        30.30        40.27        13.54       100.00   1,404.13
AMZN              31.45        26.59        33.75         8.21       100.00   1,695.67
MSFT              17.20        33.28        41.70         7.82       100.00   1,690.46
TotalRatio        21.86        30.04        38.47         9.64       100.00
     Total     1,047.18     1,438.84     1,842.65       461.59                4,790.26
```

Note that the percentages in each row sum to 100%.

We can check the math by computing the ratio of AAPL's February sales to AAPL's total sales:

```
>>> filt_feb_aapl = (ds.Symbol == 'AAPL') & (ds.Month.as_string_array == rt.Date(
↪'20210201'))
>>> filt_total_aapl = ds.Symbol == 'AAPL'
>>> ds.Size[filt_feb_aapl].nansum() / ds.Size[filt_total_aapl].nansum()
0.3030291108538412
```

This table shows that AAPL's sales are 29.57% of February sales:

```
>>> rt.accum_ratiop(ds.Symbol, ds.Month, ds.Size, norm_by='C')
*Symbol      2021-01-01   2021-02-01   2021-03-01   2021-04-01   TotalRatio      Total
----------   ----------   ----------   ----------   ----------   ----------   --------
AAPL              21.31        29.57        30.68        41.19        29.31   1,404.13
AMZN              50.93        31.33        31.06        30.16        35.40   1,695.67
MSFT              27.77        39.10        38.26        28.65        35.29   1,690.46
TotalRatio       100.00       100.00       100.00       100.00       100.00
     Total     1,047.18     1,438.84     1,842.65       461.59                4,790.26
```

Note that the percentages in each column sum to 100%.

Check the math:

```
>>> filt_feb_total = ds.Month.as_string_array == rt.Date('20210201')
>>> ds.Size[filt_feb_aapl].nansum() / ds.Size[filt_feb_total].nansum()
0.29571866540362846
```

This table shows that AAPL's February sales represent 8.88% of all sales:

```
>>> rt.accum_ratiop(ds.Symbol, ds.Month, ds.Size, norm_by='T')
*Symbol      2021-01-01   2021-02-01   2021-03-01   2021-04-01   TotalRatio      Total
----------   ----------   ----------   ----------   ----------   ----------   --------
AAPL               4.66         8.88        11.80         3.97        29.31   1,404.13
AMZN              11.13         9.41        11.95         2.91        35.40   1,695.67
MSFT               6.07        11.74        14.72         2.76        35.29   1,690.46
TotalRatio        21.86        30.04        38.47         9.64       100.00
     Total     1,047.18     1,438.84     1,842.65       461.59                4,790.26
```

Note that the "TotalRatio" row and column percentages each sum to 100%.

Check the math:

```
>>> ds.Size[filt_feb_aapl].nansum() / ds.Size.nansum()
0.08882445025331744
```

Next, for something completely different, we'll explore ways to *Concatenate Datasets*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.7 Concatenate Datasets

Concatentating Datasets is straightforward, with two Dataset methods: You can concatenate rows (vertically) with `concat_rows()` or columns (horizontally) with `concat_columns()`.

However, it's good to be aware of what happens when you concatenate two Datasets that have different shapes or column names, so we'll look at a few examples.

### 1.7.1 Concatenate Rows

Concatenating data row-wise is sometimes called vertical stacking. When two Datasets have identical column names, `concat_rows()` simply stacks the data:

```
>>> ds1 = rt.Dataset({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
>>> ds2 = rt.Dataset({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5']})
>>> ds1
#   A    B
-   --   --
0   A0   B0
1   A1   B1
2   A2   B2

>>> ds2
#   A    B
```

(continues on next page)

(continued from previous page)

```
-   --   --
0   A3   B3
1   A4   B4
2   A5   B5

>>> rt.Dataset.concat_rows([ds1, ds2])
#   A    B
-   --   --
0   A0   B0
1   A1   B1
2   A2   B2
3   A3   B3
4   A4   B4
5   A5   B5
```

When the two Datasets have only some column names in common, the result has a gap in the data:

```
>>> # Create two Datasets with two out of three columns in common.
>>> ds3 = rt.Dataset({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
>>> ds4 = rt.Dataset({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5'], 'C': ['C3', 'C4',
↪ 'C5'] })
>>> ds3
#   A    B
-   --   --
0   A0   B0
1   A1   B1
2   A2   B2

>>> ds4
#   A    B    C
-   --   --   --
0   A3   B3   C3
1   A4   B4   C4
2   A5   B5   C5

>>> rt.Dataset.concat_rows([ds3, ds4])
#   A    B    C
-   --   --   --
0   A0   B0
1   A1   B1
2   A2   B2
3   A3   B3   C3
4   A4   B4   C4
5   A5   B5   C5
```

As you can see, Riptable's missing string value is ''. If the values were floats, the empty spots would be filled with nans:

```
>>> rng = np.random.default_rng(seed=42)
>>> ds5 = rt.Dataset({'col_'+str(i):rng.random(2) for i in range(2)})
>>> ds6 = rt.Dataset({'col_'+str(i):rng.random(2) for i in range(3)})
>>> ds5
```

(continues on next page)

(continued from previous page)

```
#   col_0   col_1
-   -----   -----
0    0.77    0.86
1    0.44    0.70

>>> ds6
#   col_0   col_1   col_2
-   -----   -----   -----
0    0.09    0.76    0.13
1    0.98    0.79    0.45

>>> rt.Dataset.concat_rows([ds5, ds6])
#   col_0   col_1   col_2
-   -----   -----   -----
0    0.77    0.86     nan
1    0.44    0.70     nan
2    0.09    0.76    0.13
3    0.98    0.79    0.45
```

See *Working with Missing Data* for more about what to expect when you have missing values in Riptable.

You can also concatenate datasets row-wise with Categoricals if the Datasets have identical column names:

```
>>> a = rt.Cat(['a', 'a', 'a', 'b', 'b'])
>>> b = rt.FA([0, 1, 2, 3, 4])
>>> ds10 = rt.Dataset({'Cat': a, 'Val': b})
>>> c = rt.Cat(['c', 'c', 'c', 'd', 'd'])
>>> d = rt.FA([5, 6, 7, 8, 9])
>>> ds11 = rt.Dataset({'Cat': c, 'Val': d})
>>> ds10
#   Cat   Val
-   ---   ---
0   a       0
1   a       1
2   a       2
3   b       3
4   b       4

>>> ds11
#   Cat   Val
-   ---   ---
0   c       5
1   c       6
2   c       7
3   d       8
4   d       9

>>> rt.Dataset.concat_rows([ds10, ds11])
#   Cat   Val
-   ---   ---
0   a       0
1   a       1
2   a       2
```

(continues on next page)

(continued from previous page)

```
3   b       3
4   b       4
5   c       5
6   c       6
7   c       7
8   d       8
9   d       9
```

### 1.7.2 Concatenate Columns

Concatenating data column-wise is also called horizontal stacking. It's most straightforward when you're concatenating two Datasets that have no column names in common:

```
>>> ds7 = rt.Dataset({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
>>> ds8 = rt.Dataset({'C': ['C0', 'C1', 'C2'], 'D': ['D0', 'D1', 'D2']})
>>> ds7
#   A    B
-   --   --
0   A0   B0
1   A1   B1
2   A2   B2

>>> ds8
#   C    D
-   --   --
0   C0   D0
1   C1   D1
2   C2   D2

>>> ds9 = rt.Dataset.concat_columns([ds7, ds8], do_copy=True)
>>> ds9
#   A    B    C    D
-   --   --   --   --
0   A0   B0   C0   D0
1   A1   B1   C1   D1
2   A2   B2   C2   D2
```

Note that `do_copy` is a required argument for `concat_columns()`. When `do_copy=True`, changes you make to values in the original Datasets do not change the values in your new, concatenated Dataset, and vice-versa.

When your two Datasets have a column name (or names) in common, you need to specify which data you want to keep – the data from the shared column(s) in first Dataset or the data from the shared column(s) in the second Dataset.

We'll give our second Dataset an 'A' column:

```
>>> ds8.A = rt.FA(['A3', 'A4', 'A5'])
```

If you try to concatenate the two Datasets, you get an error:

```
>>> try:
...     rt.Dataset.concat_columns([ds7, ds8], do_copy=True)
... except KeyError as e:
```

(continues on next page)

(continued from previous page)

```
...     print("KeyError:", e)
KeyError: "Duplicate column 'A'"
```

To keep the column data from the first Dataset, use `on_duplicate='first'`. You'll get a warning about mismatched column names, but the concatenation is performed:

```
>>> rt.Dataset.concat_columns([ds7, ds8], do_copy=True, on_duplicate='first')
C:\\riptable\\rt_dataset.py:5628: UserWarning: concat_columns() duplicate column␣
↪mismatch: {'A'}
warnings.warn(f'concat_columns() duplicate column mismatch: {dups!r}')
#   A    B    C    D
-   --   --   --   --
0   A0   B0   C0   D0
1   A1   B1   C1   D1
2   A2   B2   C2   D2
```

You can turn off this warning by adding `on_mismatch='ignore'`.

To keep the column data from the second dataset, use `on_duplicate='last'`:

```
>>> rt.Dataset.concat_columns([ds7, ds8], on_duplicate='last', do_copy=True)
#   A    B    C    D
-   --   --   --   --
0   A3   B0   C0   D0
1   A4   B1   C1   D1
2   A5   B2   C2   D2
```

Note: To concatenate Datasets column-wise, the columns must all be the same length – Riptable does not fill in missing column values the way it does missing row values:

```
>>> ds9 = rt.Dataset({'E': ['E0', 'E1']})
>>> try:
...     rt.Dataset.concat_columns([ds8, ds9], do_copy=True)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Inconsistent Dataset lengths {2, 3}
```

Concatenation is sufficient in certain situations, but it helps to have more flexibility to bring data from two Datasets together. Next, we'll cover how to *Merge Datasets*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.8 Merge Datasets

Merging gives you more flexibility to bring data from different Datasets together.

A merge operation connects rows in Datasets using a "key" column that the Datasets have in common.

Riptable's two main Dataset merge functions are `merge_lookup()` and `merge_asof()`. Generally speaking, `merge_lookup()` aligns data based on identical keys, while `merge_asof()` aligns data based on the nearest key.

For more general merges, `merge2()` does database-style left, right, inner, and outer joins.

### 1.8.1 merge_lookup()

Let's start with `merge_lookup()`. It's common to have one Dataset that has most of the information you need, and another, usually smaller Dataset that has information you want to add to the first Dataset to enrich it.

Here we'll create a larger Dataset with symbols and size values, and a smaller Dataset that has symbols associated with trader names. We'll use the shared Symbol column as the key to add the trader info to the larger Dataset:

```
>>> rng = np.random.default_rng(seed=42)
>>> N = 25
>>> # Larger Dataset
>>> ds = rt.Dataset({'Symbol': rng.choice(['GME', 'AMZN', 'TSLA', 'SPY'], N),
...                  'Size': rng.integers(1, 1000, N),})
>>> # Smaller Dataset, with data used to enrich the larger Dataset
>>> ds_symbol_trader = rt.Dataset({'Symbol': ['GME', 'TSLA', 'SPY', 'AMZN'],
...                                'Trader': ['Nate', 'Elon', 'Josh', 'Dan']})
>>> ds.head()
 #   Symbol   Size
--   ------   ----
 0   GME       644
 1   SPY       403
 2   TSLA      822
 3   AMZN      545
 4   AMZN      443
 5   SPY       451
 6   GME       228
 7   TSLA       93
 8   GME       555
 9   GME       888
10   TSLA       64
11   SPY       858
12   TSLA      827
13   SPY       277
14   TSLA      632
15   SPY       166
16   TSLA      758
17   GME       700
18   SPY       355
19   AMZN       68

>>> ds_symbol_trader
#   Symbol   Trader
-   ------   ------
0   GME      Nate
1   TSLA     Elon
2   SPY      Josh
3   AMZN     Dan
```

### `merge_lookup()` with Key Columns That Have the Same Name

Now we'll use `merge_lookup()` to add the trader information to the larger Dataset. `merge_lookup()` will align the data based on exact matches in the shared Symbol column.

A note about terms: When you merge two Datasets, the Dataset you're merging data into is the *left Dataset*; the one you're getting data from is the *right Dataset*.

Here, we call `merge_lookup()` on our left Dataset, `ds`. We pass it the name of the right Dataset, and tell it what column to use as the key:

```
>>> ds.merge_lookup(ds_symbol_trader, on='Symbol')
 #   Symbol   Size   Trader
--   ------   ----   ------
 0   GME       644   Nate
 1   SPY       403   Josh
 2   TSLA      822   Elon
 3   AMZN      545   Dan
 4   AMZN      443   Dan
 5   SPY       451   Josh
 6   GME       228   Nate
 7   TSLA       93   Elon
 8   GME       555   Nate
 9   GME       888   Nate
10   TSLA       64   Elon
11   SPY       858   Josh
12   TSLA      827   Elon
13   SPY       277   Josh
14   TSLA      632   Elon
15   SPY       166   Josh
16   TSLA      758   Elon
17   GME       700   Nate
18   SPY       355   Josh
19   AMZN       68   Dan
20   TSLA      970   Elon
21   AMZN      446   Dan
22   GME       893   Nate
23   SPY       678   Josh
24   SPY       778   Josh
```

The left Dataset now has the trader information, correctly aligned.

You can also use the following syntax, passing `merge_lookup()` the names of the left and right Datasets, along with the key:

```
>>> rt.merge_lookup(ds, ds_symbol_trader, on='Symbol')
 #   Symbol   Size   Trader
--   ------   ----   ------
 0   GME       644   Nate
 1   SPY       403   Josh
 2   TSLA      822   Elon
 3   AMZN      545   Dan
 4   AMZN      443   Dan
 5   SPY       451   Josh
 6   GME       228   Nate
```

(continues on next page)

(continued from previous page)

```
 7   TSLA       93   Elon
 8   GME       555   Nate
 9   GME       888   Nate
10   TSLA       64   Elon
11   SPY       858   Josh
12   TSLA      827   Elon
13   SPY       277   Josh
14   TSLA      632   Elon
15   SPY       166   Josh
16   TSLA      758   Elon
17   GME       700   Nate
18   SPY       355   Josh
19   AMZN       68   Dan
20   TSLA      970   Elon
21   AMZN      446   Dan
22   GME       893   Nate
23   SPY       678   Josh
24   SPY       778   Josh
```

### `merge_lookup` with Key Columns That Have Different Names

If the key column has a different name in each Dataset, just specify each column name with `left_on` and `right_on`:

```
>>> # For illustrative purposes, rename the key column in the right Dataset.
>>> ds_symbol_trader.col_rename('Symbol', 'UnderlyingSymbol')
>>> ds.merge_lookup(ds_symbol_trader, left_on='Symbol', right_on='UnderlyingSymbol')
 #   Symbol   Size   UnderlyingSymbol   Trader
--   ------   ----   ----------------   ------
 0   GME       644   GME                Nate
 1   SPY       403   SPY                Josh
 2   TSLA      822   TSLA               Elon
 3   AMZN      545   AMZN               Dan
 4   AMZN      443   AMZN               Dan
 5   SPY       451   SPY                Josh
 6   GME       228   GME                Nate
 7   TSLA       93   TSLA               Elon
 8   GME       555   GME                Nate
 9   GME       888   GME                Nate
10   TSLA       64   TSLA               Elon
11   SPY       858   SPY                Josh
12   TSLA      827   TSLA               Elon
13   SPY       277   SPY                Josh
14   TSLA      632   TSLA               Elon
15   SPY       166   SPY                Josh
16   TSLA      758   TSLA               Elon
17   GME       700   GME                Nate
18   SPY       355   SPY                Josh
19   AMZN       68   AMZN               Dan
20   TSLA      970   TSLA               Elon
21   AMZN      446   AMZN               Dan
22   GME       893   GME                Nate
```

(continues on next page)

(continued from previous page)

```
23   SPY       678   SPY                Josh
24   SPY       778   SPY                Josh
```

Notice that when the key columns have different names, both are kept. If you want keep only certain columns from the left or right Dataset, you can specify them with `columns_left` or `columns_right`:

```
>>> ds.merge_lookup(ds_symbol_trader, left_on='Symbol', right_on='UnderlyingSymbol',
...                 columns_right='Trader')
 #   Symbol   Size   Trader
--   ------   ----   ------
 0   GME       644   Nate
 1   SPY       403   Josh
 2   TSLA      822   Elon
 3   AMZN      545   Dan
 4   AMZN      443   Dan
 5   SPY       451   Josh
 6   GME       228   Nate
 7   TSLA       93   Elon
 8   GME       555   Nate
 9   GME       888   Nate
10   TSLA       64   Elon
11   SPY       858   Josh
12   TSLA      827   Elon
13   SPY       277   Josh
14   TSLA      632   Elon
15   SPY       166   Josh
16   TSLA      758   Elon
17   GME       700   Nate
18   SPY       355   Josh
19   AMZN       68   Dan
20   TSLA      970   Elon
21   AMZN      446   Dan
22   GME       893   Nate
23   SPY       678   Josh
24   SPY       778   Josh
```

### Note: `merge_lookup()` Keeps Only the Keys in the Left Dataset

One thing to note about `merge_lookup()` is that it keeps only the keys are that are in the left Dataset (it's equivalent to a SQL left join). If there are keys in the right Dataset that aren't in the left Dataset, they're discarded in the merged data:

```
>>> # Create a right Dataset with an extra symbol key ('MSFT').
>>> ds_symbol_trader2 = rt.Dataset({'Symbol': ['GME', 'TSLA', 'SPY', 'AMZN', 'MSFT'],
...                                 'Trader': ['Nate', 'Elon', 'Josh', 'Dan', 'Lauren']})
>>> # Change 'UnderlyingSymbol' back to 'Symbol' for simplicity.
>>> ds_symbol_trader.col_rename('UnderlyingSymbol', 'Symbol')
>>> ds.merge_lookup(ds_symbol_trader2, on='Symbol', columns_right='Trader')
 #   Symbol   Size   Trader
--   ------   ----   ------
 0   GME       644   Nate
```

(continues on next page)

(continued from previous page)

```
 1   SPY       403   Josh
 2   TSLA      822   Elon
 3   AMZN      545   Dan
 4   AMZN      443   Dan
 5   SPY       451   Josh
 6   GME       228   Nate
 7   TSLA       93   Elon
 8   GME       555   Nate
 9   GME       888   Nate
10   TSLA       64   Elon
11   SPY       858   Josh
12   TSLA      827   Elon
13   SPY       277   Josh
14   TSLA      632   Elon
15   SPY       166   Josh
16   TSLA      758   Elon
17   GME       700   Nate
18   SPY       355   Josh
19   AMZN       68   Dan
20   TSLA      970   Elon
21   AMZN      446   Dan
22   GME       893   Nate
23   SPY       678   Josh
24   SPY       778   Josh
```

### `merge_lookup()` with Overlapping Columns That Aren't Keys

As we saw above, if the two key columns have the same name in both Datasets, only one is kept. For columns that aren't used as keys, you'll get a name collision error when you try to merge:

```
>>> # Add a Size column to the right Dataset
>>> ds_symbol_trader.Size = rng.integers(1, 1000, 4)

>>> try:
...     rt.merge_lookup(ds, ds_symbol_trader, on='Symbol')
... except ValueError as e:
...     print("ValueError:", e)
ValueError: columns overlap but no suffix specified: {'Size'}
```

If you want to keep both columns, add a suffix to each column name to disambiguate them:

```
>>> rt.merge_lookup(ds, ds_symbol_trader, on='Symbol', suffixes=('_1', '_2'))
 #   Symbol   Size_1   Trader   Size_2
--   ------   ------   ------   ------
 0   GME         644   Nate        760
 1   SPY         403   Josh        364
 2   TSLA        822   Elon        195
 3   AMZN        545   Dan         467
 4   AMZN        443   Dan         467
 5   SPY         451   Josh        364
 6   GME         228   Nate        760
```

(continues on next page)

(continued from previous page)

```
 7   TSLA         93   Elon        195
 8   GME         555   Nate        760
 9   GME         888   Nate        760
10   TSLA         64   Elon        195
11   SPY         858   Josh        364
12   TSLA        827   Elon        195
13   SPY         277   Josh        364
14   TSLA        632   Elon        195
15   SPY         166   Josh        364
16   TSLA        758   Elon        195
17   GME         700   Nate        760
18   SPY         355   Josh        364
19   AMZN         68   Dan         467
20   TSLA        970   Elon        195
21   AMZN        446   Dan         467
22   GME         893   Nate        760
23   SPY         678   Josh        364
24   SPY         778   Josh        364
```

### `merge_lookup()` with a Right Dataset That Has Duplicate Keys

If the right Dataset has more than one match for a unique key in the left Dataset, you can specify whether to use the first or the last match encountered in the right Dataset:

```
>>> # Create a right Dataset with a second GME key, associated to Lauren
>>> ds_symbol_trader3 = rt.Dataset({'Symbol': ['GME', 'TSLA', 'SPY', 'AMZN', 'GME'],
...                                 'Trader': ['Nate', 'Elon', 'Josh', 'Dan', 'Lauren']})
>>> ds_symbol_trader3
#   Symbol   Trader
-   ------   ------
0   GME      Nate
1   TSLA     Elon
2   SPY      Josh
3   AMZN     Dan
4   GME      Lauren
```

We'll keep the last match:

```
>>> ds.merge_lookup(ds_symbol_trader3, on='Symbol', columns_right='Trader', keep='last')
 #   Symbol   Size   Trader
--   ------   ----   ------
 0   GME       644   Lauren
 1   SPY       403   Josh
 2   TSLA      822   Elon
 3   AMZN      545   Dan
 4   AMZN      443   Dan
 5   SPY       451   Josh
 6   GME       228   Lauren
 7   TSLA       93   Elon
 8   GME       555   Lauren
 9   GME       888   Lauren
```

(continues on next page)

(continued from previous page)

```
10   TSLA       64   Elon
11   SPY       858   Josh
12   TSLA      827   Elon
13   SPY       277   Josh
14   TSLA      632   Elon
15   SPY       166   Josh
16   TSLA      758   Elon
17   GME       700   Lauren
18   SPY       355   Josh
19   AMZN       68   Dan
20   TSLA      970   Elon
21   AMZN      446   Dan
22   GME       893   Lauren
23   SPY       678   Josh
24   SPY       778   Josh
```

### 1.8.2 merge_asof()

In a `merge_asof()`, Riptable matches on the nearest key rather than an equal key.

This is useful for merges based on keys that are times, where the times in one Dataset are not an exact match for the times in another Dataset, but they're close enough to be used to merge the data.

Note: To most efficiently find the nearest match, `merge_asof()` requires both key columns to be sorted. The key columns must also be numeric, such as a datetime, integer, or float. You can check whether a column is sorted with `issorted()`, or just sort it using `sort_inplace()`. (If the key columns aren't sorted, Riptable will give you an error when you try to merge.)

With `merge_asof()`, you need to specify how you want to find the closest match:

- `direction='forward'` matches based on the closest key in the right Dataset that's greater than the key in the left Dataset.
- `direction='backward'` matches based on the closest key in the right Dataset that's less than the key in the left Dataset.
- `direction='nearest'` matches based on the closest key in the right Dataset, regardless of whether it's greater than or less than the key in the left Dataset.

Let's see an example based on closest times. The left Dataset has three trades and their times. The right Dataset has spot prices and times that are not all exact matches. We'll merge the spot prices from the right Dataset by getting the values associated with the nearest earlier times.

```
>>> # Left Dataset with trades and times
>>> ds = rt.Dataset({'Symbol': ['AAPL', 'AMZN', 'AAPL'],
...                  'Venue': ['A', 'I', 'A'],
...                  'Time': rt.TimeSpan(['09:30', '10:00', '10:20'])})
>>> # Right Dataset with spot prices and nearby times
>>> spot_ds = rt.Dataset({'Symbol': ['AMZN', 'AMZN', 'AMZN', 'AAPL', 'AAPL', 'AAPL'],
...                       'Spot Price': [2000.0, 2025.0, 2030.0, 500.0, 510.0, 520.0],
...                       'Time': rt.TimeSpan(['09:30', '10:00', '10:25', '09:25', '10:00
↪', '10:25'])})
>>> ds
#   Symbol   Venue                Time
```

(continues on next page)

(continued from previous page)

```
-   ------   -----   ------------------
0   AAPL     A       09:30:00.000000000
1   AMZN     I       10:00:00.000000000
2   AAPL     A       10:20:00.000000000
```

```
>>> spot_ds
#   Symbol   Spot Price                Time
-   ------   ----------   ------------------
0   AMZN       2,000.00   09:30:00.000000000
1   AMZN       2,025.00   10:00:00.000000000
2   AMZN       2,030.00   10:25:00.000000000
3   AAPL         500.00   09:25:00.000000000
4   AAPL         510.00   10:00:00.000000000
5   AAPL         520.00   10:25:00.000000000
```

Note that an as-of merge requires the `on` columns to be sorted. Before the merge, the `on` columns are always checked. If they're not sorted, by default they are sorted before the merge; the original order is then restored in the returned merged Dataset.

If you don't need to preserve the existing ordering, it's faster to sort the `on` columns in place first:

```
>>> spot_ds.sort_inplace('Time')
#   Symbol   Spot Price                Time
-   ------   ----------   ------------------
0   AAPL         500.00   09:25:00.000000000
1   AMZN       2,000.00   09:30:00.000000000
2   AMZN       2,025.00   10:00:00.000000000
3   AAPL         510.00   10:00:00.000000000
4   AAPL         520.00   10:25:00.000000000
5   AMZN       2,030.00   10:25:00.000000000
```

Now we can merge based on the nearest earlier time. But not just any nearest earlier time – we want to make sure it's the nearest earlier time associated with the same symbol. We use the optional `by` parameter to make sure we match on the symbol before getting the nearest earlier time. We'll also use the `matched_on` argument to show which key in `spot_ds` was matched on:

```
>>> ds.merge_asof(spot_ds, on='Time', by='Symbol', direction='backward', matched_on=True)
#   Symbol   Time                Venue   Spot Price   matched_on
-   ------   ------------------  -----   ----------   ------------------
0   AAPL     09:30:00.000000000  A           500.00   09:25:00.000000000
1   AMZN     10:00:00.000000000  I         2,025.00   10:00:00.000000000
2   AAPL     10:20:00.000000000  A           510.00   10:00:00.000000000
```

We can see that both AAPL trades were matched based on the nearest earlier time.

Merge based on the nearest later time:

```
>>> ds.merge_asof(spot_ds, on='Time', by='Symbol', direction='forward', matched_on=True)
#   Symbol   Time                Venue   Spot Price   matched_on
-   ------   ------------------  -----   ----------   ------------------
0   AAPL     09:30:00.000000000  A           510.00   10:00:00.000000000
1   AMZN     10:00:00.000000000  I         2,025.00   10:00:00.000000000
2   AAPL     10:20:00.000000000  A           520.00   10:25:00.000000000
```

Both AAPL trades were matched based on the nearest later time.

Here, we get the spot price associated with whatever time is nearest, whether it's earlier or later:

```
>>> ds.merge_asof(spot_ds, on='Time', by='Symbol', direction='nearest', matched_on=True)
#   Symbol                Time   Venue   Spot Price           matched_on
-   ------   ------------------   -----   ----------   ------------------
0   AAPL     09:30:00.000000000   A           500.00   09:25:00.000000000
1   AMZN     10:00:00.000000000   I         2,025.00   10:00:00.000000000
2   AAPL     10:20:00.000000000   A           520.00   10:25:00.000000000
```

For the first AAPL trade, the nearest time is earlier. For the second AAPL trade, the nearest time is later.

We won't spend time on examples of `merge2()`, which is Riptable's more general merge function that does database-style joins (left, right, inner, outer). Check out the API Reference for details.

Next, we'll briefly cover a couple of ways to change the shape of your Dataset: *Reshape Data with Pivot and Transpose*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.9 Reshape Data with Pivot and Transpose

Riptable is designed to efficiently work with column-oriented data, which is also called long-format data. This isn't always the best format for displaying data for human consumption, however.

For example, suppose your data consists of a measurement (say, trade volume) per date and symbol. The long-format, Riptable-friendly way to represent this is to have three columns – for date, symbol, and volume:

```
>>> long_ds = rt.Dataset({'Date': ['20191111', '20191111', '20191111', '20191112',
...                                '20191112', '20191112'],
...                       'Symbol': ['AAPL', 'MSFT', 'TSLA', 'MSFT', 'AAPL', 'TSLA'],
...                       'Volume': [10, 20, 30, 20, 10, 30]})
>>> long_ds
#   Date       Symbol   Volume
-   --------   ------   ------
0   20191111   AAPL         10
1   20191111   MSFT         20
2   20191111   TSLA         30
3   20191112   MSFT         20
4   20191112   AAPL         10
5   20191112   TSLA         30
```

While this format is ideal for Riptable's work, the repeated date and symbol values make it a bit unintuitive for humans to read and make sense of.

In this case, a simple transform from long format to wide doesn't help much:

```
>>> long_ds._T
Fields:            0              1              2              3              4  ␣
↪          5
   Date     20191111       20191111       20191111       20191112       20191112  ␣
↪   20191112
 Symbol         AAPL           MSFT           TSLA           MSFT           AAPL  ␣
```

(continues on next page)

(continued from previous page)

```
↪       TSLA
 Volume          10             20             30             20             10    ␣
↪         30
```

A more human-friendly presentation can be gotten from the Dataset method `pivot()`, which reorganizes data with multiple keys (here, our keys are the date and the symbol).

We can use `pivot()` to show one row per date and one column for each symbol:

```
>>> wide_ds = long_ds.pivot('Date', 'Symbol', 'Volume')
>>> wide_ds
*Date      AAPL   MSFT   TSLA
--------   ----   ----   ----
20191111     10     20     30
20191112     10     20     30
```

The first argument passed is used for the row labels; the second is for the column labels. The third argument specifies which column's (or columns') data to use to populate the table. (If none are specified, all remaining columns are used.)

Notice the output's similarity to that of `Accum2()`:

```
>>> long_ds.accum2(long_ds.Date, long_ds.Symbol).sum(long_ds.Volume)
*Date      AAPL   MSFT   TSLA   Total
--------   ----   ----   ----   -----
20191111     10     20     30      60
20191112     10     20     30      60
   Total     20     40     60     120
```

Also note that some wide-format data may be too wide for reasonable display.

To undo your pivot (or "unpivot"), use `melt()`:

```
>>> melted_ds = wide_ds.melt('Date')
>>> melted_ds
#   Date       variable   value
-   --------   --------   -----
0   20191111   AAPL          10
1   20191112   AAPL          10
2   20191111   MSFT          20
3   20191112   MSFT          20
4   20191111   TSLA          30
5   20191112   TSLA          30
```

Here, we specified the Date column as the "identifier variable." The other columns, considered "measured variables," are unpivoted to the row axis with the column headers "variable" and "value."

We could have specified our original column labels with `var_name` and `value_name`:

```
>>> melted_ds = wide_ds.melt('Date', var_name='Symbol', value_name='Volume')
>>> melted_ds
#   Date       Symbol   Volume
-   --------   ------   ------
0   20191111   AAPL         10
1   20191112   AAPL         10
2   20191111   MSFT         20
```

(continues on next page)

(continued from previous page)

```
3   20191112   MSFT         20
4   20191111   TSLA         30
5   20191112   TSLA         30
```

Next, we'll change gears to give you a high-level overview of tools you can use to *Visualize Data*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.10 Visualize Data

Riptable can work with any of the standard plotting tools, including Matplotlib, to create visualizations of your data. You can also take advantage of the plotting and HTML styling tools offered by Pandas.

In this section we'll look at a couple of simple examples using Matplotlib, Pandas, and Playa.

```
import pandas as pd
import matplotlib.pyplot as plt
```

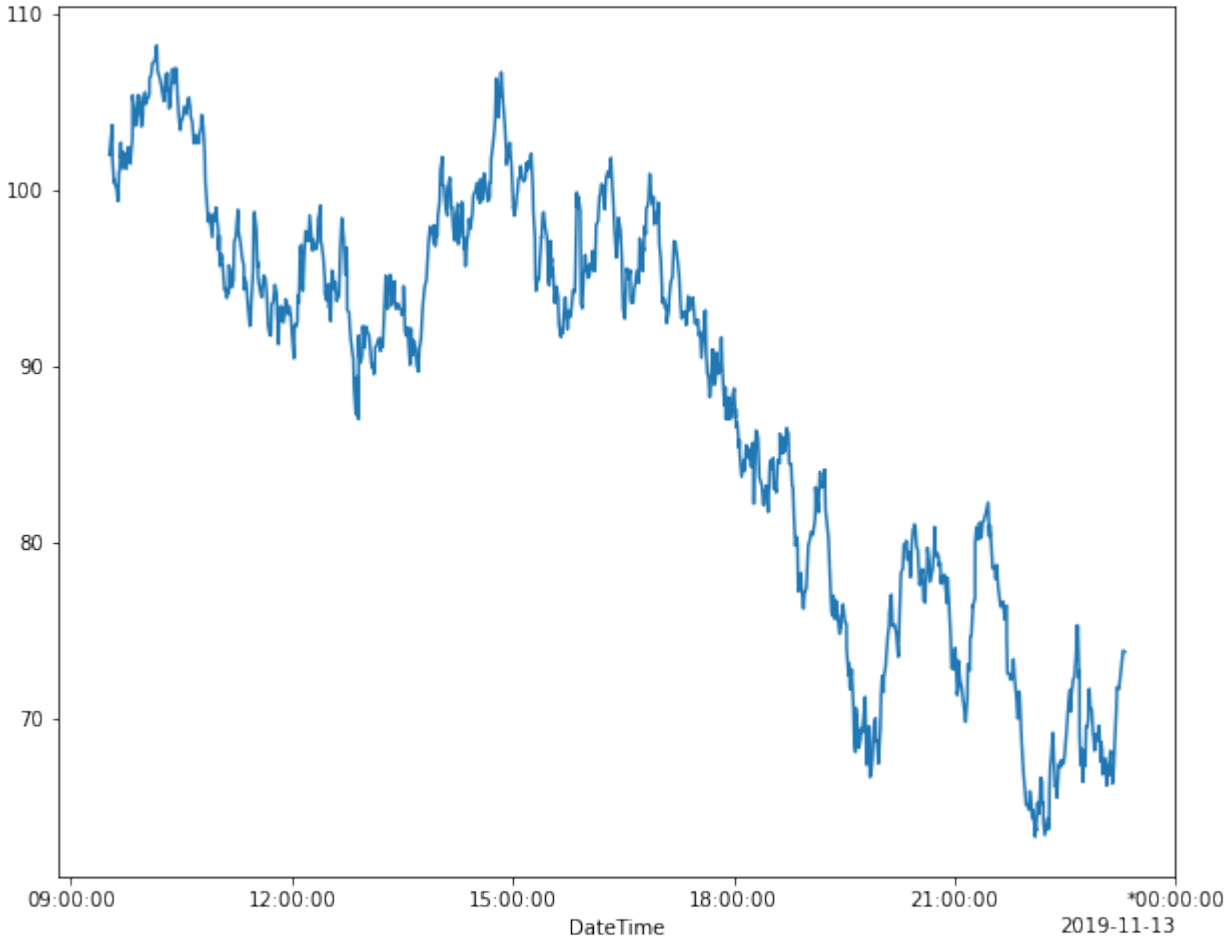
We'll use this Dataset in examples:

```
>>> rng = np.random.default_rng(seed=42)
>>> N = 1_000
>>> symbols = ['IBM', 'AAPL', 'F', 'CSCO', 'SPY']
>>> start_time = rt.DateTimeNano('20191112 09:30:00', from_tz='NYC')
>>> ds_list = []
>>> for symbol in symbols:
...     temp_ds = rt.Dataset({'Symbol': rt.full(N, symbol),
...                           'Price': 100.0 + 10 * rng.standard_normal(1) + rng.standard_
↪normal(N).cumsum(),
...                           'Size': rng.integers(1, 50, 1) + rng.integers(1, 50, N),
...                           'Time': start_time + rt.TimeSpan(rng.integers(1, 100, N).
↪cumsum(), 's'),
...                          })
...     ds_list.append(temp_ds)
>>> ds = rt.hstack(ds_list)
>>> ds = ds.sort_inplace('Time')
>>> ds.sample()
#   Symbol    Price   Size                          Time
-   ------   ------   ----   ---------------------------
0   SPY      114.70     36   20191112 10:53:02.000000000
1   CSCO      88.83     74   20191112 16:24:38.000000000
2   SPY      140.37     66   20191112 16:42:23.000000000
3   SPY      147.62     53   20191112 17:40:02.000000000
4   AAPL     125.24     68   20191112 18:02:39.000000000
5   SPY      141.60     67   20191112 18:19:48.000000000
6   SPY      139.88     61   20191112 18:41:54.000000000
7   SPY      139.81     69   20191112 19:32:47.000000000
8   SPY      143.03     50   20191112 20:40:38.000000000
9   AAPL     130.09     87   20191112 22:22:09.000000000
```

### 1.10.1 Matplotlib Plotting

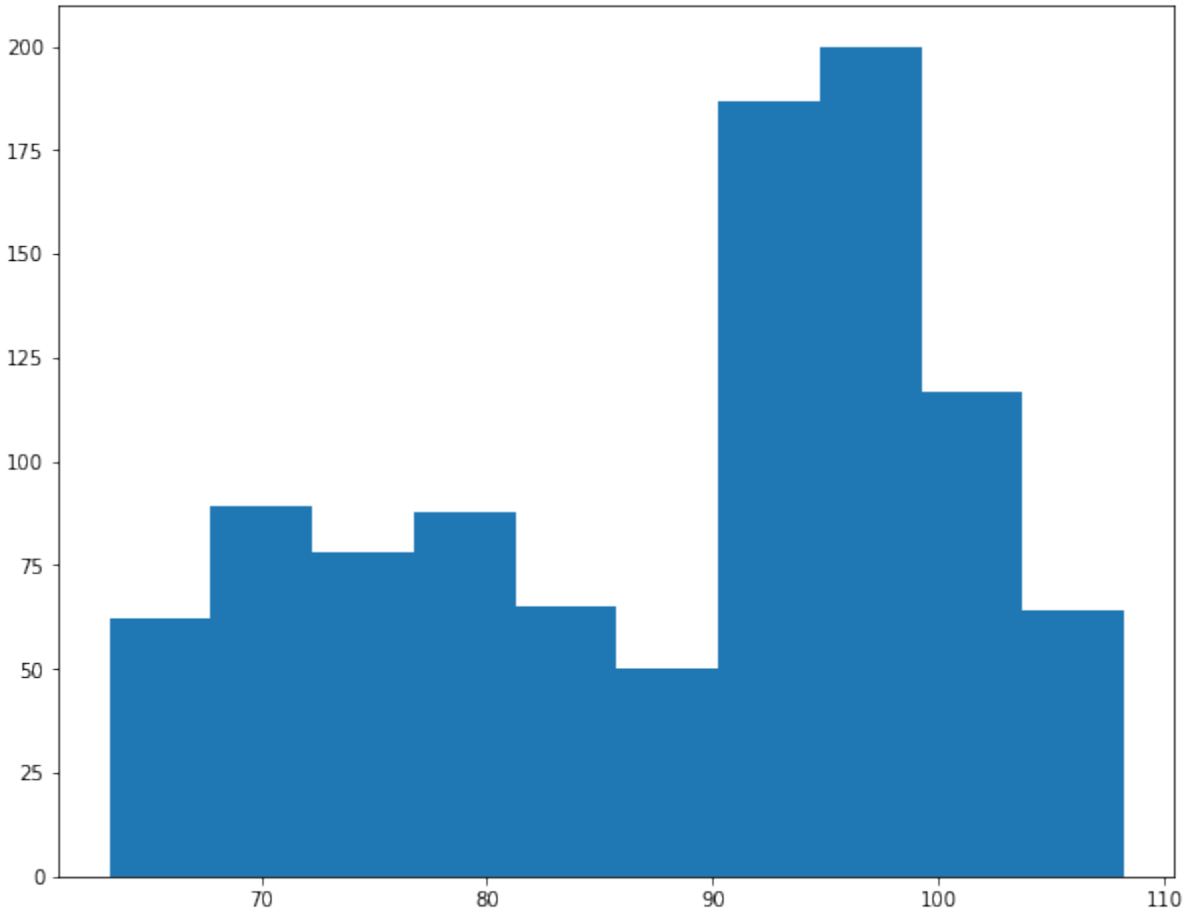
Example of a basic plot of IBM's share price:

```
>>> f = ds.Symbol=='IBM'
>>> plt.figure(figsize=(10,8))
>>> plt.plot(ds.Time[f], ds.Price[f])
>>> plt.show()
```



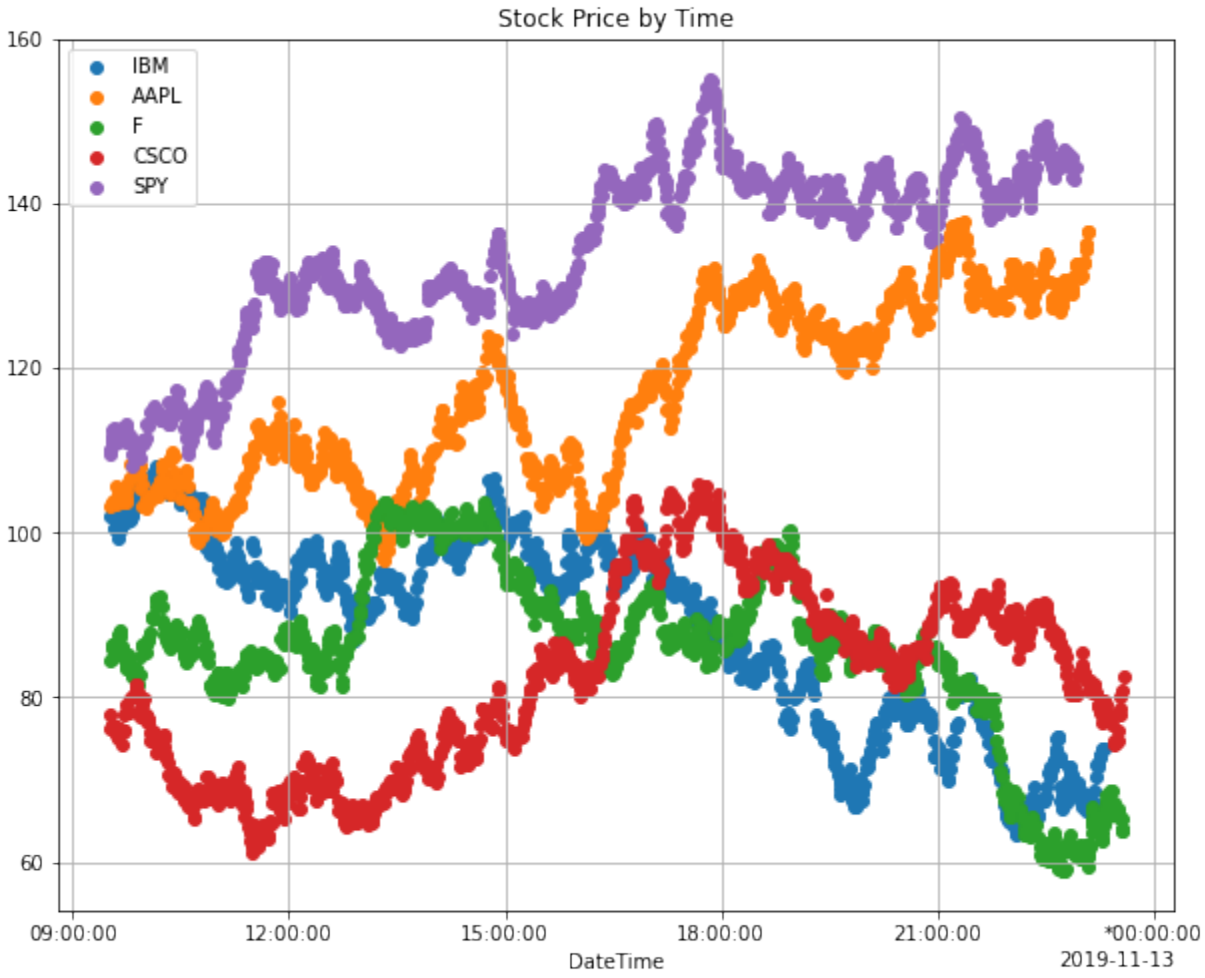

And a histogram:

```
>>> plt.figure(figsize=(10,8))
>>> plt.hist(ds.Price[f])
>>> plt.show()
```




And a scatter plot:

```
>>> plt.figure(figsize=(10,8))
>>> for symbol in symbols:
...     f = ds.Symbol==symbol
...     plt.scatter(ds.Time[f], ds.Price[f], label=symbol)
>>> plt.grid()
>>> plt.legend()
>>> plt.title('Stock Price by Time')
>>> plt.show()
```




### 1.10.2 Pandas HTML Styling

If you want to use the Pandas Styler methods, call `to_pandas()` on your Dataset for the rendering:

```
>>> def color_smaller_red(val):
...     color = 'red' if type(val)==float and val < 100 else 'gray'
...     return 'color: %s' % color
>>> ds.to_pandas().head(10).style.applymap(color_smaller_red)
```

### 1.10.3 Groupscatter Plots with Playa

Playa's `GroupScatter()` method groups data into buckets based on x-values and returns a Matplotlib plot summarizing the data.

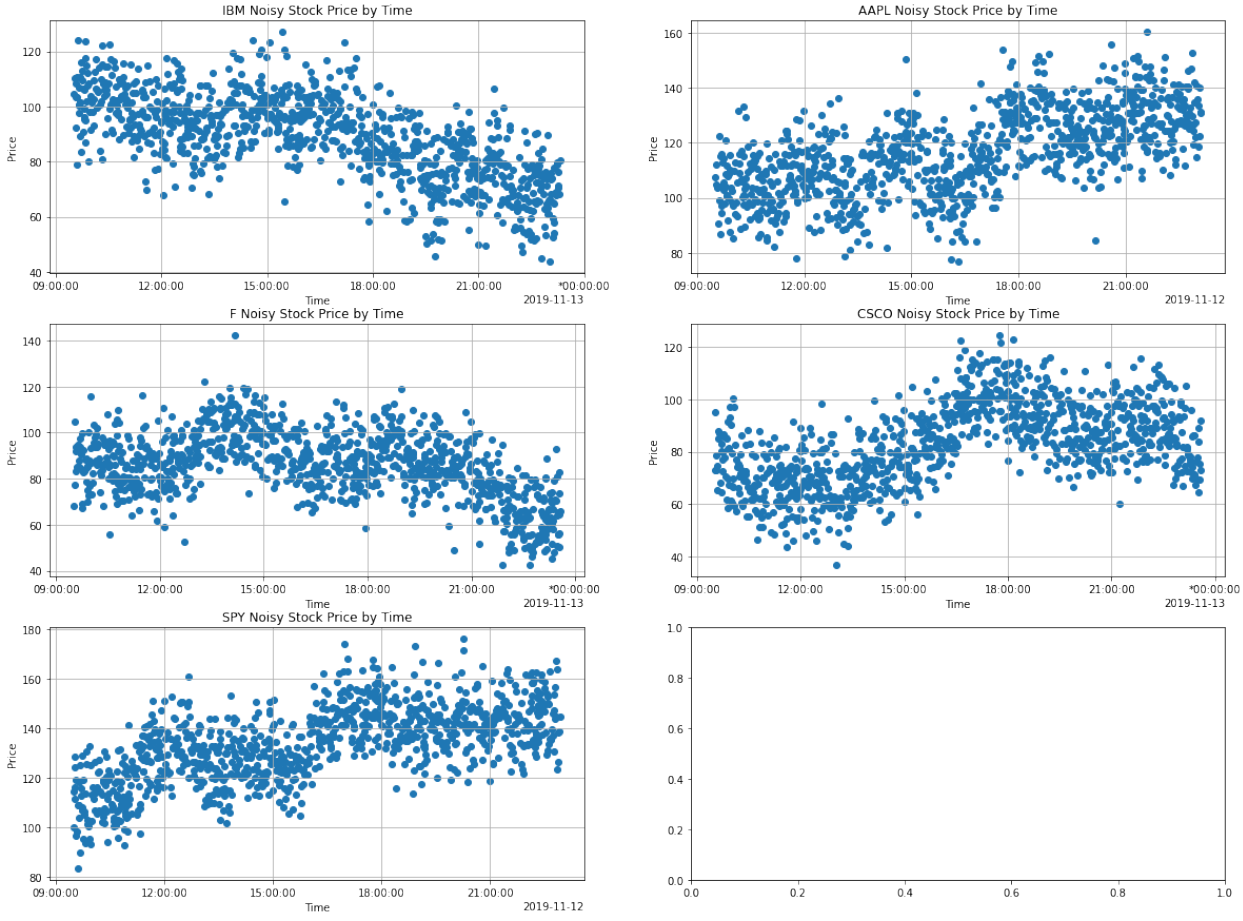
```
from playa.plot import GroupScatter
```

### Make a noisier price signal

```
>>> ds.NoisyPrice = ds.Price + rng.normal(0, 10, ds.shape[0])
```

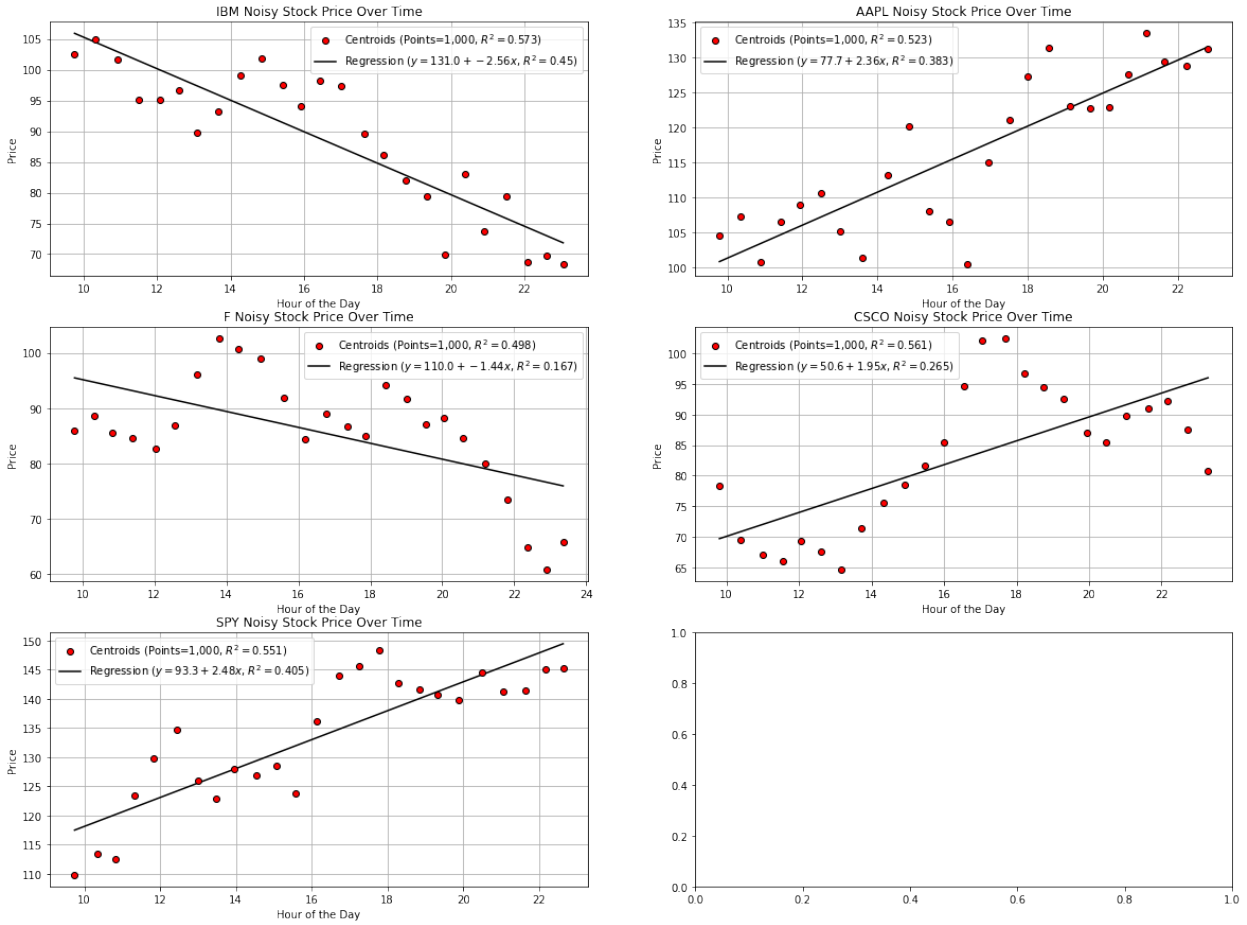
### A regular Matplotlib scatter plot, for comparison

```
>>> num_rows = int(rt.ceil(len(symbols)/2))
>>> fig, axes = plt.subplots(num_rows, 2, figsize=(20, 5 * num_rows))
>>> for (ax, symbol) in zip(axes.flatten(), symbols):
...     f = ds.Symbol==symbol
...     ax.scatter(ds.Time[f], ds.NoisyPrice[f])
...     ax.grid()
...     ax.set_xlabel('Time')
...     ax.set_ylabel('Price')
...     ax.set_title(f'{symbol} Noisy Stock Price by Time')
>>> plt.show()
```



**Now a GroupScatter for each one, you can see how it clarifies the point cloud and reveals the shape.**

```
>>> fig, axes = plt.subplots(num_rows, 2, figsize=(20, 5 * num_rows))
>>> for (ax, symbol) in zip(axes.flatten(), symbols):
...     f = ds.Symbol==symbol
...     gs = GroupScatter(ds.Time[f].hour, ds.NoisyPrice[f])
...     gs.plot(title=f'{symbol} Noisy Stock Price Over Time', x_label='Hour of the Day',
↪ y_label='Price', ax=ax)
>>> plt.show()
```




This was just a brief introduction – check out the Matpotlib, Pandas, and Playa documentation for more details and possibilities.

Next we cover useful tools for working with `NaN`s and other missing values: *Working with Missing Data*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.11 Working with Missing Data

When you work with real-world data, you often have to deal with missing values. It's useful to know how Riptable stores and represents missing values, how to detect missing values in your data, and how you can use a few strategies to fill in missing values so you can continue to work with the data effectively.

If you convert data between Riptable and other libraries, it's also important to know how conversions of missing values are handled. In this section, we show how missing values are converted between Riptable and Pandas.

### 1.11.1 Riptable Sentinel Values

Riptable uses sentinel values for missing data. Missing floating-point numbers are `NaN`s (Not a Number), per the IEEE Standard for Floating-Point Arithmetic. In Riptable, missing floating-point numbers are indicated by `nan`. Missing integers are indicated by `Inv`:

```
>>> ds = rt.Dataset({'Ints': [1, 2, 3], 'Floats': [0.1, 1.5, 2.7]})
>>> ds.Ints[0] = ds.Ints.inv
>>> ds.Floats[0] = ds.Floats.inv
>>> ds
#   Ints   Floats
-   ----   ------
0    Inv      nan
1      2     1.50
2      3     2.70
```

Note the difference in how they're stored. The floating-point `NaN` is stored as `nan`:

```
>>> ds.Floats
FastArray([nan, 1.5, 2.7])
```

The missing integer is a large negative number:

```
>>> ds.Ints
FastArray([-2147483648,           2,           3])
```

In Riptable, missing interger values are stored as `-MAXINT` for ints and `MAXINT` for unsigned ints. This has the potential to cause problems, which we'll look at below.

Tip: To find out what the missing/invalid value is for an array, use `inv` property. The array doesn't necessarily contain the invalid value; what's returned is the invalid value for the array's dtype:

```
>>> ds.Ints.inv
-2147483648
```

Arithmetic with floating-point `NaN` values is well-established: any operation involving a `NaN` is another `NaN`:

```
>>> ds.Floats.sum()
>>> ds.FloatsPlus = ds.Floats * 2
>>> ds
#   Ints   Floats   FloatsPlus
-   ----   ------   ----------
0    Inv      nan          nan
1      2     1.50         3.00
2      3     2.70         5.40
```

To help, many arithmetic functions have NaN versions that ignore NaN values:

```
>>> ds.Ints.nansum()
5
```

Be careful with missing integers, however! As of this writing, missing integer values are treated at face value in arithmetic operations:

```
>>> ds.Ints.sum()
-2147483643
```

Fortunately, the NaN versions ignore the missing values:

```
>>> ds.Ints.nansum()
5
```

There are a few methods for detecting missing values in Riptable structures.

For FastArrays, isnan() and notna() both return Boolean mask arrays. As you might expect, isnan() returns True where it finds a NaN value:

```
>>> ds.Ints.isnan()
FastArray([ True, False, False])
```

And notna() returns True where it finds a non-NaN value:

```
>>> ds.Floats.notna()
FastArray([False,  True,  True])
```

A more general approach is to use isfinite(). It returns a Boolean array where False indicates either a NaN or a value of positive or negative infinity:

```
>>> ds.Floats[1] = np.inf
>>> ds.Floats.isfinite()
FastArray([False, False,  True])
```

And as you might imagine, isnotfinite() does the opposite:

```
>>> ds.Floats.isnotfinite()
FastArray([ True,  True, False])
```

Note that inf is not considered a NaN. The NaN versions of functions don't ignore infinite values (the result is positive or negative inf), so it can be good to check for them:

```
>>> ds.Floats.nansum()
inf
```

For Datasets, mask_and_isnan() and mask_or_isnan() each return a FastArray of Booleans with a value for each row.

mask_and_isnan() returns True for each row in which every value is NaN:

```
>>> ds.mask_and_isnan()
FastArray([ True, False, False])
```

mask_or_isnan() returns True for each row in which at least one value is NaN:

```
ds.mask_or_isnan()
FastArray([ True, False, False])
```

### 1.11.2 Merging with Missing Values

Missing values are not equivalent:

```
>>> rt.nan == rt.nan
False
```

This is true for integer invalid values, string invalid values, filtered values of a Categorical, etc. That means that merge functions do not treat invalid keys as equal values.

For example, these two Datasets each have an invalid floating-point value in the Key column:

```
>>> ds1 = rt.Dataset({'Key': [1.0, rt.nan, 2.0],
...                   'Value1': ['a', 'b', 'c']})
>>> ds2 = rt.Dataset({'Key': [1.0, 2.0, rt.nan],
...                   'Value2': [1, 2, 3]})
```

Now we do a `merge_lookup()` on the Key columns:

```
>>> ds1.merge_lookup(ds2, on='Key')
#    Key   Value1   Value2
-   ----   ------   ------
0   1.00   a             1
1    nan   b           Inv
2   2.00   c             2
```

The `NaN` key and its associated value in `ds2` were ignored, and the invalid integer value was filled in.

### 1.11.3 Replacing Missing Values

For both FastArrays and Datasets, calling `fillna()` with a constant is a quick way to replace missing values:

```
>>> ds.fillna(123)
#   Ints   Floats   FloatsPlus
-   ----   ------   ----------
0    123   123.00       123.00
1      2      inf         3.00
2      3     2.70         5.40
```

Note that by default `fillna()` returns a copy; to modify the original data, use `inplace=True`.

For a little more nuance in how the gaps are filled, use `fillna()` with `method='ffill'` or `method='bfill'`.

`fillna(method='ffill')` propagates non-`NaN` values forward:

```
>>> rt.FA([1.0, 2.0, np.nan, 4.0, 5.0]).fillna(method='ffill')
FastArray([1., 2., 2., 4., 5.])
```

`fillna(method='bfill')` propagates non-NaN values backward:

```
>>> rt.FA([1.0, 2.0, np.nan, 4.0, 5.0]).fillna(method='bfill')
FastArray([1., 2., 4., 4., 5.])
```

For Categoricals, `fill_forward()` and `fill_backward()` propagate values within categories:

```
>>> # Create a Categorical with a NaN in each category
>>> ds = rt.Dataset()
>>> ds.Cat = rt.Cat(['A', 'B', 'A', 'B', 'A', 'B'])
>>> ds.x = rt.FA([1, 4, rt.nan, rt.nan, 9, 16])
>>> ds
#   Cat       x
-   ---   -----
0   A      1.00
1   B      4.00
2   A       nan
3   B       nan
4   A      9.00
5   B     16.00
```

Propagate forward the last encountered non-`NaN` value for the category:

```
>>> ds.Cat.fill_forward(ds.x)
*gb_key_0       x
---------   -----
A            1.00
B            4.00
A            1.00
B            4.00
A            9.00
B           16.00
```

Note that until a reported bug is fixed, explicit column name declarations might not be displayed for grouping operations.

Propagate backward the next encountered non-NaN value for the category:

```
>>> ds.Cat.fill_backward(ds.x)
*gb_key_0       x
---------   -----
A            1.00
B            4.00
A            9.00
B           16.00
A            9.00
B           16.00
```

Both `fill_forward()` and `fill_backward()` can take a list of arrays to fill, and both can modify data in place with `inplace=True`.

Note that if there is no value available to propagate forward or backward, the `NaN` value isn't changed:

```
>>> ds.x[1] = rt.nan
>>> ds.Cat.fill_forward(ds.x)
*gb_key_0       x
---------   -----
```

(continues on next page)

(continued from previous page)

```
A            1.00
B             nan
A            1.00
B             nan
A            9.00
B           16.00
```

### 1.11.4 Convert Missing Values to/from Pandas

This section covers some things to be aware of when you convert data with missing values between Pandas and Riptable.

Note that while you can convert Pandas DataFrames to Riptable Datasets using Riptable's Dataset constructor, you should use the Dataset methods `to_pandas` and `from_pandas` to convert data with missing values.

#### Converting Floats

To represent missing floating-point values, both Pandas and Riptable use the special floating-point `NaN` value that's part of the IEEE standard (though in Riptable, it's displayed as `nan`). Converting floating-point `NaN` values between Pandas and Riptable poses no issues:

```
>>> df = pd.DataFrame({'A': [0.0, np.nan, 1.0]})
>>> ds = rt.Dataset.from_pandas(df)
>>> ds
#      A
-   ----
0   0.00
1    nan
2   1.00

>>> df_again = ds.to_pandas()
>>> df_again
     A
0  0.0
1  NaN
2  1.0
```

#### Converting Integers

Converting integers gets more interesting. Pandas has a new nullable integer data type (Int64, not to be confused with NumPy's int64 dtype). A missing value in an Int64 column is represented by the native `pd.NA` value and displayed as `<NA>`.

Before this new dtype was created, the only numeric `NaN` used by Pandas was a floating-point `NaN`, so any `NaN` value added to an integer array in Pandas would cause the array to become an array of floating-point numbers:

```
>>> s1 = pd.Series([1, 2, 3, 4, 5])
>>> s1[1] = np.nan
>>> s1
0    1.0
1    NaN
```

(continues on next page)

(continued from previous page)

```
2    3.0
3    4.0
4    5.0
dtype: float64
```

Since this is now just a column of floats, converting it to Riptable is just as shown above.

Now, in Pandas, you can specify the new Int64 dtype (it's not yet used by default). Missing values are represented by `pd.NA`, displayed as <NA>:

```
>>> s2 = pd.Series([1, 2, 3, 4, 5], dtype='Int64')
>>> s2[1] = np.nan
>>> s2
0       1
1    <NA>
2       3
3       4
4       5
dtype: Int64
```

When we convert these to Riptable, the Int64 <NA> remains an integer (but now the int64 dtype):

```
>>> # Create a DataFrame with the series from above.
>>> df = pd.DataFrame({'Float': s1, 'Int64': s2})
>>> # Convert the DataFrame to a Riptable Dataset and display its dtypes.
>>> ds2 = rt.Dataset.from_pandas(df)
>>> ds2.dtypes
{'Float': dtype('float64'), 'Int64': dtype('int64')}
```

When you convert data with missing integer values from Riptable to Pandas, by default `to_pandas()` converts to the new Int64 dtype:

```
>>> df_again2 = ds2.to_pandas()
>>> df_again2.dtypes
Float    float64
Int64      Int64
dtype: object
```

You can choose to not convert to the new nullable dtype, but your integers might not be very useful:

```
>>> df_again3 = ds2.to_pandas(use_nullable=False)
>>> df_again3
   Float                Int64
0    1.0                    1
1    NaN -9223372036854775808
2    3.0                    3
3    4.0                    4
4    5.0                    5
```

### Converting Datetimes

In Pandas, missing datetime values are represented as `NaT`. When those are converted to Riptable, they become an `Inv`:

```
>>> date_arr = pd.Series(pd.to_datetime(['01/01/2022', '02/01/2022', np.nan]))
>>> df2 = pd.DataFrame({'Timestamp': date_arr})
>>> ds3 = rt.Dataset.from_pandas(df2)
>>> ds3
#                    Timestamp
-   --------------------------
0   20220101 00:00:00.000000000
1   20220201 00:00:00.000000000
2                           Inv
```

The missing value becomes `NaT` again when converted back to Pandas:

```
>>> df_again3 = ds3.to_pandas()
>>> df_again3
                  Timestamp
0 2022-01-01 00:00:00+00:00
1 2022-02-01 00:00:00+00:00
2                       NaT
```

### Converting Missing Booleans and Strings from Pandas to Riptable

```
>>> str_arr = pd.Series(["aaa", "bbb"])
>>> bool_arr = pd.Series([True, False])
>>> df = pd.DataFrame({"Strings": str_arr, "Bools": bool_arr})
>>> df2 = df.reindex({0, 1, 2})  # Add a row of missing values
>>> df2
  Strings  Bools
0     aaa   True
1     bbb  False
2     NaN    NaN
```

When we convert Pandas `NaN` strings and Booleans to Riptable, the results are perhaps not quite what we expect:

```
>>> ds = rt.Dataset.from_pandas(df2)
>>> ds
#   Strings   Bools
-   -------   -----
0   aaa        1.00
1   bbb        0.00
2   nan         nan
```

As you can see, the Boolean column became a column of floating-point values with an `rt.nan`. If we try to recast the values, we get an unexpected result:

```
>>> ds.Bools = ds.Bools.astype(bool)
>>> ds
#   Strings   Bools
-   -------   -----
```

(continues on next page)

(continued from previous page)

```
0   aaa        True
1   bbb       False
2   nan        True
```

As for the "nan" in the Strings column, it is a string literal:

```
>>> ds.Strings
FastArray([b'aaa', b'bbb', b'nan'], dtype='|S3')
```

One way to avoid getting the string literal is to replace the missing value in Pandas (with a space, for example). Another way to deal with these values is to create a Boolean column that's True if the Pandas object is a `NaN`, then use that column as a mask array.

**Riptable NaN values**

- Int: -MAXINT (signed), MAXINT (unsigned)
- Float: nan
- String: b''
- Bool: False
- Date (stored as int): -MAXINT
- DTN (stored as int): -MAXINT
- TS (stored as float): nan

Next we cover a few ways to *Instantiate with Placeholder Values and Generate Sample Data*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.12 Instantiate with Placeholder Values and Generate Sample Data

It's useful to have a few tools in your back pocket for generating data quickly – either placeholder values (like 1s or 0s) meant to temporarily fill a certain structure you're instantiating, or sample values that mimic real data, which you can use to explore and experiment with Riptable.

Here's a brief sampling of Riptable and NumPy methods you can use. For complete details about these functions, see their API reference documentation.

### 1.12.1 Generate Placeholder Values

The following methods generate repeated 0s and 1s.

10 floating-point zeros:

```
>>> rt.zeros(10)
FastArray([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

10 integer zeros:

```
>>> rt.zeros(10, int)
FastArray([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

10 floating-point ones:

```
>>> rt.ones(10)
FastArray([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

10 integer ones:

```
>>> rt.ones(10, int)
FastArray([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

The following methods generate repeated specified values.

10 fives:

```
>>> rt.repeat(5, 10)
FastArray([5, 5, 5, 5, 5, 5, 5, 5, 5, 5])
```

10 repeats of each array element:

```
>>> rt.repeat([1, 2], 10)
FastArray([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

10 "tiles" of the entire array:

```
>>> rt.tile([1, 2], 10)
FastArray([1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2])
```

10 twos:

```
>>> rt.full(10, 2)
FastArray([2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

### 1.12.2 Generate Sample Data

Most of these methods generate a range of values.

`arange()` generates evenly spaced floating-point or integer values (depending on the input) within a given interval, including the start value but excluding the stop value. You can also specify a step size (the spacing between the values; the default is 1). It's like Python's *range* function, but it returns a FastArray rather than a list.

Numbers 0 through 9:

```
>>> rt.arange(10)
FastArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Every second number from 1 to 9:

```
>>> rt.arange(1, 10, 2)
FastArray([1, 3, 5, 7, 9])
```

For evenly spaced values where the step is a non-integer, it's better to use `np.linspace`. Instead of specifying the step value, you specify the number of elements. Both the start and stop values are included:

```
>>> np.linspace(2.0, 3.0, num=5)  # Five evenly spaced numbers from 2.0 to 3.0 (the step␣
↪is 0.25)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

For randomly generated values, you have several options.

For integers and floating-point values, NumPy has you covered. Call `default_rng` to get a new instance of a NumPy Generator, then call its methods. To generate values that can be replicated, initialize with a seed value of your choice to initialize the BitGenerator:

```
>>> rng = np.random.default_rng(seed=42)
```

10 floating-point numbers between 0.0 and 1.0 (1.0 excluded):

```
>>> rng.random(10)
array([0.77395605, 0.43887844, 0.85859792, 0.69736803, 0.09417735,
       0.97562235, 0.7611397 , 0.78606431, 0.12811363, 0.45038594])
```

10 uniformly distributed floats between 0 and 50 (50 excluded):

```
>>> rng.uniform(0, 50, 10)
array([18.53990121, 46.33824944, 32.193256  , 41.13808066, 22.17070994,
       11.36193609, 27.72923935,  3.19086281, 41.3815586 , 31.58321996])
```

10 integers between 1 and 50 (50 excluded):

```
>>> rng.integers(1, 50, 10)
array([ 9, 38, 35, 18,  4, 48, 22, 44, 34, 39], dtype=int64)
```

10 strings chosen from a list:

```
>>> rng.choice(['GME', 'AMZN', 'TSLA', 'SPY'], 10)
array(['SPY', 'GME', 'AMZN', 'AMZN', 'AMZN', 'GME', 'TSLA', 'GME', 'TSLA', 'TSLA'],␣
↪dtype='<U4')
```

10 random Booleans:

```
>>> rng.choice([True, False], 10)
array([False, False,  True, False,  True,  True, False,  True,  True, True])
```

See NumPy's documentation for more details and other methods.

Riptable has methods for generating random Date and DateTimeNano arrays.

5 DateTimeNanos with NYT time zone:

```
>>> rt.DateTimeNano.random(5)
DateTimeNano(['20000507 22:02:14.350793900', '20040720 00:24:28.668289697', '19771017 22:
↪34:39.521017110', '20130819 05:29:22.584265022', '20170622 00:50:06.970974486'], to_tz=
↪'NYC')
```

Dates between a start date and an end date (start and end dates included; the default step is 1 day):

```
>>> rt.Date.range('20190201', '20190208')
Date(['2019-02-01', '2019-02-02', '2019-02-03', '2019-02-04', '2019-02-05', '2019-02-06',
↪ '2019-02-07', '2019-02-08'])
```

5 dates, spaced two days apart, with a specified start date (start date included):

```
>>> rt.Date.range('20190201', days=5, step=2)
Date(['2019-02-01', '2019-02-03', '2019-02-05', '2019-02-07', '2019-02-09'])
```

Though `Date` objects don't (yet) have a `random` method, you can use `rng.choice` to pick dates from a range:

```
>>> rt.Date(rng.choice(rt.Date.range('20220201', '20220430'), 5))
Date(['2022-04-12', '2022-02-17', '2022-03-14', '2022-02-12', '2022-04-03'])
```

Next we cover ways to get data in and out of Riptable: *Work with Riptable Files and Other File Formats*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.13 Work with Riptable Files and Other File Formats

SDS is Riptable's native file format, and it's the only data format fully supported directly within Riptable. That said, there are ways to get data that's in other formats in and out of Riptable.

### 1.13.1 SDS

We'll start with the most straightforward case – saving and loading SDS files. You can save Datasets, FastArrays, or Structs.

Create a Dataset:

```
>>> ds = rt.Dataset({'Ints': rt.arange(10, dtype=int), 'Floats': rt.arange(1, step=0.1),
...                  'Categoricals': rt.Categorical(['a','a','b','a','c','c','b','a','a',
↪'b'])})
>>> ds
#   Ints   Floats   Categoricals
-   ----   ------   ------------
0      0     0.00   a
1      1     0.10   a
2      2     0.20   b
3      3     0.30   a
4      4     0.40   c
5      5     0.50   c
6      6     0.60   b
7      7     0.70   a
8      8     0.80   a
9      9     0.90   b
```

Save the Dataset:

```
>>> ds.save('ds.sds')
```

Load the Dataset:

```
>>> ds_load_ds = rt.load_sds('ds.sds')
>>> ds_load_ds
#   Ints   Floats   Categoricals
-   ----   ------   ------------
0      0     0.00   a
1      1     0.10   a
2      2     0.20   b
```

(continues on next page)

(continued from previous page)

```
3      3     0.30   a
4      4     0.40   c
5      5     0.50   c
6      6     0.60   b
7      7     0.70   a
8      8     0.80   a
9      9     0.90   b
```

Load a subset of columns:

```
>>> rt.load_sds('ds.sds', include=['Ints', 'Categoricals'])
#   Ints   Categoricals
-   ----   ------------
0      0   a
1      1   a
2      2   b
3      3   a
4      4   c
5      5   c
6      6   b
7      7   a
8      8   a
9      9   b
```

Create a FastArray:

```
>>> fa = rt.FastArray(np.arange(10))
>>> fa
FastArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Save the FastArray:

```
>>> fa.save('fa.sds')
```

Load the FastArray:

```
>>> fa_load_sds = rt.load_sds('fa.sds')
>>> fa_load_sds
FastArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int32)
```

Warning: Multi-key Categoricals can't be saved in SDS files. When you try to load the SDS file, it fails with an error: "Categories dict was empty."

### 1.13.2 CSV Files

Saving to CSV isn't supported by Riptable, but you can do it by first converting your Riptable Dataset to a Pandas DataFrame, then calling the Pandas `to_csv()` method. Later, you can load your CSV file into Riptable as a Dataset.

Note that Categorical information will be lost in the `to_csv()` process. When you load the CSV file into Riptable as a Dataset, any Categorical column will be a FastArray. You can always change the FastArray back into a Categorical in Riptable.

The `index` parameter for the `to_csv()` method indicates whether you want to write row (index) names. Because Riptable doesn't use explicit row indexing, set `index=False`.

Convert the Dataset to a Pandas DataFrame, then save the DataFrame as a CSV:

```
>>> ds.to_pandas().to_csv('ds.csv', index=False)
```

Read the CSV a into Pandas DataFrame, then convert the DataFrame to a Riptable Dataset using the Dataset constructor:

```
>>> ds_from_csv = rt.Dataset(pd.read_csv('ds.csv'))
>>> ds_from_csv
#   Ints   Floats   Categoricals
-   ----   ------   ------------
0      0     0.00   a
1      1     0.10   a
2      2     0.20   b
3      3     0.30   a
4      4     0.40   c
5      5     0.50   c
6      6     0.60   b
7      7     0.70   a
8      8     0.80   a
9      9     0.90   b
```

As you can see, the Categorical is now a FastArray:

```
>>> ds_from_csv.Categoricals
FastArray([b'a', b'a', b'b', b'a', b'c', b'c', b'b', b'a', b'a', b'b'], dtype='|S1')
```

But we can change it back:

```
>>> ds_from_csv.Categoricals = rt.Cat(ds_from_csv.Categoricals)
>>> ds_from_csv.Categoricals
Categorical([a, a, b, a, c, c, b, a, a, b]) Length: 10
  FastArray([1, 1, 2, 1, 3, 3, 2, 1, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

### 1.13.3 SQL Files

Working with SQL files and Riptable is much like working with CSV files and Riptable. To save a Riptable Dataset to SQL format, first convert the Dataset to a Pandas DataFrame, then use the Pandas `to_SQL()` method to save it.

To get the file back into Riptable, first load it in Pandas as a DataFrame using `read_csv()`, then convert it to a Riptable Dataset.

### 1.13.4 H5 Files

H5 files can be loaded in Riptable using `rt.load_h5()`. To save your data as an H5 file, convert to Pandas and use the Pandas `to_h5()` method.

### 1.13.5 NPY Files

Like Pandas, NumPy has various IO tools for saving and loading data. See the NumPy docs for details. Note that Riptable can initialize Datasets only from NumPy arrays that are record arrays.

### 1.13.6 Convert data for Use in Other Libraries

Sometimes, you need to access a function available only in NumPy or Pandas. Here's how to convert a Riptable data structure to its equivalent in NumPy or Pandas, and then back to Riptable.

#### Riptable FastArray to/from NumPy Array

When we first introduced FastArrays, we created one from a NumPy array:

```
>>> my_fa = rt.FA(np.array([0.1, 0.2, 0.3]))
```

To access a FastArray's underlying NumPy array, use `_np`:

```
>>> np_arr = my_fa._np
>>> np_arr
array([0.1, 0.2, 0.3])
```

This is the same result you'd get in Pandas by calling `Series.values`.

#### Riptable Dataset to/from NumPy Array

Converting a Dataset to a 2-dimensional NumPy array is a two-step process. First, use `imatrix_make()` to convert the Dataset to a 2-dimensional FastArray (`imatrix_make()` saves only the values – your column names will be lost). FastArrays above 1-d are not technically supported by Riptable, so don't stop here! Convert the FastArray to a NumPy array with `._np`:

```
>>> ds1 = rt.Dataset({'A':[0,6,9], 'B': [1.2,3.1,9.6], 'C':[-1.6,2.7,4.6], 'D': [2.4,6.2,
↪19.2]})
>>> np_2d_arr = ds1.imatrix_make()._np
>>> np_2d_arr
array([[ 0. ,  1.2, -1.6,  2.4],
       [ 6. ,  3.1,  2.7,  6.2],
       [ 9. ,  9.6,  4.6, 19.2]])
```

A few things to note about `imatrix_make()`:

- As noted above, imatrix_make saves only column values, not column names.
- Non-numerical columns are ignored.
- You can specify which columns to convert: `ds1[['A', 'B']].imatrix_make()._np`
- Watch out for integer columns! Since NumPy arrays can't have mixed types, if your `imatrix_make` input contains any float columns, the entire array will be converted to floats. It's also possible that the integers in your original Dataset will be converted.
- Also watch out for NaNs in integer columns ("Inv"). "Inv" is stored internally by Riptable as an out-of-bounds number, and it will be sent to NumPy as that number. See *Working with Missing Data* for more on dealing with NaNs.

- If there are Categoricals in the Dataset, you can preserve the integer mapping codes by passing `cats=True`.

To convert a 2-dimensional NumPy array back to Riptable, add it to a Dataset using `add_matrix()`:

```
>>> ds2 = rt.Dataset()
>>> ds2.add_matrix(np_2d_arr)
>>> ds2
#   col_0   col_1   col_2   col_3
-   -----   -----   -----   -----
0    0.00    1.20   -1.60    2.40
1    6.00    3.10    2.70    6.20
2    9.00    9.60    4.60   19.20
```

To add it with rows and columns transposed:

```
>>> ds3 = rt.Dataset()
>>> ds3.add_matrix(np_2d_arr.T)
>>> ds3
C:\\riptable\\rt_fastarray.py:561: UserWarning: FastArray initialized with strides.
  warnings.warn(warning_string)
#   col_0   col_1   col_2
-   -----   -----   -----
0    0.00    6.00    9.00
1    1.20    3.10    9.60
2   -1.60    2.70    4.60
3    2.40    6.20   19.20
```

### Riptable Dataset to/from Pandas DataFrame

Generally, you can use `from_pandas()` and `to_pandas()` to convert a Pandas DataFrame to a Riptable Dataset and vice-versa.

We'll create a Pandas DataFrame with categorical, timestamp, float and integer columns. We won't deal with NaN values here – see *Working with Missing Data* for guidance:

```
>>> rng = np.random.default_rng(seed=42)
>>> N = 10
>>> dates = pd.date_range('20191111','20191119')
>>> df = pd.DataFrame(
...     dict(Time = rng.choice(dates, N),
...          Symbol = pd.Categorical(rng.choice(['SPY','IBM'], N)),
...          Exchange = pd.Categorical(rng.choice(['AMEX','NYSE'], N)),
...          TradeSize = rng.choice([1,5,10], N),
...          TradePrice = rng.choice([1.1,2.2,3.3], N),
...         )
... )
>>> df
        Time Symbol Exchange  TradeSize  TradePrice
0 2019-11-11    IBM     NYSE          5         1.1
1 2019-11-17    IBM     AMEX          1         3.3
2 2019-11-16    IBM     AMEX          1         3.3
3 2019-11-14    IBM     NYSE          5         2.2
4 2019-11-14    IBM     NYSE         10         1.1
5 2019-11-18    IBM     NYSE          1         3.3
```

(continues on next page)

(continued from previous page)

```
6 2019-11-11    IBM     AMEX         10         2.2
7 2019-11-17    SPY     NYSE         10         3.3
8 2019-11-12    IBM     NYSE          1         3.3
9 2019-11-11    SPY     AMEX          5         3.3
```

The DataFrame dtypes before conversion:

```
>>> df.dtypes
Time          datetime64[ns]
Symbol              category
Exchange            category
TradeSize              int32
TradePrice           float64
dtype: object
```

Use `from_pandas()` to convert to a Dataset:

```
>>> ds = rt.Dataset.from_pandas(df)
>>> ds.head(5)
#                          Time   Symbol   Exchange   TradeSize   TradePrice
-   --------------------------   ------   --------   ---------   ----------
0   20191111 00:00:00.000000000   IBM      NYSE               5         1.10
1   20191117 00:00:00.000000000   IBM      AMEX               1         3.30
2   20191116 00:00:00.000000000   IBM      AMEX               1         3.30
3   20191114 00:00:00.000000000   IBM      NYSE               5         2.20
4   20191114 00:00:00.000000000   IBM      NYSE              10         1.10
```

Note: You can also convert a Pandas DataFrame in the Dataset constructor, but only if the DataFrame has no null values:

```
>>> ds = rt.Dataset(df)
```

If we check the Dataset dtypes after conversion, we see only the underlying NumPy data type:

```
>>> ds.dtypes
{'Time': dtype('int64'),
 'Symbol': dtype('int8'),
 'Exchange': dtype('int8'),
 'TradeSize': dtype('int32'),
 'TradePrice': dtype('float64')}
```

To see the Riptable column types, we'll use a Python list comprehension:

```
>>> {(c,ds[c].dtype ,type(ds[c])) for c in ds.keys()}
{('Exchange', dtype('int8'), riptable.rt_categorical.Categorical),
 ('Symbol', dtype('int8'), riptable.rt_categorical.Categorical),
 ('Time', dtype('int64'), riptable.rt_datetime.DateTimeNano),
 ('TradePrice', dtype('float64'), riptable.rt_fastarray.FastArray),
 ('TradeSize', dtype('int32'), riptable.rt_fastarray.FastArray)}
```

Use `to_pandas()` to convert the Dataset back to a Pandas DataFrame:

```
>>> df1 = ds.to_pandas()
>>> df1.dtypes
Time          datetime64[ns, GMT]
Symbol                   category
Exchange                 category
TradeSize                   Int32
TradePrice                float64
dtype: object
```

### Convert Dates to/from Matlab (and Other Libraries)

To use Matlab (or another library) to visualize data by date, convert the Riptable Date objects to an array of integers:

```
>>> dates = rt.Date(ds.Time)
>>> int_dates = dates.yyyymmdd
>>> int_dates.dtype
dtype('int32')
```

MATLAB stores dates as days since 0000-01-01. To convert an array of Matlab datenums to a Riptable `Date` object, first convert the datenums to a FastArray, then to a Date object using the `from_matlab` keyword argument:

```
>>> dates = rt.FA([737061.0, 737062.0, 737063.0, 737064.0, 737065.0])
>>> rt_dates = rt.Date(dates, from_matlab=True)
>>> rt_dates
Date(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04', '2018-01-05'])
```

Next, we review some things to keep in mind to get the best performance out of Riptable: *Performance Considerations*.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.14 Performance Considerations

Riptable uses multi-threaded and vectorized operations to work quickly and efficiently with large amounts of data. However, because memory is a finite resource, it's good to keep some things in mind.

Whenever possible:

- Use universal functions (ufuncs) and ufunc methods. (Ufuncs take array inputs and produce array outputs.)
- To work with a subset of an array, use slicing instead of fancy indexing. Fancy indexing creates a copy of the array; slicing instead gives you a "view" of the array. (This differs from slicing lists in Python, which creates copies.) Be aware that changes to the slice also change the original data. Also note that a slice creates a reference to the original data, and the original data won't be cleared from memory until the reference is also deleted.
- In general, be aware of which operations make copies of data. Use flags to do operations in place when you can.
- Avoid filtering entire Datasets using `ds.filter()`. Use Boolean mask arrays, or use filter keyword arguments in operations on columns.
  - When it makes sense, you can use `ds.filter(inplace=True)` to modify the original Dataset.
- Avoid string operations (creating strings, parsing strings, etc.). When you need to parse a string, use the FastArray string methods and try to do it in as few operations as possible.

- Use Categoricals for string arrays, especially for repeated strings or if you're converting data between Pandas and Riptable.
- Delete datasets you're not using. Though be aware that if you have any references to the Dataset in other objects (for example, a slice or any operation that gives you a "view" of the data), you might not actually free up memory.
- Avoid using `apply()` – it's not a vectorized operation.

### 1.14.1 Multiprocessing

If you need to use multiprocessing with Riptable, this project may be helpful: https://github.com/joblib/loky.

---

Questions or comments about this guide? Email RiptableDocumentation@sig.com.

## 1.15 Riptable Exercises

This workbook is meant to give practical experience with the key ideas & functionality of Riptable.

To complete it, you'll need to consult the *Intro to Riptable*.

Depending on your preferred learning style, you can read through the entire intro guide first or start with the exercises and refer to the guide as needed.

Note that the intro guide has more coverage and detail, so it's well worth reading in full at some point.

If you have any questions or comments, email RiptableDocumentation@sig.com.

[1]:
```
import riptable as rt
import numpy as np
```

### 1.15.1 Introduction to the Riptable Dataset

**Datasets** are the core class of riptable.

They are tables of data, consisting of a series of **columns** of the same length (sometimes referred to as **fields**).

Structurally, they behave like python dictionaries, and can be created directly from one.

We'll familiarize ourselves with Datasets by manually constructing one by generating fake sample data using `np.random.default_rng().choice(...)` or similar.

In real life they will essentially always be generated from world data.

**First, create a python dictionary with two fields of the same length (>1000); one column of stock prices and one of symbols.**

**Make sure the symbols have duplicates, for later aggregation exercises.**

[ ]:
```

```

[ ]:
```

```

**Create a riptable dataset from this, using `rt.Dataset(my_dict)`.**

[ ]:

You can easily append more columns to a dataset.

**Add a new column of integer trade size, using** `my_dset.Size =`.

[ ]:

Columns can be referred with brackets around a string name as well. This is typically used when the column name comes from a variable.

**Add a new column of booleans indicating whether you traded this trade, using** `my_dset['MyTrade'] =`.

[ ]:

**Add a new column of string "Buy" or "Sell" indicating the customer direction.**

[ ]:

Riptable will convert these lists to the riptable **FastArray** container and cast the data to an appropriate numpy datatype.

**View the datatypes with** `my_dset.dtypes`.

[ ]:

**View some sample rows of the dataset using** `.sample()`.

You should use this instead of `.head()` because the initial rows of a dataset are often unrepresentative.

[ ]:

**View distributional stats of the numerical fields of your dataset with** `.describe()`.

You can call this on a single column as well.

[ ]:

### 1.15.2 Manipulating data

You can perform simple operation on riptable columns with normal python syntax. Riptable will do them to the whole column at once, efficiently.

**Create a new column by performing scalar arithmetic on one of your numeric columns.**

[ ]:

[ ]:

As long as the columns are the same size (as is guaranteed if they're in the same dataset) you can perform combining operations the same way.

**Create a new column of total price paid for the trade by multiplying two existing columns together.**

Riptable will automatically upcast types as necessary to preserve information.

[ ]:

[ ]:

There are many built-in functions as well, which you call with either `my_dset.field.function()` or `rt.function(my_dset.field)` syntax.

**Find the unique Symbols in your dataset.**

[ ]:

### 1.15.3 Date/Time

Riptable has three main date/time types: `Date`, `DateTimeNano`, and `TimeSpan`.

**Give each row of your dataset an `rt.Date`.**

**Make sure they're not all different, but still include days from multiple months.**

Note that due to Riptable idiosyncracies you need to generate a list of yyyymmdd strings and pass into the `rt.Date(...)` constructor, not construct Dates individually.

[ ]:

[ ]:

**Give each row a unique(ish) `TimeSpan` as a trade time.**

You can instantiate them using `rt.TimeSpan(hours_var, unit='h')`.

[ ]:

[ ]:

**Create a DateTimeNano of the combined TradeTime + Date by simple addition. Riptable knows how to sum the types.**

Be careful here, by default you'll get a GMT timezone, you can force NYC with `rt.DateTimeNano(..., from_tz='NYC')`.

[ ]:

[ ]:

To reverse this operation and get out separate dates and times from a DateTimeNano, you can call `rt.Date(my_DateTimeNano)` and `my_DateTimeNano.time_since_midnight()`.

**Create a new month name column by using the `.strftime` function.**

[ ]:

[ ]:

**Create another new month column by using the `.start_of_month` attribute.**

This is nice for grouping because it will automatically sort correctly.

[ ]:

[ ]:

### 1.15.4 Sorting

Riptable has two sorts, `sort_copy` (which preserves the original dataset) and `sort_inplace`, which is faster and more memory-efficient if you don't need the original data order.

**Sort your dataset by TradeDateTime.**

This is the natural ordering of a list of trades, so do it in-place.

[ ]:

[ ]:

### 1.15.5 Filtering

Filtering is the principal way to work with a subset of your data in riptable. It is commonly used for looking at a restricted set of trades matching some criterion you care about.

Except in rare instances, though, you should maintain your dataset in its full size, and only apply a filter when performing a final computation.

This will avoid unnecessary data duplication and improve speed & memory usage.

**Construct a filter of only your sales. (A filter is a column of Booleans which is true only for the rows you're interested in.)**

You can combine filters using & or |. Be careful to always wrap expressions in parentheses to avoid an extremely slow call into native python followed by a crash.

Always `(my_dset.field1 > 10) & (my_dset.field2 < 5)`, never `my_dset.field1 > 10 & my_dset.field2 > 5`.

[ ]:

**Compute the total Trade Size, filtered for only your sales.**

For this and many other instances, you can & should pass your filter into the `filter` kwarg of the `.nansum(...)` call.

This allows riptable to perform the filtering during the nansum computation, rather than instantiating a new column and then summing it.

[ ]:

**Count how many times you sold each symbol.**

Here the `.count()` function doesn't accept a `filter` kwarg, so you must fall back to explicitly filtering the `Symbol` field before counting.

Be careful that you only filter down the `Symbol` field, not the entire dataset, otherwise you are wasting a lot of compute.

[ ]:

### 1.15.6 Categoricals

So far, we've been operating on your symbol column as a column of strings.

However, it's far more efficient when you have a large column with many repeats to use a categorical, which assigns each unique value a number, and stores the labels & numbers separately.

This is memory-efficient, and also computationally efficient, as riptable can peform operations on the unique values, then expand out to the full vector appropriately.

**Make a new column of your string column converted to a categorical, using `rt.Cat(column)`.**

[ ]:

**Perform the same filtered count from above, on the categorical.**

The categorical `.count()` admits a `filter` kwarg, which makes it simpler.

[ ]:

Categoricals can be used as groupings. When you call a numeric function on a categorical and pass numeric columns in, riptable knows to do the calculation per-group.

**Compute the total amount of contracts sold by customers in each symbol.**

[ ]:

The `transform=True` kwarg in a categorical operation performs the aggregation, then *transforms* it back up to the original shape of the categorical, giving each row the appropriate value from its group.

**Make a new column which is the average trade price, per symbol.**

[ ]:

**Inspect with `.sample()` to confirm that this value is consistent for rows with matching symbol.**

[ ]:

If you need to perform a custom operation on each categorical, you can pass in a function with `.apply_reduce` (which aggregates) or `.apply_nonreduce` (which is like `transform=True`).

Note that the custom function you pass needs to expect a FastArray, and output a scalar (`apply_reduce`) or same-length FastArray (`apply_nonreduce`).

**Find, for each symbol, the trade size of the second trade occuring in the dataset.**

[ ]:

Sometimes you want to aggregate based on multiple values. In these cases we use multi-key categoricals.

**Use a multi-key categorical to compute the average size per symbol-month pair.**

[ ]:

[ ]:

### 1.15.7 Accumulating

Aggregating over two values for human viewing is often most conveniently done with an accum.

**Use `Accum2` to compute the average size per symbol-month pair.**

[ ]:

Average numbers can be meaningless. It is often better to consider relative percentage instead.

**Use `accum_ratiop` to compute the fraction of total volume done by each symbol-month pair.**

[ ]:

### 1.15.8 Merging

There are two main types of merges.

First is `merge_lookup`. This is used for enriching one (typically large) dataset with information from another (typically small) dataset.

**Create a new dataset with one row per symbol from your dataset, and a second column of who trades each symbol.**

[ ]:

[ ]:

**Enrich the main dataset by putting the correct trader into each row.**

[ ]:

[ ]:

The second type of merge is `merge_asof`, which is used for fuzzy alignment between two datasets, typically by time (though often by other variables).

**Create a new index price dataset with one price per minute, which covers all the Dates in your dataset.**

The index price doesn't need to be reasonable.

Each row should have a DateTimeNano as the datetime.

[ ]:

[ ]:

[ ]:

**Use `merge_asof` to get the most recent Index Price associated with each trade in your main dataset.**

Note both datasets need to be sorted for merge_asof.

The `on` kwarg is the numeric/time field that looks for close matches.

The `by` kwarg is not necessary here, but could constrain the match to a subset if, for example, you had multiple indices and a column of which one each row is associated with.

**Use `direction='backward'` to ensure you're not biasing your data by looking into the future!**

[ ]:

### 1.15.9 Saving/Loading

The native riptable filetype is .sds. It's the fastest way to save & load your data.

**Save out your dataset to file using `rt.save_sds`.**

[ ]:

**Delete your dataset to free up memory using the native python `del my_dset`.**

Note that if there are references to the dataset in other objects you may not actually free up memory.

[ ]:

**Reload your saved dataset from disk with `rt.load_sds`.**

[ ]:

[ ]:

To load from h5 files (a common file type at SIG), use `rt.load_h5(file)`.

To load from csv files, use the slow but robust pandas loader, with `rt.Dataset.from_pandas(pd.read_csv(file))`.

## 1.16 Solutions to Riptable Exercises

This notebook contains the solutions to the *Riptable Exercises*.

Your solutions may be implemented slightly differently, but they should get the same essential results.

If you have any questions or comments, email RiptableDocumentation@sig.com.

[1]:
```
import riptable as rt
import numpy as np
```

### 1.16.1 Introduction to the Riptable Dataset

**Datasets** are the core class of riptable.

They are tables of data, consisting of a series of **columns** of the same length (sometimes referred to as **fields**).

Structurally, they behave like python dictionaries, and can be created directly from one.

We'll familiarize ourselves with Datasets by manually constructing one by generating fake sample data using `np.random.default_rng().choice(...)` or similar.

In real life they will essentially always be generated from world data.

**First, create a python dictionary with two fields of the same length (>1000); one column of stock prices and one of symbols.**

**Make sure the symbols have duplicates, for later aggregation exercises.**

[2]:
```
rng = np.random.default_rng()
dset_length = 5_000
```

[3]:
```
my_dict = {'Price': rng.uniform(0, 1000, dset_length), 'Symbol': rng.choice(['GME', 'AMZN
↪', 'TSLA', 'SPY'], dset_length)}
```

**Create a riptable dataset from this, using `rt.Dataset(my_dict)`.**

[4]:
```
my_dset = rt.Dataset(my_dict)
```

You can easily append more columns to a dataset.

**Add a new column of integer trade size, using `my_dset.Size =`.**

[5]:
```
my_dset.Size = rng.integers(1, 1000, dset_length)
```

Columns can be referred with brackets around a string name as well. This is typically used when the column name comes from a variable.

**Add a new column of booleans indicating whether you traded this trade, using `my_dset['MyTrade'] =`.**

[6]:
```
my_dset['MyTrade'] = rng.choice([True, False], dset_length)
```

**Add a new column of string "Buy" or "Sell" indicating the customer direction.**

[7]:
```
my_dset.CustDirection = rng.choice(['Buy', 'Sell'], dset_length)
```

Riptable will convert these lists to the riptable **FastArray** container and cast the data to an appropriate numpy datatype.

**View the datatypes with `my_dset.dtypes`.**

[8]:
```
my_dset.dtypes
```

[8]:
```
{'Price': dtype('float64'),
 'Symbol': dtype('S4'),
 'Size': dtype('int64'),
 'MyTrade': dtype('bool'),
 'CustDirection': dtype('S4')}
```

**View some sample rows of the dataset using `.sample()`.**

You should use this instead of `.head()` because the initial rows of a dataset are often unrepresentative.

[9]:
```
my_dset.sample()
```

[9]:

| # | Price | Symbol | Size | MyTrade | CustDirection |
|---|---|---|---|---|---|
| 0 | 294.84 | SPY | 18 | False | Buy |
| 1 | 80.15 | TSLA | 939 | True | Buy |
| 2 | 189.83 | AMZN | 919 | True | Buy |
| 3 | 795.83 | SPY | 324 | True | Sell |
| 4 | 111.57 | AMZN | 53 | True | Buy |
| 5 | 695.09 | AMZN | 173 | True | Sell |
| 6 | 109.21 | AMZN | 810 | True | Buy |
| 7 | 83.23 | SPY | 674 | True | Sell |

(continues on next page)

(continued from previous page)

```
8   388.80   AMZN      872     False   Sell
9   744.19   SPY       164     False   Buy

[10 rows x 5 columns] total bytes: 250.0 B
```

**View distributional stats of the numerical fields of your dataset with `.describe()`.**

You can call this on a single column as well.

```
[10]: my_dset.describe()
```

```
[10]:
```

### 1.16.2 Manipulating data

You can perform simple operation on riptable columns with normal python syntax. Riptable will do them to the whole column at once, efficiently.

**Create a new column by performing scalar arithmetic on one of your numeric columns.**

```
[11]: my_dset.SharesOfStock = 100 * my_dset.Size
```

```
[12]: my_dset.sample()
```

```
[12]:
```

As long as the columns are the same size (as is guaranteed if they're in the same dataset) you can perform combining operations the same way.

**Create a new column of total price paid for the trade by multiplying two existing columns together.**

Riptable will automatically upcast types as necessary to preserve information.

```
[13]: my_dset.TotalCash = my_dset.Price * my_dset.Size
```

```
[14]: my_dset.sample()
```

```
[14]:
```

There are many built-in functions as well, which you call with either `my_dset.field.function()` or `rt.function(my_dset.field)` syntax.

**Find the unique Symbols in your dataset.**

```
[15]: my_dset.Symbol.unique()
```

```
[15]: FastArray([b'AMZN', b'GME', b'SPY', b'TSLA'], dtype='|S4')
```

### 1.16.3 Date/Time

Riptable has three main date/time types: `Date`, `DateTimeNano`, and `TimeSpan`.

**Give each row of your dataset an `rt.Date`.**

**Make sure they're not all different, but still include days from multiple months.**

Note that due to Riptable idiosyncracies you need to generate a list of yyyymmdd strings and pass into the `rt.Date(...)` constructor, not construct Dates individually.

[16]:
```
my_dset.Date = rt.Date(rng.choice(rt.Date.range('20220201', '20220430'), dset_length))
```

[17]:
```
my_dset.sample()
```

[17]:

**Give each row a unique(ish) `TimeSpan` as a trade time.**

You can instantiate them using `rt.TimeSpan(hours_var, unit='h')`.

[18]:
```
my_dset.TradeTime = rt.TimeSpan(rng.uniform(9.5, 16, dset_length), unit='h')
```

[19]:
```
my_dset.sample()
```

[19]:

**Create a DateTimeNano of the combined TradeTime + Date by simple addition. Riptable knows how to sum the types.**

Be careful here, by default you'll get a GMT timezone, you can force NYC with `rt.DateTimeNano(..., from_tz='NYC')`.

[20]:
```
my_dset.TradeDateTime = rt.DateTimeNano(my_dset.Date + my_dset.TradeTime, from_tz='NYC')
```

[21]:
```
my_dset.sample()
```

[21]:

To reverse this operation and get out separate dates and times from a DateTimeNano, you can call `rt.Date(my_DateTimeNano)` and `my_DateTimeNano.time_since_midnight()`.

**Create a new month name column by using the `.strftime` function.**

[22]:
```
my_dset.month_name = my_dset.Date.strftime('%b%y')
```

[23]:
```
my_dset.sample()
```

[23]:

**Create another new month column by using the `.start_of_month` attribute.**

This is nice for grouping because it will automatically sort correctly.

[24]:
```
my_dset.month = my_dset.Date.start_of_month
```

[25]:
```
my_dset.sample()
```

[25]:

### 1.16.4 Sorting

Riptable has two sorts, `sort_copy` (which preserves the original dataset) and `sort_inplace`, which is faster and more memory-efficient if you don't need the original data order.

**Sort your dataset by TradeDateTime.**

This is the natural ordering of a list of trades, so do it in-place.

[26]:
```
my_dset = my_dset.sort_inplace('TradeDateTime')
```

[27]:
```
my_dset.sample()
```

[27]:

### 1.16.5 Filtering

Filtering is the principal way to work with a subset of your data in riptable. It is commonly used for looking at a restricted set of trades matching some criterion you care about.

Except in rare instances, though, you should maintain your dataset in its full size, and only apply a filter when performing a final computation.

This will avoid unnecessary data duplication and improve speed & memory usage.

**Construct a filter of only your sales. (A filter is a column of Booleans which is true only for the rows you're interested in.)**

You can combine filters using & or |. Be careful to always wrap expressions in parentheses to avoid an extremely slow call into native python followed by a crash.

Always `(my_dset.field1 > 10) & (my_dset.field2 < 5)`, never `my_dset.field1 > 10 & my_dset.field2 > 5`.

[28]:
```
f_my_sales = my_dset.MyTrade & (my_dset.CustDirection == 'Buy')
```

**Compute the total Trade Size, filtered for only your sales.**

For this and many other instances, you can & should pass your filter into the `filter` kwarg of the `.nansum(...)` call.

This allows riptable to perform the filtering during the nansum computation, rather than instantiating a new column and then summing it.

[29]:
```
my_dset.Size.nansum(filter=f_my_sales)
```

[29]:
```
621241
```

**Count how many times you sold each symbol.**

Here the `.count()` function doesn't accept a `filter` kwarg, so you must fall back to explicitly filtering the `Symbol` field before counting.

Be careful that you only filter down the `Symbol` field, not the entire dataset, otherwise you are wasting a lot of compute.

```
[30]: my_dset.Symbol[f_my_sales].count()
```

```
[30]:
```

### 1.16.6 Categoricals

So far, we've been operating on your symbol column as a column of strings.

However, it's far more efficient when you have a large column with many repeats to use a categorical, which assigns each unique value a number, and stores the labels & numbers separately.

This is memory-efficient, and also computationally efficient, as riptable can peform operations on the unique values, then expand out to the full vector appropriately.

**Make a new column of your string column converted to a categorical, using `rt.Cat(column)`.**

```
[31]: my_dset.Symbol_cat = rt.Cat(my_dset.Symbol)
      my_dset.Symbol_cat
```

```
[31]: Categorical([AMZN, SPY, SPY, SPY, SPY, ..., TSLA, GME, SPY, AMZN, SPY]) Length: 5000
        FastArray([1, 3, 3, 3, 3, ..., 4, 2, 3, 1, 3], dtype=int8) Base Index: 1
        FastArray([b'AMZN', b'GME', b'SPY', b'TSLA'], dtype='|S4') Unique count: 4
```

**Perform the same filtered count from above, on the categorical.**

The categorical `.count()` admits a `filter` kwarg, which makes it simpler.

```
[32]: my_dset.Symbol_cat.count(filter=f_my_sales)
```

```
[32]:
```

Categoricals can be used as groupings. When you call a numeric function on a categorical and pass numeric columns in, riptable knows to do the calculation per-group.

**Compute the total amount of contracts sold by customers in each symbol.**

```
[33]: my_dset.Symbol_cat.sum(my_dset.Size, filter=my_dset.CustDirection == 'Sell')
```

```
[33]:
```

The `transform=True` kwarg in a categorical operation performs the aggregation, then *transforms* it back up to the original shape of the categorical, giving each row the appropriate value from its group.

**Make a new column which is the average trade price, per symbol.**

```
[34]: my_dset.average_trade_price = my_dset.Symbol_cat.mean(my_dset.Price, transform=True)
```

**Inspect with `.sample()` to confirm that this value is consistent for rows with matching symbol.**

```
[35]: my_dset.sample()
```

```
[35]:
```

If you need to perform a custom operation on each categorical, you can pass in a function with `.apply_reduce` (which aggregates) or `.apply_nonreduce` (which is like `transform=True`).

Note that the custom function you pass needs to expect a FastArray, and output a scalar (`apply_reduce`) or same-length FastArray (`apply_nonreduce`).

**Find, for each symbol, the trade size of the second trade occuring in the dataset.**

```
[36]: my_dset.Symbol_cat.apply_reduce(lambda x: x[1], my_dset.Size)
```

```
[36]:
```

Sometimes you want to aggregate based on multiple values. In these cases we use multi-key categoricals.

**Use a multi-key categorical to compute the average size per symbol-month pair.**

```
[37]: my_dset.Symbol_month_cat = rt.Cat([my_dset.Symbol, my_dset.month])
```

```
[38]: my_dset.Symbol_month_cat.nanmean(my_dset.Size).sort_inplace('Symbol')
```

```
[38]:
```

### 1.16.7 Accumulating

Aggregating over two values for human viewing is often most conveniently done with an accum.

**Use `Accum2` to compute the average size per symbol-month pair.**

```
[39]: rt.Accum2(my_dset.Symbol, my_dset.month).nanmean(my_dset.Size)
```

```
[39]:
```

Average numbers can be meaningless. It is often better to consider relative percentage instead.

**Use `accum_ratiop` to compute the fraction of total volume done by each symbol-month pair.**

```
[40]: rt.accum_ratiop(my_dset.Symbol, my_dset.month, my_dset.Size, norm_by='R')
```

```
[40]:
```

### 1.16.8 Merging

There are two main types of merges.

First is `merge_lookup`. This is used for enriching one (typically large) dataset with information from another (typically small) dataset.

**Create a new dataset with one row per symbol from your dataset, and a second column of who trades each symbol.**

```
[41]: symbol_trader = rt.Dataset({'UnderlyingSymbol': ['GME', 'TSLA', 'SPY', 'AMZN'],
                                  'Trader': ['Nate', 'Elon', 'Josh', 'Dan']})
```

```
[42]: symbol_trader
```

```
[42]:
```

**Enrich the main dataset by putting the correct trader into each row.**

```
[43]: my_dset.Trader = my_dset.merge_lookup(symbol_trader, on=('Symbol', 'UnderlyingSymbol'),␣
      ↪columns_left=[])['Trader']
```

[44]:
```
my_dset.sample()
```

[44]:

The second type of merge is `merge_asof`, which is used for fuzzy alignment between two datasets, typically by time (though often by other variables).

**Create a new index price dataset with one price per minute, which covers all the Dates in your dataset.**

The index price doesn't need to be reasonable.

Each row should have a DateTimeNano as the datetime.

[45]:
```
num_minutes = int((my_dset.TradeDateTime.max() - my_dset.TradeDateTime.min()).minutes[0])
start_datetime = rt.Date(my_dset.TradeDateTime.min())
```

[46]:
```
index_price = rt.Dataset({'DateTime': start_datetime + rt.TimeSpan(range(num_minutes),␣
↪unit='m'),
                          'IndexPrice': rng.uniform(3500, 4500, num_minutes)})
```

[47]:
```
index_price.sample()
```

[47]:

**Use `merge_asof` to get the most recent Index Price associated with each trade in your main dataset.**

Note both datasets need to be sorted for merge_asof.

The on kwarg is the numeric/time field that looks for close matches.

The by kwarg is not necessary here, but could constrain the match to a subset if, for example, you had multiple indices and a column of which one each row is associated with.

**Use `direction='backward'` to ensure you're not biasing your data by looking into the future!**

[48]:
```
my_dset.IndexPrice = my_dset.merge_asof(index_price, on=('TradeDateTime', 'DateTime'),␣
↪direction='backward', columns_left=[])['IndexPrice']
```

### 1.16.9 Saving/Loading

The native riptable filetype is .sds. It's the fastest way to save & load your data.

**Save out your dataset to file using `rt.save_sds`.**

[49]:
```
rt.save_sds('my_dset.sds', my_dset)
```

**Delete your dataset to free up memory using the native python `del my_dset`.**

Note that if there are references to the dataset in other objects you may not actually free up memory.

[50]:
```
del my_dset
```

**Reload your saved dataset from disk with `rt.load_sds`.**

[51]:
```
my_dset = rt.load_sds('my_dset.sds')
```

```
[52]: my_dset.sample()
```

```
[52]:
```

To load from h5 files (a common file type at SIG), use `rt.load_h5(file)`.

To load from csv files, use the slow but robust pandas loader, with `rt.Dataset.from_pandas(pd.read_csv(file))`.

## 1.17 Appendix

### 1.17.1 NumPy Methods Optimized by Riptable for FastArrays

When you call the NumPy method, the Riptable version is called instead.

#### Basic NumPy Methods

- copy()
- astype()
- squeeze()

#### NumPy Unary Operators

- absolute()
- negative()
- cos()
- sin()
- exp()
- log()
- log10()
- sqrt()
- ceil()
- trunc()
- floor()
- fabs()
- modf()
- logical_not()

#### NumPy Binary Operators

- add()
- subtract()
- multiply()
- divide()
- floor_divide()
- true_divide()
- greater()
- greater_equal()
- less()
- less_equal()
- equal()
- not_equal()
- logical_and()
- logical_or()
- logical_xor()
- maximum()
- minimum()

#### NumPy Reduce Methods

- nansum()

### 1.17.2 Reducing Functions Supported by Categoricals

The following reducing/aggregating methods are supported by Categoricals. The NaN versions exclude NaN values. When a method is called on a Categorical, it's applied to each group.

| Reducing Function | Description |
|---|---|
| `count()` | Total number of items |
| `count_uniques()` | Total number of each unique item |
| `first()`, `last()` | First item, last item |
| `mean()`, `nanmean()` | Mean, NaN version |
| `median()`, `nanmedian()` | Median, NaN version |
| `min()`, `nanmin()` | Minimum, NaN version |
| `max()`, `nanmax()` | Maximum, NaN version |
| `mode()` | The number that occurs most often (skips NaNs) |
| `nth()` | Take nth value, or a subset if n is a list |
| `prod()` | Product of all items |
| `std()`, `nanstd()` | Standard deviation, NaN version |
| `sum()`, `nansum()` | Sum of all items, NaN version |
| `trimbr()` | Trimmed mean br (skips NaNs) |
| `var()`, `nanvar()` | Variance, NaN version |

### 1.17.3 A Useful Way to Instantiate a Categorical

It can sometimes be useful to instantiate a Categorical with only one category, then fill it in as needed.

For example, let's say we have a Dataset with a column that has a lot of categories, and we want to create a new Categorical column that keeps two of those categories, properly aligned with the rest of the data in the Dataset, and lumps the other categories into a category called 'Other.'

Our Dataset, with a column of many categories:

```
>>> rng = np.random.default_rng(seed=42)
>>> N = 50
>>> ds_buildcat = rt.Dataset({'big_cat': rng.choice(['A', 'B', 'C', 'D', 'E', 'F', 'G',
↪'H', 'I', 'J'], N)})
>>> ds_buildcat
  #   big_cat
---   -------
  0   D
  1   I
  2   A
  3   I
  4   F
  5   B
  6   D
  7   F
  8   D
  9   B
 10   G
 11   G
 12   B
 13   C
 14   C
...   ...
 35   I
 36   J
 37   D
```

(continues on next page)

(continued from previous page)

```
 38   C
 39   J
 40   G
 41   C
 42   G
 43   F
 44   J
 45   C
 46   J
 47   J
 48   B
 49   B
```

We create our 'small' Categorical instantiated with 3s, which fills the column with the 'Other' category:

```
>>> ds_buildcat.small_cat = rt.Cat(rt.full(ds_buildcat.shape[0], 3), categories=['B', 'D
↪', 'Other'])
>>> ds_buildcat.small_cat
>>> ds_buildcat
  #   big_cat   small_cat
---   -------   ---------
  0   D         Other
  1   I         Other
  2   A         Other
  3   I         Other
  4   F         Other
  5   B         Other
  6   D         Other
  7   F         Other
  8   D         Other
  9   B         Other
 10   G         Other
 11   G         Other
 12   B         Other
 13   C         Other
 14   C         Other
...   ...       ...
 35   I         Other
 36   J         Other
 37   D         Other
 38   C         Other
 39   J         Other
 40   G         Other
 41   C         Other
 42   G         Other
 43   F         Other
 44   J         Other
 45   C         Other
 46   J         Other
 47   J         Other
 48   B         Other
 49   B         Other
```

Now we can fill in the aligned 'B' and 'D' categories:

```
>>> ds_buildcat.small_cat[ds_buildcat.big_cat == 'B'] = 'B'
>>> ds_buildcat.small_cat[ds_buildcat.big_cat == 'D'] = 'D'
>>> ds_buildcat
  #   big_cat   small_cat
---   -------   ---------
  0   D         D
  1   I         Other
  2   A         Other
  3   I         Other
  4   F         Other
  5   B         B
  6   D         D
  7   F         Other
  8   D         D
  9   B         B
 10   G         Other
 11   G         Other
 12   B         B
 13   C         Other
 14   C         Other
...   ...       ...
 35   I         Other
 36   J         Other
 37   D         D
 38   C         Other
 39   J         Other
 40   G         Other
 41   C         Other
 42   G         Other
 43   F         Other
 44   J         Other
 45   C         Other
 46   J         Other
 47   J         Other
 48   B         B
 49   B         B
```

## 1.18 More In-Depth Topics

### 1.18.1 Riptable Categoricals User Guide

This guide covers a few topics in more depth than the *Categoricals* section of the *Intro to Riptable* and the API reference docs for the `Categorical` class.

### Riptable Categoricals – Constructing

There are many ways to construct a Categorical – here are some of the more common ones.

On this page:


- *From a list of strings*
- *From a list of non-unique strings and a list of unique categories*
- *From a list of numeric values that index into a list of string categories*
- *From a list of numeric values with no categories provided*
- *From an integer-string dictionary and an array of integer mapping codes*
- *From an IntEnum and an array of integer mapping codes*
- *From a list of arrays or a dictionary: a multi-key Categorical*
- *From a list of float values (Matlab indexing)*
- *From a Pandas Categorical*
- *Using the categories of another Categorical*
- *From an array of values using rt.cut or rt.qcut*


### From a list of strings

A Categorical is typically created from a list of strings (unicode or byte strings). An array of integer mapping codes is created, along with an array of the unique categories:

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([b, a, b, a, c, c, b]) Length: 7
  FastArray([2, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

By default, the integer mapping arrary uses base-1 indexing, with 0 reserved for Filtered values.

### From a list of non-unique strings and a list of unique categories

When categories are provided, they're always held in the same order, so you can preserve a non-lexicographical ordering.

The provided categories don't need to represent all of the provided values – note that the "xsmall" category has no index in the mapping array:

```
>>> rt.Categorical(["small", "small", "medium", "small", "large", "large", "medium"],
...                categories=["xsmall", "small", "medium", "large"])
Categorical([small, small, medium, small, large, large, medium]) Length: 7
  FastArray([2, 2, 3, 2, 4, 4, 3], dtype=int8) Base Index: 1
  FastArray([b'xsmall', b'small', b'medium', b'large'], dtype='|S6') Unique count: 4
```

However, all of the values must appear in the provided categories, otherwise an error is raised:

```
>>> try:
...     rt.Categorical(["small", "small", "medium", "small", "large", "large", "medium",
↪"xlarge"],
...                    categories=["xsmall", "small", "medium", "large"])
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Found values that were not in provided categories: [b'xlarge']
```

### From a list of numeric values that index into a list of string categories

If you have an array of integers that indexes into an array of provided unique categories, the integers are used for the integer mapping array and the categories are held in the order provided.

Because this is a base-1 Categorical, 0 is reserved for the Filtered category, and 1 and 2 are mapped to "small" and "medium", respectively:

```
>>> rt.Categorical([0, 1, 1, 2, 2, 0, 1, 1, 2, 1], categories=["small", "medium", "large
↪"])
Categorical([Filtered, small, small, medium, medium, Filtered, small, small, medium,␣
↪small]) Length: 10
  FastArray([0, 1, 1, 2, 2, 0, 1, 1, 2, 1]) Base Index: 1
  FastArray([b'small', b'medium', b'large'], dtype='|S6') Unique count: 3
```

You can set `base_index=0` to make the 0 not Filtered:

```
>>> rt.Categorical([0, 1, 1, 2, 2, 0, 1, 1, 2, 1], categories=["small", "medium", "large
↪"], base_index=0)
Categorical([small, medium, medium, large, large, small, medium, medium, large, medium])␣
↪Length: 10
  FastArray([0, 1, 1, 2, 2, 0, 1, 1, 2, 1]) Base Index: 0
  FastArray([b'small', b'medium', b'large'], dtype='|S6') Unique count: 3
```

### From a list of numeric values with no categories provided

Note that when no categories are provided, the integer mapping codes start at 1 so that 0 values are not Filtered:

```
>>> rt.Categorical([10, 0, 0, 5, 5, 10, 0, 0, 5, 0])
Categorical([10, 0, 0, 5, 5, 10, 0, 0, 5, 0]) Length: 10
  FastArray([3, 1, 1, 2, 2, 3, 1, 1, 2, 1], dtype=int8) Base Index: 1
  FastArray([ 0,  5, 10]) Unique count: 3
```

### From an integer-string dictionary and an array of integer mapping codes

A dictionary can be used for the `categories` argument to provide a mapping between possibly non-consecutive or non-sequential mapping codes and strings. The dictionary can map integers to strings or string to integers.

Provide a list of integer mapping codes as the first argument to the constructor (notice here that the provided codes have duplication and a missing entry):

```
>>> # Integer to string mapping.
>>> d = {44: "StronglyAgree", 133: "Agree", 75: "Disagree", 1: "StronglyDisagree", 144:
↪"NeitherAgreeNorDisagree" }
>>> codes = [1, 44, 44, 133, 75]
>>> rt.Categorical(codes, categories=d)
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]) Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4

>>> # String to integer mapping.
>>> d = {"StronglyAgree": 44, "Agree": 133, "Disagree": 75, "StronglyDisagree": 1,
↪"NeitherAgreeNorDisagree": 144 }
>>> codes = [1, 44, 44, 133, 75]
>>> c = rt.Categorical(codes, categories=d)
>>> c
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]) Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4
```

Note that Categoricals created from a mapping dictionary have no base index. To see how this affects filtering, see the page on *Filters*.

Also note that groupby results are displayed *not* in the order of the provided mapping dictionary, but the order of the underlying mapping codes array, unless you set `sort_gb=True` at Categorical creation:

```
>>> vals = rt.arange(5)
>>> ds = rt.Dataset({"c": c, "vals": vals})
>>> ds
#   c                 vals
-   ---------------   ----
0   StronglyDisagree     0
1   StronglyAgree        1
2   StronglyAgree        2
3   Agree                3
4   Disagree             5

>>> c.sum(vals)
*c               vals
--------------   ----
StronglyDisagre     0
StronglyAgree       3
Agree               3
Disagree            4
```

See *Sorting and Display Order* for examples.

### From an IntEnum and an array of integer mapping codes

Similar to a dictionary, a Python `IntEnum` class defines a mapping between strings and possibly non-consecutive, non-sequential integer mapping codes. Similarly, the list of the integer codes is supplied as the first argument to the constructor, and the `IntEnum` is provided as the `categories` argument:

```
>>> from enum import IntEnum
>>> class LikertDecision(IntEnum):
...     # A Likert scale with the typical five-level Likert item format.
...     StronglyAgree = 44
...     Agree = 133
...     Disagree = 75
...     StronglyDisagree = 1
...     NeitherAgreeNorDisagree = 144

>>> codes = [1, 44, 44, 133, 75]
>>> c = rt.Categorical(codes, categories=LikertDecision)
>>> c
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]) Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4
```

As with Categoricals created from dictionaries, a Categorical created from an `IntEnum` has no base index. To see how this affects filtering, see the page on *Filters*.

Also similarly, aggregation results are displayed in the order of the mapping codes unless you set `sort_gb=True` at Categorical creation:

```
>>> c.sum(vals)
*key_0            vals
---------------   ----
StronglyDisagre      1
StronglyAgree        5
Agree                4
Disagree             5
```

See *Sorting and Display Order* for examples.

### From a list of arrays or a dictionary: a multi-key Categorical

Multi-key Categoricals let you create and operate on groupings based on multiple associated categories. The associated keys form a group:

```
>>> strs = rt.FastArray(["a", "b", "b", "a", "b", "a"])
>>> ints = rt.FastArray([2, 1, 1, 2, 1, 3])
>>> c = rt.Categorical([strs, ints])  # Create with a list of arrays.
>>> c
Categorical([(a, 2), (b, 1), (b, 1), (a, 2), (b, 1), (a, 3)]) Length: 6
  FastArray([1, 2, 2, 1, 2, 3], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'a'], dtype='|S1'), 'key_1': FastArray([2, 1, 3])}␣
↪Unique count: 3
```

(continues on next page)

(continued from previous page)

```
>>> c.count()
*key_0   *key_1   Count
------   ------   -----
a             2       2
b             1       3
a             3       1

>>> c2 = rt.Categorical({"Key1": strs, "Key2": ints})  # Create with a dict of key-value␣
↪pairs.
>>> c2
Categorical([(a, 2), (b, 1), (b, 1), (a, 2), (b, 1), (a, 3)]) Length: 6
  FastArray([1, 2, 2, 1, 2, 3], dtype=int8) Base Index: 1
  {'Key1': FastArray([b'a', b'b', b'a'], dtype='|S1'), 'Key2': FastArray([2, 1, 3])}␣
↪Unique count: 3

>>> c2.count()
*Key1   *Key2   Count
-----   -----   -----
a           2       2
b           1       3
a           3       1
```

### From a list of float values (Matlab indexing)

To convert a Matlab Categorical that uses float indices, set `from_matlab=True`. The indices are converted to an integer type, and any 0.0 values are Filtered:

```
>>> rt.Categorical([0.0, 1.0, 2.0, 3.0, 1.0, 1.0], categories=["b", "c", "a"], from_
↪matlab=True)
Categorical([Filtered, b, c, a, b, b]) Length: 6
  FastArray([0, 1, 2, 3, 1, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'c', b'a'], dtype='|S1') Unique count: 3
```

### From a Pandas Categorical

Categoricals created from Pandas Categoricals must have a base-1 index to preserve invalid values. The invalid values become Filtered:

```
>>> import pandas as pd
>>> pdc = pd.Categorical(["a", "a", "z", "b", "c"], ["c", "b", "a"])
>>> pdc
['a', 'a', NaN, 'b', 'c']
Categories (3, object): ['c', 'b', 'a']

>>> rt.Categorical(pdc)
Categorical([a, a, Filtered, b, c]) Length: 5
  FastArray([3, 3, 0, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'c', b'b', b'a'], dtype='|S1') Unique count: 3
```

### Using the categories of another Categorical

```
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"], categories=["c", "b", "a"])
>>> c.category_array
FastArray([b'c', b'b', b'a'], dtype='|S1')
```

```
>>> c2 = rt.Categorical(["b", "c", "c", "b"], categories=c.category_array)
>>> c2
Categorical([b, c, c, b]) Length: 4
  FastArray([2, 1, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'c', b'b', b'a'], dtype='|S1') Unique count: 3
```

Note that the `c2.category_array` has the same values as `c.category_array`, but it is a copy of and not a reference to the latter:

```
>>> c.category_array is c2.category_array
False
```

To create a `Categorical` that references the same categorical array, it must be constructed with indices and categories:

```
>>> c2 = rt.Categorical([1, 2, 1, 2], categories=c.category_array)
>>> c.category_array is c2.category_array
True
```

### From an array of values using `rt.cut` or `rt.qcut`

Both `cut` and `qcut` partition values into discrete bins that form the categories of a Categorical.

With `cut`, values can be parititioned into a specified number of equal-width bins or bins bounded by specified endpoints. Here, they're parititioned into 3 equal-width bins:

```
>>> rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]), bins=3)
Categorical([1.0->3.0, 5.0->7.0, 3.0->5.0, 3.0->5.0, 5.0->7.0, 1.0->3.0]) Length: 6
  FastArray([1, 3, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'1.0->3.0', b'3.0->5.0', b'5.0->7.0'], dtype='|S8') Unique count: 3
```

Here the bins are bounded by specified endpoints. Values that fall outside of the bins are put in the Filtered category:

```
rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]), bins=[1, 3, 6])
Categorical([1.0->3.0, Filtered, 3.0->6.0, 3.0->6.0, 3.0->6.0, 1.0->3.0]) Length: 6
  FastArray([1, 0, 2, 2, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'1.0->3.0', b'3.0->6.0'], dtype='|S8') Unique count: 2
```

The `qcut` function lets you partition values into bins based on sample quantiles:

```
>>> rt.qcut(rt.arange(5), q=4)
Categorical([0.0->1.0, 0.0->1.0, 1.0->2.0, 2.0->3.0, 3.0->4.0]) Length: 5
  FastArray([2, 2, 3, 4, 5], dtype=int8) Base Index: 1
  FastArray([b'Clipped', b'0.0->1.0', b'1.0->2.0', b'2.0->3.0', b'3.0->4.0'], dtype='|S8
↪') Unique count: 5
```

The 'Clipped' bin is created to hold any out-of-bounds values, such as when a value falls outside of a specified range. A 'Clipped' bin is different from a 'Filtered' bin:

```
>>> rt.qcut(rt.arange(5), q=[.1, .25, .5, .75, 1.], filter=[True, False, True, True,␣
↪True])
Categorical([Clipped, Filtered, 1.5->2.5, 2.5->3.25, 3.25->4.0]) Length: 5
  FastArray([1, 0, 3, 4, 5], dtype=int8) Base Index: 1
  FastArray([b'Clipped', b'0.6->1.5', b'1.5->2.5', b'2.5->3.25', b'3.25->4.0'], dtype=
↪'|S9') Unique count: 5
```

### Riptable Categoricals – Accessing Parts of the Categorical

Use Categorical methods and properties to access the stored data.

Get the array of Categorical values with *expand_array*. Note that because the expansion constructs the complete list of values from the list of unique categories, it is an expensive operation:

```
>>> c = rt.Categorical(["b", "a", "b", "c", "a", "c", "b"])
>>> c
Categorical([b, a, b, c, a, c, b]) Length: 7
  FastArray([2, 1, 2, 3, 1, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c.expand_array
FastArray([b'b', b'a', b'b', b'c', b'a', b'c', b'b'], dtype='|S8')


>>> c2 = rt.Categorical([10, 0, 0, 5, 5, 10, 0, 0, 5, 0])
>>> c2
Categorical([10, 0, 0, 5, 5, 10, 0, 0, 5, 0]) Length: 10
  FastArray([3, 1, 1, 2, 2, 3, 1, 1, 2, 1], dtype=int8) Base Index: 1
  FastArray([ 0,  5, 10]) Unique count: 3

>>> c2.expand_array
FastArray([10,  0,  0,  5,  5, 10,  0,  0,  5,  0])
```

Note that in this base-1 Categorical with an integer mapping array and unique categories provided, 0 is mapped to Filtered, 1 is mapped to “b”, and 2 is mapped to “a”; there is no 3 to be mapped to “c”, so it doesn’t appear in the expanded array.

```
>>> c3 = rt.Categorical([0, 1, 1, 2, 2, 0, 1, 1, 2, 1], categories=["b", "a", "c"])
>>> c3
Categorical([Filtered, b, b, a, a, Filtered, b, b, a, b]) Length: 10
  FastArray([0, 1, 1, 2, 2, 0, 1, 1, 2, 1]) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3
```

```
>>> c3.expand_array
FastArray([b'Filtered', b'b', b'b', b'a', b'a', b'Filtered', b'b', b'b', b'a', b'b'],␣
↪dtype='|S8')
```

Get the integer mapping array with *_fa*:

```
>>> c._fa
FastArray([2, 1, 2, 3, 1, 3, 2], dtype=int8)
```

(continues on next page)

(continued from previous page)

```
>>> c2._fa
FastArray([3, 1, 1, 2, 2, 3, 1, 1, 2, 1], dtype=int8)

>>> c3._fa
FastArray([0, 1, 1, 2, 2, 0, 1, 1, 2, 1])
```

Get the array of unique categories with *category_array*:

```
>>> c.category_array
FastArray([b'a', b'b', b'c'], dtype='|S1')

>>> c2.category_array
FastArray([ 0,  5, 10])

>>> c3.category_array
FastArray([b'b', b'a', b'c'], dtype='|S1')
```

Note that if you want to use *_fa* to index into *category_array*, you'll need to subtract 1:

```
>>> c.category_array[c._fa[0]-1]
b'b'
```

For multi-key Categoricals, use *category_dict* to get a dictionary of the two category arrays:

```
>>> strs = rt.FastArray(["a", "b", "b", "a", "b", "a"])
>>> ints = rt.FastArray([2, 1, 1, 2, 1, 3])
>>> c = rt.Categorical([strs, ints])
>>> c
Categorical([(a, 2), (b, 1), (b, 1), (a, 2), (b, 1), (a, 3)]) Length: 6
  FastArray([1, 2, 2, 1, 2, 3], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'a'], dtype='|S1'), 'key_1': FastArray([2, 1, 3])}␣
↪Unique count: 3

>>> c.category_dict
{'key_0': FastArray([b'a', b'b', b'a'], dtype='|S1'),
'key_1': FastArray([2, 1, 3])}
```

Use *category_mapping* to get the mapping dictionary from a Categorical created with an IntEnum or mapping dictionary:

```
>>> d = {"StronglyAgree": 44, "Agree": 133, "Disagree": 75, "StronglyDisagree": 1,
↪"NeitherAgreeNorDisagree": 144 }
>>> codes = [1, 44, 44, 133, 75]
>>> c = rt.Categorical(codes, categories=d)
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]) Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4
>>> c.category_mapping
{44: 'StronglyAgree',
 133: 'Agree',
 75: 'Disagree',
```

(continues on next page)

(continued from previous page)

```
 1: 'StronglyDisagree',
 144: 'NeitherAgreeNorDisagree'}
```

### Riptable Categoricals – Indexing

Bracket indexing traverses the FastArray of indices/codes and returns the corresponding category.

When a Categorical is indexed with a single integer, the corresponding category is returned as a unicode string.

When multiple integers or a boolean array are used, a copy of the Categorical is returned that has the same categories as the original Categorical but with an index/code array limited to the selected elements. If you modify the returned subset, it won't affect the original Categorical.

When a slice is used, the returned Categorical is a view, not a copy. If you modify the view, the original Categorical is also modified.

To set a value to a new value, the new value must be already represented in the existing categories array.

The following examples use this Categorical:

```
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([a, a, b, a, c, c, b]) Length: 7
  FastArray([1, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

### Single integer

Use bracket indexing to get a single value:

```
>>> c[0]
'a'

>>> c[1]
'a'

>>> c[2]
'b'
```

You can also index from the end of the array with negative indices:

```
>>> c[-1]
'b'

>>> c[-2]
'c'
```

Set a value:

```
>>> c[0] = "c"
>>> c
Categorical([c, a, b, a, c, c, b]) Length: 7
```

(continues on next page)

(continued from previous page)

```
  FastArray([3, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

The value must be already represented in the existing categories array (adding categories using `auto_add_categories` isn't working correctly at the time of this writing):

```
>>> try:
...     c[0] = "d"
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Cannot automatically add categories [b'd'] while auto_add_categories is
set to False. Set flag to True in Categorical init.
```

### Multiple integers

```
>>> c
Categorical([c, a, b, a, c, c, b]) Length: 7
  FastArray([3, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Pass a list of indices (a fancy index, which also specifies ordering). The returned Categorical is a copy of the original Categorical:

```
>>> c[[0, 2]]
Categorical([c, b]) Length: 2
  FastArray([3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c[[2, 0]]
Categorical([b, c]) Length: 2
  FastArray([2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c[[-1, 1]]
Categorical([b, a]) Length: 2
  FastArray([2, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Or pass an array:

```
>>> c[rt.arange(1, 3)]  # Indices 1 and 2.
Categorical([a, b]) Length: 2
  FastArray([1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Set values:

```
>>> c[[0, 2]] = "a"
>>> c
Categorical([a, a, a, a, c, c, b]) Length: 7
  FastArray([1, 1, 1, 1, 3, 3, 2], dtype=int8) Base Index: 1
```

(continues on next page)

(continued from previous page)

```
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c[rt.arange(1, 3)] = "b"
>>> c
Categorical([a, b, b, a, c, c, b]) Length: 7
  FastArray([1, 2, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

### Boolean mask array

```
>>> c
Categorical([a, b, b, a, c, c, b]) Length: 7
  FastArray([1, 2, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

The returned Categorical is a copy of the original Categorical:

```
>>> mask = rt.FA([False, True, True, True, True, True, False])
>>> c[mask]
Categorical([a, b, a, c, c]) Length: 5
  FastArray([1, 2, 1, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Set values:

```
>>> c[mask] = "c"
>>> c
Categorical([a, c, c, c, c, c, b]) Length: 7
  FastArray([1, 3, 3, 3, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

### Slice

```
>>> c
Categorical([a, c, c, c, c, c, b]) Length: 7
  FastArray([1, 3, 3, 3, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

The returned Categorical is a view of the original Categorical. Any changes to the view also modify the original (see below):

```
>>> c[:3]  # Indices 0-2.
Categorical([a, c, c]) Length: 3
  FastArray([1, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c[1:6]  # Indices 1-5.
Categorical([c, c, c, c, c]) Length: 5
```

(continues on next page)

(continued from previous page)

```
  FastArray([3, 3, 3, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Set values:

```
>>> c[1:6] = "a"
Categorical([a, a, a, a, a, a, b]) Length: 7
  FastArray([1, 1, 1, 1, 1, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Slicing returns a view, not a copy. So if you set values in the returned subset, the original Categorical is modified:

```
>>> c2 = c[1:6]
>>> c2
Categorical([a, a, a, a, a]) Length: 5
  FastArray([1, 1, 1, 1, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c2[1:5] = "c"  # Modify the returned view.
>>> c2
Categorical([a, c, c, c, c]) Length: 5
  FastArray([1, 3, 3, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c  # The original is also modified.
Categorical([a, a, c, c, c, c, b]) Length: 7
  FastArray([1, 1, 3, 3, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

### Riptable Categoricals – Filtering

Categoricals that use base-1 indexing can be filtered when they're created or anytime afterwards. Filters can also be applied on a one-off basis at the time of an operation.

Values or entire categories can be filtered. Filtered items are mapped to 0 in the integer mapping array and omitted from operations.

On this page:


- *Filtering at Categorical creation*
- *Filtering after Categorical creation*
- *Filter an operation on a Categorical*
- *Set a name for filtered values*
- *See the name set for filtered values*

### Filtering at Categorical creation

Provide a `filter` argument to filter values at Categorical creation. Filtered values are omitted from all operations on the Categorical.

Notes:

- Only base-1 indexing is supported – the 0 is reserved for Filtered values.
- You can't use a dictionary or `IntEnum` to create a Categorical with a filter.

You can filter out certain values or an entire category:

```
>>> f = rt.FA([True, True, False, True, True, True, True])  # The mask must be an array,␣
↪not a list.
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"], filter=f)  # One "b" value␣
↪is filtered.
>>> c
Categorical([a, a, Filtered, a, c, c, b]) Length: 7
  FastArray([1, 1, 0, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c.count()
*key_0   Count
------   -----
a            3
b            1
c            2
```

In the example below, an entire category is filtered. If the Categorical is constructed from values without provided categories, categories that are entirely filtered out do not appear in the array of unique categories or in the results of operations:

```
>>> vals = rt.FA(["a", "a", "b", "a", "c", "c", "b"])
>>> f = (vals != "b")  # Filter out all "b" values.
>>> c = rt.Categorical(vals, filter=f)
>>> c
Categorical([a, a, Filtered, a, c, c, Filtered]) Length: 7
  FastArray([1, 1, 0, 1, 2, 2, 0], dtype=int8) Base Index: 1
  FastArray([b'a', b'c'], dtype='|S1') Unique count: 2

>>> c.count()
*key_0   Count
------   -----
a            3
c            2
```

If categories are provided, entirely filtered-out categories do appear in the array of unique categories and the results of operations:

```
>>> c = rt.Categorical(vals, categories=["a", "b", "c"], filter=f)
>>> c
Categorical([a, a, Filtered, a, c, c, Filtered]) Length: 7
  FastArray([1, 1, 0, 1, 3, 3, 0], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

(continues on next page)

(continued from previous page)

```
>>> c.count()
*key_0   Count
------   -----
a            3
b            0
c            2
```

Multi-key Categoricals can also be filtered at creation.

```
>>> f = rt.FA([False, False, True, False, True, True])
>>> vals1 = rt.FastArray(["a", "b", "b", "a", "b", "a"])
>>> vals2 = rt.FastArray([2, 1, 1, 3, 2, 1])
>>> rt.Categorical([vals1, vals2], filter=f)
Categorical([Filtered, Filtered, (b, 1), Filtered, (b, 2), (a, 1)]) Length: 6
  FastArray([0, 0, 1, 0, 2, 3], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'b', b'b', b'a'], dtype='|S1'), 'key_1': FastArray([1, 2, 1])}␣
↪Unique count: 3
```

Categoricals using base-0 indexing can't be filtered at creation:

```
>>> f = rt.FA([False, False, True, False, True, True, False])
>>> try:
...     rt.Categorical([0, 1, 1, 2, 2, 0, 1], base_index=0, filter=f)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Filtering is not allowed for base index 0. Use base-1 indexing instead.
```

Categoricals created using a dictionary or `IntEnum` can't be filtered by passing a `filter` argument at creation, but a Filtered category can be included by by using the integer sentinel value as the Filtered mapping code. They can also be filtered after creation using `set_valid()`.

Using the `filter` argument gets an error:

```
>>> f = rt.FA([True, False, False, False, False])
>>> d = {44: "StronglyAgree", 133: "Agree", 75: "Disagree", 1: "StronglyDisagree", 144:
↪"NeitherAgreeNorDisagree" }
>>> codes = [1, 44, 144, 133, 75]
>>> try:
...     rt.Categorical(codes, categories=d, filter=f)
... except TypeError as e:
...     print("TypeError:", e)
TypeError: Grouping from enum does not support pre-filtering.
```

However, you can include a Filtered category by using the integer sentinel value in your mapping:

```
>>> d = {-2147483648: "Filtered", 44: "StronglyAgree", 133: "Agree", 75: "Disagree", 1:
↪"StronglyDisagree", 144: "NeitherAgreeNorDisagree" }
>>> codes = [-2147483648, 44, 144, 133, 75]
>>> c = rt.Categorical(codes, categories=d)
>>> c
Categorical([Filtered, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree]) Length:␣
↪5
```

(continues on next page)

(continued from previous page)

```
  FastArray([-2147483648,          44,         144,         133,          75]) Base␣
↪Index: None
  {-2147483648:'Filtered', 44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:
↪'StronglyDisagree', 144:'NeitherAgreeNorDisagree'} Unique count: 5

>>> from enum import IntEnum
>>> class LikertDecision(IntEnum):
...     # A Likert scale with the typical five-level Likert item format.
...     Filtered = -2147483648
...     StronglyAgree = 44
...     Agree = 133
...     Disagree = 75
...     StronglyDisagree = 1
...     NeitherAgreeNorDisagree = 144
>>> codes = [-2147483648, 1, 44, 144, 133, 75]
>>> rt.Categorical(codes, categories=LikertDecision)
Categorical([Filtered, StronglyDisagree, StronglyAgree, NeitherAgreeNorDisagree, Agree,␣
↪Disagree]) Length: 6
  FastArray([-2147483648,           1,          44,         144,         133,         ␣
↪75]) Base Index: None
  {-2147483648:'Filtered', 44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:
↪'StronglyDisagree', 144:'NeitherAgreeNorDisagree'} Unique count: 6
```

You can also filter an existing category after creation using *set_valid* (see below).

### Filtering after Categorical creation

Calling *set_valid* on a Categorical returns a filtered copy of the Categorical.

```
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([a, a, b, a, c, c, b]) Length: 7
  FastArray([1, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

```
>>> f = rt.FA([True, True, False, True, True, True, True])  # Filter out 1 "b" value.
>>> c.set_valid(f)
Categorical([a, a, Filtered, a, c, c, b]) Length: 7
  FastArray([1, 1, 0, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

The original Categorical isn't modified:

```
>>> c
Categorical([a, a, b, a, c, c, b]) Length: 7
  FastArray([1, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Entirely filtered-out bins are removed from the array of unique categories:

```
>>> vals = rt.FA(["a", "a", "b", "a", "c", "c", "b"])
>>> f = (vals != "b")  # Filter out all "b" values.
>>> c.set_valid(f)
Categorical([a, a, Filtered, a, c, c, Filtered]) Length: 7
  FastArray([1, 1, 0, 1, 2, 2, 0], dtype=int8) Base Index: 1
  FastArray([b'a', b'c'], dtype='|S1') Unique count: 2
```

A Categorical created with a mapping dictionary or `IntEnum` can be filtered after creation. Filtered values are mapped to the integer sentinel value:

```
>>> d = {44: "StronglyAgree", 133: "Agree", 75: "Disagree", 1: "StronglyDisagree", 144:
↪"NeitherAgreeNorDisagree" }
>>> codes = [1, 44, 144, 133, 75]
>>> c = rt.Categorical(codes, categories=d)
>>> c
Categorical([StronglyDisagree, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree]),␣
↪Length: 5
  FastArray([  1,  44, 144, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 5
>>> f = rt.FA([False, True, True, True, True])  # Filter out 1: "StronglyDisagree".
>>> c.set_valid(f)
Categorical([Filtered, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree]) Length:␣
↪5
  FastArray([-2147483648,          44,         144,         133,          75]) Base␣
↪Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 144:'NeitherAgreeNorDisagree', -
↪2147483648:'Filtered'} Unique count: 5

>>> class LikertDecision(IntEnum):
...     # A Likert scale with the typical five-level Likert item format.
...     StronglyAgree = 44
...     Agree = 133
...     Disagree = 75
...     StronglyDisagree = 1
...     NeitherAgreeNorDisagree = 144
>>> codes = [1, 44, 144, 133, 75]
>>> c = rt.Categorical(codes, categories=LikertDecision)
>>> c
Categorical([StronglyDisagree, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree]),␣
↪Length: 5
  FastArray([  1,  44, 144, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 5
>>> f = rt.FA([False, True, True, True, True])  # Filter out 1: "StronglyDisagree".
>>> c.set_valid(f)
Categorical([Filtered, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree]) Length:␣
↪5
  FastArray([-2147483648,          44,         144,         133,          75]) Base␣
↪Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 144:'NeitherAgreeNorDisagree', -
↪2147483648:'Filtered'} Unique count: 5
```

Filtering can be useful to re-index a Categorical so only its occurring uniques are shown:

```
>>> f = (vals != "b")
>>> c2 = c[f]
>>> c2
Categorical([a, a, a, c, c]) Length: 5
  FastArray([1, 1, 1, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c2.sum(rt.arange(5))
*key_0   col_0
------   -----
a            3
b            0
c            7

>>> # Use set_valid to create a re-indexed Categorical:.
>>> c3 = c2.set_valid()
>>> c3
Categorical([a, a, a, c, c]) Length: 5
  FastArray([1, 1, 1, 2, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'c'], dtype='|S1') Unique count: 2

>>> c3.count()
*key_0   Count
------   -----
a            3
c            2

>>> c3.sum(rt.arange(5))
*key_0   col_0
------   -----
a            3
c            7
```

### Filter an operation on a Categorical

To filter one operation (such as a sum), use the `filter` argument for the operation. Filtered results are omitted, but any entirely filtered categories still appear in the results:

```
>>> # Put the Categorical in a Dataset to better see
>>> # the associated values used in the operation.
>>> ds = rt.Dataset()
>>> vals = rt.FA(["a", "a", "b", "a", "c", "c", "b"])
>>> c = rt.Categorical(vals)
>>> ds.cats = c
>>> ds.ints = rt.arange(7)
>>> ds
#   cats   ints
-   ----   ----
0   a         0
1   a         1
2   b         2
```

(continues on next page)

(continued from previous page)

```
3   a         3
4   c         4
5   c         5
6   b         6

>>> f = rt.FA([True, True, False, True, True, True, True])  # One "b" value is filtered.
>>> c.sum(ints, filter=f)
*key_0   ints
------   ----
a           4
b           6
c           9

>>> f = (cats != "b")  # Filter out all "b" values.
>>> c.sum(ints, filter=f)
*key_0   ints
------   ----
a           4
b           0
c           9
```

The Categorical doesn't retain the filter:

```
>>> c
Categorical([a, a, b, a, c, c, b]) Length: 7
  FastArray([1, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

To see the results of the operation applied to all Filtered values (irrespective of their group), use the `showfilter` argument:

```
>>> # A "b" value (2) and a "c" value (5) are filtered.
>>> f = rt.FA([True, True, False, True, True, False, True])
>>> c.sum(ints, filter=f, showfilter=True)
*key_0     ints
--------   ----
Filtered      7
a             4
b             6
c             4

>>> f = (cats != "a")  # Filter out all "a" values.
>>> c.sum(ints, filter=f, showfilter=True)
*key_0     ints
--------   ----
Filtered      4
a             0
b             8
c             9
```

### Set a name for filtered values

You can set a string for displaying filtered values using *filtered_set_name*:

```
>>> vals = rt.FA(["a", "a", "b", "a", "c", "c", "b"])
>>> f = (vals != "b")
>>> c = rt.Categorical(vals, filter=f)
>>> c.filtered_set_name("FNAME")
>>> c
Categorical([a, a, FNAME, a, c, c, FNAME]) Length: 7
  FastArray([1, 1, 0, 1, 2, 2, 0], dtype=int8) Base Index: 1
  FastArray([b'a', b'c'], dtype='|S1') Unique count: 2
```

### See the name set for filtered values

To see the string used when filtered values are displayed, use the *filtered_name* property:

```
>>> c.filtered_name
'FNAME'
```

### Riptable Categoricals – Base Index

### Categoricals default to base-1 indexing

The 0 index is reserved for Filtered values and categories:

```
>>> vals = rt.FA(["b", "a", "a", "c", "a", "b"])
>>> f = rt.FA([False, True, True, True, True, True])
>>> rt.Categorical(vals, filter=f)
Categorical([Filtered, a, a, c, a, b]) Length: 6
  FastArray([0, 1, 1, 3, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Note that “b” doesn’t appear in the array of unique categories because it’s entirely filtered out:

```
>>> f = (vals != "b")
>>> rt.Categorical(vals, filter=f)
Categorical([Filtered, a, a, c, a, Filtered]) Length: 6
  FastArray([0, 1, 1, 2, 1, 0], dtype=int8) Base Index: 1
  FastArray([b'a', b'c'], dtype='|S1') Unique count: 2
```

Provided indices are assumed to be base-1, with the 0 index indicating invalid values:

```
>>> cats = rt.FA(["a", "b", "c"])
>>> rt.Categorical([1, 0, 0, 2, 0, 1], categories=cats])
Categorical([a, Filtered, Filtered, b, Filtered, a]) Length: 6
  FastArray([1, 0, 0, 2, 0, 1]) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Matlab also reserves 0 for invalid values, so a Categorical created with `from_matlab=True` must have a base-1 index:

```
>>> rt.Categorical([0.0, 1.0, 1.0, 3.0, 1.0, 2.0], categories=cats, from_matlab=True)
Categorical([Filtered, a, a, c, a, b]) Length: 6
  FastArray([0, 1, 1, 3, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Same with a Categorical converted from Pandas:

```
>>> import pandas as pd
>>> pdc = pd.Categorical(["a", "a", "z", "b", "c"], categories=cats)
>>> pdc
['a', 'a', NaN, 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
>>> rt.Categorical(pdc)
Categorical([a, a, Filtered, b, c]) Length: 5
  FastArray([1, 1, 0, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Multi-key Categorical:

```
>>> f = rt.FA([False, False, True, False, True, True])
>>> rt.Categorical([rt.FA(["b", "a", "a", "c", "a", "b"]), rt.arange(6)], filter=f)
Categorical([Filtered, Filtered, (a, 2), Filtered, (a, 4), (b, 5)]) Length: 6
  FastArray([0, 0, 1, 0, 2, 3], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'a', b'b'], dtype='|S1'), 'key_1': FastArray([2, 4, 5])}␣
↪Unique count: 3
```

### Categoricals with no base index

Categoricals created from a mapping dictionary or `IntEnum` have no base index:

```
>>> # Integer to string mapping.
>>> d = {44: "StronglyAgree", 133: "Agree", 75: "Disagree", 1: "StronglyDisagree", 144:
↪"NeitherAgreeNorDisagree" }
>>> codes = [1, 44, 144, 133, 75]
>>> rt.Categorical(codes, categories=d)
Categorical([StronglyDisagree, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree])␣
↪Length: 5
  FastArray([  1,  44, 144, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 5

>>> # String to integer mapping.
>>> d = {"StronglyAgree": 44, "Agree": 133, "Disagree": 75, "StronglyDisagree": 1,
↪"NeitherAgreeNorDisagree": 144 }
>>> codes = [1, 44, 144, 133, 75]
>>> rt.Categorical(codes, categories=d)
Categorical([StronglyDisagree, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree])␣
↪Length: 5
  FastArray([  1,  44, 144, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 5
```

(continues on next page)

(continued from previous page)

```
>>> from enum import IntEnum
>>> class LikertDecision(IntEnum):
...     # A Likert scale with the typical five-level Likert item format.
...     StronglyAgree = 44
...     Agree = 133
...     Disagree = 75
...     StronglyDisagree = 1
...     NeitherAgreeNorDisagree = 144
>>> codes = [1, 44, 144, 133, 75]
>>> rt.Categorical(codes, categories=LikertDecision)
Categorical([StronglyDisagree, StronglyAgree, NeitherAgreeNorDisagree, Agree, Disagree]),␣
↪Length: 5
  FastArray([  1,  44, 144, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 5
```

Note: Categoricals that have no base index can't be filtered by passing a `filter` argument at creation, but they can be filtered by using the integer sentinel value as the Filtered mapping code. They can also be filtered after creation using `set_valid`. For examples, see *Filters*.

### Some Categoricals can opt for base-0 indexing

Base-0 can be used if:

- A mapping dictionary isn't used. A `Categorical` created from a mapping dictionary does not have a base index.
- A filter isn't used at creation.
- A Matlab or Pandas Categorical isn't being converted. These both reserve 0 for invalid values.

```
>>> rt.Categorical(["b", "a", "a", "c", "a", "b"], base_index=0)
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 0, 0, 2, 0, 1]) Base Index: 0
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> rt.Categorical(["b", "a", "a", "c", "a", "b"], categories=cats, base_index=0)
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 0, 0, 2, 0, 1], dtype=int8) Base Index: 0
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> rt.Categorical([1, 0, 0, 2, 0, 1], categories=cats, base_index=0)
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 0, 0, 2, 0, 1]) Base Index: 0
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

### Filtering at Categorical creation prevents base-0 indexing

```
>>> f = rt.FA([True, True, False, True, True, True])
```

```
>>> try:
...     rt.Categorical(["b", "a", "a", "c", "a", "b"], filter=f, base_index=0)
... except ValueError as e:
...    print("ValueError:", e)
ValueError: Filtering is not allowed for base index 0. Use base-1 indexing instead.
```

```
>>> try:
...     rt.Categorical(["b", "a", "a", "c", "a", "b"], categories=cats, filter=f, base_
↪index=0)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Filtering is not allowed for base index 0. Use base-1 indexing instead.
```

```
>>> try:
...     rt.Categorical([1, 0, 0, 2, 0, 1], categories=cats, filter=f, base_index=0)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Filtering is not allowed for base index 0. Use base-1 indexing instead.
```

### Categoricals created from Matlab or Pandas Categoricals can't use base-0 indexing

Categoricals created from Matlab Categoricals must use a base-1 index in order to preserve invalid values (which are also indexed as 0 in Matlab):

```
>>> import pandas as pd
>>> pdc = pd.Categorical(["b", "a", "a", "c", "a", "b"])
>>> try:
...     rt.Categorical(pdc, base_index=0)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: To preserve invalids, pandas categoricals must be 1-based.

>>> try:
...     rt.Categorical([2.0, 1.0, 1.0, 3.0, 1.0, 2.0], categories=cats, from_matlab=True,
↪ base_index=0)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Categoricals from matlab must have a base index of 1, got 0.
```

### Riptable Categoricals – Sorting and Display Order

Whether a Categorical's categories are lexicographically sorted or considered to be "ordered" as specified at creation depends on two parameters: `ordered` and `lex`.

- `ordered` controls whether categories are sorted lexicographically before they are mapped to integers.
- `lex` controls whether hashing- or sorting-based logic is used to find unique values in the input array. Note that if `lex=True`, the categories become sorted even if `ordered=False`.

Additionally, the results of groupby operations can be displayed in sorted order with the `sort_gb` parameter.

The way these parameters interact depends on whether categories are provided when the Categorical is created.

**If categories are not provided,** then if `ordered=True` (the default) or `lex=True` they are sorted in the Categorical and in groupby results, even if `sort_gb=False`. If `ordered=False` and `lex=False`, the categories are held in the order of first appearance, and groupby results are sorted only if `sort_gb=True`.

| ordered | lex | categories sorted? | groupby results sorted? |
|---|---|---|---|
| T | T | Y | Y |
| T | F | Y | Y |
| F | T | Y | Y |
| F | F | N | only if sort_gb=True |

**If categories are provided,** they are always held in the same order. The `ordered` argument is ignored, and `lex` can't be specified. Groupby results can be displayed in sorted order with `sort_gb=True`.

#### Categorical created from values (no user-provided categories)

##### With the default `ordered=True`

- Categories are sorted.
- Groupby results are sorted regardless of whether `sort_gb=True`.

```
>>> vals = rt.arange(6)
>>> c = rt.Categorical(["b", "a", "a", "c", "a", "b"])
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([2, 1, 1, 3, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c.sum(vals)
*key_0   col_0
------   -----
a            7
b            5
c            3
```

### With `ordered=False`

- Categories are not sorted unless `lex=True`. Setting `lex=True` causes a lexicographical sort to be performed to find the uniques, and the Categorical's categories become sorted even if `ordered=False`.
- Groupby results are not displayed sorted unless `lex=True` or `sort_gb=True`.

```
>>> vals = rt.arange(6)
>>> c = rt.Categorical(["b", "a", "a", "c", "a", "b"], ordered=False)
>>> c
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 2, 2, 3, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3

>>> c.sum(vals)
*key_0   col_0
------   -----
b            5
a            7
c            3
```

Here, `lex=True` causes the categories to become sorted even though `ordered=False`.

```
>>> c = rt.Categorical(["b", "a", "a", "c", "a", "b"], ordered=False, lex=True)
>>> c
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([2, 1, 1, 3, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

```
>>> c.sum(vals)
*key_0   col_0
------   -----
a            7
b            5
c            3
```

```
>>> c = rt.Categorical(["b", "a", "a", "c", "a", "b"], ordered=False, sort_gb=True)
>>> c
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 2, 2, 3, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3
```

```
>>> c.sum(vals)
*key_0   col_0
------   -----
a            7
b            5
c            3
```

### Categorical created from values and user-provided categories (unsorted)

- If categories are provided, they are always held in the same order. The `ordered` argument is ignored, and you can't set `lex=True`.
- Groupby results can be displayed in sorted order with `sort_gb=True`.

Categorical with unsorted categories:

```
>>> c = rt.Categorical(["b", "a", "a", "c", "a", "b"], categories=["b", "a", "c"])
>>> c
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 2, 2, 3, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3
```

Groupby results are in the order of provided categories:

```
>>> vals = rt.arange(6)
>>> c.sum(vals)
*key_0   col_0
------   -----
b            5
a            7
c            3
```

With provided categories, `lex` can't be set to `True`:

```
>>> try:
...  rt.Categorical(["b", "a", "a", "c", "a", "b"], categories=["b", "a", "c"], lex=True)
... except TypeError as e:
...    print("TypeError:", e)
TypeError: Cannot bin using lexsort and user-suplied categories.
```

With `sort_gb=True`, categories are held in the order provided but displayed lexicographically sorted in groupby results:

```
>>> c = rt.Categorical(["b", "a", "a", "c", "a", "b"], categories=["b", "a", "c"], sort_
↪gb=True)
>>> c
Categorical([b, a, a, c, a, b]) Length: 6
  FastArray([1, 2, 2, 3, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3

>>> c.sum(vals)
*key_0   col_0
------   -----
a            7
b            5
c            3
```

If the categories are provided in a mapping dictionary or `IntEnum`, the groupby results are in the order of the underlying mapping codes array unless `sort_gb=True`:

```
>>> d = {"StronglyAgree": 44, "Agree": 133, "Disagree": 75, "StronglyDisagree": 1,
↪"NeitherAgreeNorDisagree": 144 }
```

(continues on next page)

(continued from previous page)

```
>>> codes = [1, 44, 44, 133, 75]  # Note duplication and missing entry.
>>> c = rt.Categorical(codes, categories=d)
>>> c
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]) Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4
>>> vals = rt.arange(5)
>>> ds = rt.Dataset({"c": c, "vals": vals})
>>> ds
#   c                 vals
-   ---------------   ----
0   StronglyDisagree     0
1   StronglyAgree        1
2   StronglyAgree        2
3   Agree                3
4   Disagree             4

>>> c.sum(vals)
*c               vals
--------------   ----
StronglyDisagre     0
StronglyAgree       3
Agree               3
Disagree            4
```

With `sort_gb=True`, categories are displayed lexicographically sorted in groupby results:

```
>>> c = rt.Categorical(codes, categories=d, sort_gb=True)
>>> c
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]) Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4

>>> c.sum(vals)
*key_0           vals
--------------   ----
Agree               3
Disagree            4
StronglyAgree       3
StronglyDisagre     0
```

### Ordering of results from rt.cut and rt.qcut operations

With `cut` and `qcut`, when labels are provided they are held and displayed in the order of first appearance and are considered ordered in the context of logical comparisons:

```
>>> c = rt.cut(rt.arange(10), bins=3, labels=["z-label1", "y-label2", "x-label3"])
>>> c
Categorical([z-label1, z-label1, z-label1, z-label1, y-label2, y-label2, y-label2, x-
↪label3, x-label3, x-label3]) Length: 10
  FastArray([1, 1, 1, 1, 2, 2, 2, 3, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'z-label1', b'y-label2', b'x-label3'], dtype='|S8') Unique count: 3

>>> c.sum(rt.arange(10))
*key_0     col_0
--------   -----
z-label1       6
y-label2      15
x-label3      24

>>> c > "z-label1"
FastArray([False, False, False, False,  True,  True,  True,  True,  True, True])
```

See *Comparisons* for more examples.

### Riptable Categoricals – Comparisons

When you provide a value to compare against a Categorical, it's important to remember that the Categorical may be ordered in a non-lexicographic way. (See *Sorting and Display Order* for more details.)

For example, this Categorical's categories are held in the order in which they appear in the provided values (when `ordered=True`, they are lexicographically sorted instead):

```
>>> c = rt.Categorical([4, 1, 2, 2, 3, 4, 4], ordered=False)
>>> c
Categorical([4, 1, 2, 2, 3, 4, 4]) Length: 7
  FastArray([1, 2, 3, 3, 4, 1, 1], dtype=int8) Base Index: 1
  FastArray([4, 1, 2, 3]) Unique count: 4

>>> c.category_array
FastArray([4, 1, 2, 3])
```

The ordering of the category array is the ordering used for comparisons: 4 < 1 < 2 < 3.

So when we check whether each Categorical value is greater than 2:

```
>>> c > 2
FastArray([False, False, False, False,  True, False, False])
```

… the result reflects that in this Categorical, the only value greater than 2 is 3.

In practice, the comparison is performed on the array of indices. Notice that the indices reflect the order of the category array: The 4 category is mapped to 1 in the index array, the 1 category is mapped to 2, the 2 category is mapped to 3, and the 3 category is mapped to 4.

The value you provide for comparison is converted to its index and compared against the index array.

So 2's index, 3, is compared to the other indices:

```
>>> 3 < c._fa
FastArray([False, False, False, False,  True, False, False])
```

(If the value you provide doesn't have an index because it doesn't exist in the Categorical, the comparison is adjusted so that it still works out.)

When you provide categories, they are always held in the order they're provided:

```
>>> c = rt.Categorical([1, 1, 2, 2, 3, 4, 4], categories=[5, 7, 3, 6])
>>> c
Categorical([5, 5, 7, 7, 3, 6, 6]) Length: 7
  FastArray([1, 1, 2, 2, 3, 4, 4]) Base Index: 1
  FastArray([5, 7, 3, 6]) Unique count: 4

>>> c > 3
FastArray([False, False, False, False, False,  True,  True])
```

When you do comparisons with strings, unicode strings and byte strings are properly translated internally.

Ordering is that of the provided categories:

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"], categories=["b", "a", "c"])
>>> c
Categorical([b, a, b, a, c, c, b]) Length: 7
  FastArray([1, 2, 1, 2, 3, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3

>>> c > "b"
FastArray([False,  True, False,  True,  True,  True, False])
```

When categories aren't provided, by default they are sorted lexicographically:

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([b, a, b, a, c, c, b]) Length: 7
  FastArray([2, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3

>>> c > "b"
FastArray([False, False, False, False,  True,  True, False])
```

The equality operator and *isin* are more straightforward, and can be used to construct boolean filters based on categories.

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([a, a, b, a, c, c, b]) Length: 7
  FastArray([1, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

```
>>> c == "a"
FastArray([ True,  True, False,  True, False, False, False])
```

```
>>> c == b"a"
FastArray([ True,  True, False,  True, False, False, False])
```

```
>>> c.isin("a")
FastArray([ True,  True, False,  True, False, False, False])
```

```
>>> c.isin(["a", "b"])
FastArray([ True,  True,  True,  True, False, False,  True])
```

```
>>> c = rt.Categorical([5, 6, 6, 7, 7, 6, 6, 6, 7, 5])
>>> c
Categorical([5, 6, 6, 7, 7, 6, 6, 6, 7, 5]) Length: 10
  FastArray([1, 2, 2, 3, 3, 2, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([5, 6, 7]) Unique count: 3
```

```
>>> c == 1
FastArray([False, False, False, False, False, False, False, False, False, False])
```

```
>>> c == 6
FastArray([False,  True,  True, False, False,  True,  True,  True, False, False])
```

```
>>> c.isin(5)
FastArray([ True, False, False, False, False, False, False, False, False,  True])
```

```
>>> c.isin([5, 6])
FastArray([ True,  True,  True, False, False,  True,  True,  True, False,  True])
```

The underlying integer mapping array can also be used for both string and integer Categoricals:

```
>>> c._fa == 2
FastArray([False,  True,  True, False, False,  True,  True,  True, False, False])
```

### Riptable Categoricals – Final dtype of Integer Mapping Array

#### Final dtype from provided mapping code/index array

If the user provides the integer array of mapping codes, the array will not be recast unless:

- The integer type is unsigned. If a large enough dtype is specified with the `dtype` argument, it will be used; otherwise the smallest possible dtype will be used based on the number of unique categories or the maximum value provided in a mapping.
- A dtype is specified that's large enough to accommodate the provided codes. If the dtype isn't large enough, the array is upcast to the smallest possible dtype that can be used.

Codes with a signed integer dtype:

```
>>> codes = rt.FastArray([2, 4, 4, 3, 2, 1, 3, 2, 0, 1, 3, 4, 2, 0, 4,
...                       3, 1, 0, 1, 2, 3, 1, 4, 2, 2, 3, 4, 2, 0, 2], dtype=rt.int64)
>>> cats = rt.FastArray(["a", "b", "c", "d", "e"])
```

It is unchanged:

```
>>> rt.Categorical(codes, categories=cats)
Categorical([b, d, d, c, b, ..., c, d, b, Filtered, b]) Length: 30
  FastArray([2, 4, 4, 3, 2, ..., 3, 4, 2, 0, 2], dtype=int64) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

The codes have an unsigned dtype. No `dtype` argument is provided, so the smallest dtype is found:

```
>>> c = rt.Categorical(codes.astype(rt.uint64), categories=cats)
Categorical([b, d, d, c, b, ..., c, d, b, Filtered, b]) Length: 30
  FastArray([2, 4, 4, 3, 2, ..., 3, 4, 2, 0, 2]) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5

>>> c._fa.dtype
dtype('int8')
```

The codes have an unsigned dtype, and the specified dtype is large enough to be used:

```
>>> rt.Categorical(codes.astype(rt.uint8), categories=cats, dtype=rt.int64)
Categorical([b, d, d, c, b, ..., c, d, b, Filtered, b]) Length: 30
  FastArray([2, 4, 4, 3, 2, ..., 3, 4, 2, 0, 2], dtype=int64) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

The codes have a signed dtype, and a different dtype is specified that's large enough to accommodate the provided codes:

```
>>> rt.Categorical(codes.astype(rt.int16), categories=cats, dtype=rt.int64)
Categorical([b, d, d, c, b, ..., c, d, b, Filtered, b]) Length: 30
  FastArray([2, 4, 4, 3, 2, ..., 3, 4, 2, 0, 2], dtype=int64) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

The codes have a signed dtype, but the specified dtype is too small:

```
>>> big_cats = rt.FastArray(['string'+str(i) for i in range(2000)])
>>> rt.Categorical(codes, big_cats, dtype=rt.int8)
UserWarning: A type of <class 'riptable.rt_numpy.int8'> was too small, upcasting.
Categorical([string1, string3, string3, string2, string1, ..., string2, string3, string1,
↪ Filtered, string1]) Length: 30
  FastArray([2, 4, 4, 3, 2, ..., 3, 4, 2, 0, 2], dtype=int16) Base Index: 1
  FastArray([b'string0', b'string1', b'string2', b'string3', b'string4', ..., b
↪'string1995', b'string1996', b'string1997', b'string1998', b'string1999'], dtype='|S10
↪') Unique count: 2000
```

### Final dtype from Matlab index array

If the index array is from Matlab, it is often floating-point. Unless a dtype is specified, the smallest dtype will be found:

No dtype is specified; the smallest usable dtype is found:

```
>>> matlab_codes = (codes + 1).astype(rt.float32)
>>> rt.Categorical(matlab_codes, categories=cats, from_matlab=True)
Categorical([c, e, e, d, c, ..., d, e, c, a, c]) Length: 30
  FastArray([3, 5, 5, 4, 3, ..., 4, 5, 3, 1, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

A dtype is specified that's large enough to be used:

```
>>> rt.Categorical(matlab_codes, categories=cats, from_matlab=True, dtype=rt.int64)
Categorical([c, e, e, d, c, ..., d, e, c, a, c]) Length: 30
  FastArray([3, 5, 5, 4, 3, ..., 4, 5, 3, 1, 3], dtype=int64) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

### Final dtype from strings or strings + categories

A new index array is generated. Unless a dtype is specified, the smallest usable dtype will be found.

No dtype specified; the smallest usable dtype is found:

```
>>> str_fa = rt.FastArray(["c", "e", "e", "d", "c", "b", "d", "c", "a", "b",
...                        "d", "e", "c", "a", "e", "d", "b", "a", "b", "c",
...                        "d", "b", "e", "c", "c", "d", "e", "c", "a", "c"])
>>> rt.Categorical(str_fa)
Categorical([c, e, e, d, c, ..., d, e, c, a, c]) Length: 30
  FastArray([3, 5, 5, 4, 3, ..., 4, 5, 3, 1, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

A large enough dtype is specified:

```
>>> rt.Categorical(str_fa, dtype=rt.int64)
Categorical([c, e, e, d, c, ..., d, e, c, a, c]) Length: 30
  FastArray([3, 5, 5, 4, 3, ..., 4, 5, 3, 1, 3], dtype=int64) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

### Final dtype from a multi-key Categorical

This follows the same rules as string construction. Unless a dtype is specified, the smallest usable dtype is found.

No dtype specified; the smallest usable dtype is found:

```
>>> rt.Categorical([str_fa, codes])
Categorical([(c, 2), (e, 4), (e, 4), (d, 3), (c, 2), ..., (d, 3), (e, 4), (c, 2), (a, 0),
↪ (c, 2)]) Length: 30
  FastArray([1, 2, 2, 3, 1, ..., 3, 2, 1, 5, 1], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'c', b'e', b'd', b'b', b'a'], dtype='|S1'), 'key_1':␣
↪FastArray([2, 4, 3, 1, 0])} Unique count: 5
```

A large enough dtype is specified:

```
>>> rt.Categorical([str_fa, codes], dtype=rt.int64)
Categorical([(c, 2), (e, 4), (e, 4), (d, 3), (c, 2), ..., (d, 3), (e, 4), (c, 2), (a, 0),
↪ (c, 2)]) Length: 30
  FastArray([1, 2, 2, 3, 1, ..., 3, 2, 1, 5, 1], dtype=int64) Base Index: 1
  {'key_0': FastArray([b'c', b'e', b'd', b'b', b'a'], dtype='|S1'), 'key_1':␣
↪FastArray([2, 4, 3, 1, 0])} Unique count: 5
```

### Final dtype from a Pandas Categorical

Pandas already attempts to find the smallest dtype during Categorical construction. If a Riptable Categorical is created from a Pandas Categorical and a dtype is specified, Riptable uses the specified dtype.

Construction from Pandas always generates a new array because Riptable adds 1 to the indices:

```
>>> import pandas as pd
>>> pdc = pd.Categorical(str_fa)
>>> pdc._codes
array([2, 4, 4, 3, 2, 1, 3, 2, 0, 1, 3, 4, 2, 0, 4, 3, 1, 0, 1, 2, 3, 1,
       4, 2, 2, 3, 4, 2, 0, 2], dtype=int8)
>>> c = rt.Categorical(pdc)
>>> c
Categorical([c, e, e, d, c, ..., d, e, c, a, c]) Length: 30
  FastArray([3, 5, 5, 4, 3, ..., 4, 5, 3, 1, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5

>>> c = rt.Categorical(pdc, dtype=rt.int32)
Categorical([c, e, e, d, c, ..., d, e, c, a, c]) Length: 30
  FastArray([3, 5, 5, 4, 3, ..., 4, 5, 3, 1, 3]) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
```

### Riptable Categoricals – Invalid Categories

A category set to be invalid at Categorical creation is considered to be NaN in the sense that *isnan* returns `True` for the category, but it's mapped to a valid index and not excluded from any operations on the Categorical. To exclude values or categories from operations, use the `filter` argument.

Note that this behavior differs from *Previous invalid behavior*.

Warning: If the invalid category isn't in the provided list of unique categories and a filter is also provided at Categorical creation, the invalid category also becomes Filtered.

### Categorical created from values (no user-provided categories)

Because it's assigned to a regular bin, an invalid category is allowed for base-0 and base-1 indexing:

```
>>> c = rt.Categorical(["b", "a", "a", "Inv", "c", "a", "b"], invalid="Inv", base_
↪index=0)
>>> c
Categorical([b, a, a, Inv, c, a, b]) Length: 7
FastArray([2, 1, 1, 0, 3, 1, 2]) Base Index: 0
FastArray([b'Inv', b'a', b'b', b'c'], dtype='|S3') Unique count: 4

>>> c.isnan()
FastArray([False, False, False,  True, False, False, False])

>>> c = rt.Categorical(['b', 'a', 'Inv', 'a'], invalid='Inv')
>>> c
Categorical([b, a, Inv, a]) Length: 4
  FastArray([3, 2, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'Inv', b'a', b'b'], dtype='|S3') Unique count: 3
```

(continues on next page)

(continued from previous page)

```
>>> c.isnan()
FastArray([False, False,  True, False])
```

### Categorical created from values and user-provided categories

If an invalid category is specified, it must also be in the list of unique categories, otherwise an error is raised:

```
>>> # Included.
>>> c = rt.Categorical(["b", "a", "Inv", "a"], categories=["a", "b", "Inv"], invalid="Inv
↪")
>>> c
Categorical([b, a, Inv, a]) Length: 4
  FastArray([2, 1, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'Inv'], dtype='|S3') Unique count: 3

>>> # Not included.
>>> try:
...     rt.Categorical(["b", "a", "Inv", "a"], categories=["a", "b"], invalid="Inv")
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Found values that were not in provided categories: [b'Inv']. The
user-supplied categories (second argument) must also contain the invalid item Inv.
For example: Categorical(['b','a','Inv','a'], ['a','b','Inv'], invalid='Inv')
```

### Categorical created with a filter

Be careful when mixing invalid categories and filters.

If you filter an invalid category, it becomes Filtered and no longer invalid:

```
>>> c = rt.Categorical(["Inv", "a", "b", "a"], categories=["Inv", "a", "b"],
...                    filter=rt.FA([False, True, True, True]), invalid="Inv")

>>> c
Categorical([Filtered, a, b, a]) Length: 4
  FastArray([0, 2, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'Inv', b'a', b'b'], dtype='|S3') Unique count: 3

>>> c.isnan()
FastArray([False, False, False, False])
```

Warning: If the invalid category *isn't* included in the array of unique cagtegories and you *also* provide a filter, the invalid category *also becomes Filtered* even if it isn't filtered out directly.

For comparison, here's an example where the invalid category *is* included in the list of unique categories and a filter is provided. You get a warning that doesn't apply in this case, and the filter is applied:

```
>>> c = rt.Categorical(["Inv", "a", "b", "a"], categories=["Inv", "a", "b"],
...                    filter=rt.FA([True, True, False, False]), invalid="Inv")
UserWarning: Invalid category was set to Inv. If not in provided categories, will
```

(continues on next page)

(continued from previous page)

```
also appear as filtered. For example: print(Categorical(['a','a','b'], ['b'],
filter=FA([True, True, False]), invalid='a')) -> Filtered, Filtered, Filtered
```

The second two values are filtered:

```
>>> c
Categorical([Inv, a, Filtered, Filtered]) Length: 4
  FastArray([1, 2, 0, 0], dtype=int8) Base Index: 1
  FastArray([b'Inv', b'a', b'b'], dtype='|S3') Unique count: 3
```

And the invalid category is still invalid:

```
>>> c.isnan()
FastArray([ True, False, False, False])
```

However, when the invalid category *is not* included in the list of unique categories, the warning does apply, and the invalid category also becomes Filtered:

```
>>> c = rt.Categorical(["Inv", "a", "b", "a"], categories=["a", "b"],
...                    filter=rt.FA([True, True, False, False]), invalid="Inv")
UserWarning: Invalid category was set to Inv. If not in provided categories, will
also appear as filtered. For example: print(Categorical(['a','a','b'], ['b'],
filter=FA([True, True, False]), invalid='a')) -> Filtered, Filtered, Filtered

>>> c
Categorical([Filtered, a, Filtered, Filtered]) Length: 4
  FastArray([0, 1, 0, 0], dtype=int8) Base Index: 1
  FastArray([b'a', b'b'], dtype='|S1') Unique count: 2
```

And "Inv" is no longer considered an invalid category:

```
>>> c.isnan()
FastArray([False, False, False, False])
```

### Invalid categories are not excluded from operations

Although invalid categories are recognized by the Categorical *isnan* method, they are not excluded from operations as filtered values and categories are.

Here, "Inv" is invalid and the "b" category is filtered:

```
>>> vals = rt.FA(["Inv", "b", "a", "b", "c", "c", "Inv"])
>>> f = vals != "b"
>>> c = rt.Categorical(vals, invalid="Inv", filter=f)
>>> c
Categorical([Inv, Filtered, a, Filtered, c, c, Inv]) Length: 7
  FastArray([1, 0, 2, 0, 3, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'Inv', b'a', b'c'], dtype='|S3') Unique count: 3

>>> c.isnan()
FastArray([ True, False, False, False, False, False,  True])
```

Create some values to sum and put in a Dataset to see their relationsips to the catgegories:

```
>>> vals = rt.FA([1, 2, 3, 4, 5, 6, 7])
>>> ds = rt.Dataset({"c": c, "vals": vals})
>>> ds
#   c          vals
-   --------   ----
0   Inv           1
1   Filtered      2
2   a             3
3   Filtered      4
4   c             5
5   c             6
6   Inv           7
```

Get the nansum:

```
>>> c.nansum(vals)
*c    vals
---   ----
Inv      8
a        3
c       11
```

The showfilter argument confirms that only the "b" values were excluded:

```
>>> c.nansum(vals, showfilter=True)
*c         vals
--------   ----
Filtered      6
Inv           8
a             3
c            11
```

If you use the filter argument with nansum and filter out an invalid, the filtered invalid value is excluded from the operation:

```
>>> # Filter the first Inv, one of the already-filtered "b"s, and the first "c".
>>> f2 = rt.FA([False, False, True, True, True, False, True])
>>> c.nansum(vals, filter=f2, showfilter=True)
*key_0     col_0
--------   -----
Filtered      13
Inv            7
a              3
c              5
```

If both invalid values are filtered by the nansum operation, the category still appears in the result:

```
>>> f3 = rt.FA([False, False, True, True, True, False, False])
>>> c.nansum(vals, filter=f3)
*c    vals
---   ----
Inv      0
a        3
c        5
```

And both invalid values are still invalid:

```
>>> c.isnan()
FastArray([ True, False, False, False, False, False,  True])
```

### Previous invalid behavior

Previously, the specified string was used to represent an invalid catgegory when values missing in the categories list were encountered. The invalid category was mapped to 0 in the index/codes array.

This is similar to how Pandas works, except that Pandas uses -1 for its NaN index:

```
>>> import pandas as pd
>>> pdc = pd.Categorical(["a", "a", "z", "b", "c"], ["a", "b", "c"])
>>> pdc
['a', 'a', NaN, 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
>>> pdc._codes
array([ 0,  0, -1,  1,  2], dtype=int8)
>>> pd.Series([1, 1, 1, 1, 1]).groupby(pdc).count()
a    2
b    1
c    1
dtype: int64
```

### Riptable Categoricals – Get Bins from Categories and Vice-Versa

You can use the *from_bin* method and *from_category* methods to retrieve corresponding index / mapping codes and categories.

### Category from bin

Provide the bin to get its associated category.

With base index 1:

```
>>> c = rt.Categorical(["a", "a", "b", "c", "d"])
>>> c
Categorical([a, a, b, c, d]) Length: 5
  FastArray([1, 1, 2, 3, 4], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd'], dtype='|S1') Unique count: 4

>>> c.from_bin(1)
b'a'
```

With base index 0:

```
>>> c = rt.Categorical(["a", "a", "b", "c", "d"], base_index=0)
>>> c
Categorical([a, a, b, c, d]) Length: 5
  FastArray([0, 0, 1, 2, 3]) Base Index: 0
```

(continues on next page)

(continued from previous page)

```
  FastArray([b'a', b'b', b'c', b'd'], dtype='|S1') Unique count: 4

>>> c.from_bin(1)
b'b'
```

An out-of-range bin returns an error:

```
>>> try:
...     c.from_bin(10)
... except IndexError as e:
...     print("IndexError:", e)
IndexError: index 9 is out of bounds for axis 0 with size 4
```

Note that with a base-1 index, `from_bin(0)` is also considered out of range:

```
>>> try:
...     c.from_bin(0)
... except ValueError as e:
...     print("ValueError:", e)
ValueError: Bin 0 is out of range for categorical with base index 1
```

You can also provide the code from a mapping dictionary:

```
>>> c = rt.Categorical([10, 20, 20, 30, 40], categories={"a": 10, "b": 20, "c": 30, "d":␣
↪40})
>>> c
Categorical([a, b, b, c, d]) Length: 5
  FastArray([10, 20, 20, 30, 40]) Base Index: None
  {10:'a', 20:'b', 30:'c', 40:'d'} Unique count: 4
>>> c.from_bin(40)
'd'
```

A multi-key Categorical returns a tuple of unique values:

```
>>> c = rt.Categorical([rt.FastArray(["a", "b", "c", "d", "e"]), rt.arange(5)])
>>> c
Categorical([(a, 0), (b, 1), (c, 2), (d, 3), (e, 4)]) Length: 5
  FastArray([1, 2, 3, 4, 5], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1'), 'key_1':␣
↪FastArray([0, 1, 2, 3, 4])} Unique count: 5
>>> c.from_bin(1)
(b'a', 0)
```

### Bin from category

Provide the unique category to get its associated bin. Note that the bin isn't always an index into the stored category array.

Unicode strings and byte strings are properly translated internally.

With base index 1:

```
>>> c = rt.Categorical(["a", "a", "b", "c", "d"])
>>> c
Categorical([a, a, b, c, d]) Length: 5
  FastArray([1, 1, 2, 3, 4], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd'], dtype='|S1') Unique count: 4

>>> c.from_category("a")
1
```

With base index 0:

```
>>> c = rt.Categorical(["a", "a", "b", "c", "d"], base_index=0)
>>> c
Categorical([a, a, b, c, d]) Length: 5
  FastArray([0, 0, 1, 2, 3]) Base Index: 0
  FastArray([b'a', b'b', b'c', b'd'], dtype='|S1') Unique count: 4

>>> c.from_category("a")
0
```

A non-existent category returns an error:

```
>>> try:
...     c.from_category("z")
... except ValueError as e:
...     print("ValueError:", e)
ValueError: z not found in uniques.
```

Provide a tuple for a multi-key Categorical:

```
>>> c = rt.Categorical([rt.FastArray(["a", "b", "c", "d", "e"]), rt.arange(5)])
>>> c
Categorical([(a, 0), (b, 1), (c, 2), (d, 3), (e, 4)]) Length: 5
  FastArray([1, 2, 3, 4, 5], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1'), 'key_1':␣
↪FastArray([0, 1, 2, 3, 4])} Unique count: 5

>>> c.from_category(("d", 3))
4
```

Riptable Categoricals have two related uses:

- They efficiently store string (or other large dtype) arrays that have repeated values. The repeated values are partitioned into groups (a.k.a. categories), and each group is mapped to an integer. For example, in a Categorical that contains three "AAPL" symbols and four "MSFT" symbols, the data is partitioned into an "AAPL" group that's mapped to 1 and a "MSFT" group that's mapped to 2. This integer mapping allows the data to be stored and operated on more efficiently.

- They're Riptable's class for doing group operations. A method applied to a Categorical is applied to each group separately.

A Categorical is typically created from a list of strings:

```
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([a, a, b, a, c, c, b]) Length: 7
  FastArray([1, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

The output shows:

- The Categorical values. These are made unique to form the categories.
- The integer mapping codes that correspond to the unique categories. Because the integers can be used to index into the Categorical, they're also referred to as the indices. Notice that the base index of the array is also shown. By default, the integer index is 1-based; 0 is reserved for Filtered categories. The integer array is an `int8`, `int16`, `int32`, or `int64` array, depending on the number of unique categories or the maximum value provided in a mapping.
- The unique categories. (Both the categories and their associated integer codes are sometimes called bins.) Each represents a group for groupby operations. The categories are held in an array (sorted by default), a dictionary that maps integers to strings or strings to integers, or a multi-key dictionary.

Use Categorical objects to perform aggregations over arbitrary arrays of the same dimension as the Categorical:

```
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"])
>>> ints = rt.FA([3, 10, 2, 5, 4, 1, 1])
>>> flts = rt.FA([1.2, 3.4, 5.6, 4.0, 2.1, 0.6, 11.3])
>>> c.sum([ints, flts])
*key_0   col_0   col_1
------   -----   -----
a           18    8.60
b            3   16.90
c            5    2.70
```

Multi-key Categoricals let you create and operate on groupings based on multiple related categories:

```
>>> strs1 = rt.FastArray(["a", "b", "b", "a", "b", "a"])
>>> strs2 = rt.FastArray(["x", "y", "y", "z", "x", "x"])
>>> c = rt.Categorical([strs1, strs2])
>>> c
Categorical([(a, x), (b, y), (b, y), (a, z), (b, x), (a, x)]) Length: 6
  FastArray([1, 2, 2, 3, 4, 1], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'a', b'b'], dtype='|S1'), 'key_1': FastArray([b'x', b
↪'y', b'z', b'x'], dtype='|S1')} Unique count: 4

>>> c.count()
*key_0   *key_1   Count
------   ------   -----
a        x            2
b        y            2
a        z            1
b        x            1
```

To see more ways to create a Categorical, go to *Constructing Categoricals*. To see more operations on Categoricals, see the *Categoricals* section of the *Intro to Riptable*.

CHAPTER

# TWO

# RIPTABLE

## 2.1 Subpackages

### 2.1.1 riptable.Utils

#### Submodules

##### riptable.Utils.appdirs

Utilities for determining application-specific dirs.

See <http://github.com/ActiveState/appdirs> for details and usage.

#### Classes

| | |
|---|---|
| *AppDirs* | Convenience wrapper for getting application dirs. |

### Functions

| | |
|---|---|
| _get_win_folder_from_registry(csidl_name) | This is a fallback technique at best. I'm not sure if using the |
| _get_win_folder_with_ctypes(csidl_name) | |
| _get_win_folder_with_jna(csidl_name) | |
| _get_win_folder_with_pywin32(csidl_name) | |
| site_config_dir([appname, appauthor, version, multipath]) | Return full path to the user-shared data dir for this application. |
| site_data_dir([appname, appauthor, version, multipath]) | Return full path to the user-shared data dir for this application. |
| user_cache_dir([appname, appauthor, version, opinion]) | Return full path to the user-specific cache dir for this application. |
| user_config_dir([appname, appauthor, version, roaming]) | Return full path to the user-specific config dir for this application. |
| user_data_dir([appname, appauthor, version, roaming]) | Return full path to the user-specific data dir for this application. |
| user_log_dir([appname, appauthor, version, opinion]) | Return full path to the user-specific log dir for this application. |
| user_state_dir([appname, appauthor, version, roaming]) | Return full path to the user-specific state dir for this application. |

### Attributes

| |
|---|
| PY3 |
| __version__ |
| __version_info__ |
| _get_win_folder |
| appname |
| os_name |
| unicode |

**class** riptable.Utils.appdirs.**AppDirs**(*appname=None*, *appauthor=None*, *version=None*, *roaming=False*, *multipath=False*)

Bases: object

Convenience wrapper for getting application dirs.

**property site_config_dir**

**property site_data_dir**

**property user_cache_dir**

**property user_config_dir**

**property user_data_dir**

**property user_log_dir**

**property user_state_dir**

riptable.Utils.appdirs.**_get_win_folder_from_registry**(*csidl_name*)

This is a fallback technique at best. I'm not sure if using the registry for this guarantees us the correct answer for all CSIDL_* names.

riptable.Utils.appdirs.**_get_win_folder_with_ctypes**(*csidl_name*)

riptable.Utils.appdirs.**_get_win_folder_with_jna**(*csidl_name*)

riptable.Utils.appdirs.**_get_win_folder_with_pywin32**(*csidl_name*)

riptable.Utils.appdirs.**site_config_dir**(*appname=None*, *appauthor=None*, *version=None*, *multipath=False*)

Return full path to the user-shared data dir for this application.

Typical site config directories are:

- Mac OS X: same as *site_data_dir*
- Unix: /etc/xdg/<AppName> or $XDG_CONFIG_DIRS[i]/<AppName> for each value in $XDG_CONFIG_DIRS
- Win: same as *site_data_dir*
- Vista: (Fail! "C:\ProgramData" is a hidden *system* directory on Vista.)

For Unix, this is using the $XDG_CONFIG_DIRS[0] default, if multipath=False.

> **Warning:** Do not use this on Windows. See the Vista-Fail note above for why.

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass False to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when *appname* is present.
- **multipath** (*bool, default False*) – An optional parameter only applicable to Unix-like operating systems that indicates that the entire list of config dirs should be returned. By default, the first item from XDG_CONFIG_DIRS is returned, or '/etc/xdg/<AppName>', if XDG_CONFIG_DIRS is not set.

riptable.Utils.appdirs.**site_data_dir**(*appname=None*, *appauthor=None*, *version=None*, *multipath=False*)

Return full path to the user-shared data dir for this application.

Typical site data directories are:

- Mac OS X: /Library/Application Support/<AppName>
- Unix: /usr/local/share/<AppName> or /usr/share/<AppName>
- Win XP: C:\Documents and Settings\All Users\Application Data\<AppAuthor>\<AppName>
- Vista: (Fail! "C:\ProgramData" is a hidden *system* directory on Vista.)
- Win 7: C:\ProgramData\<AppAuthor>\<AppName> (Hidden, but writeable on Win 7.)

For Unix, this is using the $XDG_DATA_DIRS[0] default.

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass False to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when *appname* is present.
- **multipath** (*bool, default False*) – An optional parameter only applicable to Unix-like operating systems that indicates that the entire list of data dirs should be returned. By default, the first item from XDG_DATA_DIRS is returned, or '/usr/local/share/<AppName>' if XDG_DATA_DIRS is not set.

**Warning:** Do not use this on Windows. See the Vista-Fail note above for why.

riptable.Utils.appdirs.**user_cache_dir**(*appname=None*, *appauthor=None*, *version=None*, *opinion=True*)

Return full path to the user-specific cache dir for this application.

Typical user cache directories are:

- Mac OS X: ~/Library/Caches/<AppName>
- Unix: ~/.cache/<AppName> (XDG default)
- Win XP: C:\Documents and Settings\<username>\Local Settings\Application Data\<AppAuthor>\<AppName>\Cache
- Vista: C:\Users\<username>\AppData\Local\<AppAuthor>\<AppName>\Cache

On Windows the only suggestion in the MSDN docs is that local settings go in the CSIDL_LOCAL_APPDATA directory. This is identical to the non-roaming app data dir (the default returned by *user_data_dir* above). Apps typically put cache data somewhere *under* the given dir here. Some examples:

- ...\Mozilla\Firefox\Profiles\<ProfileName>\Cache
- ...\Acme\SuperApp\Cache\1.0

OPINION: This function appends "Cache" to the CSIDL_LOCAL_APPDATA value. This can be disabled with the opinion=False option.

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass `False` to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.
- **opinion** (*bool, default True*) – Can be `False` to disable the appending of "Cache" to the base app data dir for Windows. See discussion above.

riptable.Utils.appdirs.**user_config_dir**(*appname=None*, *appauthor=None*, *version=None*, *roaming=False*)

Return full path to the user-specific config dir for this application.

Typical user config directories are:

- Mac OS X: ~/Library/Preferences/<AppName>
- Unix: ~/.config/<AppName> or in $XDG_CONFIG_HOME, if defined
- Win: same as *user_data_dir*

For Unix, we follow the XDG spec and support $XDG_CONFIG_HOME. That means ~/.config/<AppName> by default.

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass `False` to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when *appname* is present.
- **roaming** (*bool, default False*) – Can be set `True` to use the Windows roaming appdata directory. That means that for users on a Windows network setup for roaming profiles, this user data will be sync'd on login. See Microsoft's Managing Roaming User Data Deployment Guide for a discussion of issues.

riptable.Utils.appdirs.**user_data_dir**(*appname=None*, *appauthor=None*, *version=None*, *roaming=False*)

Return full path to the user-specific data dir for this application.

Typical user data directories are:

- Mac OS X: ~/Library/Application Support/<AppName>
- Unix: ~/.local/share/<AppName> or in $XDG_DATA_HOME, if defined
- Win XP (not roaming): C:\Documents and Settings\<username>\Application Data\<AppAuthor>\<AppName>
- Win XP (roaming): C:\Documents and Settings\<username>\Local Settings\Application Data\<AppAuthor>\<AppName>
- Win 7 (not roaming): C:\Users\<username>\AppData\Local\<AppAuthor>\<AppName>
- Win 7 (roaming): C:\Users\<username>\AppData\Roaming\<AppAuthor>\<AppName>

For Unix, we follow the XDG spec and support `$XDG_DATA_HOME`. That means `~/.local/share/<AppName>` by default .

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass `False` to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be `"<major>.<minor>"`. Only applied when *appname* is present.
- **roaming** (*bool, default False*) – Can be set `True` to use the Windows roaming appdata directory. That means that for users on a Windows network setup for roaming profiles, this user data will be sync'd on login. See Microsoft's Managing Roaming User Data Deployment Guide for a discussion of issues.

riptable.Utils.appdirs.**user_log_dir**(*appname=None*, *appauthor=None*, *version=None*, *opinion=True*)

Return full path to the user-specific log dir for this application.

Typical user log directories are:

- Mac OS X: `~/Library/Logs/<AppName>`
- Unix: `~/.cache/<AppName>/log` or under `$XDG_CACHE_HOME` if defined
- Win XP: `C:\Documents and Settings\<username>\Local Settings\Application Data\<AppAuthor>\<AppName>\Logs`
- Vista: `C:\Users\<username>\AppData\Local\<AppAuthor>\<AppName>\Logs`

On Windows the only suggestion in the MSDN docs is that local settings go in the `CSIDL_LOCAL_APPDATA` directory. (Note: I'm interested in examples of what some windows apps use for a logs dir.)

OPINION: This function appends "Logs" to the `CSIDL_LOCAL_APPDATA` value for Windows and appends "log" to the user cache dir for Unix. This can be disabled with the `opinion=False` option.

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass `False` to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be `"<major>.<minor>"`. Only applied when *appname* is present.
- **opinion** (*bool, default True*) – Can be `False` to disable the appending of "Logs" to the base app data dir for Windows, and "log" to the base cache dir for Unix. See discussion above.

riptable.Utils.appdirs.**user_state_dir**(*appname=None*, *appauthor=None*, *version=None*, *roaming=False*)

Return full path to the user-specific state dir for this application.

Typical user state directories are:

- Mac OS X: same as *user_data_dir*

- Unix: ~/.local/state/<AppName> or in $XDG_STATE_HOME, if defined
- Win: same as *user_data_dir*

For Unix, we follow this Debian proposal to extend the XDG spec and support $XDG_STATE_HOME. That means ~/.local/state/<AppName> by default.

**Parameters**

- **appname** – The name of application. If None, just the system directory is returned.
- **appauthor** – Only used on Windows. The name of the appauthor or distributing body for this application. Typically it is the owning company name. This falls back to *appname*. You may pass False to disable it.
- **version** – An optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when *appname* is present.
- **roaming** (*bool, default False*) – Can be set True to use the Windows roaming appdata directory. That means that for users on a Windows network setup for roaming profiles, this user data will be sync'd on login. See Microsoft's Managing Roaming User Data Deployment Guide for a discussion of issues.

riptable.Utils.appdirs.**PY3**

riptable.Utils.appdirs.**__version__ = '1.4.4'**

riptable.Utils.appdirs.**__version_info__**

riptable.Utils.appdirs.**_get_win_folder**

riptable.Utils.appdirs.**appname = 'MyApp'**

riptable.Utils.appdirs.**os_name**

riptable.Utils.appdirs.**unicode**

### riptable.Utils.common

Logic / helper functions used throughout the riptable benchmark suite.

#### Classes

*cached_weakref_property*

### Functions

| | |
|---|---|
| *integer_range*(dtype[, include_invalid]) | Given an integer dtype, return the range [lo, hi] of values representable |
| *integer_valid_range*(dtype) | Given an integer dtype, return the range [lo, hi] of valid values |
| *rand_floats*(gen[, low, high, size, dtype, include_invalid]) | Generate a random array of floating-point values with the specified length and dtype. |
| *rand_integers*(gen[, size, dtype, include_invalid]) | Generate a random array of integers with the specified length and dtype. |
| *trial_size*([low, high, scale_factor]) | Generates a list of input sizes to benchmark against. |
| *zeros_eager*(shape, dtype) | |

### Attributes

| | |
|---|---|
| *_DOUBLING_TRIAL_16* | |
| *_DOUBLING_TRIAL_32* | |
| *_DOUBLING_TRIAL_MAX* | |
| *_INT_16_MAX* | |
| *_INT_32_MAX* | |
| *_INT_MAX* | |
| *_SEED* | Value used to seed random number generators so we get "random" data for |
| *_dtypes_by_group* | |
| *cached_property* | |
| *dtypes_by_group* | `np.typecodes` but the value for each entry is a list containing the |

**class** riptable.Utils.common.**cached_weakref_property**(*func*)

**_NO_OBJECT**

**_WEAKREF_CACHE_NAME = '_weakref_cache'**

**__get__**(*instance*, *owner=None*)

**__set_name__**(*owner*, *name*)

riptable.Utils.common.**integer_range**(*dtype*, *include_invalid=False*)

Given an integer dtype, return the range [lo, hi] of values representable by the type, optionally excluding the invalid value (if any) for the type.

riptable.Utils.common.**integer_valid_range**(*dtype*)

Given an integer dtype, return the range [lo, hi] of valid values representable by that dtype.

**Parameters**
**dtype** (*data-type*) – An integer dtype.

riptable.Utils.common.**rand_floats**(*gen*, *low=0.0*, *high=1.0*, *size=None*, *dtype=np.float64*, *include_invalid=False*)

Generate a random array of floating-point values with the specified length and dtype.

The elements of the array are drawn from the uniform distribution over the range `[low, high)`.

riptable.Utils.common.**rand_integers**(*gen*, *size=None*, *dtype=np.int64*, *include_invalid=False*)

Generate a random array of integers with the specified length and dtype.

The elements of the array will span the representable range of the dtype, optionally including the 'invalid' value for the type. The elements of the array are drawn from the 'discrete uniform' distribution.

riptable.Utils.common.**trial_size**(*low=250*, *high=_INT_16_MAX*, *scale_factor=2*)

Generates a list of input sizes to benchmark against.

riptable.Utils.common.**zeros_eager**(*shape*, *dtype*)

riptable.Utils.common.**_DOUBLING_TRIAL_16**

riptable.Utils.common.**_DOUBLING_TRIAL_32**

riptable.Utils.common.**_DOUBLING_TRIAL_MAX**

riptable.Utils.common.**_INT_16_MAX**

riptable.Utils.common.**_INT_32_MAX**

riptable.Utils.common.**_INT_MAX**

riptable.Utils.common.**_SEED = 1234**

Value used to seed random number generators so we get "random" data for the benchmarks while also allowing the benchmarks to be repeatable.

riptable.Utils.common.**_dtypes_by_group**

riptable.Utils.common.**cached_property**

riptable.Utils.common.**dtypes_by_group: Mapping[str, List[numpy.dtype]]**

`np.typecodes` but the value for each entry is a list containing the dtypes corresponding to the typecode(s) for the original entry.

**See also:**

*riptable.rt_enum.NumpyCharTypes*

## riptable.Utils.conversion_utils

### Functions

| | |
|---|---|
| *append_dataset_dict*(ds_dict, key_field_name) | Converts a dictionary of Datasets to a single Dataset appending the dictionary |
| *dataset_as_matrix*(ds[, save_metadata, column_data]) | |
| *dset_dict_to_list*(ds_dict, key_field_name[, ...]) | Converts a dict of Datasets to a list, appending the keyname as a new field key_field_name. |
| *numpy2d_to_dict*(arr, columns) | Converts arr 2D ndarray and column names to arr dict (is ordered) of ndarray's |
| *numpy_array_to_dataset*(inarray[, columns]) | |
| *numpy_array_to_dict*(inarray[, columns]) | |

riptable.Utils.conversion_utils.**append_dataset_dict**(*ds_dict*, *key_field_name*)

Converts a dictionary of Datasets to a single Dataset appending the dictionary keys as key_field_name to distinguish them. NB: This modifies the original Datasets!

**TODO: add support for harmonizing fields, e.g.,**
fill missing values and deal with differing types

**Parameters**

- **ds_dict** – dictionary of Datasets. Keys MUST be ascii strings (or bytes)!
- **key_field_name** – New column to add to each to which Dataset will be assigned the constant value of the key.

**Returns**
New dataset.

riptable.Utils.conversion_utils.**dataset_as_matrix**(*ds*, *save_metadata=True*, *column_data={}*)

riptable.Utils.conversion_utils.**dset_dict_to_list**(*ds_dict*, *key_field_name*, *allow_overwrite=False*)

Converts a dict of Datasets to a list, appending the keyname as a new field key_field_name. NB: This modifies the Datasets!

TODO: allow option of inplace or copy

**Parameters**

- **ds_dict** – dictionary of Datasets. Keys MUST be ascii strings (or bytes)!
- **key_field_name** – New column to add to each to which Dataset will be assigned the constant value of the key.
- **allow_overwrite** – Unless set to True the key_field_name may not exist in any of the input Datasets.

**Returns**
list of original Datasets _modified_.

riptable.Utils.conversion_utils.**numpy2d_to_dict**(*arr*, *columns*)

Converts arr 2D ndarray and column names to arr dict (is ordered) of ndarray's suitable for the Dataset constructor:

**Parameters**

- **arr** – numpy NxM ndarray
- **columns** – list of M column names

**Returns**

dictionary suitable for rt.Dataset constructor

Example: import numpy as np import riptable as rt from rt.Utils.conversion_utils import numpy2d_to_dict

arr = np.array([[1, 2, 3], [4, 5, 6]]) columns = ['c1', 'c2', 'c3'] dset = rt.Dataset(numpy2d_to_dict(arr, columns)) print(dset)

riptable.Utils.conversion_utils.**numpy_array_to_dataset**(*inarray*, *columns=None*)

riptable.Utils.conversion_utils.**numpy_array_to_dict**(*inarray*, *columns=None*)

## riptable.Utils.display_options

Display options for formatting and displaying numeric values, datasets, and multisets.

### Classes

| | |
|---|---|
| *DisplayOptions* | Provides display options for: |

**class** riptable.Utils.display_options.**DisplayOptions**

Bases: object

**Provides display options for:**

1) console screen for customizing width, height, and character buffers
2) row, columns, and styles when displaying datasets and multisets
3) formatting headers, footers, and general string widths
4) formatting numeric types using scientific notation or specifying precision
5) other miscellaneous display options such as prefixing group by column names

**CONSOLE_X_BUFFER**

Number of characters for buffer width on console display (default 30).

**Type**

int

**CONSOLE_X_HTML**

Number of characters for buffer width on HTML display (default 340).

**Type**

int

**CONSOLE_X**

Number of characters for console display width (default 150).

**Type**
int

**CONSOLE_Y**

Number of characters for console display height (default 25).

**Type**
int

**HTML_DISPLAY**

Toggle HTML display mode (default True).

**Type**
bool

**X_PADDING**

Number of characters for column buffer in console display (default 4).

**Type**
int

**Y_PADDING**

Number of characters for row buffer in console display (default 3).

**Type**
int

**ROW_ALL**

Toggle display of all rows for Dataset, Multiset, and Struct objects (default False).

**Type**
bool

**COL_ALL**

Toggle display of all columns for Dataset, Multiset, and Struct objects (default False).

**Type**
bool

**COL_MIN**

Minimum columns to display for Dataset, Multiset, and Struct objects (default 1).

**Type**
int

**COL_MAX**

Maximum columns to display for Dataset, Multiset, and Struct objects (default 50).

**Type**
int

**COL_T**

Number of transposed rows to display, which appear as columns for Dataset, Multiset, and Struct objects (default 8).

**Type**
int

**HEAD_ROWS**

Number of rows to display when calling head on a Dataset, Multiset, or Struct object (default 15).

**Type**
int

**TAIL_ROWS**

Number of rows to display when calling tail on a Dataset, Multiset, or Struct object (default 15).

**Type**
int

**MAX_ROWS**

Maximum number of rows to display for Dataset, Multiset, and Struct objects (default 30).

**Type**
int

**NO_STYLES**

Toggle for colors in IPython console (default False). Note, may be difficult to see with light background.

**Type**
bool

**COLOR_MODE**

Color mode for display (default None).

**Type**
DisplayColorMode or None, optional

**CUSTOM_COMPLETION = bool**

Toggle on for attribute completion results that show in alphanumeric key, attribute, then method ordering for `Dataset`, `Multiset`, `Struct` (default False).

This will override the default IPython `Completer._complete` to a custom variant that allows custom completer dispatching using the `IPython.utils.generics.complete_object` hook while preserving the custom ordering.

Caution, below are the side effects when toggling this on: - IPython `use_jedi` is set to False since this approach is currently incompatible with Jedi completion because the code is actually evaluated on TAB. - IPython `Completer._complete` is monkey patched to change use the custom completion that is backwards compatible with `Completer._complete`, but allows preserving the order. - As of 20191218, if `CUSTOM_COMPLETION` is toggled on it results in a one-time registration of custom attribute completion per IPython session as opposed to supporting deregistration.

**MAX_HEADER_WIDTH**

Maximum number of characters for header strings in a Dataset, Multiset, or Struct object (default 15).

**Type**
int

**MAX_FOOTER_WIDTH**

Maximum number of characters for footer strings in Dataset, Multiset, or Struct object (default 15).

**Type**
int

**MAX_STRING_WIDTH**

Maximum number of characters for all strings (default 15).

**Type**
int

**PRECISION**

Number of digits to display to the right of the decimal (default 2).

**E_PRECISION**

Number of digits to display to the right of the decimal in scientific notation (default 3).

**E_THRESHOLD**

Power of 10 at which the float flips to scientific notation 10**+/- (default 6).

**E_MIN**

lower limit before going to scientific notation (default None).

**Type**
int or None, optional

**E_MAX**

Upper limit before going to scientific notation (default None).

**Type**
int or None, optional

**P_THRESHOLD**

Precision threshold for area in between - so small values don't display as zero (default None).

**Type**
int or None, optional

**NUMBER_SEPARATOR**

Flag for separating thousands in floats and ints (default False).

**Type**
bool

**NUMBER_SEPARATOR_CHAR**

Character for separating `,`, , or _ (default DisplayNumberSeparator.Comma).

**Type**
DisplayNumberSeparator

**GB_PREFIX**

Prefix for column names to indicate that they are groupby keys (default "*").

**Type**
str

**save_config**()

**load_config**()

**reset_config**()

**e_min**()

**e_max**()

**p_threshold**()

**no_colors()**

**See also:**

**DisplayTable**
builds result tables with display options.

**DisplayColorMode**
enumerates supported color modes.

**get_terminal_size**
calculates console height and width.

### Examples

*CONSOLE_X_HTML* sets the number of characters for buffer width on the HTML display. Truncated characters are replaced by ellipsis.

```
>>> from riptable.Utils.display_options import DisplayOptions
>>> ds = rt.Dataset({'A':[0,6,9], 'B' : [1.2,3.1,9.6], 'C': [-1.6,2.7,4.6], 'D' :␣
↪[2.4,6.2,19.2]})
>>> ds
#   A      B       C       D
-   -   ----   -----   -----
0   0   1.20   -1.60    2.40
1   6   3.10    2.70    6.20
2   9   9.60    4.60   19.20
>>> DisplayOptions.CONSOLE_X_HTML = 25
>>> ds
#   A       B       ...     D
0   0       1.20    ...     2.40
1   6       3.10    ...     6.20
2   9       9.60    ...     19.20
```

**COLOR_MODE**

**COL_ALL = False**

**COL_MAX = 50**

**COL_MIN = 1**

**COL_T = 8**

**CONSOLE_X = 150**

**CONSOLE_X_BUFFER = 30**

**CONSOLE_X_HTML = 340**

**CONSOLE_Y = 25**

**CUSTOM_COMPLETION: bool = False**

**E_MAX**

**E_MIN**

```
E_PRECISION = 3

E_THRESHOLD = 6

GB_PREFIX = '*'

HEAD_ROWS = 15

HTML_DISPLAY = True

MAX_FOOTER_WIDTH = 15

MAX_HEADER_WIDTH = 15

MAX_ROWS = 30

MAX_STRING_WIDTH = 15

NO_STYLES = False

NUMBER_SEPARATOR = False

NUMBER_SEPARATOR_CHAR

PRECISION = 2

P_THRESHOLD

ROW_ALL = False

TAIL_ROWS = 15

X_PADDING = 4

Y_PADDING = 3

_AUTO_SAVE = False

_BAR_GRAPH = False

_BOUNDS

_CONFIG_LOADED = False

_HEAT_MAP = False

_PAINT_MAX = False

_PAINT_MIN = False

_PAINT_SIGNS = False

_PAINT_ZEROS = False

_RESET_OPTIONS = False

_TEST_FOOTERS = False

_TEST_ONE_PASS = False

_TRANSPOSE = False
```

_USERNAME

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

static _get_default_path()

static _get_username()

classmethod e_max()

Returns the upper limit integer before displaying in scientific notation.

**Returns**
**e_max** – Upper limit before going to scientific notation

**Return type**
int

classmethod e_min()

Returns the lower limit integer before displaying in scientific notation.

**Returns**
**e_min** – lower limit before going to scientific notation.

**Return type**
int

static load_config(*path=None*, *name=None*)

Load display config file from the default location if path and name are not supplied. Otherwise load display config settings using path and name. Return bool if applied correctly, otherwise return -1 if resetting display options.

**Parameters**

- **path** (*str, optional*) – Path to display config file
- **name** (*str, optional*) – Name of display config file

**Returns**
**result** – True if config was loaded correctly, otherwise False. -1 if a new default config will be saved after a reset

**Return type**
bool or int

static no_colors()

Turn off all non-default table styles.

**Return type**
str, optional

classmethod p_threshold()

Returns DisplayOption.P_THRESHOLD. Defaults to 10 ** (-1 * DisplayOptions.PRECISION) - 1e-5.

**Returns**
**p_threshold** – The precision threshold for area in between - so small values don't display as zero

**Return type**
float

**static reset_config**(*path=None*, *name=None*)

Reapply display config file from default location if path and name are not supplied. Otherwise override display config settings using path and name.

**Parameters**

- **path** (*str, optional*) – Path to display config file
- **name** (*str, optional*) – Name of display config file

**Return type**
None

**static save_config**(*path=None*, *name=None*, *force_overwrite=False*)

Save display options at the default config file path if path and name are not supplied. Otherwise save display options using path and name. If `force_overwrite` is True, then silently overwrite any previous configs at that file path, otherwise prompt for user input before overwrite.

**Parameters**

- **path** (*str, optional*) – Path to display config
- **name** (*str, optional*) – Name of display config
- **force_overwrite** (*bool*) – True to overwrite if file already exists, otherwise prompt user whether to overwrite the file.

**Returns**
**result** – True if config was saved, otherwise False.

**Return type**
bool

## riptable.Utils.ipython_utils

This module has side effects on import. If DisplayOptions.CUSTOM_COMPLETION is True then this will disable IPython Jedi and use a IPCompleter custom complete.

### Functions

| | |
|---|---|
| *enable_custom_attribute_completion*() | Registers the custom attribute completer and sets our monkey patched complete method to preserve the custom |

riptable.Utils.ipython_utils.**enable_custom_attribute_completion**()

Registers the custom attribute completer and sets our monkey patched complete method to preserve the custom attribute completer ordering. If an object has keys then the completion results will show the keys then any existing completions in case insensitive sorted order.

Notable side effect - this will disable Jedi support since IPCompleter will prefer to use Jedi completions over any custom completer.

### riptable.Utils.pandas_utils

Utility function for rt. These functions (may) have dependence on additional libraries and therefore should _NOT_ be imported in __init__.py or any other such core (like rt_appconfig.py).

## Functions

| | |
|---|---|
| *dataset_as_pandas_df*(ds) | This function is deprecated, please use riptable.Dataset.as_pandas_df method. |
| *dataset_from_pandas_df*(df[, tz]) | This function is deprecated, please use riptable.Dataset.from_pandas. |
| *fastarray_to_pandas_series*(arr[, unicode, use_nullable]) | |
| *pandas_series_to_riptable*(series[, tz]) | |

riptable.Utils.pandas_utils.**dataset_as_pandas_df**(*ds*)

This function is deprecated, please use riptable.Dataset.as_pandas_df method.

Create a pandas DataFrame from a riptable Dataset. Will attempt to preserve single-key categoricals, otherwise will appear as an index array. Any bytestrings will be converted to unicode.

**Parameters**
**ds** (Dataset) – The riptable Dataset to be converted.

**Return type**
DataFrame

**See also:**

riptable.Dataset.to_pandas

riptable.Utils.pandas_utils.**dataset_from_pandas_df**(*df*, *tz='UTC'*)

This function is deprecated, please use riptable.Dataset.from_pandas.

Creates a riptable Dataset from a pandas DataFrame. Pandas categoricals and datetime arrays are converted to their riptable counterparts. Any timezone-unaware datetime arrays (or those using a timezone not recognized by riptable) are localized to the timezone specified by the tz parameter.

**Recognized pandas timezones:**
UTC, GMT, US/Eastern, and Europe/Dublin

**Parameters**

- **df** (*DataFrame*) – The pandas DataFrame to be converted
- **tz** (*string*) – A riptable-supported timezone (‘UTC’, ‘NYC’, ‘DUBLIN’, ‘GMT’)

**Return type**
*Dataset*

**See also:**

riptable.Dataset.from_pandas, riptable.Dataset.to_pandas

riptable.Utils.pandas_utils.**fastarray_to_pandas_series**(*arr*, *unicode=True*, *use_nullable=True*)

riptable.Utils.pandas_utils.**pandas_series_to_riptable**(*series*, *tz='UTC'*)

### riptable.Utils.rt_display_nested

#### Classes

| | |
|---|---|
| *AttributeTraversal* | Attribute traversal. |
| *BoxStyle* | A rendering style that uses box draw characters and a common layout. |
| *DictTraversal* | Traversal suitable for a dictionary. Keys are tree labels, all values |
| *DisplayNested* | |
| *KeyArgsConstructor* | |
| *LeftAligned* | Creates a renderer for a left-aligned tree. |
| *Style* | Rendering style for trees. |

#### Functions

| | |
|---|---|
| *_arr_info*(arr) | |
| *_cat_info*(arr) | |
| *_default_info*(item) | |
| *_mask_flags*(flagnum) | |
| *_scalar_info*(item) | |
| *treedir*(path[, name]) | |

#### Attributes

| |
|---|
| *BOX_LIGHT* |

**class** riptable.Utils.rt_display_nested.**AttributeTraversal**(***kwargs*)

Bases: *KeyArgsConstructor*

Attribute traversal.

Uses an attribute of a node as its list of children.

**attribute = 'children'**

**get_children**(*node*)

class riptable.Utils.rt_display_nested.**BoxStyle**(***kwargs*)

Bases: *Style*

A rendering style that uses box draw characters and a common layout.

**gfx**

**horiz_len = 4**

**indent = 1**

**label_space = 1**

**child_head**(*label*)

Render a node label into final output.

**child_tail**(*line*)

Render a node line that is not a label into final output.

**last_child_head**(*label*)

Like `child_head()` but only called for the last child.

**last_child_tail**(*line*)

Like `child_tail()` but only called for the last child.

class riptable.Utils.rt_display_nested.**DictTraversal**(***kwargs*)

Bases: *KeyArgsConstructor*

Traversal suitable for a dictionary. Keys are tree labels, all values must be dictionaries as well.

**get_children**(*node*)

**get_root**(*tree*)

**get_text**(*node*)

class riptable.Utils.rt_display_nested.**DisplayNested**

**property fmtend**

**property fmtstart**

**inline_svg**

**_build_nested_ascii**(*data*, *name=None*, *showpaths=False*, *info=False*)

**_build_nested_html**(*data*, *name=None*)

**_map_asciitree**(*data*, *structure*, *name=None*, *info=False*)

**_map_full_paths**(*data*, *structure*, *name=None*, *prefix=''*)

**_map_htmltree**(*data*, *structure*, *name=None*, *html_str=[]*, *showicon=True*)

**build_nested_html**(*data={}*, *name=None*)

**build_nested_string**(*data={}*, *name=None*, *showpaths=False*, *info=False*)

class riptable.Utils.rt_display_nested.**KeyArgsConstructor**(***kwargs*)

Bases: object

class riptable.Utils.rt_display_nested.**LeftAligned**(***kwargs*)

Bases: *KeyArgsConstructor*

Creates a renderer for a left-aligned tree.

Any attributes of the resulting class instances can be set using constructor arguments.

**draw**

The draw style used. See Style.

**traverse**

Traversal method. See Traversal.

**__call__**(*tree*)

Render the tree into string suitable for console output.

**Parameters**
**tree** – A tree.

**render**(*node*)

Renders a node. This function is used internally, as it returns a list of lines. Use __call__() instead.

class riptable.Utils.rt_display_nested.**Style**(***kwargs*)

Bases: *KeyArgsConstructor*

Rendering style for trees.

**label_format = '{}'**

**child_head**(*label*)

Render a node label into final output.

**child_tail**(*line*)

Render a node line that is not a label into final output.

**last_child_head**(*label*)

Like child_head() but only called for the last child.

**last_child_tail**(*line*)

Like child_tail() but only called for the last child.

**node_label**(*text*)

Render a node text into a label.

riptable.Utils.rt_display_nested.**_arr_info**(*arr*)

riptable.Utils.rt_display_nested.**_cat_info**(*arr*)

riptable.Utils.rt_display_nested.**_default_info**(*item*)

riptable.Utils.rt_display_nested.**_mask_flags**(*flagnum*)

riptable.Utils.rt_display_nested.**_scalar_info**(*item*)

riptable.Utils.rt_display_nested.**treedir**(*path*, *name=None*)

riptable.Utils.rt_display_nested.**BOX_LIGHT**

riptable.Utils.rt_display_properties

## Classes

| | |
|---|---|
| *DisplayConvert* | Will analyze an array of a default type and return the appropriate conversion function. |
| *ItemFormat* | A container for display options for different data types in FastArray and numpy arrays. |

## Functions

| | |
|---|---|
| *format_scalar*(sc) | Convert a scalar to a string for display - the same way it would be if contained in a FastArray. |
| *get_array_formatter*(arr) | FastArray/subclasses have display_query_properties defined for custom string formatting. |
| *trim_string*(i, itemformat) | If maxwidth was specified and is larger than the global default, it will be used instead. |

## Attributes

| |
|---|
| *default_item_formats* |

**class** riptable.Utils.rt_display_properties.**DisplayConvert**

Will analyze an array of a default type and return the appropriate conversion function. Anything that subclasses from FastArray will always fall back on the dtype of its underlying array.

**ConvertFuncCache**

**ConvertTypeCache**

**Verbose = 0**

**convert_func_dict**

**static convertBool**(*b*, *itemformat*)

**static convertBytes**(*i*, *itemformat*)

**static convertDefault**(*i*, *itemformat*)

**static convertFloat**(*f*, *itemformat*)

**static convertInt**(*i*, *itemformat*)

**static convertMultiDims**(*i*, *itemformat*)

For displaying multi-dimensional arrays (currently only supports 2-dims). ItemFormat object contains a convert function for the dtype of the multidimensional array.

**static convertRecord**(*i*, *itemformat*)

**static convertString**(*i*, *itemformat*)

**static get_display_array_type**(*dtype*)

For FastArray and numpy array of basic type (not for objects that are subclasses of FastArray)

**static get_display_convert**(*arr*)

class riptable.Utils.rt_display_properties.**ItemFormat**(*length=DisplayLength.Short*, *justification=DisplayJustification.Right*, *invalid=None*, *can_have_spaces=False*, *html=False*, *color=DisplayColumnColors.Default*, *decoration=None*, *convert=None*, *convert_format=None*, *format_string=None*, *timezone_str=None*, *maxwidth=None*)

A container for display options for different data types in FastArray and numpy arrays. Basic numpy types have defaults. (see below) New types (subclassed from FastArray) will be queried to get formatting options. New types have the option of overwriting display_query_properties to set their own defaults.

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**copy**()

**summary**()

riptable.Utils.rt_display_properties.**format_scalar**(*sc*)

Convert a scalar to a string for display - the same way it would be if contained in a FastArray. Returns the converted scalar.

riptable.Utils.rt_display_properties.**get_array_formatter**(*arr*)

FastArray/subclasses have display_query_properties defined for custom string formatting. Numpy arrays have defaults in DisplayConvert. Returns ItemFormat object and display function for items in array based on type.

riptable.Utils.rt_display_properties.**trim_string**(*i*, *itemformat*)

If maxwidth was specified and is larger than the global default, it will be used instead. See also ColumnStyle()

riptable.Utils.rt_display_properties.**default_item_formats**

## riptable.Utils.rt_metadata

### Classes

| |
|---|
| *MetaData* |

### Functions

| | |
|---|---|
| *meta_from_version*(cls, vnum) | Returns a dictionary of meta data defaults. |

### Attributes

| |
|---|
| *META_VERSION* |

**class** riptable.Utils.rt_metadata.**MetaData**(*metadict={}*)

**property dict**

**property itemclass**

Starting 4/29/2019 item classes will be saved as strings in json meta data in `classname`. For backwards compatibility, will also check *typeid*. The TypeId class is an enum of ItemClass -> typeid. Both will lookup the items class in the TypeRegister, which holds classname -> itemclass.

**property name**

**property string**

**property typeid**

**default_dict**

**__getitem__**(*idx*)

**__repr__**()

Return repr(self).

**__setitem__**(*idx*, *value*)

**__str__**()

Return str(self).

**get**(*key*, *default*)

**setdefault**(*k*, *v*)

riptable.Utils.rt_metadata.**meta_from_version**(*cls*, *vnum*)

Returns a dictionary of meta data defaults.

riptable.Utils.rt_metadata.**META_VERSION = 0**

#### riptable.Utils.teamcity_helper

TeamCity utility that wraps TeamCity build metadata. TeamCity server predefined build parameters can be found here: https://www.jetbrains.com/help/teamcity/predefined-build-parameters.html

### Functions

| | |
|---|---|
| *get_build_conf_name*() | |
| *is_running_in_teamcity*() | |

riptable.Utils.teamcity_helper.**get_build_conf_name**()

riptable.Utils.teamcity_helper.**is_running_in_teamcity**()

#### riptable.Utils.terminalsize

### Functions

| | |
|---|---|
| *get_terminal_size*() | getTerminalSize() |

riptable.Utils.terminalsize.**get_terminal_size**()

getTerminalSize() - get width and height of console - works on linux,os x,windows,cygwin(windows) originally retrieved from: http://stackoverflow.com/questions/566746/how-to-get-console-window-width-in-python

Returns: (x,y) in screen size Returns: (None, None) on failure

## 2.1.2 riptable.numba

### Submodules

#### riptable.numba.indexing

Indexing-related helper functions for use when implementing other numba-based functions.

### Functions

| | |
|---|---|
| *deref_idx*(deref, idx) | A numba function for turning an 'indirect' index into a true index if a deference idx exists. |
| *scalar_or_lookup*(val, idx) | Allows a numba-based function to accept either a scalar value or an array for some parameter. |

riptable.numba.indexing.**deref_idx**(*deref*, *idx*)

A numba function for turning an 'indirect' index into a true index if a deference idx exists.

This main use of this function is to seamlessly deal with the existence/non-existence of `iGroup` (from a `Grouping` object).

- When writing a function to operate on a single array, there are no groups. The data is contiguous, so there is no `iGroup` and an index doesn't need to be 'dereferenced' to get a 'real' index.
- When writing a function to operate over `Grouping` data, we need to use `Grouping.igroup` to turn a 0-based index (scalar) within a particular group's data into an 0-based index within the larger array whose shape matches the `Grouping` object itself.

**Parameters**

- **deref** (*None or Array*) – an array to use for dereferencing or None.
- **idx** (*Integer*) – the index to grab (or to be returned if deref is None).

**Returns**
`idx` if `deref` is `None`; otherwise `deref[idx]`.

**Return type**
int

riptable.numba.indexing.**scalar_or_lookup**(*val*, *idx*)

Allows a numba-based function to accept either a scalar value or an array for some parameter. Scalars are passed through, but if an array is provided, the array element at the specified index is returned.

For example, in a numba-based function operating over grouped data (with a `Grouping`), the function could accept a scalar parameter; by using *scalar_or_lookup*, the code can easily accept either a scalar (to be applied to all groups) or an array containing a specific value for each group.

**Parameters**

- **val** –
- **idx** –

**Return type**
retval

## riptable.numba.invalid_values

Functions for working with numba.

Functionality includes:

- Functions for numba <-> riptable interop
- Convenience functions and decorators used when implementing numba-accelerated functions.

### Notes

Many functions in this module implement both a standard Python version and an overload using `numba.extending.overload`. This is done to allow the code to work when `NUMBA_DISABLE_JIT=1` is specified on the command line, e.g. to allow functions to be debugged.

### Functions

| | |
|---|---|
| *get_invalid*(x) | A function for getting the invalid for a type of element. |
| *get_max_valid*(x) | Get the maximum valid value for the dtype of an array or scalar. |
| *get_min_valid*(x) | Get the minimum valid value for the dtype of an array or scalar. |
| *is_valid*(x) | A function for checking if data is valid. This works for both floats and integers. |

riptable.numba.invalid_values.**get_invalid**(*x*)

A function for getting the invalid for a type of element.

This works for both floats and integers.

- For floats, the invalid is NaN.
- For signed integers, the invalid is the most NEGATIVE value of the type.
- For unsigned integers, the invalid is the most POSITIVE value of the type.

For arrays, the invalid of the dtype of the array is returned.

**Parameters**
**x** – An element of the type you want the invalid for

**Return type**
The invalid value for **x**'s type/dtype.

riptable.numba.invalid_values.**get_max_valid**(*x*)

Get the maximum valid value for the dtype of an array or scalar.

This function supports integer and floating-point values.

**Parameters**
**x** – An array or scalar value of the type you want the maximum valid value for.

**Returns**
The maximum valid value for **x**'s dtype.

**Return type**
max_valid

riptable.numba.invalid_values.**get_min_valid**(*x*)

Get the minimum valid value for the dtype of an array or scalar.

**Parameters**
**x** – An array or scalar value of the type you want the minimum valid value for.

**Returns**
The minimum valid value for **x**'s dtype.

**Return type**
min_valid

riptable.numba.invalid_values.**is_valid**(*x*)

A function for checking if data is valid. This works for both floats and integers.

- For floats, the invalid is NaN.
- For signed integers, the invalid is the most NEGATIVE value of the type.
- For unsigned integers, the invalid is the most POSITIVE value of the type.

**Parameters**
**x** – The value to check

**Return type**
A bool for whether the data is valid.

## 2.2 Submodules

### 2.2.1 riptable.config

General-purpose settings for configuring riptable behavior.

This module provides a class encapsulating riptable settings and feature flags, along with functions for retrieving a top-level, process-wide instance of the class.

#### Classes

| | |
|---|---|
| *Settings* | Encapsulates process-wide settings and feature-flags for riptable. |

#### Functions

| | |
|---|---|
| *get_global_settings*() | Get the global (process-wide) *Settings* instance. |

**class** riptable.config.**Settings**

Bases: NamedTuple

Encapsulates process-wide settings and feature-flags for riptable.

**enable_numba_cache: bool = False**

Controls whether the numba JIT cache is enabled for functions within riptable. This is disabled (False) by default because the caching can lead to occasional segfaults in numba-compiled code for some users, possibly caused by a race condition or filesystem non-atomicity.

riptable.config.**get_global_settings**()

Get the global (process-wide) *Settings* instance.

**Returns**
Global (process-wide) settings.

**Return type**
*Settings*

### 2.2.2 riptable.conftest

#### Functions

| | |
|---|---|
| *docstring_imports*(doctest_namespace) | |
| *docstring_merge_datasets*(doctest_namespace) | |
| *get_doctest_dataset_data*() | |
| *register_null_log_handler*() | Session-level fixture that installs a top-level log handler (or formatter) at the DEBUG level (or anything higher than NOTSET). |

riptable.conftest.**docstring_imports**(*doctest_namespace*)

riptable.conftest.**docstring_merge_datasets**(*doctest_namespace*)

riptable.conftest.**get_doctest_dataset_data**()

riptable.conftest.**register_null_log_handler**()

Session-level fixture that installs a top-level log handler (or formatter) at the DEBUG level (or anything higher than NOTSET). It formats the messages in the typical way then just throws away the result. The idea is to just force all logging code to run, even if guarded with something like `if logger.isEnabledFor(logging.DEBUG):` so we exercise that code within the tests. This helps guard against bad logging code that's only discovered when debug-level logging is enabled.

### 2.2.3 riptable.rt_accum2

#### Classes

| | |
|---|---|
| *Accum2* | The Accum2 object is very similar to a GroupBy object that has been initialized with a multikey Categorical. |

**class** riptable.rt_accum2.**Accum2**(*cat_rows*, *cat_cols*, *filter=None*, *showfilter=False*, *ordered=None*, *sort_gb=False*, *totals=True*, *ylabel=None*)

Bases: *riptable.rt_groupbyops.GroupByOps*, *riptable.rt_fastarray.FastArray*

The Accum2 object is very similar to a GroupBy object that has been initialized with a multikey Categorical.

The Accum2 object is very similar to a GroupBy object that has been initialized with a multikey Categorical. Because it also inherits from GroupByOps, all calculations will be sent to _calculate_all in a Grouping object. Accum2 generates a single array of data, and splits it into multiple columns - one for each x-axis bin. There is always an invalid bin, but it is omitted by default when the single array is split into columns. Datasets resulting from an Accum2 groupby calculation will be displayed with a footer row of column totals, and an additional vertical column of row totals.

In addition to inheriting from GroupByOps, Accum2 also inherits from FastArray. This way, it can exist as a column in a Dataset. Its cell data will appear as a tuple of values from its X and Y axis.

**Parameters**

- **cat_rows** (Categorical) – Categorical for the rows axis, or an array which will be converted to a Categorical.
- **cat_cols** (Categorical) – Categorical for the column axis, or an array which will be converted to a Categorical.
- **Keywords** –
- **--------** –
- **invalid** (*defaults to False. Set to True to show filtered columns*) –
- **ordered** (*defaults to None. See Categorical*) –
- **sort_gb** (*defaults to False. See Categorical*) –
- **ylabel** (*defaults to None. Set to a string to override the name of the left column*) –
- **totals** (*defaults to True.*) –
- **option** (*There is no sort_display*) –

**Returns**

- *Accum2 object which can be used to perform calculations*
- *Accum2 subclasses from FastArray and can be added to a dataset*
- *Accum2.operation is then supported. Accum2(catx, caty).min(array1)*
- **See** (*groupbyops*)

### Examples

```
>>> int_fa = FastArray([1,2,3,4]*4)
>>> str_fa = FastArray(['a','b','c','d','b','c','d','a','c','d','b','a','d','a','b',
↪'c'])
>>> data_col = np.random.rand(16)*10
>>> data_col
array([6.7337479 , 1.69561884, 8.20657899, 6.12821287, 3.95380641,
       1.06706672, 9.51679965, 3.57184704, 7.86268264, 9.0136061 ,
       2.12355667, 3.64954958, 8.40952542, 0.06431684, 9.52872172,
       3.94938333])   #random
```

```
>>> c_x = Categorical(str_fa)
>>> c_y = Categorical(int_fa)
>>> ac = Accum2(c_x, c_y)
>>> ac
Accum2 Keys
 X:[b'a' b'b' b'c' b'd']
 Y:{'key_0': FastArray([1, 2, 3, 4])}
 Bins:25   Rows:16

*YLabel   a   b   c   d   Total
```

(continues on next page)

(continued from previous page)

```
-------   -   -   -   -   -----
      1   1   1   1   1       4
      2   1   1   1   1       4
      3   0   2   1   1       4
      4   2   0   1   1       4
-------   -   -   -   -   -----
  Total   4   4   4   4      16
```

```
>>> ac.sum(data_col)
*YLabel       a       b       c       d   Total
-------   -----   -----   -----   -----   -----
      1    6.73    3.95    7.86    8.41   26.96
      2    0.06    1.70    1.07    9.01   11.84
      3    0.00   11.65    8.21    9.52   29.38
      4    7.22    0.00    3.95    6.13   17.30
-------   -----   -----   -----   -----   -----
  Total   14.02   17.30   21.09   33.07   85.48
```

**property gb_keychain**

Request a GroupByKeys from the y-axis categorical.

This provides unique keys, a possible sorted index, and the ability to add a filtered bin to the final table from groupby calculations.

**property gbkeys**

**property ikey**

**property isortrows**

**property ncountgroup**

Grouping.ncountgroup

**Type**
See

**property ncountkey**

Grouping.ncountkey

**Type**
See

**property size**

**ACCUM_X_MAX: int = 10000**

**DebugMode: bool = False**

**__del__**()

Called when a Categorical is deleted.

**__getitem__**(*fld*)

Bracket indexing for Accum2.

**__len__**()

__repr__()

Return repr(self).

__str__()

Return str(self).

classmethod _accum1_pass(*cat*, *origarr*, *funcNum*, *showfilter=False*, *filter=None*, *func_param=0*, ***kwargs*)

internal call to calculate the Y or X summary axis the filter muse be passed correctly returns array with result of operation, size of array is number of uniques

classmethod _add_totals(*cat_rows*, *newds*, *name*, *totalsX*, *totalsY*, *totalOfTotals*)

Adds a summary column on the right (totalsY) Adds a footer on the bottom (totalsX)

classmethod _apply_2d_operation(*func*, *imatrix*, *showfilter=True*, *filter_rows=None*, *filter_cols=None*)

Called from routines like sum or min where we can make one pass

If there are badrows, then filter_rows is set to the row indexes that are bad If there are badcols, then filter_cols is set to the col indexes that are bad filter_rows is a fancy index or none

_build_sds_meta_data(*name*, ***kwargs*)

_build_string()

classmethod _calc_badslots(*cat*, *badslots*, *filter*, *wantfancy*)

internal routine will combine (row or col filter) badslots with common filter

if there are not badslots, the common filter is returned otherwise a new filter is returned the filter is negative (badslots locations are false)

if wantfancy is true, returns fancy index to cols or rows otherwise full boolean mask combined with existing filter (if exists)

classmethod _calc_multipass(*cat_cols*, *cat_rows*, *newds*, *origarr*, *funcNum*, *func*, *imatrix*, *name=None*, *showfilter=False*, *filter=None*, *badrows=None*, *badcols=None*, *badcalc=True*, ***kwargs*)

For functions that require multiple passes to get the proper result. such as mean or median.

If the grid is 7 x 11: there will be 77 + 11 + 7 + 1 => 96 passes

**Parameters**

- **func** (*userfunction to call calculate*) –
- **name** (*optional column name (otherwise function name used)*) –
- **badrows** (*optional list of bad row keys, will be combined with filter*) –
- **badcols** (*optional list of bad col keys, will be combined with filter*) –
- **filter)** (*badrows/cols is just the keys that are bad (not a boolean*) –
- **badrows=['AAPL'** (*for example*) –
- **'GOOG']** –
- **take** (*Need new algo to*) – bad bins + ikey + existing boolean filter ==> create a new boolean filter walk ikey, see if bin is bad in lookup table, if so set filter to False else copy from existing filter value

**classmethod _calc_onepass**(*cat_cols*, *cat_rows*, *newds*, *origarr*, *funcNum*, *func*, *imatrix*, *name=None*, *showfilter=False*, *filter=None*, *badrows=None*, *badcols=None*, *badcalc=True*, ***kwargs*)

For functions such as sum or min that require one pass to get the proper result.

The first pass calculates all the cells. Once the cells are calculated, an imatrix is made. Since functions like sum or min can calculate proper values for horizontal or vertical operations without making another pass, we use the imatrix to calculate the rest.

The user may also pass in badrows or badcols, or both. When badrows is passed, the CELLS for that row are still calculated normally. However, the totalOfTotals will not include the badrows or cols.

**_calculate_all**(*funcNum*, **args*, *func_param=0*, ***kwargs*)

Can be called from apply_reduce

**_finish_calculate_all**(*origdict*, *accum_dict*, *funcNum*, *func_param=0*, *tups=0*, *transform=False*, ***kwargs*)

**Parameters**

- **origdict** (*original dataset input*) –
- **accum_list** (*input data we can calculate on*) –
- **funcNum** (*internal riptable groupby function number OR*) – a callable reduce function
- **func_param** (*optional, parameters for the function*) –

**_get_gbkeyname**()

**_get_gbkeys**(*showfilter=False*)

**_internal_getitem**(*matrix_index*)

**classmethod _load_from_sds_meta_data**(*name*, *arr*, *cols*, *meta*)

**_make_imatrix**(*input_arr*, *col_keys*, *row_keys*, *showfilter=False*)

Return a Fortran-ordered 2d matrix.

```
if showfilter is False, the first column is removed
    shape is (row_keys.unique_count+1, col_keys.unique_count)
else if showfilter is True
    shape is (row_keys.unique_count +1, col_keys.unique_count+1)
```

**_stack_dataset**(*arr*, *origarr*, *funcNum*, *showfilter=False*, *tups=0*, ***kwargs*)

Accum2 uses a single array but returns a dataset that is stacked. The long column is unrolled into columns.

**Parameters**

- **arr** –
- **origarr** –
- **funcNum** –
- **showfilter** (*bool*) –
- **kwargs** (*dict-like*) – Keyword args to pass to the function specified by `funcNum`.

**apply_reduce**(*userfunc*, **args*, *dataset=None*, *label_keys=None*, *func_param=None*, *dtype=None*, *transform=False*, ***kwargs*)

Accum2:apply_reduce calls Grouping:apply_helper

**Parameters**

- **userfunc** (*callable*) – A callable that takes a contiguous array as its first argument, and returns a scalar In addition the callable may take positional and keyword arguments.
- **args** – Used to pass in columnar data from other datasets
- **dataset** (*None*) – User may pass in an entire dataset to compute.
- **label_keys** (*None*) – Not supported, will use the existing groupby keys as labels.
- **func_param** (*tuple, optional*) – Set to a tuple to pass as arguments to the routine.
- **dtype** (*str or np.dtype, optional*) – Change to a numpy dtype to return an array with that dtype. Defaults to None.
- **transform** (*bool*) – Set to True to re-expand the results of the calculation. Defaults to False.
- **filter** –
- **kwargs** – Optional positional and keyword arguments to pass to `userfunc`

### Notes

See Grouping.apply_reduce

**count**(***kwargs*)

Compute count of group

**display_convert_func**(*index*, *itemformat*)

**display_query_properties**()

Take over display query properties from parent class FastArray.

When displayed in a Dataset, Accum2 data will be displayed as a tuple composite of its categorical (x,y) bin values.

**make_dataset**(*arr*, *showfilter=False*)

**Parameters**

**arr** (*input array of data*) –

**Returns**

- *ds*
- *col_keys*
- *row_keys*

### 2.2.4 riptable.rt_accumtable

#### Classes

| | |
|---|---|
| *AccumTable* | AccumTable is a wrapper on Accum2 that enables the creation of tables that |

#### Functions

| | |
|---|---|
| *accum_cols*(cat, val_list[, name_list, filt_list, ...]) | Compute multiple accum calculations on the same categorical label, output as a single dataset. |
| *accum_ratio*(cat1[, cat2, val1, val2, filt1, filt2, ...]) | Compute a bucketed ratio of two accums, using AccumTable. |
| *accum_ratiop*(cat1[, cat2, val, filter, func, norm_by, ...]) | Compute an internal ratio, either by Total (T), Row (R), or Column (C). |

**class** riptable.rt_accumtable.**AccumTable**(*cat_rows*, *cat_cols*, *filter=None*, *showfilter=False*)

Bases: *riptable.rt_accum2.Accum2*

AccumTable is a wrapper on Accum2 that enables the creation of tables that combine the results of multiple reductions generated from the Accum2 object. The three parts of a table generated by the AccumTable gen() method are these:

- **Inner Table** - a table of values indexed by row labels and column key names. A generated table contains only one inner table, but any number of inner tables may be created and used to create margin columns and footer rows, or as a reference for display formatting (future functionality).
- **Margin Columns** - columns of values on the right margin, associated with an inner table, and indexed by and representing a value associated with a given row label. Call set_margin_columns() to adjust them.
- **Footer Rows** - rows of values on the bottom margin, associated with an inner table and indexed by and representing a value associated with a given column key. Call set_footer_rows() to adjust them.

**Parameters**

- **cat_rows** (*Categorical or an array converted to same*) – The array used to create the row labels in the AccumTable
- **cat_cols** (*Categorical or an array converted to same*) – The array used to create the column keys in the AccumTable
- **filter** (*ndarray*) – Boolean mask array applied as filter before constructing the groupings
- **showfilter** (*bool*) – Whether to include groupings whose values were all filtered out

**__getitem__**(*index*)

**Parameters**
**index** (*str*) – Inner table name

**Returns**
The specified inner table

**Return type**
*Dataset*

**Raises**
IndexError – If index is not a string (table name).

__repr__()
Return a string representation of the object

**Returns**
The repr string

**Return type**
str

__setitem__(*name*, *ds*)

**Parameters**

- **name** (*str*) – Inner table name
- **ds** (Dataset) – The dataset

**Raises**

- IndexError – If name is not a string (table name).
- ValueError – If ds is not a Dataset

_rename_summary_row_and_col(*ds*, *new_name*)

**Parameters**

- **ds** (Dataset) – The dataset
- **new_name** (*str*) – the new name for the summary column and footer row

**Return type**
*Dataset*

gen(*table_name=None*, *format=None*, *ref_table=None*, *remove_blanks=True*)
Generate an AccumTable view.

**Parameters**

- **table_name** (*string or tuple (not implemented yet)*) – The name of the AccumTable table to display, or a tuple of table names if more than one value is to be displayed in each cell (not implemented yet).
- **yet)** (*ref_table (not implemented*) – A dictionary used to specify the formatting of each cell in the table. The keys are formatting types, such as 'bold', 'color', and 'background', and the values are functions that are applied to the value (or tuple) in each table cell to determine the applicability of a formatting type. For example, once could set format={'bold': lambda v: v > 0} to make all positive values in the table bold.
- **yet)** – The name of the AccumTable table, or a Dataset of the same shape, to be referenced for formatting the displayed table (not implemented yet).
- **remove_blanks** (*bool*) – Do not display rows or columns containing all zeros or NaNs

**Returns**
The generated table

**Return type**
*Dataset*

### Examples

View the pnl values in the table, coloring negative values red (not implemented yet):

```
>>> at.gen('pnl', format={'color': lambda v: return 'red' if v < 0 else 'black'}
↪)
```

**set_footer_rows**(*rows*)

Specify the names of the inner tables whose footer rows should appear in the generated AccumTable view.

**Parameters**
**rows** (*list*) – The list of inner table names, in order.

**set_margin_columns**(*cols*)

Specify the names of the inner tables whose margin columns should appear in the generated AccumTable view.

**Parameters**
**cols** (*list of str*) – The list of inner table names, in order.

riptable.rt_accumtable.**accum_cols**(*cat*, *val_list*, *name_list=None*, *filt_list=None*, *func_list='nansum'*, *remove_blanks=False*)

Compute multiple accum calculations on the same categorical label, output as a single dataset.

**Parameters**

- **cat** (Categorical) – Categorical label to group by
- **val_list** (*list*) – List of data columns. If an element is a two-element list itself, a ratio will be calculated. If an element is a two-element list of type [val, 'p'], an accum_ratiop-style percentile will be calculated
- **name_list** (*list*) – List of column names in the eventual dataset. Defaults to colN.
- **filt_list** (*list*) – List of filters, either one for all or one for each. Defaults to truecol.
- **func_list** (*str or list of str*) – String of function name (or list of strings of function names) to pass into AccumTable call, either one for all or one for each. Defaults to 'nansum'.
- **remove_blanks** (*bool*) – If set to true, blanks will be removed from the output. Defaults to True.

**Returns**
Accum2 view of calculated data.

**Return type**
*Dataset*

riptable.rt_accumtable.**accum_ratio**(*cat1*, *cat2=None*, *val1=None*, *val2=None*, *filt1=None*, *filt2=None*, *func1='nansum'*, *func2=None*, *return_table=False*, *include_numer=False*, *include_denom=True*, *remove_blanks=False*)

Compute a bucketed ratio of two accums, using AccumTable.

**Parameters**

- **cat1** (Categorical) – First categorical label to group by.
- **cat2** (Categorical, *optional*) – Second categorical label to group by.
- **val1** – Numerator data
- **val2** – Denominator data

- **filt1** – Filter for val1 data
- **filt2** – Filter for val2 data. Optional, defaults to filter1
- **func1** – String of function name to pass into numerator AccumTable call
- **func2** – String of function name to pass into denominator AccumTable call. Defaults to func1.
- **include_numer** (*bool*) – If set to True, include the totals from the numerator data in the output. Ignored if return_table is True.
- **include_denom** (*bool*) – If set to True, include the totals from the denominator data in the output. Ignored if return_table is True.
- **return_table** (*bool*) – If set to True, returns the whole AccumTable instead of just the gen'd ratio data.
- **remove_blanks** (*bool*) – If set to True, blanks will be removed from the output.

**Returns**
Either a view of the ratio data, or the entire AccumTable, depending on return_table flag.

**Return type**
*Dataset* or *AccumTable*

riptable.rt_accumtable.**accum_ratiop**(*cat1*, *cat2=None*, *val=None*, *filter=None*, *func='nansum'*, *norm_by='T'*, *include_total=True*, *remove_blanks=False*, *filt=None*)

Compute an internal ratio, either by Total (T), Row (R), or Column (C).

**Parameters**

- **cat1** (Categorical) – First categorical label to group by
- **cat2** (Categorical, *optional*) – Second categorical label to group by.
- **val** – Data column
- **filter** (*boolean column*) – Filter for var data. Replacing filt.
- **func** (*str*) – String of function name to pass into AccumTable call
- **norm_by** (*{'T', 'C', 'R'}*) – what to use as the denominator
- **include_total** (*bool*) – Include the total amounts in addition to the ratios, defaults to True.
- **remove_blanks** (*bool*) – If set to True, blanks will be removed from the output; defaults to True.
- **filt** – DEPRECATED FOR "filter".

**Returns**
AccumTable view of ratios

**Return type**
*Dataset*

### 2.2.5 riptable.rt_algos

#### Functions

| | |
|---|---|
| *merge_index*(indices, listcats[, idx_cutoffs, ...]) | For hstacking Categoricals or fixing indices in a categorical from a stacked .sds load. |

riptable.rt_algos.**merge_index**(*indices*, *listcats*, *idx_cutoffs=None*, *unique_cutoffs=None*, *from_mapping=False*, *stack=True*)

For hstacking Categoricals or fixing indices in a categorical from a stacked .sds load.

Supports categoricals from single array or dictionary mapping.

**Parameters**

- **indices** (*single stacked array or list of indices*) – if single array, needs idx_cutoffs for slicing
- **listcats** (*list*) – list of stacked unique category arrays (needs unique_cutoffs) or list of lists of uniques
- **idx_cutoffs** – (TODO)
- **unique_cutoffs** – (TODO)
- **from_mapping** (*bool*) – (TODO)
- **stack** (*bool*) – (TODO)

**Returns**

- *Tuple containing two items*
- *- list of fixed indices, or array of fixed contiguous indices.*
- *- stacked unique values*

### 2.2.6 riptable.rt_bin

#### Functions

| | |
|---|---|
| *cut*(x, bins[, labels, right, retbins, precision, ...]) | Partition values into discrete bins. |
| *qcut*(x, q[, labels, retbins, precision, duplicates, ...]) | Quantile-based discretization function. |
| *quantile*(x, q[, interpolation_method]) | Compute sample quantile or quantiles of the input array. For example, q=0.5 computes the median. |

riptable.rt_bin.**cut**(*x*, *bins*, *labels=True*, *right=True*, *retbins=False*, *precision=3*, *include_lowest=False*, *filter=None*, *duplicates='raise'*)

Partition values into discrete bins.

This function is also useful for converting a continuous variable to a *Categorical* variable.

Values can be partitioned into a specified number of equal-width bins or bins bounded by specified endpoints.

For bins bounded by specified endpoints, values that fall outside of the bin range are put into the 'Filtered' bin, which is mapped to 0 in the returned *Categorical*. See the exception (caused by a known issue) noted in the description of the `right` parameter, below.

Other known issues are noted in the parameter descriptions and shown in the Examples section, below.

**Parameters**

- **x** (array) – The input array to be partitioned. Must be 1-dimensional. NaN values are put into the 'Filtered' bin.
- **bins** (int or sequence of scalar) – Determines how bins are created:
  - int: Creates int number of equal-width bins in the range of x.
  - sequence of scalar: Creates bins based on the specified endpoints. Bins can be of non-uniform width.
- **right** (bool, default True) – Indicates whether each bin includes its right endpoint or not. Note: Until known issues are fixed:
  - Each bin includes its right endpoint, even if right is set to False.
  - If right is True (the default), the first bin includes its left endpoint even if include_lowest is False (the default).
  - If right is False, values of x that fall outside of the last bin's right endpoint are put into a bin labeled with an integer representing the bin number. For example, if bins=[1, 2, 3, 4], a value of 5 in x is put in a bin labeled !<4>. This bin is mapped to 4 in the integer mapping array.
- **labels** (bool, array, or None, default True) – Specify the labels for the returned bins. If an array, it must be the same length as the number of resulting bins (that is, its length should be one fewer than the number of endpoints). If True (the default) or None, the labels are created based on the bin endpoints. If False, only a *FastArray* of the integer bin mappings is returned.
- **retbins** (bool, default False) – Whether to return an array of the bin endpoints. Useful when bins is provided as a scalar or other labels are specified. See the Returns section below for details of the output.
- **precision** (*int, default 3*) – The precision at which to display the bin labels. Note that the endpoints used for partitioning are not changed.
- **include_lowest** (bool, default False) – Indicates whether the first bin should include its left endpoint or not. Note: Until a known issue is fixed, the first bin always includes its left endpoint, except when right is set to False.
- **filter** (array of bool, optional) – A boolean mask array. If a filter is provided, any values of x corresponding to False values are put in the 'Filtered' bin and mapped to 0 in the integer bin mapping array. Note that until a known issue is fixed, this parameter accepts a mask array that is shorter than x and ignores values of x that are past the last corresponding value of the mask.
- **duplicates** (*{'raise', 'drop'}, default 'raise'*) – If bin endpoints are not unique, raise an error or drop duplicate values.

**Returns**

- **bins** (*Categorical or FastArray*) –
  - If labels is True or None, a *Categorical* is returned, consisting of the bins, the integer mapping codes for the bins, and the unique bin labels.
  - If labels is False, a *FastArray* is returned that contains the integer mapping codes.
- **endpoints (optional)** (ndarray of str) – An array of the bin endpoints. Returned as a separate value, only when retbins is True.

**See also:**

**riptable.qcut**
Partition values into bins based on rank or sample quantiles.

### Examples

Partition values into three equal-sized bins.

```
>>> rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]), bins=3)
Categorical([1.0->3.0, 5.0->7.0, 3.0->5.0, 3.0->5.0, 5.0->7.0, 1.0->3.0]) Length: 6
  FastArray([1, 3, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'1.0->3.0', b'3.0->5.0', b'5.0->7.0'], dtype='|S8') Unique count: 3
```

Also return an array of the bin endpoints.

```
>>> cat, endpoints = rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]), bins=3, retbins=True)
>>> cat
Categorical([1.0->3.0, 5.0->7.0, 3.0->5.0, 3.0->5.0, 5.0->7.0, 1.0->3.0]) Length: 6
  FastArray([1, 3, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'1.0->3.0', b'3.0->5.0', b'5.0->7.0'], dtype='|S8') Unique count: 3
>>> endpoints
array([1., 3., 5., 7.])
```

Return just the array of integer bin mappings.

```
>>> rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]), bins=3, labels=False)
FastArray([1, 3, 2, 2, 3, 1], dtype=int8)
```

Assign the bins specific labels. Notice that the returned *Categorical* object's categories are `labels`.

```
>>> rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]),
...        bins=3, labels=["bad", "medium", "good"])
Categorical([bad, good, medium, medium, good, bad]) Length: 6
  FastArray([1, 3, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'bad', b'medium', b'good'], dtype='|S6') Unique count: 3
```

Partition values into bins with specified endpoints. Values that fall outside of the bins are put in the 'Filtered' category.

```
>>> rt.cut(x=rt.FA([1, 7, 5, 4, 6, 3]), bins=[1, 3, 6])
Categorical([1.0->3.0, Filtered, 3.0->6.0, 3.0->6.0, 3.0->6.0, 1.0->3.0]) Length: 6
  FastArray([1, 0, 2, 2, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'1.0->3.0', b'3.0->6.0'], dtype='|S8') Unique count: 2
```

**Known Issues**

Each bin includes its right endpoint, even if `right` is set to `False`.

```
>>> rt.cut(x=rt.FA([2, 3, 4]), bins=[1, 2, 3, 4], right=False)
Categorical([1.0->2.0, 2.0->3.0, 3.0->4.0]) Length: 3
  FastArray([1, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'1.0->2.0', b'2.0->3.0', b'3.0->4.0'], dtype='|S8') Unique count: 3
```

If `right` is `True` (the default), the first bin includes its left endpoint even if `include_lowest` is `False` (the default).

```
>>> rt.cut(x=rt.FA([1, 2, 3, 4]), bins=3, include_lowest=False)
Categorical([1.0->2.0, 1.0->2.0, 2.0->3.0, 3.0->4.0]) Length: 4
  FastArray([1, 1, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'1.0->2.0', b'2.0->3.0', b'3.0->4.0'], dtype='|S8') Unique count: 3
```

If `right` is `False`, values of `x` that fall outside of the last bin's right endpoint are put into a bin labeled with an integer representing the bin number.

```
>>> rt.cut(x=rt.FA([1, 2, 3, 4, 5, 6]), bins=[1, 2, 3, 4], right=False)
Categorical([Filtered, 1.0->2.0, 2.0->3.0, 3.0->4.0, !<4>, !<4>]) Length: 6
  FastArray([0, 1, 2, 3, 4, 4], dtype=int8) Base Index: 1
  FastArray([b'1.0->2.0', b'2.0->3.0', b'3.0->4.0'], dtype='|S8') Unique count: 3
```

If a boolean mask filter is provided that's shorter than the length of `x`, values of `x` that are past the length of the mask are ignored.

```
>>> rt.cut(x=rt.FA([1, 2, 3, 4]), bins=2, filter=rt.FA([False, True, True]))
Categorical([Filtered, 2.0->2.5, 2.5->3.0]) Length: 3
  FastArray([0, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'2.0->2.5', b'2.5->3.0'], dtype='|S8') Unique count: 2
```

riptable.rt_bin.**qcut**(*x*, *q*, *labels=True*, *retbins=False*, *precision=3*, *duplicates='raise'*, *filter=None*)

Quantile-based discretization function.

Discretize variable into equal-sized buckets based on rank or based on sample quantiles. For example, 1000 values for 10 quantiles would produce a Categorical object indicating quantile membership for each data point.

**Parameters**

- **x** (*1d ndarray*) –
- **q** (*integer or array of quantiles*) – Number of quantiles. 10 for deciles, 4 for quartiles, etc. Alternately, array of quantiles, e.g. [0, .25, .5, .75, 1.] for quartiles
- **labels** (*boolean, array, or None*) – Used as labels for the resulting bins. If an array, must be of the same length as the resulting bins. If False, returns only integer indicators of the bins. If None or True, the labels are created based on the bins. This affects the type of the output container (see below).
- **retbins** (*bool, optional*) – Whether to return the (bins, labels) or not.
- **precision** (*int, optional*) – The precision at which to store and display the bins labels
- **duplicates** (*{default 'raise', 'drop'}, optional*) – If bin edges are not unique, raise ValueError or drop non-uniques.
- **filter** (*ndarray of bool, default None*) – If provided, any False values will be ignored in the calculation.

**Returns**

- **out** (*Categorical or FastArray*) – An array-like object representing the respective bin for each value of `x`. The type depends on the value of `labels`:
  - False : returns a FastArray of integers
  - array, True, or None : returns a Categorical

- **bins** (*ndarray of floats*) – The computed or specified bins. Only returned when `retbins=True`.

### Notes

Out of bounds values will be represented as 'Clipped' in the resulting Categorical object

**See also:**

**`cut()`**
Bin values into discrete intervals.

**`Categorical`**
Array type for storing data that come from a fixed set of values.

### Examples

```
>>> rt.qcut(range(5), 4)
Categorical([0.0->1.0, 0.0->1.0, 1.0->2.0, 2.0->3.0, 3.0->4.0]) Length: 5
  FastArray([2, 2, 3, 4, 5], dtype=int8) Base Index: 1
  FastArray([b'Clipped', b'0.0->1.0', b'1.0->2.0', b'2.0->3.0', b'3.0->4.0'], dtype=
↪'|S8') Unique count: 5
```

```
>>> rt.qcut(range(5), 3, labels=["good", "medium", "bad"])
Categorical([good, good, medium, bad, bad]) Length: 5
  FastArray([2, 2, 3, 4, 4], dtype=int8) Base Index: 1
  FastArray([b'Clipped', b'good', b'medium', b'bad'], dtype='|S7') Unique count: 4
```

```
>>> rt.qcut(range(5), 4, labels=False)
FastArray([2, 2, 3, 4, 5], dtype=int8)
```

riptable.rt_bin.**quantile**(*x*, *q*, *interpolation_method='fraction'*)

Compute sample quantile or quantiles of the input array. For example, q=0.5 computes the median.

The `interpolation_method` parameter supports three values, namely `fraction` (default), `lower` and `higher`. Interpolation is done only, if the desired quantile lies between two data points `i` and `j`. For `fraction`, the result is an interpolated value between `i` and `j`; for `lower`, the result is `i`, for `higher` the result is `j`.

**Parameters**

- **x** (*ndarray*) – Values from which to extract score.
- **q** (*scalar or array*) – Percentile at which to extract score.
- **interpolation_method** (*{'fraction', 'lower', 'higher'}, optional*) – This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points `i` and `j`: - fraction: `i + (j - i)*fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`. - lower: `i`. - higher: `j`.

**Returns**
**score** – Score at percentile.

**Return type**
float

### Examples

```
>>> from scipy import stats
>>> a = np.arange(100)
>>> stats.scoreatpercentile(a, 50)
49.5
```

## 2.2.7 riptable.rt_categorical

### Classes

| | |
|---|---|
| *Categorical* | A *Categorical* efficiently stores an array of repeated strings and is used for |
| *Categories* | Holds categories for each Categorical instance. This adds a layer of abstraction to Categorical. |

### Functions

| | |
|---|---|
| *CatZero*(values[, categories, ordered, sort_gb, lex, ...]) | Calls Categorical() with base_index keyword set to 0. |
| *categorical_convert*(v[, base_index]) | **param v** |
| *categorical_merge_dict*(list_categories[, ...]) | Checks to make sure all unique string values in all dictionaries have the same corresponding integer in every categorical they appear in. |

**class** riptable.rt_categorical.**Categorical**(*values*, *categories=None*, *ordered=None*, *sort_gb=None*, *sort_display=None*, *lex=None*, *base_index=None*, *filter=None*, *dtype=None*, *unicode=None*, *invalid=None*, *auto_add=False*, *from_matlab=False*, *_from_categorical=None*)

Bases: *riptable.rt_groupbyops.GroupByOps*, *riptable.rt_fastarray.FastArray*

A *Categorical* efficiently stores an array of repeated strings and is used for groupby operations.

Riptable *Categorical* objects have two related uses:

- They efficiently store string (or other large dtype) arrays that have repeated values. The repeated values are partitioned into groups (a.k.a. categories), and each group is mapped to an integer. The mapping codes allow the data to be stored and operated on more efficiently.
- They're Riptable's class for doing groupby operations. A method applied to a *Categorical* is applied to each group separately.

A *Categorical* is typically created from a list of strings:

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"])
>>> c
Categorical([b, a, b, a, c, c, b]) Length: 7
  FastArray([2, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

The output shows:

- The *Categorical* values. These are grouped into unique categories (here, "a", "b", and "c"), which are also stored in the *Categorical* (see below).
- The integer mapping codes (also called bins). Each integer is mapped to a unique category (here, 1 is mapped to "a", 2 is mapped to "b", and 3 is mapped to "c"). Because these codes can also be used to index into the *Categorical*, they're also referred to as indices. By default, the index is 1-based, with 0 reserved for Filtered values.
- The unique categories. Each category represents a group for groupby operations.

Use *Categorical* objects to perform aggregations over arbitrary arrays of the same dimension as the *Categorical*:

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"])
>>> ints = rt.FA([3, 10, 2, 5, 4, 1, 1])
>>> flts = rt.FA([1.2, 3.4, 5.6, 4.0, 2.1, 0.6, 11.3])
>>> c.sum([ints, flts])
*key_0   col_0   col_1
------   -----   -----
a           15    7.40
b            6   18.10
c            5    2.70

[3 rows x 3 columns] total bytes: 51.0 B
```

### Multi-Key Categoricals

The *Categorical* above is a single-key *Categorical* – it groups one array of values into keys (the categories) for groupby operations.

Multi-key *Categorical* objects let you create and operate on groupings based on multiple associated categories. The associated keys form a group:

```
>>> strs = rt.FastArray(["a", "b", "b", "a", "b", "a"])
>>> ints = rt.FastArray([2, 1, 1, 2, 1, 1])
>>> c = rt.Categorical([strs, ints])  # Create a with a list of arrays.
>>> c
Categorical([(a, 2), (b, 1), (b, 1), (a, 2), (b, 1), (a, 1)]) Length: 6
  FastArray([1, 2, 2, 1, 2, 3], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'a'], dtype='|S1'), 'key_1': FastArray([2, 1,␣
↪1])} Unique count: 3
>>> c.count()
*key_0   *key_1   Count
------   ------   -----
a             2       2
b             1       3
a             1       1

[3 rows x 3 columns] total bytes: 27.0 B
```

### Filtered Values and Categories

Filter values and categories to exclude them from operations on the *Categorical*.

*Categorical* objects can be filtered when they're created or anytime afterwards. Because filtered items are mapped to 0 in the integer mapping array, filters can be used only in base-1 *Categorical* objects.

Filters can also be applied on a one-off basis at the time of an operation. See the Filtering topic under More About Categoricals for examples.

**More About Categorials**

For more about using *Categorical* objects, see the *Categoricals* section of the *Intro to Riptable* or these more in-depth topics:

- *Constructing Categoricals*
- *Accessing Parts of the Categorical*
- *Indexing*
- *Comparisons*
- *Filtering*
- *Base Index*
- *Sorting and Display Order*
- *Final dtype of Integer Mapping Array*
- *Invalid Categories*
- *Get Bins from Categories and Vice-Versa*

**Parameters**

- **values** (*array of str, int, or float, list of arrays, dict, or Categorical or pandas.Categorical*) –
  - Strings: Unicode strings and byte strings are supported.
  - Integers without provided categories: The integer mapping codes start at 1.
  - Integers with provided categories: If you have an array of integers that indexes into an array of provided unique categories, the integers are used for the integer mapping array. Any 0 values are mapped to the Filtered category.
  - Floats are supported with no user-provided categories. If you have a Matlab Categorical with categories, set `from_matlab` to `True`. *Categorical* objects created from Matlab Categoricals must have a base-1 index; any 0.0 values become Filtered.
  - A list of arrays or a dictionary with multiple key-value pairs creates a multi-key *Categorical*.
  - For a *Categorical* created from a *Categorical*, a deep copy of categories is performed.
  - For a *Categorical* created from a Pandas Categorical, a deep copy is performed and indices start at 1 to preserve invalid values. *Categorical* objects created from Pandas Catagoricals must have a base-1 index.
- **categories** (*array of str, int, or float, dict of {str : int} or {int : str}, or IntEnum, optional*) – The unique categories. Can be:
  - An array of strings, integers, or floats. Floats can be used only when `values` is numeric. Warning: Non-unique categories may give unexpected results in operations.
  - A dictionary or `IntEnum` that maps integers to strings or strings to integers. Provided `values` must be integers.

  Note:
  - User-provided categories are always held in the order provided.

  - Multi-key *Categorical* objects don't support user-provided categories.

- **ordered** (*bool, default None/True*) – Controls whether categories are sorted lexicographically before they are mapped to integers:
  - If categories are not provided, by default they are sorted. If `ordered=False`, the order is first appearance unless `lex=True`. To sort categories for groupby operations, use `sort_gb=True` (see below).
  - If categories are provided, they are always held in the order they're provided in; they can't be sorted with `ordered` or `lex`.
- **sort_gb** (*bool, default None/False*) – Controls whether groupby operation results are displayed in sorted order. Note that results may already appear sorted based on `ordered` or `lex` settings.
- **sort_display** (*bool, optional*) – See `sort_gb`.
- **lex** (*bool, default None/False*) – Controls whether hashing- or sorting-based logic is used to find unique values in the input array. By default hashing is used. If more than 50% of the values are unique, set `lex=True` for a possibly faster lexicographical sort (not supported if categories are provided).
- **base_index** (*{None, 0, 1}, default None/1*) – By default, base-1 indexing is used. Base-0 can be used if:
  - A mapping dictionary isn't used. A *Categorical* created from a mapping dictionary does not have a base index.
  - A `filter` isn't used at creation.
  - A Matlab or Pandas Categorical isn't being converted. These both reserve 0 for invalid values.

  If base-0 indexing is used, 0 becomes a valid category.
- **filter** (*array of bool, optional*) – Must be the same length as `values`. Values that are `False` become Filtered and mapped to 0 in the integer mapping array, and they are ignored in groupby operations. A filter can't be used with a base-0 *Categorical* or one created with a mapping dictionary or `IntEnum`.
- **dtype** (*riptable.dtype, numpy.dtype, or str, optional*) – Force the dtype of the underlying integer mapping array. Must be a signed integer dtype. By default, the constructor uses the smallest dtype based on the number of unique categories or the maximum value provided in a mapping.
- **unicode** (*bool, default False*) – By default, the array of unique categories is stored as byte strings. Set to `True` to store as unicode strings.
- **invalid** (*str, optional*) – Specify a value in `values` to be treated as an invalid category. Note: Invalid categories are not excluded from aggregations; use `filter` instead. Warning: If the invalid category isn't included in `categories` and a `filter` is used, the invalid category becomes Filtered.
- **auto_add** (*bool, default False*) – Warning: Until a known issue is fixed, adding categories can have unexpected results. Intended behavior: When set to `True`, categories that do not exist in the unique categories can be added using *category_add*.
- **from_matlab** (*bool, default False*) – Set to `True` to convert a Matlab Categorical. The float indices are converted to an integer type. To preserve invalid values, only base-1 indexing is supported.

**See also:**

**Accum2**
Class for multi-key aggregations with summary data displayed.

*Categorical._fa*
Return the array of integer category mapping codes that corresponds to the array of *Categorical* values.

*Categorical.category_array*
Return the array of unique categories of a *Categorical*.

*Categorical.category_dict*
Return a dictionary of the unique categories.

*Categorical.category_mapping*
Return a dictionary of the integer category mapping codes for a *Categorical* created with an IntEnum or a mapping dictionary.

*Categorical.base_index*
See the base index of a *Categorical*.

*Categorical.isnan*
See which *Categorical* category is invalid.

### Examples

A single-key *Categorical* created from a list of strings:

```
>>> c = rt.Categorical(["b", "a", "b", "a", "c", "c", "b"])
Categorical([b, a, b, a, c, c, b]) Length: 7
  FastArray([2, 1, 2, 1, 3, 3, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

A *Categorical* created from list of non-unique string values and a list of unique category strings. All values must appear in the provided categories, otherwise an error is raised:

```
>>> rt.Categorical(["b", "a", "b", "c", "a", "c", "c", "c"], categories=["b", "a",
↪"c"])
Categorical([b, a, b, c, a, c, c, c]) Length: 8
  FastArray([1, 2, 1, 3, 2, 3, 3, 3], dtype=int8) Base Index: 1
  FastArray([b'b', b'a', b'c'], dtype='|S1') Unique count: 3
```

A *Categorical* created from a list of integers that index into a list of unique strings. The integers are used for the mapping array. Note that 0 becomes Filtered:

```
>>> rt.Categorical([0, 1, 1, 0, 2, 1, 2], categories=["c", "a", "b"])
Categorical([Filtered, c, c, Filtered, a, c, a]) Length: 7
  FastArray([0, 1, 1, 0, 2, 1, 2]) Base Index: 1
  FastArray([b'c', b'a', b'b'], dtype='|S1') Unique count: 3
```

If integers are provided with no categories and 0 is included, the integer mapping codes are incremented by 1 so that 0 is not Filtered:

```
>>> rt.Categorical([0, 1, 1, 0, 2, 1, 2])
Categorical([0, 1, 1, 0, 2, 1, 2]) Length: 7
  FastArray([1, 2, 2, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([0, 1, 2]) Unique count: 3
```

Use `from_matlab=True` to create a *Categorical* from Matlab data. The float indices are converted to an integer type. To preserve invalid values, only base-1 indexing is supported:

```
>>> rt.Categorical([0.0, 1.0, 2.0, 3.0, 1.0, 1.0], categories=["b", "c", "a"], from_
↪matlab=True)
Categorical([Filtered, b, c, a, b, b]) Length: 6
  FastArray([0, 1, 2, 3, 1, 1], dtype=int8) Base Index: 1
  FastArray([b'b', b'c', b'a'], dtype='|S1') Unique count: 3
```

A *Categorical* created from a Pandas Categorical with an invalid value:

```
>>> import pandas as pd
>>> pdc = pd.Categorical(["a", "a", "z", "b", "c"], ["c", "b", "a"])
>>> pdc
['a', 'a', NaN, 'b', 'c']
Categories (3, object): ['c', 'b', 'a']
>>> rt.Categorical(pdc)
Categorical([a, a, Filtered, b, c]) Length: 5
  FastArray([3, 3, 0, 2, 1], dtype=int8) Base Index: 1
  FastArray([b'c', b'b', b'a'], dtype='|S1') Unique count: 3
```

A *Categorical* created from a Python dictionary of strings to integers. The dictionary is provided as the `categories` argument, with a list of the mapping codes provided as the first argument:

```
>>> d = {"StronglyAgree": 44, "Agree": 133, "Disagree": 75, "StronglyDisagree": 1,
↪"NeitherAgreeNorDisagree": 144 }
>>> codes = [1, 44, 44, 133, 75]
>>> rt.Categorical(codes, categories=d)
Categorical([StronglyDisagree, StronglyAgree, StronglyAgree, Agree, Disagree]),␣
↪Length: 5
  FastArray([  1,  44,  44, 133,  75]) Base Index: None
  {44:'StronglyAgree', 133:'Agree', 75:'Disagree', 1:'StronglyDisagree', 144:
↪'NeitherAgreeNorDisagree'} Unique count: 4
```

A *Categorical* created using the categories of another *Categorical*:

```
>>> c = rt.Categorical(["a", "a", "b", "a", "c", "c", "b"], categories=["c", "b", "a
↪"])
>>> c.category_array
FastArray([b'c', b'b', b'a'], dtype='|S1')
>>> c2 = rt.Categorical(["b", "c", "c", "b"], categories=c.category_array)
>>> c2
Categorical([b, c, c, b]) Length: 4
  FastArray([2, 1, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'c', b'b', b'a'], dtype='|S1') Unique count: 3
```

Multi-key Categoricals let you create and operate on groupings based on multiple associated categories:

```
>>> strs = rt.FastArray(["a", "b", "b", "a", "b", "a"])
>>> ints = rt.FastArray([2, 1, 1, 2, 1, 3])
>>> c = rt.Categorical([strs, ints]) # Create with a list of arrays.
>>> c
Categorical([(a, 2), (b, 1), (b, 1), (a, 2), (b, 1), (a, 3)]) Length: 6
  FastArray([1, 2, 2, 1, 2, 3], dtype=int8) Base Index: 1
```

(continues on next page)

(continued from previous page)

```
  {'key_0': FastArray([b'a', b'b', b'a'], dtype='|S1'), 'key_1': FastArray([2, 1,␣
↪3])} Unique count: 3
>>> c.count()
*key_0   *key_1   Count
------   ------   -----
a             2       2
b             1       3
a             3       1

[3 rows x 3 columns] total bytes: 27.0 B
```

**property _categories**

**property _fa: *riptable.rt_fastarray.FastArray***

Return the array of integer category mapping codes that corresponds to the array of *Categorical* values.

**Returns**

A *FastArray* of the integer category mapping codes of the *Categorical*.

**Return type**

*FastArray*

**See also:**

***Categorical.category_array***

Return the array of unique categories of a *Categorical*.

***Categorical.categories***

Return the unique categories of a single-key or multi-key *Categorical*, prepended with the 'Filtered' category.

***Categorical.category_dict***

Return a dictionary of the unique categories.

***Categorical.category_mapping***

Return a dictionary of the integer category mapping codes for a *Categorical* created with an IntEnum or a mapping dictionary.

### Examples

Single-key string *Categorical*:

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c
Categorical([a, a, b, c, a]) Length: 5
  FastArray([1, 1, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c._fa
FastArray([1, 1, 2, 3, 1], dtype=int8)
```

Multi-key *Categorical*:

```
>>> c2 = rt.Categorical([rt.FA([1, 2, 3, 3, 3, 1]), rt.FA(['a','b','c','c','c',
↪'a'])])
>>> c2
```

(continues on next page)

(continued from previous page)

```
Categorical([(1, a), (2, b), (3, c), (3, c), (3, c), (1, a)]) Length: 6
  FastArray([1, 2, 3, 3, 3, 1], dtype=int8) Base Index: 1
  {'key_0': FastArray([1, 2, 3]), 'key_1': FastArray([b'a', b'b', b'c'], dtype=
↪'|S1')} Unique count: 3
>>> c2._fa
FastArray([1, 2, 3, 3, 3, 1], dtype=int8)
```

A *Categorical* constructed with an IntEnum or a mapping dictionary returns the provided integer category mapping codes:

```
>>> log_levels = {10: "DEBUG", 20: "INFO", 30: "WARNING", 40: "ERROR", 50:
↪"CRITICAL"}
>>> c3 = rt.Categorical([10, 10, 40, 0, 50, 10, 30], log_levels)
>>> c3
Categorical([DEBUG, DEBUG, ERROR, !<0>, CRITICAL, DEBUG, WARNING]) Length: 7
  FastArray([10, 10, 40,  0, 50, 10, 30]) Base Index: None
  {10:'DEBUG', 20:'INFO', 30:'WARNING', 40:'ERROR', 50:'CRITICAL'} Unique count:
↪ 5
>>> c3._fa
FastArray([10, 10, 40,  0, 50, 10, 30])
```

A 'Filtered' category is mapped to 0 in the integer array:

```
>>> c4 = rt.Categorical(['b','b','c','d','e','b','c'])
>>> c4
Categorical([b, b, c, d, e, b, c]) Length: 7
  FastArray([1, 1, 2, 3, 4, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 4
>>> c4._fa
FastArray([1, 1, 2, 3, 4, 1, 2], dtype=int8)
>>> c4.category_remove('c')  # A removed category becomes 'Filtered'.
>>> c4
Categorical([b, b, Filtered, d, e, b, Filtered]) Length: 7
  FastArray([1, 1, 0, 2, 3, 1, 0], dtype=int8) Base Index: 1
  FastArray([b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 4
>>> c4._fa
FastArray([1, 1, 0, 2, 3, 1, 0], dtype=int8)
```

**property _total_size: int**

Returns total size in bytes of Categorical's Index FastArray and category array(s).

**property as_string_array: *riptable.rt_fastarray.FastArray***

Return the full list of values of a *Categorical* as a string array.

For multi-key *Categorical* objects, the corresponding keys are concatenated with a '_' separator.

Filtered values become the string 'Filtered'. Values from invalid categories are treated the same way as values from valid categories.

NOTE: This routine is costly because it re-expands the full list of values as strings.

**Returns**

A FastArray of the string values of the *Categorical*.

**Return type**

*FastArray*

**See also:**

*Categorical.expand_array*
Return the full list of *Categorical* values.

### Notes

This method works by applying an index mask to the unique categories.

### Examples

Single-key string *Categorical*:

```
>>> c = rt.Categorical(['AAPL','MSFT','AAPL','TSLA','MSFT','TSLA','AAPL'])
>>> c
Categorical([AAPL, MSFT, AAPL, TSLA, MSFT, TSLA, AAPL]) Length: 7
  FastArray([1, 2, 1, 3, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'AAPL', b'MSFT', b'TSLA'], dtype='|S4') Unique count: 3
>>> c.as_string_array
FastArray([b'AAPL', b'MSFT', b'AAPL', b'TSLA', b'MSFT', b'TSLA', b'AAPL'],␣
↪dtype='|S8')
```

Single-key integer *Categorical*:

```
>>> c = rt.Categorical([1, 2, 1, 1, 3, 2, 3])
>>> c.as_string_array
FastArray(['1', '2', '1', '1', '3', '2', '3'], dtype='<U11')
```

Multi-key *Categorical*:

```
>>> key1 = rt.FastArray(['AAPL','MSFT','AAPL','TSLA','MSFT','TSLA','AAPL'])
>>> key2 = rt.FastArray([1, 1, 2, 2, 3, 3, 4])
>>> mk_cat = rt.Categorical([key1, key2])
>>> mk_cat
Categorical([(AAPL, 1), (MSFT, 1), (AAPL, 2), (TSLA, 2), (MSFT, 3), (TSLA, 3),␣
↪(AAPL, 4)]) Length: 7
  FastArray([1, 2, 3, 4, 5, 6, 7], dtype=int8) Base Index: 1
  {'key_0': FastArray([b'AAPL', b'MSFT', b'AAPL', b'TSLA', b'MSFT', b'TSLA', b
↪'AAPL'], dtype='|S4'), 'key_1': FastArray([1, 1, 2, 2,
3, 3, 4])} Unique count: 7
>>> mk_cat.as_string_array
FastArray([b'AAPL_1', b'MSFT_1', b'AAPL_2', b'TSLA_2', b'MSFT_3', b'TSLA_3', b
↪'AAPL_4'], dtype='|S16')
```

**property base_index: enum.IntEnum**

**property category_array: *riptable.rt_fastarray.FastArray***

Return the array of unique categories of a *Categorical*.

Unlike *Categorical.categories*, this method does not prepend the 'Filtered' category to the returned array.

Raises an error for multi-key *Categorical* objects. To get the categories of a multi-key *Categorical*, use *Categorical.categories*.

**Returns**
A *FastArray* of the unique categories of the *Categorical*.

**Return type**
*FastArray*

**See also:**

*Categorical._fa*
Return the array of integer category mapping codes that corresponds to the array of *Categorical* values.

*Categorical.categories*
Return the unique categories of a single-key or multi-key *Categorical*, prepended with the 'Filtered' category.

*Categorical.category_dict*
Return a dictionary of the unique categories.

*Categorical.category_mapping*
Return a dictionary of the integer category mapping codes for a *Categorical* created with an IntEnum or a mapping dictionary.

### Examples

Single-key string *Categorical*:

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c
Categorical([a, a, b, c, a]) Length: 5
  FastArray([1, 1, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.category_array
FastArray([b'a', b'b', b'c'], dtype='|S1')
```

Single-key integer *Categorical*:

```
>>> c2 = rt.Categorical([4, 5, 4, 4, 6, 5, 6])
>>> c2
Categorical([4, 5, 4, 4, 6, 5, 6]) Length: 7
  FastArray([1, 2, 1, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([4, 5, 6]) Unique count: 3
>>> c2.category_array
FastArray([4, 5, 6])
```

Single-key integer *Categorical* with categories provided:

```
>>> c3 = rt.Categorical([2, 3, 4, 2, 3, 4], categories=['a', 'b', 'c', 'd', 'e
↪'])
>>> c3
Categorical([b, c, d, b, c, d]) Length: 6
  FastArray([2, 3, 4, 2, 3, 4]) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1') Unique count: 5
>>> c3.category_array
FastArray([b'a', b'b', b'c', b'd', b'e'], dtype='|S1')
```

The 'Filtered' category isn't included:

```
>>> c4 = rt.Categorical([0, 1, 1, 0, 2, 1, 1, 1, 2, 0], categories=['a', 'b', 'c
↪'])
>>> c4
Categorical([Filtered, a, a, Filtered, b, a, a, a, b, Filtered]) Length: 10
  FastArray([0, 1, 1, 0, 2, 1, 1, 1, 2, 0]) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c4.category_array
FastArray([b'a', b'b', b'c'], dtype='|S1')
```

A *Categorical* constructed with an IntEnum or a mapping dictionary returns the provided string categories:

```
>>> log_levels = {10: "DEBUG", 20: "INFO", 30: "WARNING", 40: "ERROR", 50:
↪"CRITICAL"}
>>> c5 = rt.Categorical([10, 10, 40, 0, 50, 10, 30], log_levels)
>>> c5
Categorical([DEBUG, DEBUG, ERROR, !<0>, CRITICAL, DEBUG, WARNING]) Length: 7
  FastArray([10, 10, 40,  0, 50, 10, 30]) Base Index: None
  {10:'DEBUG', 20:'INFO', 30:'WARNING', 40:'ERROR', 50:'CRITICAL'} Unique count:
↪ 5
>>> c5.category_array
FastArray([b'DEBUG', b'INFO', b'WARNING', b'ERROR', b'CRITICAL'],
        dtype='|S8')
```

**property category_codes: *riptable.rt_fastarray.FastArray***

**property category_dict: Mapping[str, *riptable.rt_fastarray.FastArray*]**

When possible, returns the dictionary of stored unique categories, otherwise raises an error.

Unlike the default for categories(), this will not prepend the invalid category to each array.

**property category_mapping: dict**

**property category_mode: riptable.rt_enum.CategoryMode**

Returns the category mode of the Categorical's Categories object. List modes are when the categorical has gone through the unique/mbget process of binning. Dict modes are when the categorical was constructed with a dictionary mapping or IntEnum. Grouping mode is when the categorical was binned with the groupby hash (numeric list, multikey, etc.)

**Returns**
see CategoryMode in rt_enum.py

**Return type**
IntEnum

**property expand_array: numpy.ndarray | Tuple[numpy.ndarray, Ellipsis]**

Return the full list of values of a *Categorical*.

If the *Categorical* is constructed with an IntEnum or a mapping dictionary, the integer mapping codes are returned.

Filtered *Categorical* values are returned as "Filtered" for string arrays or numeric sentinel values for numeric arrays.

Note that because the expansion constructs the complete list of values from the list of unique categories, it is an expensive operation.

**Returns**
For single-key *Categorical* objects, a FastArray is returned. For multi-key *Categorical* objects, a tuple of FastArray objects is returned.

**Return type**
*FastArray* or tuple of *FastArray*

**Warns**
**Performance warning** – Will warn the user if a large *Categorical* (more than 100,000 items) is being re-expanded.

**See also:**

*Categorical.as_string_array*
Return the full list of values of a *Categorical* as a string array.

### Examples

Single-key *Categorical*:

```
>>> c = rt.Categorical(["a", "a", "b", "c", "a"])
>>> c.expand_array
FastArray([b'a', b'a', b'b', b'c', b'a'], dtype='|S3')
```

Multi-key *Categorical*:

```
>>> c = rt.Categorical([rt.FastArray(["a", "b", "c", "a"]), rt.FastArray([1, 2,␣
↪3, 1])])
>>> c.expand_array
(FastArray([b'a', b'b', b'c', b'a'], dtype='|S8'), FastArray([1, 2, 3, 1]))
```

For a *Categorical* constructed with an IntEnum or a mapping dictionary, the array of integer mapping codes (c._fa) is returned:

```
>>> c = rt.Categorical([2, 2, 2, 1, 3], {"a": 1, "b": 2, "c": 3})
>>> c
Categorical([b, b, b, a, c]) Length: 5
  FastArray([2, 2, 2, 1, 3]) Base Index: None
  {1:'a', 2:'b', 3:'c'} Unique count: 3
>>> c.expand_array
FastArray([2, 2, 2, 1, 3])
>>> c._fa
FastArray([2, 2, 2, 1, 3])
```

Filtered string *Categorical* values are returned as the string "Filtered":

```
>>> a = rt.FastArray(["a", "c", "b", "b", "c", "a"])
>>> f = rt.FastArray([False, False, True, True, True, True])
>>> c = rt.Categorical(a, filter=f)
>>> c
Categorical([Filtered, Filtered, b, b, c, a]) Length: 6
  FastArray([0, 0, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.expand_array
FastArray([b'Filtered', b'Filtered', b'b', b'b', b'c', b'a'], dtype='|S8')
```

Filtered integer *Categorical* values are returned as the integer sentinel value:

```
>>> a = rt.FastArray([1, 3, 2, 2, 3, 1])
>>> f = rt.FastArray([False, False, True, True, True, True])
>>> c = rt.Categorical(a, filter=f)
>>> c
Categorical([Filtered, Filtered, 2, 2, 3, 1]) Length: 6
  FastArray([0, 0, 2, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([1, 2, 3]) Unique count: 3
>>> c.expand_array
FastArray([-2147483648, -2147483648,           2,           2,
            3,           1])
```

**property expand_dict: Dict[str, *riptable.rt_fastarray.FastArray*]**

returns: A dictionary of expanded single or multikey columns. :rtype: dict

### Notes

Will warn the user if a large categorical ( > 100,000 items ) is being re-expanded.

### Examples

```
>>> c = rt.Categorical([FA(['a','a','b','c','a']), rt.arange(5)])
>>> c.expand_dict
{'key_0': FastArray([b'a', b'a', b'b', b'c', b'a'], dtype='|S3'),
 'key_1': FastArray([0, 1, 2, 3, 4])}
```

**property filtered_name: str**

Item displayed when a 0 bin is encountered. Will be omitted from groupby results by default.

**property filtered_string**

**property gb_keychain**

**property groupby_data**

All GroupByOps objects can hold a default dataset to perform operations on. GroupBy always holds a dataset. Categorical and Accum2 do not.

### Examples

By default, requires data to be passed:

```
>>> c = rt.Categorical(['a','b','c'])
>>> c.sum()
ValueError: Useable data has not been specified in (). Pass in array data to␣
↪operate on.
```

After the result of a Dataset.cat() operation, groupby data is set.

```
>>> ds = rt.Dataset({'groups':np.random.choice(['a','b','c'],10), 'data': rt.
↪arange(10), 'data2': rt.arange(10)})
>>> ds
```

(continues on next page)

(continued from previous page)

```
#   groups   data   data2
-   ------   ----   -----
0   a           0       0
1   a           1       1
2   c           2       2
3   c           3       3
4   a           4       4
5   a           5       5
6   c           6       6
7   b           7       7
8   c           8       8
9   a           9       9
>>> c = ds.cat('groups')
>>> c.sum()
*groups   data   data2
-------   ----   -----
a           19      19
b            7       7
c           19      19
```

**property grouping**

Grouping object that is called to perform calculations on grouped data. In the constructor, a grouping object provides a categorical with its instance array. The grouping object stores and generates other groupby information, like grouping indices, first occurrence, count, etc. The grouping object should be queried for all grouping-related properties. This is also a property in GroupBy, and is called by many routines in the GroupByOps parent class.

See Also: Grouping

**property grouping_dict**

Grouping dict held by Grouping object. May trigger lazy build of Grouping object.

**property ifirstkey**

Index of first occurrence of each unique key. May also trigger lazy evaluation of grouping object. If grouping object used the Groupby hash, it will have an iFirstKey array, otherwise returns None.

**property ikey**

Returns the grouping object's iKey. This will always be a 1-base index, and is often the same array as the Categorical. See also: grouping.ikey (may return base 0 index)

**property ilastkey**

Index of last occurrence of each unique key. May also trigger lazy evaluation of grouping object. If grouping object used the Groupby hash, it will have an iLastKey array, otherwise returns None.

**property invalid_category**

The *Categorical* object's invalid category.

An invalid category is specified when the *Categorical* is created or set afterward using *Categorical.invalid_set*. An invalid category is different from a Filtered category or a NaN value.

**Returns**

The invalid category of the *Categorical*. Returns None if there's no invalid category.

**Return type**

str or int or float or None

**See also:**

*Categorical.filtered_name*
Item displayed when a 0 bin is encountered in a *Categorical*.

*Categorical.isnan*
Find the invalid elements of a *Categorical*.

*Categorical.isnotnan*
Find the valid elements of a `Categorical`.

### Examples

```
>>> c = rt.Categorical(values=["b", "a", "c", "b", "c"], invalid="b")
>>> c
Categorical([b, a, c, b, c]) Length: 5
  FastArray([2, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.invalid_category
'b'
>>> c.isnan()  # Returns True for invalid category.
FastArray([ True, False, False,  True, False])
```

Invalid categories are different from Filtered categories:

```
>>> f = rt.FA([False, True, True, False, True])
>>> c2 = rt.Categorical(values=["b", "a", "c", "b", "c"], invalid="a", filter=f)
>>> c2
Categorical([Filtered, a, c, Filtered, c]) Length: 5
  FastArray([0, 1, 2, 0, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'c'], dtype='|S1') Unique count: 2
>>> c2.invalid_category
'a'
>>> c2.isnan()  # Show which values are in the invalid category.
FastArray([False,  True, False, False, False])
>>> c2.isfiltered()  # Show which values are Filtered.
FastArray([ True, False, False,  True, False])
```

Invalid categories in a *Categorical* are different from regular integer NaN values. An integer NaN is a valid category and is `False` for `Cat.isnan()`:

```
>>> a = rt.FA([1, 2, 3, 4])
>>> a[3] = a.inv  # Set the last value to an integer NaN.
>>> a
FastArray([          1,           2,           3, -2147483648])
>>> c3 = rt.Categorical(values=a, invalid=2)  # Make 2 an invalid category.
>>> c3
Categorical([1, 2, 3, -2147483648]) Length: 4
  FastArray([2, 3, 4, 1], dtype=int8) Base Index: 1
  FastArray([-2147483648,           1,           2,           3]) Unique count:␣
↪4
>>> c3.invalid_category()
2
>>> c3.isnan()  # Only the invalid category returns True for Cat.isnan.
```

(continues on next page)

(continued from previous page)

```
FastArray([False,  True, False, False])
>>> c3.expand_array.isnan()  # Only the integer NaN returns True for FA.isnan.
FastArray([False, False, False,  True])
```

**property isenum: bool**

See Categories.enum

**property ismultikey: bool**

See Categories.multikey

**property issinglekey: bool**

See Categories.singlekey

**property nan_index: int**

**property ordered: bool**

If the categorical is tagged as ordered, the unique categories will remain in the order they were provided in.

*ordered* is also true if a sort was performed when generating the unique categories.

**property sort_gb: bool**

**property sorted: bool**

If the categorical is tagged as sorted, it can use a binary search when performing a lookup in the unique categories.

If a sorted groupby operation is performed, no sort will need to be applied.

**property transform**

TO BE DEPRECATED

### Examples

```
>>> c = rt.Categorical(ds.symbol)
>>> c.transform.sum(ds.TradeSize)
```

**property unique_count**

Number of unique values in the categorical. It is necessary for every groupby operation.

### Notes

For categoricals in dict / enum mode that have generated their grouping object, this will reflect the number of unique values that `occur` in the non-unique values. Empty bins will not be included in the count.

**property unique_repr**

**DebugMode = False**

**GroupingDebugMode = False**

**MetaDefault**

**MetaVersion = 1**

**TestIsMemberVerbose = False**

**_test_cat_ismember = ''**

**__arrow_array__**(*type=None*)

Implementation of the `__arrow_array__` protocol for conversion to a pyarrow array.

**Parameters**
**type** (*pyarrow.DataType, optional, defaults to None*) –

**Return type**
pyarrow.Array or pyarrow.ChunkedArray

### Notes

https://arrow.apache.org/docs/python/extending_types.html#controlling-conversion-to-pyarrow-array-with-the-arrow-array

**__del__**()

Called when a Categorical is deleted.

**__eq__**(*other*)

Return self==value.

**__ge__**(*other*)

Return self>=value.

**__getitem__**(*fld*)

Indexing: Bracket indexing for Categoricals will *always* hit the FastArray of indices/codes first. If indexed by integer, the retrieved index or code will be passed to the Categories object so the corresponding Category can be returned. Otherwise, a new Categorical will be returned, using the same Categories as the original Categorical with a different index/code array.

The following examples will use this Categorical:

```
>>> c = rt.Categorical(['a','a','a','b','c','a','b'])
>>> c
Categorical([a, a, a, b, c, a, b]) Length: 7
  FastArray([1, 1, 1, 2, 3, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Single Integer:

For convenience, any bytestrings will be returned/displayed as unicode strings.

```
>>> c[3]
'b'
```

Multiple Integers:

```
>>> c[[1,2,3,4]]
Categorical([a, a, b, c]) Length: 4
  FastArray([1, 1, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

```
>>> c[np.arange(4,6)]
Categorical([c, a]) Length: 2
  FastArray([3, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Boolean Array:

```
>>> mask = FastArray([False,  True,  True,  True,  True,  True, False])
>>> c[mask]
Categorical([a, a, b, c, a]) Length: 5
  FastArray([1, 1, 2, 3, 1], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

Slice:

```
>>> c[2:5]
Categorical([a, b, c]) Length: 3
  FastArray([1, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

**__gt__**(*other*)

Return self>value.

**__le__**(*other*)

Return self<=value.

**__lt__**(*other*)

Return self<value.

**__ne__**(*other*)

Return self!=value.

**__repr__**(*verbose=False*)

Return repr(self).

**__setitem2__**(*key*, *value*)

Use grouping object isin, single item accessor instead of Categories object.

**__setitem__**(*index*, *value*)

**Parameters**

- **index** (*int or string (depends on category mode)*) –
- **value** (*sequence or scalar value*) – The value may represent a category or category index.

**Raises**

**IndexError** –

**__str__**()

Return str(self).

**static _array_compiled_numba_apply**(*iGroup*, *iFirstGroup*, *nCountGroup*, *userfunc*, *args*)

**_as_meta_data**(*name=None*)

**Parameters**
**name** (*string, optional*) – If not specified, will attempt to get name with get_name(), otherwise use class name.

**Returns**

- **arrdict** (*dictionary*) – Dictionary of column names -> arrays. Extra columns (for unique categories) will have the name+'!' before their keys.
- **arrtypes** (*list*) – List of SDSFlags, same length as arrdict.
- **meta** (*json-encoded string*) – Meta data for the categorical.

**See also:**

*_from_meta_data*

**_attach_self_as_key_column**(*result*)

**_autocomplete**()

**_build_sds_meta_data**(*name*, ***kwargs*)

Generates meta data from calling categorical, assembles arrays to represent its unique categories.

**Parameters**
**name** (*name of the categorical in the calling structure, or Categorical by default*) –

**Returns**

- **meta** (*MetaData*) – Metadata object for final save
- **cols** (*list of FastArray*) – arrays to represent unique categories - regardless of CategoryMode
- **tups** (*tuples with names of addtl. cols - still determining enum for second item in tuple (will relate to multiday load/concatenation)*) – names will be in the format 'name!col_' followed by column number

**_build_string**()

**_calculate_all**(*funcNum*, **args*, *func_param=0*, ***kwargs*)

**_categorical_compare_check**(*func_name*, *other*)

Converts a category to a valid index for faster logical comparison operations on the underlying index fastarray.

**_category_make_unique_multi_key**()

Remove duplicated categories by replacing categories with the unique set and remapping codes. Gets out early if categories are already unique.

**_copy_extra**(*cat_copy*)

Internal routine to move over some extra data from self

**_expand_array**(*arr*, *index=None*)

Internal routine to h-stack an invalid with an array for re-expanding single or multikey categoricals. This allows invalids to be retained in the re-expanded array(s)

**static _from_arrow**(*arr*, *zero_copy_only=True*, *writable=False*)

Create a *Categorical* instance from a dictionary-encoded pyarrow.Array.

For certain special cases, namely CategoryMode.IntEnum, CategoryMode.Dictionary, and CategoryMode.MultiKey, this method accepts an instance of pyarrow.Table, since *Categorical* instances with these CategoryMode`s don't have an encoding in pyarrow that'd directly preserve their structure. (For example, the direct mapping between the case labels and values for a `CategoryMode.IntEnum or CategoryMode.Dictionary-mode *Categorical*.)

**Parameters**

- **arr** (*pyarrow.Array or pyarrow.ChunkedArray*) – Must be a dictionary-encoded pyarrow array or a Struct-type array (e.g. pyarrow.StructArray).
- **zero_copy_only** (*bool, optional, defaults to True*) –
- **writable** (*bool, optional, defaults to False*) –

**Return type**
*Categorical*

**classmethod _from_maybe_non_unique_labels**(*values*, *categories*, *base_index=1*)

Remove duplicated categories by replacing categories with the unique set and remapping codes. Gets out early if categories are already unique.

**classmethod _from_meta_data**(*arrdict*, *arrflags*, *meta*)

**_getsingleitem**(*fld*)

If the getitem indexing operation returned a scalar, translate it according to how the uniques are being held.

**Return type**
Scalar or tuple based on unique type.

**_ipython_key_completions_**()

For tab completions with bracket indexing (__getitem__) The IPython completer needs a python list or dict keys/values. If no return (e.g. multikey categorical), return an empty list. Also returns empty if categorical has > 10_000 unique values. If an IPython environment is detected, the 'greedy' property is set to True in riptable's __init__

**classmethod _load_from_sds_meta_data**(*name*, *arr*, *cols*, *meta*)

Builds a categorical object from metadata and arrays.

Will translate metadata, array/column layout from older versions to be compatible with current loader. Raises an error if the metadata version is higher than the class's meta version (user will need to update riptable)

**Parameters**

- **name** (*item's name in the calling container, or the classname Categorical by default*) –
- **arr** (*the underlying index array for the categorical*) –
- **cols** (*additional arrays to rebuild unique categories*) –
- **meta** (*meta data generated by build_sds_meta_data() routine*) –

**Returns**
Reconstructed categorical object.

**Return type**
*Categorical*

### Examples

```
>>> m = y._build_sds_meta_data('y')
>>> rt.Categorical._load_from_sds_meta_data('y', y._fa, m[1], m[0])
```

**_meta_dict**(*name=None*)

**_nan_idx**()

Internal - for isnan, isnotnan

**_nanfunc**(*func*, *fillval*)

**_prepend_invalid**(*arr*)

For base index 1 categoricals, add the invalid category to the beginning of the array of unique categories.

**Parameters**

**arr** (FastArray) – The array holding the unique category values for this Categorical. This array may be a `FastArray` or a subclass of `FastArray`.

**Returns**

An array of the same type as `arr` whose length is `len(arr) + 1`, where the first (0th) element of the array is the invalid value for that array type.

**Return type**

*FastArray*

**static _scalar_compiled_numba_apply**(*iGroup*, *iFirstGroup*, *nCountGroup*, *userfunc*, *args*)

**_tf_spacer**(*tf_string*)

**static _transformed_scalar_compiled_numba_apply**(*iGroup*, *iFirstGroup*, *nCountGroup*, *userfunc*, *args*)

**classmethod align**(*cats*)

Cats must be a list of categoricals. The unique categories will be merged into a new unique list. The indices will be fixed to point to the new category array.

**Return type**

A list of (possibly) new categoricals which share the same categories (and thus bin numbering).

### Examples

```
>>> c1 = rt.Categorical(['a','b','c'])
>>> c2 = rt.Categorical(['d','e','f'])
>>> c3 = rt.Categorical(['c','f','z'])
>>> rt.Categorical.align([c1,c2,c3])
[Categorical([a, b, c]) Length: 3
  FastArray([1, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e', b'f', b'z'], dtype='|S1') Unique␣
↪count: 7
Categorical([d, e, f]) Length: 3
  FastArray([4, 5, 6], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e', b'f', b'z'], dtype='|S1') Unique␣
↪count: 7
```

(continues on next page)

(continued from previous page)

```
Categorical([c, f, z]) Length: 3
  FastArray([3, 6, 7], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e', b'f', b'z'], dtype='|S1') Unique␣
↪count: 7]
```

**apply**(*userfunc=None*, **args*, *dataset=None*, ***kwargs*)

See Grouping.apply for examples. Categorical needs remove unused bins from its uniques before an apply.

**apply_nonreduce**(*userfunc=None*, **args*, *dataset=None*, ***kwargs*)

See GroupByOps.apply_nonreduce for examples. Categorical needs remove unused bins from its uniques before an apply.

**argsort**()

**as_singlekey**(*ordered=False*, *sep='_'*)

Normalizes categoricals by returning a base 1 single key categorical.

Enum or dict based categoricals will be converted to single key categoricals. Multikey categoricals will be converted to single key categoricals. If the categorical is already single key, base 0 it will be returned as base 1. If the categorical is already single key, base 1 it will be returned as is.

**Parameters**

- **ordered** (*bool, defaults False*) – whether or not to sort the result
- **sep** (*char, defaults '_'*) – only valid for multikey since this is the multikey separator

### Examples

```
>>> c=rt.Cat([5, -3, 7], {-3:'one', 2:'two', 5: 'three', 7:'four'})
>>> d=c.as_singlekey()
>>> c._fa
FastArray([ 5, -3,  7])
```

```
>>> d._fa
FastArray([3, 2, 1], dtype=int8)
```

**Return type**
A single key base 1 categorical.

**auto_add_off**()

Sets the _auto_add_categories flag to False. Category assignment with a non-existing categorical will raise an error.

### Examples

```
>>> c = rt.Categorical(['a','a','b','c','a'], auto_add_categories=True)
>>> c._categories
FastArray([b'a', b'b', b'c'], dtype='|S1')
>>> c.auto_add_off()
>>> c[0] = 'z'
ValueError: Cannot automatically add categories [b'z'] while auto_add_
↪categories is set to False.
```

**auto_add_on**()

If the categorical is unlocked, this sets the _auto_add_categories flag to be True. If _auto_add_categories is set to False, the following assignment will raise an error. If the categorical is locked, auto_add_on() will warn the user and the flag will not change.

### Examples

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c._categories
FastArray([b'a', b'b', b'c'], dtype='|S1')
>>> c.auto_add_on()
>>> c[0] = 'z'
>>> print(c)
z, a, b, c, a
>>> c._categories
FastArray([b'a', b'b', b'c', b'z'], dtype='|S1')
```

**categories**(*showfilter=True*)

If the categories are stored in a single array or single-key dictionary, an array will be returned. If the categories are stored in a multikey dictionary, a dictionary will be returned. If the categories are a mapping, a dictionary of the mapping will be returned (int -> string)

Note: you can also request categories in a certain format when possible using properties: *category_array*, *category_dict*, *category_mapping*.

**Parameters**

**showfilter** (*bool, defaults to True*) – If True (default), the invalid category will be prepended to the returned array or multikey columns. Does not apply when mapping is returned.

**Return type**

np.ndarray or dict

### Examples

```
>>> c = rt.Categorical(['a','a','b','c','d'])
>>> c.categories()
FastArray([b'Inv', b'a', b'b', b'c', b'd'], dtype='|S1')
```

```
>>> c = rt.Categorical([rt.arange(3), rt.FA(['a','b','c'])])
>>> c.categories()
```

(continues on next page)

(continued from previous page)

```
{'key_0': FastArray([-2147483648,           0,           1,           2]),
 'key_1': FastArray([b'Inv', b'a', b'b', b'c'], dtype='|S3')}
```

```
>>> c = rt.Categorical(rt.arange(3), {'a':0, 'b':1, 'c':2})
>>> c.categories()
{0: 'a', 1: 'b', 2: 'c'}
```

**classmethod categories_equal**(*cats*)

Check if every *Categorical* or array has the same categories (same unique values in the same order).

**Parameters**

**cats** (*list of Categorical or np.ndarray or tuple of np.ndarray*) – `cats` must be a list of *Categorical* objects or arrays that can be converted to *Categorical* objects.

**Returns**

- **match** (*bool*) – True if every *Categorical* has the same categories (same unique values in same order), otherwise False.
- **fixed_cats** (*list of Categorical*) – List of *Categorical* objects which may have been fixed up.

### Notes

TODO: Can the type annotation for `cats` be relaxed to Collection instead of List?

**category_add**(*value*)

New category will always be added to the end of the category array.

**category_make_unique**()

Remove duplicated categories by replacing categories with the unique set and remapping codes. Gets out early if categories are already unique.

**category_remove**(*value*)

Performance may suffer as indices need to be fixed up. All previous matches to the removed category will be flipped to invalid.

**category_replace**(*value*, *new_value*)

**copy**(*categories=None*, *ordered=None*, *sort_gb=None*, *lex=None*, *base_index=None*, *filter=None*, *dtype=None*, *unicode=None*, *invalid=None*, *auto_add=False*, *from_matlab=False*, *_from_categorical=None*, *deep=True*, *order='K'*)

Return a copy of the input `FastArray`.

**Parameters**

**order** (*{'K', 'C', 'F', 'A'}, default 'K'*) – Controls the memory layout of the copy: ‘K’ means match the layout of the input array as closely as possible; ‘C’ means row-based (C-style) order; ‘F’ means column-based (Fortran-style) order; ‘A’ means ‘F’ if the input array is formatted as ‘F’, ‘C’ if not.

**Returns**

A copy of the input `FastArray`.

**Return type**

*FastArray*

**See also:**

*Categorical.copy*
Return a copy of the input *Categorical*.

*Dataset.copy*
Return a copy of the input *Dataset*.

*Struct.copy*
Return a copy of the input *Struct*.

### Examples

Copy a FastArray:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a2 = a.copy()
>>> a2
FastArray([1, 2, 3, 4, 5])
>>> a2 is a
False  # The copy is a separate object.
```

**copy_invalid**()

Return a copy of a FastArray filled with the invalid value for the array's data type.

**Returns**
A copy of the input array, filled with the invalid value for the array's dtype.

**Return type**
*FastArray*

**See also:**

**FastArray.inv**
Return the invalid value for the input array's dtype.

**FastArray.fill_invalid**
Replace the values of a FastArray with the invalid value for the array's dtype.

### Examples

Copy an integer array and replace with invalids:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a2 = a.copy_invalid()
>>> a2
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
>>> a
FastArray([1, 2, 3, 4, 5])  # a is unchanged.
```

Copy a floating-point array and replace with invalids:

```
>>> a3 = rt.FA([0., 1., 2., 3., 4.])
>>> a3
FastArray([0., 1., 2., 3., 4.])
>>> a3.copy_invalid()
FastArray([nan, nan, nan, nan, nan])
```

Copy a string array and replace with invalids:

```
>>> a4 = rt.FA(['AMZN', 'IBM', 'MSFT', 'AAPL'])
>>> a4
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a4.copy_invalid()
FastArray([b'', b'', b'', b''], dtype='|S4')  # Invalid string value is an␣
↪empty string.
```

**count**(*filter=None*, *transform=False*)

Return the unique values of a *Categorical* and their counts.

Unlike other *Categorical* operations, this does not take a parameter for data.

**Parameters**

- **filter** (*array of bool, optional*) – *Categorical* values that correspond to False filter values are excluded from the count. The array must be the same length as the *Categorical*.
- **transform** (*bool, default False*) – Set to True to return a `Dataset` that's the length of the *Categorical*, with counts aligned to the ungrouped *Categorical* values. Only the counts are included.

**Returns**

A `Dataset` containing each unique category and its count. If *transform* is True, the `Dataset` is the same length as the original *Categorical* and contains only the counts.

**Return type**

`Dataset`

**See also:**

*Grouping.count*
Called by this method.

*Categorical.unique_count*
Return the number of unique values in a *Categorical*.

*FastArray.count*
Return the unique values of a `FastArray` and their counts.

### Examples

```
>>> c = rt.Categorical(['a','a','b','c','a','c'])
Categorical([a, a, b, c, a, c]) Length: 6
  FastArray([1, 1, 2, 3, 1, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.count()
*key_0   Count
------   -----
a            3
b            1
c            2
```

With a filter:

```
>>> f = c.count(c == 'a')  # Filter based on Categorical values
>>> c.count(filter=f)
*key_0   Count
------   -----
a            3
b            0
c            0
>>> vals = rt.arange(6)  # Filter based on a same-length array of values
>>> f = vals > 2
>>> c.count(filter=f)
*key_0   Count
------   -----
a            1
b            0
c            2
```

With `transform=True`, a `Dataset` is returned with counts aligned to the ungrouped *Categorical* values:

```
>>> c.count(transform=True)
#   Count
-   -----
0       3
1       3
2       1
3       2
4       3
5       2
```

**static display_convert_func**(*item*, *itemformat*)

Used in conjunction with display_query_properties for final display of a categorical in a dataset. Removes quotation marks from multikey categorical tuples so display is easier to read.

**display_query_properties**()

Takes over display query properties for fastarray. By default, all categoricals will use left alignment.

**expand_any**(*categories*)

**Parameters**

**categories** (*list or np.ndarray same size as categories array*) –

**Return type**
A re-expanded array of mapping categories passed in.

### Examples

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c.expand_any(['d','e','f'])
FastArray(['d', 'd', 'e', 'f', 'd'], dtype='<U8')
```

**fill_backward**(**args*, *limit=0*, *fill_val=None*, *inplace=False*)

Replace NaN and invalid array values by propagating the next encountered valid group value backward.

Optionally, you can modify the original array if it's not locked.

**Parameters**

- ***args** (*array or list of arrays*) – The array or arrays that contain NaN or invalid values you want to replace.
- **limit** (*int, default 0 (disabled)*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.
- **fill_val** (*scalar, default None*) – The value to use where there is no valid group value to propagate backward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate backward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object. This fails if the array is locked.

**Returns**
The *Categorical* will be the same size and have the same dtypes as the original input.

**Return type**
*Categorical*

**See also:**

*Categorical.fill_forward*
Replace NaN and invalid array values with the last valid group value.

**GroupBy.fill_backward**
Replace NaN and invalid array values with the next valid group value.

**riptable.fill_backward**
Replace NaN and invalid values with the next valid value.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**FastArray.fillna**
Replace NaN and invalid values with a specified value or nearby data.

### Examples

```
>>> cat = rt.Categorical(['A', 'B', 'A', 'B', 'A', 'B'])
>>> x = rt.FA([rt.nan, rt.nan, 2, 3, 4, 5])
>>> cat.fill_backward(x)
*gb_key_0   col_0
---------   -----
A            2.00
B            3.00
A            2.00
B            3.00
A            4.00
B            5.00
```

Use a `fill_val` to replace values where there's no valid group value to propagate backward:

```
>>> x = rt.FastArray([0, 1, 2, 3, rt.nan, rt.nan])
>>> cat.fill_backward(x, fill_val = 0)[0]
FastArray([0., 1., 2., 3., 0., 0.])
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values in a group:

```
>>> x = rt.FastArray([rt.nan, rt.nan, rt.nan, rt.nan, 4, 5])
>>> cat.fill_backward(x, limit = 1)[0]
FastArray([nan, nan,  4.,  5.,  4.,  5.])
```

**fill_forward**(**args*, *limit=0*, *fill_val=None*, *inplace=False*)

Replace NaN and invalid array values by propagating the last encountered valid group value forward.

Optionally, you can modify the original array if it's not locked.

**Parameters**

- ***args** (*array or list of arrays*) – The array or arrays that contain NaN or invalid values you want to replace.
- **limit** (*int, default 0 (disabled)*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.
- **fill_val** (*scalar, default None*) – The value to use where there is no valid group value to propagate forward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate forward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object. This fails if the array is locked.

**Returns**

The *Categorical* will be the same size and have the same dtypes as the original input.

**Return type**

*Categorical*

**See also:**

*Categorical.fill_backward*
Replace NaN and invalid array values with the next valid group value.

**GroupBy.fill_forward**
Replace NaN and invalid array values with the last valid group value.

**riptable.fill_forward**
Replace NaN and invalid values with the last valid value.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**FastArray.fillna**
Replace NaN and invalid values with a specified value or nearby data.

### Examples

```
>>> cat = rt.Categorical(['A', 'B', 'A', 'B', 'A', 'B'])
>>> x = rt.FastArray([0, 1, 2, 3, rt.nan, rt.nan])
>>> cat.fill_forward(x)
*gb_key_0   col_0
---------   -----
A            0.00
B            1.00
A            2.00
B            3.00
A            2.00
B            3.00
```

Use a `fill_val` to replace values where there's no valid group value to propagate forward:

```
>>> x = rt.FastArray([rt.nan, rt.nan, 2, 3, 4, 5])
>>> cat.fill_forward(x, fill_val = 0)[0]
FastArray([0., 0., 2., 3., 4., 5.])
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values in a group:

```
>>> x = rt.FastArray([0, 1, rt.nan, rt.nan, rt.nan, rt.nan])
>>> cat.fill_forward(x, limit = 1)[0]
FastArray([ 0.,  1.,  0.,  1., nan, nan])
```

**fill_invalid**(*shape=None*, *dtype=None*, *order=None*, *inplace=True*)
Returns a Categorical full of invalids, with reference to same categories. Must be base index 1.

**filtered_set_name**(*name*)
Set the name or value that will be displayed for filtered categories. Default is FILTERED_LONG_NAME

**from_bin**(*bin*)
Returns the category corresponding to a single integer. Raises error if index is out of range (accounts for base index) - or does not exist in mapping.

### Notes

String values will appear as the scalar type they are stored in, however FastArray, Categorical, and other riptable routines will convert/compensate for unicode/bytestring mismatches.

### Examples

Base-1 Indexing:

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c.category_array
FastArray([b'a', b'b', b'c'], dtype='|S1')
>>> c.category_from_bin(2)
b'b'
```

```
>>> c.category_from_bin(4)
IndexError
```

Base-0 Indexing:

```
>>> c = rt.Categorical(['a','a','b','c','a'], base_index=0)
>>> c.category_from_bin(2)
b'c'
```

**from_category**(*category*)

Returns the bin associated with a category. If the category doesn't exist, an error will be raised.

Note: the bin returned is the value as it appears in the underlying integer FastArray. It may not be a direct index into the stored unique categories.

Unicode/bytes conversion will be handled internally.

### Examples

Single Key (base-1):

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c.bin_from_category('a')
1
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> c.bin_from_category(b'c')
3
```

Single Key (base-0):

```
>>> c = rt.Categorical(['a','a','b','c','a'], base_index=0)
>>> c.bin_from_category('a')
0
```

Multikey:

```
>>> c = rt.Categorical([rt.FA(['a','b','c']), rt.arange(3)])
>>> c.bin_from_category(('a', 0))
1
```

Mapping:

```
>>> c = rt.Categorical([1,2,3], {'a':1, 'b':2, 'c':3})
>>> c.bin_from_category('c')
>>> 3
```

Numeric:

```
>>> c = rt.Categorical(rt.FA([3.33, 5.55, 6.66]))
>>> c.bin_from_category(3.33)
1
```

**static full**(*size*, *value*)

Create a *Categorical* of a given length, filled with a single value.

**Parameters**

- **size** (*int*) – The size/length of the *Categorical* to create.
- **value** – The value to be repeated.

**Return type**
*Categorical*

### Examples

Create a 1D *Categorical* array of length 100_000, filled with the string "example".

```
>>> rt.Categorical.full(100_000, 'example')
Categorical([example, example, example, example, example, ..., example, example,
↪ example, example, example]) Length: 100000
  FastArray([1, 1, 1, 1, 1, ..., 1, 1, 1, 1, 1], dtype=int8) Base Index: 1
  FastArray([b'example'], dtype='|S7') Unique count: 1
```

**groupby_data_clear**()

Remove any stored dataset for future groupby operations.

**groupby_data_set**(*ds*)

Store data to apply future groupby operations to. This will make the categorical behave like a groupby object that was created from a dataset. If data is specified during an operation, it will be used instead of the stored dataset.

**Parameters**
**ds** (Dataset) –

### Examples

```
>>> c = rt.Categorical(['a','b','c','c','a','a'])
>>> a = np.arange(6)
>>> ds = rt.Dataset({'col':a})
>>> c.groupby_data_set(ds)
>>> c.sum()
*gb_key   col
-------   ---
```

(continues on next page)

(continued from previous page)

```
a           9
b           1
c           5
```

**groupby_reset**()

Resets all lazily evaluated groupby information. The categorical will go back to the state it was in just after construction. This is called any time the categories are modified.

**classmethod hstack**(*cats*)

Cats must be a list of categoricals. The unique categories will be merged into a new unique list. The indices will be fixed to point to the new category array. The indices are hstacks and a new categorical is returned.

### Examples

```
>>> c1 = rt.Categorical(['a','b','c'])
>>> c2 = rt.Categorical(['d','e','f'])
>>> combined = rt.Categorical.hstack([c1,c2])
>>> combined
Categorical([a, b, c, d, e, f]) Length: 6
  FastArray([1, 2, 3, 4, 5, 6]) Base Index: 1
  FastArray([b'a', b'b', b'c', b'd', b'e', b'f'], dtype='|S1') Unique count: 6
```

**info**()

The three arrays in info: Categories mapped to their indices, often making the categorical appear to be a string array. Length of array. Underlying array of integer indices, dtype. Base index (normally 1 to reserve 0 as an invalid bin for groupby - much better for performance) Categories - list or dictionary

The CategoryMode is also displayed:

Mode:

Default - no example StringArray - categories are held in a single string array IntEnum - categories are held in a dictionary generated from an IntEnum Dictionary - categories are held in a dictionary generated from a code-mapping dictionary NumericArray - categories are held in a single numeric array MultiKey - categories are held in a dictionary (when constructed with multikey, or numeric categories the groupby hash does the binning)

Locked:

If True, categories may be changed.

**invalid_set**(*inv*)

Set a *Categorical* category to be invalid.

An invalid category is specified when the *Categorical* is created or set afterward using *Categorical.invalid_set*. An invalid category is different from a Filtered category or a NaN value.

If there's an existing invalid category in the *Categorical*, using *Categorical.invalid_set* to set a different category causes the existing invalid category to become valid.

**Parameters**
**inv** (*str or bytes*) – The category to be made invalid.

**Return type**
None

**See also:**

*Categorical.isnan*
Find the invalid elements of a *Categorical*.

*Categorical.isnotnan*
Find the valid elements of a Categorical.

*Categorical.invalid_category*
The *Categorical* object's invalid category.

### Examples

```
>>> c = rt.Categorical(values=["b", "a", "c", "b", "c"])
>>> c
Categorical([b, a, c, b, c]) Length: 5
  FastArray([2, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.invalid_set("b")
>>> c.invalid_category
'b'
>>> c.isnan()  # Returns True for invalid category.
FastArray([ True, False, False,  True, False])
```

Set a new invalid category:

```
>>> c.invalid_set("a")
>>> c.invalid_category
'a'
>>> c.isnan()
FastArray([False,  True, False, False, False])
```

**isfiltered**()

True where bin == 0. Only applies to categoricals with base index 1, otherwise returns all False. Different than invalid category.

**See also:**

*Categorical.isnan*, *Categorical.isnotnan*

**isin**(*values*)

**Parameters**
**values** (*a list-like or single value to be searched for*) –

**Returns**
Boolean array with the same size as self. True indicates that the array element occured in the provided values.

**Return type**
*FastArray*

### Notes

Behavior differs from pandas in the following ways: * Riptable favors bytestrings, and will make conversions from unicode/bytes to match for operations as necessary. * We also accept single scalars for `values`. * Pandas series will return another series - we have no series, and will return a FastArray.

### Examples

```
>>> c = rt.Categorical(['a','b','c','d','e'], unicode=False)
>>> c.isin(['a','b'])
FastArray([ True,  True, False, False, False])
```

**See also:**

`pandas.Categorical.isin`

**isna**(**args*, ***kwargs*)

See *Categorical.isnan*.

**isnan**(**args*, ***kwargs*)

Find the invalid elements of a *Categorical*.

An invalid category is specified when the *Categorical* is created or set afterward using *Categorical.invalid_set*. An invalid category is different from a Filtered category or a NaN value.

**Returns**
A boolean array the length of the values array where `True` indicates an invalid *Categorical* category.

**Return type**
*FastArray*

**See also:**

***Categorical.isnotnan***
Find the valid elements of a `Categorical`.

***Categorical.invalid_category***
The *Categorical* object's invalid category.

***Categorical.invalid_set***
Set a *Categorical* category to be invalid.

### Examples

```
>>> c = rt.Categorical(values=["b", "a", "c", "b", "c"], invalid="b")
>>> c
Categorical([b, a, c, b, c]) Length: 5
  FastArray([2, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.isnan()
FastArray([ True, False, False,  True, False])
```

Invalid categories are different from Filtered categories:

```
>>> f = rt.FA([True, False, True, True, True])
>>> c2 = rt.Categorical(values=["b", "a", "c", "b", "c"], invalid="b", filter=f)
>>> c2
Categorical([b, Filtered, c, b, c]) Length: 5
  FastArray([1, 0, 2, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'b', b'c'], dtype='|S1') Unique count: 2
>>> c2.isnan()  # Only the invalid category returns True for Cat.isnan.
FastArray([ True, False, False,  True, False])
>>> c2.isfiltered()  # Only the Filtered value returns True for Cat.isfiltered.
FastArray([False,  True, False, False, False])
```

Invalid categories in a *Categorical* are different from regular integer NaN values. An integer NaN is a valid category and is `False` for `Cat.isnan()`:

```
>>> a = rt.FA([1, 2, 3, 4])
>>> a[3] = a.inv  # Set the last value to an integer NaN.
>>> a
FastArray([          1,           2,           3, -2147483648])
>>> c3 = rt.Categorical(values=a, invalid=2)  # Make 2 an invalid category.
>>> c3
Categorical([1, 2, 3, -2147483648]) Length: 4
  FastArray([2, 3, 4, 1], dtype=int8) Base Index: 1
  FastArray([-2147483648,           1,           2,           3]) Unique count:␣
↪4
>>> c3.invalid_category()
2
>>> c3.isnan()  # Only the invalid category returns True for Cat.isnan.
FastArray([False,  True, False, False])
>>> c3.expand_array.isnan()  # Only the integer NaN returns True for FA.isnan.
FastArray([False, False, False,  True])
```

**isnotnan**(**args*, ***kwargs*)

Find the valid elements of a `Categorical`.

An invalid category is specified when the *Categorical* is created or set afterward using *Categorical.invalid_set*. An invalid category is different from a Filtered category or a NaN value.

**Returns**

A boolean array the length of the values array where `True` indicates a valid *Categorical* category.

**Return type**

*FastArray*

**See also:**

***Categorical.isnan***

Find the invalid elements of a `Categorical`.

***Categorical.invalid_category***

The *Categorical* object's invalid category.

***Categorical.invalid_set***

Set a *Categorical* category to be invalid.

### Examples

```
>>> c = rt.Categorical(values=["b", "a", "c", "b", "c"], invalid="b")
>>> c
Categorical([b, a, c, b, c]) Length: 5
  FastArray([2, 1, 3, 2, 3], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
>>> c.isnotnan()
FastArray([False,  True,  True, False,  True])
```

Invalid categories are different from Filtered categories:

```
>>> f = rt.FA([True, False, True, True, True])
>>> c2 = rt.Categorical(values=["b", "a", "c", "b", "c"], invalid="b", filter=f)
>>> c2
Categorical([b, Filtered, c, b, c]) Length: 5
  FastArray([1, 0, 2, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'b', b'c'], dtype='|S1') Unique count: 2
>>> c2.isnotnan()  # Only the invalid category returns False for Cat.isnotnan.
FastArray([False,  True,  True, False,  True])
>>> ~c2.isfiltered()  # Only the Filtered value returns False for the negation␣
↪of Cat.isfiltered.
FastArray([ True, False,  True,  True,  True])
```

Invalid categories in a *Categorical* are different from regular integer NaN values. An integer NaN is a valid category and is True for Cat.isnotnan():

```
>>> a = rt.FA([1, 2, 3, 4])
>>> a[3] = a.inv  # Set the last value to an integer NaN.
>>> a
FastArray([          1,           2,           3, -2147483648])
>>> c3 = rt.Categorical(values=a, invalid=2)  # Make 2 an invalid category.
>>> c3
Categorical([1, 2, 3, -2147483648]) Length: 4
  FastArray([2, 3, 4, 1], dtype=int8) Base Index: 1
  FastArray([-2147483648,           1,           2,           3]) Unique count:␣
↪4
>>> c3.invalid_category()
2
>>> c3.isnotnan()  # Only the invalid category returns False for Cat.isnotnan.
FastArray([ True, False,  True,  True])
>>> c3.expand_array.isnotnan()  # Only the integer NaN returns False for FA.
↪isnotnan.
FastArray([ True,  True,  True, False])
```

**lock**()

Locks the categories to none can be added, removed, or change.

**map**(*mapper*, *invalid=None*)

Maps existing categories to new categories and returns a re-expanded array.

**Parameters**

- **mapper** (*dictionary or numpy.array or* FastArray) –
  - dictionary maps existing categories -> new categories

– array must be the same size as the existing category array

- **invalid** – Optionally specify an invalid value to insert for existing categories that were not found in the new mapping. If no invalid is set, the default invalid for the result's dtype will be used.

**Returns**
Re-expanded array.

**Return type**
*FastArray*

### Notes

Maybe to add: - option to return categorical instead of re-expanding - dtype for return array

### Examples

New strings (all exist, no invalids in original):

```
>>> c = rt.Categorical(['b','b','c','a','d'], ordered=False)
>>> mapping = {'a': 'AA', 'b': 'BB', 'c': 'CC', 'd': 'DD'}
>>> c.map(mapping)
FastArray([b'BB', b'BB', b'CC', b'AA', b'DD'], dtype='|S3')
```

New strings (not all exist, no invalids in original):

```
>>> mapping = {'a': 'AA', 'b': 'BB', 'c': 'CC'}
>>> c.map(mapping, invalid='INVALID')
FastArray([b'BB', b'BB', b'CC', b'AA', b'INVALID'], dtype='|S7')
```

String to float:

```
>>> mapping = {'a': 1., 'b': 2., 'c': 3.}
>>> c.map(mapping, invalid=666)
FastArray([  2.,   2.,   3.,   1., 666.])
```

If no invalid is specified, the default invalid will be used:

```
>>> c.map(mapping)
FastArray([ 2.,  2.,  3.,  1., nan])
```

Mapping as array (must be the same size):

```
>>> mapping = rt.FastArray(['w','x','y','z'])
>>> c.map(mapping)
FastArray([b'w', b'w', b'x', b'y', b'z'], dtype='|S3')
```

**mapping_add**(*code*, *value*)

Add a new code -> value mapping to categories.

**mapping_new**(*mapping*)

Replace entire mapping dictionary. No codes in the Categorical's integer FastArray will be changed. If they are not in the new mapping, they will appear as Invalid.

**mapping_remove**(*code*)

Remove the category associated with an integer code.

**mapping_replace**(*code*, *value*)

Replace a single integer code with a single value.

**classmethod newclassfrominstance**(*instance*, *origin*)

Used when the FastArray portion of the Categorical is updated, but not the reset of the class attributes.

### Examples

```
>>> c=rt.Cat(['a','b','c'])
>>> rt.Cat.newclassfrominstance(c._fa[1:2],c)
Categorical([b]) Length: 1
  FastArray([2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

**notna**(**args*, ***kwargs*)

See *Categorical.isnotnan*.

**nth**(*arr*, *n=1*, *transform=None*, *filter=None*, *showfilter=None*)

Select the nth row from each group.

**Parameters**

- **arr** (*array or list of array*) – The array of values to select from.
- **n** (*int*) – A single nth value for the row.
- **transform** (*bool*) – If `True`, the output will have the same shape as `arr`. If `False`, the output will typically have the same shape as the *Categorical*.
- **filter** (*array of bool, optional*) – Elements to include in the operation.
- **showfilter** (*bool*) – If `True`, the output contains an extra row representing the operation applied to a stack of all the elements that were filtered out (both at *Categorical* creation and in this operation, using a filter.)

### Examples

```
>>> ds = rt.Dataset({'A': rt.Categorical(['a', 'a', 'b', 'a', 'b']),
...                  'B': [rt.nan, 2, 3, 4, 5]})
>>> c = ds.A
>>> c.nth([ds.A, ds.B], 0)
*A      B
--   ----
a     nan
b    3.00

[2 rows x 2 columns] total bytes: 18.0 B
```

```
>>> c.nth([ds.A, ds.B], 1)
*A     B
--  ----
```

(continues on next page)

(continued from previous page)

```
a   2.00
b   5.00

[2 rows x 2 columns] total bytes: 18.0 B
```

```
>>> c.nth([ds.A, ds.B], -1)
*A      B
--   ----
a    4.00
b    5.00

[2 rows x 2 columns] total bytes: 18.0 B
```

```
>>> c.nth(ds.B, -2, transform=True)
#      B
-   ----
0   2.00
1   2.00
2   3.00
3   2.00
4   3.00

[5 rows x 1 columns] total bytes: 40.0 B
```

```
>>> c.nth(ds.B, 1, filter=ds.B.isnotnan())
*A      B
--   ----
a    4.00
b    5.00

[2 rows x 2 columns] total bytes: 18.0 B
```

```
>>> c.nth(ds.B, -2, filter=ds.A!='b', showfilter=True)
*A            B
--------   ----
Filtered   3.00
a          2.00
b           nan

[3 rows x 2 columns] total bytes: 48.0 B
```

**numba_apply**(*userfunc*, **args*, *filter=None*, *transform=False*, ***kwargs*)

Applies a user numba function over the groups of a categorical. Numba function should either return a scalar or np.array the size of the input array. If numba function returns scalar, set transform = True to reshape result to size of categorical.

**Parameters**

- **userfunc** (*a numba function*) –
- **args** (*a np.array, userfunc must return scalar or np.array of same length*) –
- **filter** (*boolean filter*) –

- **kwargs** (*kwargs to pass to userfunc*) –
- **transform** (*Set to true if userfunc returns a scalar, but you want re-expanded to the size of original array*) –

**Return type**
Dataset with categorical keys for scalar function with transform = False, otherwise aligned to original categorical

**nunique**()

Number of unique values that occur in the Categorical. Does not include invalids. Not the same as the length of possible uniques.

Categoricals based on dictionary mapping / enum will return unique count including all possibly invalid values from underlying array.

**See also:**

*Categorical.unique_count*

**one_hot_encode**(*dtype=None*, *categories=None*, *return_labels=True*)

Generate one hot encoded arrays from each unique category.

**Parameters**

- **dtype** (*data-type, optional*) – The numpy data type to use for the one-hot encoded arrays. If `dtype` is not specified (i.e. is `None`), the encoded arrays will default to using a `np.float32` representation.
- **categories** (*list or array-like, optional*) – List or array containing unique category values to one-hot encode. Specify this when you only want to encode a subset of the unique category values. Defaults to None, in which case all categories are encoded.
- **return_labels** (*bool*) – Not implemented.

**Returns**

- **col_names** (*FastArray*) – FastArray of column names (unique categories as unicode strings)
- **encoded_arrays** (*list of FastArray*) – list of one-hot encoded arrays for each category

### Notes

Unicode is used because the column names are often going to a dataset.

Performance warning for large amount of uniques - an array will be generated for ALL of them

### Examples

Default:

```
>>> c = rt.Categorical(FA(['a','a','b','c','a']))
>>> c.one_hot_encode()
(FastArray(['a', 'b', 'c'], dtype='<U1'),
 [FastArray([1., 1., 0., 0., 1.], dtype=float32),
  FastArray([0., 0., 1., 0., 0.], dtype=float32),
  FastArray([0., 0., 0., 1., 0.], dtype=float32)])
```

Custom dtype:

```
>>> c.one_hot_encode(dtype=np.int8)
c.one_hot_encode(dtype=np.int8)
(FastArray(['a', 'b', 'c'], dtype='<U1'),
 [FastArray([1, 1, 0, 0, 1], dtype=int8),
  FastArray([0, 0, 1, 0, 0], dtype=int8),
  FastArray([0, 0, 0, 1, 0], dtype=int8)])
```

Specific categories:

```
>>> c.one_hot_encode(categories=['a','b'])
(FastArray(['a', 'b'], dtype='<U1'),
 [FastArray([ True,  True, False, False,  True]),
  FastArray([False, False,  True, False, False])])
```

Multikey:

```
>>> #NOTE: The double-quotes in the category names are not part of the actual␣
↪string.
>>> c = rt.Categorical([rt.FA(['a','a','b','c','a']), rt.FA([1, 1, 2, 3, 1]) ] )
>>> c.one_hot_encode()
(FastArray(["('a', '1')", "('b', '2')", "('c', '3')"], dtype='<U10'),
 [FastArray([1., 1., 0., 0., 1.], dtype=float32),
  FastArray([0., 0., 1., 0., 0.], dtype=float32),
  FastArray([0., 0., 0., 1., 0.], dtype=float32)])
```

Mapping:

```
>>> c = rt.Categorical(rt.arange(3), {'a':0, 'b':1, 'c':2})
>>> c.one_hot_encode()
(FastArray(['a', 'b', 'c'], dtype='<U1'),
 [FastArray([1., 0., 0.], dtype=float32),
  FastArray([0., 1., 0.], dtype=float32),
  FastArray([0., 0., 1.], dtype=float32)])
```

**set_name**(*name*)

If the grouping dict contains a single item, rename it.

**See also:**

`Grouping.set_name`, `FastArray.set_name`

**set_valid**(*filter=None*)

Apply a filter to the categorical's values. If values no longer occur in the uniques, the uniques will be reduced, and the index will be recalculated.

**Parameters**

**filter** (*boolean array, optional*) – If provided, must be the same size as the categorical's underlying array. Will be used to mask non-unique values. If not provided, categorical may still reduce its unique values to the unique occuring values.

**Returns**

**c** – New categorical with possibly reduced uniques.

**Return type**

*Categorical*

**shift**(*arr*, *window=None*, *, *periods=None*, *filter=None*)

Shift values in each group by the specified number of periods.

Where the shift introduces a missing value, the missing value is filled with the invalid value for the array's data type (for example, NaN for floating-point arrays or the sentinel value for integer arrays).

**Parameters**

- **arr** (*array or list of array*) – The array of values to shift.
- **window** (*int, default 1*) – The number of periods to shift. Can be a negative number to shift values backward.
- **periods** (*int, optional, default 1*) – Can use `periods` instead of `window` for Pandas parameter support.
- **filter** (*FastArray of bool, optional*) – Set of rows to include. Filtered out rows are skipped by the shift and become NaN in the output.

**Returns**

A *Dataset* containing a column of shifted values.

**Return type**

*Dataset*

**See also:**

***Categorical.shift_cat***

Shift the values of a *Categorical*.

***FastArray.shift***

Shift the values of a *FastArray*.

***DateTimeNano.shift***

Shift the values of a *DateTimeNano* array.

### Examples

With the default `window=1`:

```
>>> c = rt.Cat(['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c'])
>>> fa = rt.arange(9)
>>> shift_val = c.shift(fa)
>>> shift_val
#   col_0
-   -----
0     Inv
1       0
2       1
3     Inv
4       3
5       4
6     Inv
7       6
8       7
```

With `window=2`:

```
>>> shift_val_2 = c.shift(fa, window=2)
>>> shift_val_2
#   col_0
-   -----
0     Inv
1     Inv
2       0
3     Inv
4     Inv
5       3
6     Inv
7     Inv
8       6
```

With `window=-1`:

```
>>> shift_neg = c.shift(fa, window=-1)
>>> shift_neg
#   col_0
-   -----
0       1
1       2
2     Inv
3       4
4       5
5     Inv
6       7
7       8
8     Inv
```

With `filter`:

```
>>> filt = rt.FA([True, True, True, True, False, True, False, True, True])
>>> shift_filt = c.shift(fa, filter=filt)
>>> shift_filt
#   col_0
-   -----
0     Inv
1       0
2       1
3     Inv
4     Inv
5       3
6     Inv
7     Inv
8       7
```

Results put in a *Dataset* to show the shifts in relation to the categories:

```
>>> ds = rt.Dataset()
>>> ds.c = c
>>> ds.shift_val = shift_val
>>> ds.shift_val_2 = shift_val_2
```

(continues on next page)

(continued from previous page)

```
>>> ds.shift_neg = shift_neg
>>> ds
#   c   shift_val   shift_val_2   shift_neg
-   -   ---------   -----------   ---------
0   a         Inv           Inv           1
1   a           0           Inv           2
2   a           1             0         Inv
3   b         Inv           Inv           4
4   b           3           Inv           5
5   b           4             3         Inv
6   c         Inv           Inv           7
7   c           6           Inv           8
8   c           7             6         Inv
```

Shift two arrays:

```
>>> fa2 = rt.arange(10, 19)
>>> shift_val_3 = c.shift([fa, fa2])
>>> shift_val_3
#   col_0   col_1
-   -----   -----
0     Inv     Inv
1       0      10
2       1      11
3     Inv     Inv
4       3      13
5       4      14
6     Inv     Inv
7       6      16
8       7      17
```

**shift_cat**(*periods=1*)

See FastArray.shift() Instead of nan or sentinel values, like shift on a FastArray, the invalid category will appear. Returns a new categorical.

### Examples

```
>>> rt.Cat(['a','b','c']).shift(1)
Categorical([Filtered, a, b]) Length: 3
  FastArray([0, 1, 2], dtype=int8) Base Index: 1
  FastArray([b'a', b'b', b'c'], dtype='|S1') Unique count: 3
```

**shrink**(*newcats*, *misc=None*, *inplace=False*)

**Parameters**

- **newcats** (*array-like*) – New categories to replace the old - typically a reduced set.
- **misc** (*scalar, optional (often a string)*) – Value to use as category for items not found in new categories. This will be added to the new categories. If not provided, all items not found will be set to a filtered bin.
- **inplace** (*bool*) – If True, re-index the categorical's underlying FastArray. Otherwise, return a new categorical with a new index and grouping object.

**Returns**
A new Categorical with the new index.

**Return type**
*Categorical*

### Examples

Base index 1, no misc

```
>>> c = rt.Categorical([1,2,3,1,2,3,0], ['a','b','c'])
>>> c.shrink(['b','c'])
Categorical([Filtered, b, c, Filtered, b, c, Filtered]) Length: 7
  FastArray([0, 1, 2, 0, 1, 2, 0]) Base Index: 1
  FastArray([b'b', b'c'], dtype='|S1') Unique count: 2
```

Base index 1, filtered bins and misc

```
>>> c.shrink(['b','c'], 'AAA').sum(rt.arange(7), showfilter=True)
*key_0     col_0
--------   -----
Filtered       6
AAA            3
b              5
c              7
```

Base index 0, with misc

```
>>> c = rt.Categorical([0,1,2,0,1,2], ['a','b','c'], base_index=0)
>>> c.shrink(['b','c'], 'AAA')
Categorical([AAA, b, c, AAA, b, c]) Length: 6
  FastArray([0, 1, 2, 0, 1, 2], dtype=int8) Base Index: 0
  FastArray(['AAA', 'b', 'c'], dtype='<U3') Unique count: 3
```

**See also:**

*Categorical.map*

**str()**

Casts an array of byte strings or unicode as `FAString`.

Enables a variety of useful string manipulation methods.

**Return type**
*FAString*

**Raises**
**TypeError** – If the FastArray is of dtype other than byte string or unicode

**See also:**

`np.chararray`, `np.char`, `rt.FAString.apply`

### Examples

```
>>> s=FA(['this','that','test ']*100_000)
>>> s.str.upper
FastArray([b'THIS', b'THAT', b'TEST ', ..., b'THIS', b'THAT', b'TEST '],
          dtype='|S5')
```

```
>>> s.str.lower
FastArray([b'this', b'that', b'test ', ..., b'this', b'that', b'test '],
          dtype='|S5')
```

```
>>> s.str.removetrailing()
FastArray([b'this', b'that', b'test', ..., b'this', b'that', b'test'],
          dtype='|S5')
```

**to_arrow**(*type=None*, ***, *preserve_fixed_bytes=False*, *empty_strings_to_null=True*)

Convert this *Categorical* to a pyarrow.Array.

**Parameters**

- **type** (*pyarrow.DataType, optional, defaults to None*) – Unused.
- **preserve_fixed_bytes** (*bool, optional, defaults to False*) – Unused.
- **empty_strings_to_null** (*bool, optional, defaults To True*) – Unused.

**Return type**

pyarrow.Array or pyarrow.ChunkedArray

### Notes

**TODO: Consider whether we should store all Categoricals as Struct-type pyarrow arrays, since that'd**
allow us to preserve the key names, even for single-key Categoricals.

**unlock**()

Unlocks the categories so new categories can be added, or existing categories can be removed or changed.

**class** riptable.rt_categorical.**Categories**(**args*, *base_index=1*, *invalid_category=None*, *ordered=False*, *unicode=False*, *_from_categorical=False*, ***kwargs*)

Holds categories for each Categorical instance. This adds a layer of abstraction to Categorical.

Categories objects are constructed in Categorical's constructor and other internal routines such as merging operations. The Categories object is responsible for translating the values in the Categorical's underlying fast array into the correct bin in the categories. It performs different operations to retrieve the correct bins based on it's mode.

**Parameters**

- **categories** – main categories data - can also be empty list
- **invalid_category** (*str*) – string that will be displayed for an invalid index
- **invalid_index** – sentinel value for a particular index; this invalid will be displayed differntly in IntEnum/Dictionary modes
- **ordered** (*bool*) – flag for list list modes, ordered categories can use a binary search for finding bins

- **auto_add_categories** – if a setitem (bracket-indexing with a value) is called, and the value is not in the categories, this flag allows it to be added automatically.
- **na_added** – for some constructors, the calling Categorical has already added the invalid category
- **base_index** – the calling Categorical passes in the index offset for list and grouping modes
- **multikey** – the categories information is stored in a multikey dictionary *up for deletion*
- **groupby** – *possibly merge with the multikey flag*

### Notes

There are multiple modes in which a Categories object can operate.

**StringArray**: *(list_modes)* Two paths for initializations use the categories routines: TB Filled in LATER array and list of unique categories. String mode will be set to unicode or bytes so the correct encoding/decoding can be performed before comparison/searching operations. - from list of strings (unique/ismember) - from list of strings paired with unique string categories (unique/ismember) - from codes paired with unique string categories (assignment will happen without unique/ismember) - from pandas categoricals (with string categories) (assignment will happen without unique/ismember) - from matlab categoricals (with string categories) (assignment will happen without unique/ismember)

**NumericArray:** *(list_modes)* this is not currently implemented as default behavior, but if enabled it will handle these constructors - from list of integers - from list of floats - from codes paired with unique integer categories - from codes paired with unique float categories - from list of floats paired with unique float categories - from pandas categoricals with numeric categories

**IntEnum / Dictionary:** *(dict_modes)* Two dictionaries will be held: one mapping strings to integers, another mapping integers to strings. This mode requires that all strings and their corresponding codes are one-to-one. - from codes paired with IntEnum object - from codes paired with Integer -> String dictionary - from codes paired wtih String -> Integer dictionary *not implemented*

**Grouping** All categories objects in Grouping mode hold categories in a dictionary, even if the dictionary only contains one item. Information for indexed items will appear in a tuple if multiple columns are being held. - from list of key columns - from dictionary of key columns - from single list of numeric type - from dataset *not implemented*

**property _first_list**

Returns the first column when categories are in a dictionary, or the list if the categories are in a list mode.

**property base_index**

**property grouping**

**property int2strdict**

**property isbytes**

True if uniques are held in single array of bytes. Otherwise False.

**property isenum**

True if uniques have an enum / dictionary mapping for uniques. Otherwise False.

See also: GroupingEnum

**property ismultikey**

True if unique dict holds multiple arrays. False if unique dict holds single array or in enum mode.

**property issinglekey**

True if unique dict holds single array. False if unique dict hodls multiple arrays or in enum mode.

**property isunicode**

True if uniques are held in single array of unicode. Otherwise False.

**property mode**

**property name: str**

**property ncols: int**

Returns the number of key columns in a multikey categorical or 1 if a single key's categories are being held in a dictionary.

**property nrows: int**

Returns the number of unique categories in a multikey categorical.

**property str2intdict**

**property uniquedict**

**property uniquelist**

**_grouping: *riptable.rt_grouping.Grouping***

**default_colname = 'key_0'**

**dict_modes**

**list_modes**

**multikey_spacer = ' '**

**numeric_modes**

**string_modes**

**__getitem__**(*value*)

**__len__**()

TODO: consider changing length of enum/dict mode categories to be the length of the dictionary. using max int so the calling Categorical can properly recast the integer array.

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**_array_edit**(*value*, *new_value=None*, *how='add'*)

**_build_string**()

**_copy**(*deep=True*)

Creates a new categories object and possibly performs a deep copy of category list. Currently only supports Categories in list modes.

**_get_array**()

**_get_codes**()

**_get_dict**()

**_get_mapping**()

**_getitem_enum**(*value*)

At this point, the categorical's underlying fast array's __getitem__ has already been hit. It will only execute if the return value was scalar. No need to handle lists/arrays/etc. - which take a different path in Categorical.__getitem__

The value should always be a single integer.

this will return a single item or list of items from int/string index Enums will always return an array of values, even if there is only one entry. Enums dictionaries can only be looked up with unicode strings, so bytes will be converted.

**_getitem_multikey**(*value*)

**_getitem_singlekey**(*value*)

**_is_valid_mapping_code**(*value*)

**_mapping_edit**(*code*, *value=None*, *how='add'*)

**_mapping_new**(*mapping*)

**_possibly_add_categories**(*new_categories*)

Add non-existing categories to categories. If categories were added, an array is returned to fix the old indexes. If no categories were added, returns None.

**classmethod build_dicts_enum**(*enum*)

Builds forward/backward dictionaries from IntEnums. If there are multiple identifiers with the same, WARN!

**classmethod build_dicts_python**(*python_dict*)

Categoricals can be initialized with a dictionary of string to integer or integer to string. Python dictionaries accept multiple types for their keys, so the dictionaries need to check types as they're being constructed.

**categories_as_dict**()

Groupby keys can be prepared for the calling Categorical.

**copy**(*deep=True*)

Wrapper for internal _copy.

**classmethod from_grouping**(*grouping*, *invalid_category=None*)

**get_categories**()

TODO: decide what to return for int enum categories. for now returning list of category strings

**get_category_index**(*s*)

Returns an integer or float for logical comparisons with the Categorical's index array. Floating point return ensures that LTE/GTE functions work properly

**get_category_match_index**(*fld*)

Returns the indices of matching strings in the unique list. The Categorical instance will compare these integers to those in its underlying array to generate a boolean mask.

**get_multikey_index**(*multikey*)

Multikey categoricals can be indexed by tuple. This is an internal routine for getitem, setitem, and logical comparisons. Valid return will be adjusted for the base index of the categorical (currently always 1 for multikey)

**Parameters**
**multikey** (*tuple of items to search for in multiple columns*) –

**Returns**
location of multikey + base index, or -1 if not found

**Return type**
int

### Examples

```
>>> c = rt.Categorical([rt.arange(5), rt.arange(5)])
>>> c
Categorical([(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]) Length: 5
  FastArray([1, 2, 3, 4, 5], dtype=int8) Base Index: 1
  {'key_0': FastArray([0, 1, 2, 3, 4]), 'key_1': FastArray([0, 1, 2, 3, 4])}␣
↪Unique count: 5
```

```
>>> c._categories_wrap.get_multikey_index((0,0))
1
```

**match_str_to_category**(*fld*)

If necessary, convert the string or list of strings to the same type as the categories so that correct comparisons can be made.

**possibly_invalid**(*value*)

If the calling categorical's values are set to a bad index, the !<badindex> will be returned. If the bad index is the sentinel value for that integer type, !<inv> will be returned

riptable.rt_categorical.**CatZero**(*values*, *categories=None*, *ordered=None*, *sort_gb=None*, *lex=None*, *base_index=0*, ***kwargs*)

Calls Categorical() with base_index keyword set to 0.

riptable.rt_categorical.**categorical_convert**(*v*, *base_index=0*)

**Parameters**
**v** (*a pandas categorical*) –

**Returns**

- **Returns the two building blocks to make an rt categorical** (*integer array, and what that indexes into*)
- *whatever the pandas categorical underlying object is we try to convert it to a string to*
- *detach from object references and free of pandas references*
- *pandas also uses -1 to indicate an out of bounds value, when we detect this, we insert an item in the beginning*

#### Examples

```
>>> p=pd.Categorical(['a','b','b','a','a','c','b','c','a','a'], categories=['a','b
↪'])
>>> test=Categorical(p)
```

from a cut

```
>>> a=rt.FA(rt.arange(10.0)+.1)
>>> p=pd.cut(a,[0,3,6,7])
(0, 3], (0, 3], (3, 6], (3, 6], (3, 6], (6, 7], NaN, NaN, NaN]
>>> test=Categorical(p)
Categorical([(0, 3], (0, 3], (0, 3], (3, 6], (3, 6], (3, 6], (6, 7], nan, nan, nan])
```

riptable.rt_categorical.**categorical_merge_dict**(*list_categories*, *return_is_safe=False*, *return_type=Categorical*)

Checks to make sure all unique string values in all dictionaries have the same corresponding integer in every categorical they appear in. Checks to make sure all unique integer values in all dictionaries have the same corresponding string in every categorical they appear in.

### 2.2.8 riptable.rt_compressedarray

#### Classes

| | |
|---|---|
| *CompressedArray* | A FastArray is a 1-dimensional array of items that are the same data type. |

**class** riptable.rt_compressedarray.**CompressedArray**(*arr*)

Bases: *riptable.rt_fastarray.FastArray*

A FastArray is a 1-dimensional array of items that are the same data type.

Because it's a subclass of NumPy's numpy.ndarray, all ndarray functions and attributes can be used with FastArray objects. However, Riptable optimizes many of NumPy's functions to make them faster and more memory-efficient. Riptable has also added some methods.

FastArray objects with more than 1 dimension are not supported.

See NumPy's docs for details on all ndarray methods and attributes.

**Parameters**

- **arr** (*array, iterable, or scalar value*) – Contains data to be stored in the FastArray.
- ****kwargs** – Additional keyword arguments to be passed to the function.

### Notes

To improve performance, `FastArray` objects take over some of NumPy's universal functions (ufuncs), use array recycling and multiple threads, and pass certain method calls to Bottleneck.

Note that whenever Riptable has implemented its own version of an existing NumPy method, a call to the NumPy method results in a call to the optimized Riptable version instead. We encourage users to directly call the Riptable method in order to avoid any confusion as to what method is actually being called.

See the list of NumPy Methods Optimized by Riptable for FastArrays.

### Examples

**Construct a FastArray**

Pass a list to the constructor:

```
>>> rt.FastArray([1, 2, 3, 4, 5])
FastArray([1, 2, 3, 4, 5])
```

```
>>> #NOTE: rt.FA also works.
>>> rt.FA([1.0, 2.0, 3.0, 4.0, 5.0])
FastArray([1., 2., 3., 4., 5.])
```

Or use a utility function:

```
>>> rt.full(10, 0.7)
FastArray([0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7])
```

```
>>> rt.arange(10)
FastArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You can optionally specify a data type:

```
>>> x = rt.FastArray([3, 6, 10],  dtype = rt.float64)
>>> x, x.dtype
(FastArray([ 3.,  6., 10.]), dtype('float64'))
```

```
>>> # Using a string shortcut:
>>> x = rt.FastArray([3,6,10],  dtype = 'float64')
>>> x, x.dtype
(FastArray([ 3.,  6., 10.]), dtype('float64'))
```

By default, characters are stored as byte strings. When `unicode=True`, the `FastArray` allows Unicode characters.

```
>>> rt.FA(list('abc'), unicode=True)
FastArray(['a', 'b', 'c'], dtype='<U1')
```

To convert an existing NumPy array, use the `FastArray` constructor.

```
>>> np_arr = np.array([1, 2, 3])
>>> rt.FA(np_arr)
FastArray([1, 2, 3])
```

To view the NumPy array as a `FastArray` (which is slightly less expensive than using the constructor), use the `view` method.

```
>>> fa = np_arr.view(FA)
>>> fa
FastArray([1, 2, 3])
```

To view it as a NumPy array again:

```
>>> fa.view(np.ndarray)
array([1, 2, 3])
```

```
>>> # Alternatively:
>>> fa._np
array([1, 2, 3])
```

**Get a Subset of a FastArray**

You can use standard Python slicing notation or fancy indexing to access a subset of a `FastArray`.

```
>>> # Create a FastArray:
>>> array = rt.arange(8)**2
>>> array
FastArray([0, 1, 4, 9, 16, 25, 36, 49])
>>> # Use Python slicing to get elements 2, 3, and 4:
>>> array[2:5]
FastArray([4, 9, 16])
```

```
>>> # Use fancy indexing to get elements 2, 4, and 1 (in that order):
>>> array[[2, 4, 1]]
FastArray([4, 16, 1])
```

For more details, see the examples for 1-dimensional arrays in NumPy's docs: Indexing on ndarrays.

Note that slicing creates a view of the array and does not copy the underlying data; modifying the slice modifies the original array. Fancy indexing creates a copy of the extracted data; modifying this array does not modify the original array.

You can also pass a Boolean mask array.

```
>>> # Create a Boolean mask:
>>> evenMask = (array % 2 == 0)
>>> evenMask
FastArray([True, False, True, False, True, False, True, False])
>>> # Index using the Boolean mask:
>>> array[evenMask]
FastArray([0, 4, 16, 36])
```

**How to Subclass FastArray**

Include the required class definition:

```
>>> class TestSubclass(FastArray):
...     def __new__(cls, arr, **args):
...         # Before this call, arr needs to be a np.ndarray instance.
...         return arr.view(cls)
```

(continues on next page)

(continued from previous page)

```
...     def __init__(self, arr, **args):
...         pass
```

If the subclass is computable, you might define your own math operations. In these operations, you might define what the subclass can be computed with. For examples of new definitions, see the DateTimeNano class.

Common operations to hook are comparisons (__eq__(), __ne__(), __gt__(), __lt__(), __le__(), __ge__()) and basic math functions (__add__(), __sub__(), __mul__(), etc.).

Bracket indexing operations are very common. If the subclass needs to set or return a value other than that in the underlying array, you need to take over __getitem__() or __setitem__().

Indexing is also used in display. For regular console/notebook display, you need to take over:

- *__repr__()*
- *__str__()*
- _repr_html_() (for JupyterLab and Jupyter notebooks)

If the array is being displayed in a Dataset and you require certain formatting, you need to define two more methods:

**display_query_properties()**
Returns an ItemFormat object (see rt.Utils.rt_display_properties)

**display_convert_func()**
The conversion function returned by display_query_properties() must return a string. Each item being displayed, the result of __getitem__() at a single index, will go through this function individually, accompanied by an ItemFormat object.

Many Riptable operations need to return arrays of the same class they received. To ensure that your subclass will retain its special properties, you need to take over newclassfrominstance(). Failure to take this over will often result in an object with uninitialized variables.

copy() is another method that is called generically in Riptable routines, and needs to be taken over to retain subclass properties.

For a view of the underlying FastArray, you can use the _fa property.

**allowed_funcs = ['decompress', 'view']**

**__getattribute__**(*attr*)
Block all FastArray operations. See allowed_funcs class global.

**__repr__**()
Return repr(self).

**__str__**()
Return str(self).

**_build_string**()

**decompress**()

### 2.2.9 riptable.rt_csv

#### Functions

| | |
|---|---|
| *load_csv_as_dataset*(path_or_file[, column_names, ...]) | Load a Dataset from a comma-separated value (CSV) file. |

riptable.rt_csv.**load_csv_as_dataset**(*path_or_file*, *column_names=None*, *converters=None*, *skip_rows=0*, *version=None*, *encoding='utf-8'*, ***kwargs*)

Load a Dataset from a comma-separated value (CSV) file.

**Parameters**

- **path_or_file** – A filename or a file-like object (from open() or StringIO()); if you need a non-standard encoding, do the open yourself.
- **column_names** (*list of str, optional*) – List of column names (must be legal python var names), or None for 'use first row read from file'. Defaults to None.
- **converters** (*dict*) – {column_name -> str2type-converters}, do your own error handling, should return uniform types, and handle bad/missing data as desired missing converter will default to 'leave as string'.
- **skip_rows** (*int*) – Number of rows to skip before processing, defaults to 0.
- **version** (*int, optional*) – Selects the implementation of the CSV parser used to read the input file. Defaults to None, in which case the function chooses the best available implementation.
- **encoding** (*str*) – The text encoding of the CSV file, defaults to 'utf-8'.
- **kwargs** – Any csv 'dialect' params you like.

**Return type**
*Dataset*

#### Notes

For a dataset of shape (459302, 15) (all strings) the timings are roughly: (version=0) 6.195947s (version=1) 5.605156s (default if pandas not available) (version=2) 8.370234s (version=3) 6.994191s (version=4) 3.642205s (only available if pandas is available, default if so)

### 2.2.10 riptable.rt_dataset

#### Classes

| | |
|---|---|
| *Dataset* | The Dataset class is the workhorse of riptable; it may be considered as an NxK array of values (of mixed type, |

**class** riptable.rt_dataset.**Dataset**(*inputval=None*, *base_index=0*, *sort=False*, *unicode=False*)

Bases: *riptable.rt_struct.Struct*

The Dataset class is the workhorse of riptable; it may be considered as an NxK array of values (of mixed type, constant by column) where the rows are integer indexed and the columns are indexed by name (as well as integer index). Alternatively it may be regarded as a dictionary of arrays, all of the same length.

The Dataset constructor takes dictionaries (dict, OrderedDict, etc…), as well as single instances of Dataset or Struct (if all entries are of the same length). Dataset() := Dataset({}).

The constructor dictionary keys (or element/column names added later) must be legal Python variable names, not starting with '_' and not conflicting with any Dataset member names.

**Column indexing behavior**:

```
>>> st['b'] # get a column (equiv. st.b)
>>> st[['a', 'e']] # get some columns
>>> st[[0, 4]] # get some columns (order is that of iterating st (== list(st))
>>> st[1:5:2] # standard slice notation, indexing corresponding to previous
>>> st[bool_vector_len5] # get 'True' columns
```

In all of the above: `st[col_spec] := st[:, colspec]`

**Row indexing behavior**:

```
>>> st[2, :] # get a row (all columns)
>>> st[[3, 7], :] # get some rows (all columns)
>>> st[1:5:2, :] # standard slice notation (all columns)
>>> st[bool_vector_len5, :] # get 'True' rows (all columns)
>>> st[row_spec, col_spec] # get specified rows for specified columns
```

Note that because `st[spec] := st[:, spec]`, to specify rows one *must* specify columns as well, at least as 'the all-slice': e.g., `st[row_spec, :]`.

Wherever possible, views into the original data are returned. Use `copy()` where necessary.

### Examples

A Dataset with six integral columns of length 10:

```
>>> import string
>>> ds = rt.Dataset({_k: list(range(_i * 10, (_i + 1) * 10)) for _i, _k in␣
↪enumerate(string.ascii_lowercase[:6])})
```

Add a column of strings (stored internally as ascii bytes):

```
>>> ds.S = list('ABCDEFGHIJ')
```

Add a column of non-ascii strings (stored internally as a Categorical column):

```
>>> ds.U = list('ø-613')
>>> print(ds)
#   a    b    c    d    e    f   S   U
-   -   --   --   --   --   --   -   -
0   0   10   20   30   40   50   A
1   1   11   21   31   41   51   B
```

(continues on next page)

(continued from previous page)

```
2   2   12   22   32   42   52   C
3   3   13   23   33   43   53   D
4   4   14   24   34   44   54   E   ø
5   5   15   25   35   45   55   F
6   6   16   26   36   46   56   G   -
7   7   17   27   37   47   57   H   6
8   8   18   28   38   48   58   I   1
9   9   19   29   39   49   59   J   3
```

```
>>> ds.get_ncols()
8
>>> ds.get_nrows()
10
```

`len` applied to a Dataset returns the number of rows in the Dataset.

```
>>> len(ds)
10
>>> # Not too dissimilar from numpy/pandas in many ways.
>>> ds.shape
(10, 8)
>>> ds.size
80
>>> ds.head()
>>> ds.tail(n=3)
```

```
>>> assert (ds.c == ds['c']).all() and (ds.c == ds[2]).all()
```

```
>>> print(ds[1:8:3, :3])
#   a    b    c
-   -   --   --
0   1   11   21
1   4   14   24
2   7   17   27
```

```
>>> ds.newcol = np.arange(100, 110) # okay, a new entry
>>> ds.newcol = np.arange(200, 210) # okay, replace the entry
>>> ds['another'] = 6 # okay (scalar is promoted to correct length vector)
>>> ds['another'] = ds.another.astype(np.float32) # redefines type of column
```

```
>>> ds.col_remove(['newcol', 'another'])
```

Fancy indexing for get/set:

```
>>> ds[1:8:3, :3] = ds[2:9:3, ['d', 'e', 'f']]
```

Equivalents:

```
>>> for colname in ds: print(colname, ds[colname])
>>> for colname, array in ds.items(): print(colname, array)
>>> for colname, array in zip(ds.keys(), ds.values()): print(colname, array)
>>> for colname, array in zip(ds, ds.values()): print(colname, array)
```

```
>>> if key in ds:
...     assert getattr(ds, key) is ds[key]
```

Context manager:

```
>>> with Dataset({'a': 1, 'b': 'fish'}) as ds0:
...     print(ds0.a)
[1]
```

```
>>> assert not hasattr(ds0, 'a')
```

Dataset cannot be used in a boolean context (`if ds: ...`), use `ds.any(axis='all')` or `ds.all(axis='all')` instead:

```
>>> ds1 = ds[:-2] # Drop the string columns, Categoricals are 'funny' here.
>>> ds1.any(axis='all')
True
```

```
>>> ds1.all(axis='all')
False
```

```
>>> ds1.a[0] = -99
>>> ds1.all(axis='all')
True
```

```
>>> if (ds2 <= ds3).all(axis='all'): ...
```

Do math:

```
>>> ds1 += 5
>>> ds1 + 3 * ds2 - np.ones(10)
>>> ds1 ** 5
>>> ds.abs()
```

```
>>> ds.sum(axis=0, as_dataset=True)
    #    a     b     c     d     e     f
    -   --   ---   ---   ---   ---   ---
    0   39   238   338   345   445   545
```

```
>>> ds.sum(axis=1)
array([ 51, 249, 162, 168, 267, 180, 186, 285, 198, 204])
```

```
>>> ds.sum(axis=None)
1950
```

**property _sort_columns**

Subclasses can define their own callback function to return columns they were sorted by, and styles. Callback function will receive trimmed fancy index (based on sort index) and return a dictionary of column headers -> (masked_array, ColumnStyle objects) These columns will be moved to the left side of the table (but to the right of row labels, groupbykeys, row numbers, etc.)

**property crc: *Dataset***

Returns a new dataset with the 64 bit CRC value of every column.

Useful for comparing the binary equality of columns in two datasets

### Examples

```
>>> ds1 = rt.Dataset({'test': rt.arange(100), 'test2': rt.arange(100.0)})
>>> ds2 = rt.Dataset({'test': rt.arange(100), 'test2': rt.arange(100)})
>>> ds1.crc == ds2.crc
#   test   test2
-   ----   -----
0   True   False
```

**property dtypes: Mapping[str, numpy.dtype]**

The data type of each *Dataset* column.

**Returns**
Dictionary containing each column's name/label and dtype.

**Return type**
dict

### Examples

```
>>> ds = rt.Dataset({'Int' : [1], 'Float' : [1.0], 'String': ['aaa']})
>>> ds.dtypes
{'Int': dtype('int32'), 'Float': dtype('float64'), 'String': dtype('S3')}
```

**property imatrix: numpy.ndarray | None**

Returns the 2d array created from *imatrix_make*.

**Returns**
**imatrix** – If *imatrix_make* was previously called, returns the 2D array created and cached internally by that method. Otherwise, returns None.

**Return type**
np.ndarray, optional

### Examples

```
>>> ds = rt.Dataset({'a': np.arange(-3,3), 'b':np.arange(6), 'c':np.arange(10,
↪70,10)})
>>> ds
#    a   b    c
-   --   -   --
0   -3   0   10
1   -2   1   20
2   -1   2   30
3    0   3   40
4    1   4   50
5    2   5   60
```

```
>>> ds.imatrix  # returns nothing since we have not called imatrix_make
>>> ds.imatrix_make()
FastArray([[-3,  0, 10],
           [-2,  1, 20],
           [-1,  2, 30],
           [ 0,  3, 40],
           [ 1,  4, 50],
           [ 2,  5, 60]])
>>> ds.imatrix
FastArray([[-3,  0, 10],
           [-2,  1, 20],
           [-1,  2, 30],
           [ 0,  3, 40],
           [ 1,  4, 50],
           [ 2,  5, 60]])
```

```
>>> ds.a = np.arange(6)
>>> ds
#   a   b    c
-   -   -   --
0   0   0   10
1   1   1   20
2   2   2   30
3   3   3   40
4   4   4   50
5   5   5   60
```

```
>>> ds.imatrix    # even after changing the dataset, the matrix remains the␣
↪same.
FastArray([[-3,  0, 10],
           [-2,  1, 20],
           [-1,  2, 30],
           [ 0,  3, 40],
           [ 1,  4, 50],
           [ 2,  5, 60]])
```

**property imatrix_cls**

Returns the IMatrix class created by *imatrix_make*.

**property imatrix_ds**

Returns the dataset of the 2d array created from *imatrix_make*.

### Examples

```
>>> ds = rt.Dataset({'a': np.arange(-3,3), 'b':np.arange(6), 'c':np.arange(10,
↪70,10)})
>>> ds
#    a   b    c
-   --   -   --
0   -3   0   10
1   -2   1   20
```

(continues on next page)

(continued from previous page)

```
2   -1   2   30
3    0   3   40
4    1   4   50
5    2   5   60

[6 rows x 3 columns] total bytes: 144.0 B
```

```
>>> ds.imatrix_make(colnames = ['a', 'c'])
FastArray([[-3, 10],
           [-2, 20],
           [-1, 30],
           [ 0, 40],
           [ 1, 50],
           [ 2, 60]])
```

```
>>> ds.imatrix_ds
#    a    c
-   --   --
0   -3   10
1   -2   20
2   -1   30
3    0   40
4    1   50
5    2   60
```

**property memory_stats: None**

**property size: int**

The number of elements in the *Dataset* (the number of rows times the number of columns).

**Returns**

The number of elements in the *Dataset* (nrows x ncols).

**Return type**

int

**See also:**

***Dataset.get_nrows***

The number of elements in each column of a *Dataset*.

**Struct.get_ncols**

The number of items in a Struct or the number of elements in each row of a *Dataset*.

**Struct.shape**

A tuple containing the number of rows and columns in a Struct or *Dataset*.

### Examples

```
>>> ds = rt.Dataset({'A': [1.0, 2.0], 'B': [3, 4], 'C': ['c', 'c']})
>>> ds.size
6
```

**property total_size: int**

Returns total size of all (columnar) data in bytes.

**Returns**
The total size, in bytes, of all columnar data in this instance.

**Return type**
int

**__abs__**()

**__add__**(*lhs*)

**__and__**(*lhs*)

**__del__**()

**__eq__**(*lhs*)

Return self==value.

**__floordiv__**(*lhs*)

**__ge__**(*lhs*)

Return self>=value.

**__getitem__**(*index*)

**Parameters**
**index** (*(rowspec, colspec) or colspec*) –

**Return type**
the indexed row(s), cols(s), sub-dataset or single value

**Raises**

- **IndexError** – When an invalid column name is supplied.
- **TypeError** –

**__gt__**(*lhs*)

Return self>value.

**__iadd__**(*lhs*)

**__iand__**(*lhs*)

**__ifloordiv__**(*lhs*)

**__ilshift__**(*lhs*)

**__imod__**(*lhs*)

**__imul__**(*lhs*)

__invert__()

__ior__(*lhs*)

__ipow__(*lhs*, *modulo=None*)

__irshift__(*lhs*)

__isub__(*lhs*)

__itruediv__(*lhs*)

__ixor__(*lhs*)

__le__(*lhs*)
Return self<=value.

__len__()

__lshift__(*lhs*)

__lt__(*lhs*)
Return self<value.

__mod__(*lhs*)

__mul__(*lhs*)

__ne__(*lhs*)
Return self!=value.

__neg__()

__or__(*lhs*)

__pos__()

__pow__(*lhs*, *modulo=None*)

__radd__(*lhs*)

__rand__(*lhs*)

__repr__()
Return repr(self).

__rfloordiv__(*lhs*)

__rmod__(*lhs*)

__rmul__(*lhs*)

__ror__(*lhs*)

__rpow__(*lhs*)

__rshift__(*lhs*)

__rsub__(*lhs*)

**__rtruediv__**(*lhs*)

**__rxor__**(*lhs*)

**__setitem__**(*fld*, *value*)

**Parameters**

- **fld** (*(rowspec, colspec) or colspec (=> rowspec of :)*) –
- **value** (*scalar, sequence or dataset value*) –
  - Scalar is always valid.
  - If (rowspec, colspec) is an NxK selection:
    * (1xK), K>1: allow `|sequence| == K`
    * (Nx1), N>1: allow `|sequence| == N`
    * (NxK), N, K>1: allow only w/ `|dataset| = NxK`
  - Sequence can be list, tuple, np.ndarray, FastArray

**Raises**

IndexError –

**__str__**()

Return str(self).

**__sub__**(*lhs*)

**__truediv__**(*lhs*)

**__xor__**(*lhs*)

**_add_allnames**(*colname*, *arr*, *nrows*)

Internal routine used to add columns only when AllNames is True.

**_add_labels_footers_summaries**(*ret_obj*, *summary_colnames*, *footers*)

**_apply_outlier**(*func*, *name*, *col_keep*)

**_as_itemcontainer**(*deep=False*, *rows=None*, *cols=None*, *base_index=0*)

Returns an ItemContainer object for quick reconstruction or slicing/indexing of a dataset. Will perform a deep copy if requested and necessary.

**_autocomplete**()

**static _axis_key**(*axis*)

**_check_add_dimensions**(*col*)

Used in _init_from_dict and _replaceitem. If _nrows has not been set, it will be here.

**_check_addtype**(*name*, *value*)

override to check types

**_construct_new_footers**(*arrays*, *num_labels*, *summary_colnames*)

**_copy**(*deep=False*, *rows=None*, *cols=None*, *base_index=0*, *cls=None*)

Bracket indexing that returns a dataset will funnel into this routine.

deep : if True, perform a deep copy on column array rows : row mask cols : column mask base_index : used for head/tail slicing cls : class of return type, for subclass super() calls First argument must be deep. Deep cannnot be set to None. It must be True or False.

**_copy_attributes**(*ds*, *deep=False*)

After constructing a new dataset or pdataset, copy over attributes for sort, labels, footers, etc. Called by Dataset._copy(), PDataset._copy()

**_dataset_compare_check**(*func_name*, *lhs*)

**_ensure_vector**(*vec*)

**_footers_exist**(*labels*)

Return a list of occurring footers from user-specified labels. If labels is None, return list of all footer labels. If none occur, returns None.

**See also:**

*footer_remove*, *footer_get_values*

**_get_columns**(*cols*)

internal routine used to create a list of one or more columns

**_imatrix_y_internal**(*func*, *name=None*, *showfilter=True*)

**Parameters**
**func** (*function or method name of function*) –

**Returns**

- *Y axis calculations*
- *name of the column used*
- *func used*

**_init_columns_as_dict**(*columns*, *base_index=0*, *sort=True*, *unicode=False*)

Most methods of dataset construction will be turned into a dictionary before setting dataset columns. This will return the resulting dictionary for each type or raise an error.

**_init_from_dict**(*dictionary*, *unicode=False*)

**_init_from_itemcontainer**(*columns*)

Store the itemcontainer and set _nrows.

**_init_from_pandas_df**(*df*, *unicode=False*)

Pulls data from pandas dataframes. Uses get attribute, so does not need to import pandas.

**_ipython_key_completions_**()

**_is_float_encodable**(*xtype*)

**_labels_footers_summaries_conform**(*other*)

**_last_row_stats**()

**_makecat**(*cols*)

**_mask_reduce**(*func*, *is_ormask*)

helper function for boolean masks: see mask_or_isnan, et al

**_normalize_column**(*x*, *field_key*)

**_object_as_string**(*name*, *v*)

After failing to convert objects to a numeric type, or when the first item is a string or bytes, try to flip the array to a bytes array, then unicode array.

**_operate_iter_input_cols**(*args*, *fill_value*, *func_or_method_name*, *kwargs*, *lhs*)

Operate iteratively across all columns in the dataset and matching ones in lhs.

In order to operate on summary columns and footer rows, such as those generated by accum2, require that self and lhs conform in the sense of having the same number of labels, footers, and summary columns, with all label columns to the left and all summary columns to the right. The operation is then performed on positionally corresponding elements in the summary columns and footer rows, skipping the label column(s).

**_possibly_convert**(*name*, *v*, *unicode=False*)

Input: any data type that can be added to a dataset Returns: a numpy based array

**_possibly_convert_array**(*v*, *name*, *unicode=False*)

If an array contains objects, it will attempt to flip based on the type of the first item.

By default, flip any numpy arrays to FastArray. (See UseFastArray flag) The constructor will warn the user whenever object arrays appear, and raise an error if conversion was unsuccessful.

### Examples

String objects:

```
>>> ds = rt.Dataset({'col1': np.array(['a','b','c'], dtype=object)})
>>> ds.col1
FastArray([b'a', b'b', b'c'], dtype='|S1')
```

Numeric objects:

```
>>> ds = rt.Dataset({'col1': np.array([1.,2.,3.], dtype=object)})
>>> ds.col1
FastArray([1., 2., 3.])
```

Mixed type objects:

```
>>> ds = rt.Dataset({'col1': np.array([np.nan, 'str', 1], dtype=object)})
ValueError: could not convert string to float: 'str'
TypeError: Cannot handle a numpy object array of type <class 'float'>
```

**Note: depending on the order of mixed types in an object array, they may be converted to strings.**
for performance, only the type of the first item is examined

Mixed type objects starting with string:

```
>>> ds = rt.Dataset({'col1': np.array(['str', np.nan, 1], dtype=object)})
>>> ds.col1
FastArray([b'str', b'nan', b'1'], dtype='|S3')
```

**_post_init**()

Leave this here to chain init that only Dataset has.

**_pre_init**(*sort=False*)

Leave this here to chain init that only Dataset has.

**_prepare_display_data**()

Prepare column headers, arrays, and column footers for display. Arrays will be aranged in order: Labels, sort columns, regular columns, right columns.

**_repr_html_**()

**_sort_lexsort**(*by*, *ascending=True*)

**_sort_values**(*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='mergesort'*, *na_position='last'*, *copy=False*, *sort_rows=None*)

Accepts a single column name or list of column names and adds them to the dataset's column sort list.

The actual sort is performed during display; the dataset itself is not affected unless `inplace=True`. When the dataset is being fed into display, the sort cache gets checked to see if a sorted index index is being held for the keys with the dataset's matching unique ID. If a sorted index is found, it gets passed to display. If no index is found, a lexsort is performed, and the sort is stored in the cache.

**Parameters**

- **by** (*string or list of strings*) – The column name or list of column names by which to sort
- **axis** (*int*) – not used
- **ascending** (*bool*) – not used
- **inplace** (*bool*) – Sort the dataset itself.
- **kind** (*str*) – not used
- **na_position** (*str*) – not used
- **sortrows** (*fancy index array*) – used to pass in your own sort

**Return type**
*Dataset*

**abs**()

Return a dataset where all elements are replaced, as appropriate, by their absolute value.

**Return type**
*Dataset*

### Examples

```
>>> ds = rt.Dataset({'a': np.arange(-3,3), 'b':3*['A', 'B'], 'c':3*[True,␣
↪False]})
>>> ds
#    a   b       c
-   --   -   -----
0   -3   A    True
1   -2   B   False
2   -1   A    True
```

(continues on next page)

(continued from previous page)

```
3    0   B   False
4    1   A    True
5    2   B   False
```

```
>>> ds.abs()
#   a   b       c
-   -   -   -----
0   3   A    True
1   2   B   False
2   1   A    True
3   0   B   False
4   1   A    True
5   2   B   False
```

**accum1**(*cat_rows*, *filter=None*, *showfilter=False*, *ordered=True*, ***kwargs*)

Returns the `GroupBy` object constructed from the Dataset with a 'Totals' column and footer.

**Parameters**

- **cat_rows** (*list of str*) – The list of column names to group by on the row axis. These columns will be made into a `Categorical`.
- **filter** (*ndarray of bools, optional*) – This parameter is unused.
- **showfilter** (*bool, default False*) – This parameter is unused.
- **ordered** (*bool, default True*) – This parameter is unused.
- **sort_gb** (*bool, default True*) – Set to False to change the display order.
- **kwargs** – May be any of the arguments allowed by the Categorical constructor

**Return type**

*GroupBy*

### Examples

```
>>> ds.accum1('symbol').sum(ds.TradeSize)
```

**accum2**(*cat_rows*, *cat_cols*, *filter=None*, *showfilter=False*, *ordered=None*, *lex=None*, *totals=True*)

Returns the Accum2 object constructed from the dataset.

**Parameters**

- **cat_rows** (*list*) – The list of column names to group by on the row axis. This will be made into a categorical.
- **cat_cols** (*list*) – The list of column names to group by on the column axis. This will be made into a categorical.
- **filter** – TODO
- **showfilter** (*bool*) – Used in Accum2 to show filtered out data.
- **ordered** (*bool, optional*) – Defaults to None. Set to True or False to change the display order.
- **lex** (*bool*) – Defaults to None. Set to True for high unique counts. It will override `ordered` when set to True.

- **totals** (*bool, default True*) – Set to False to not show Total column.

**Return type**
*Accum2*

### Examples

```
>>> ds.accum2('symbol', 'exchange').sum(ds.TradeSize)
>>> ds.accum2(['symbol','exchange'], 'date', ordered=True).sum(ds.TradeSize)
```

**add_matrix**(*arr*, *names=None*)

Add a 2-dimensional matrix as columns in a dataset.

**Parameters**

- **arr** (*2-d ndarray*) –
- **names** (*list of str, optional*) – optionally provide column names

**all**(*axis=0*, *as_dataset=True*)

Returns truth value 'all' along axis. Behavior for `axis=None` differs from pandas!

**Parameters**

- **axis** (*int, optional*) –
  - **axis=0 (dflt.) -> over columns (returns Struct (or Dataset) of bools)**
    string synonyms: c, C, col, COL, column, COLUMN
  - **axis=1 -> over rows (returns array of bools)**
    string synonyms: r, R, row, ROW
  - **axis=None -> over rows and columns (returns bool)**
    string synonyms: all, ALL
- **as_dataset** (*bool*) – When `axis=0`, return Dataset instead of Struct. Defaults to False.

**Return type**
*Struct* (or *Dataset*) or list or bool

**any**(*axis=0*, *as_dataset=True*)

Returns truth 'any' value along `axis`. Behavior for `axis=None` differs from pandas!

**Parameters**

- **axis** (*int, optional, default axis=0*) –
  - **axis=0 (dflt.) -> over columns (returns Struct (or Dataset) of bools)**
    string synonyms: c, C, col, COL, column, COLUMN
  - **axis=1 -> over rows (returns array of bools)**
    string synonyms: r, R, row, ROW
  - **axis=None -> over rows and columns (returns bool)**
    string synonyms: all, ALL
- **as_dataset** (*bool*) – When `axis=0`, return Dataset instead of Struct. Defaults to False.

**Return type**
*Struct* (or *Dataset*) or list or bool

**apply**(*funcs*, **args*, *check_op=True*, ***kwargs*)

The apply method returns a Dataset the same size as the current dataset. The transform function is applied column-by-column. The transform function must:

- Return an array that is the same size as the input array.
- Not perform in-place operations on the input array. Arrays should be treated as immutable, and changes to an array may produce unexpected results.

**Parameters**

- **funcs** (*callable or list of callable*) – the function or list of functions applied to each column.
- **check_op** (*bool*) – Defaults to True. Whether or not to check if dataset has its own version, like `sum`.

**Return type**

*Dataset* or *Multiset*

### Examples

```
>>> ds = rt.Dataset({'a': rt.arange(3), 'b': rt.arange(3.0).tile(7), 'c':['Jim
↪','Jason','John']})
>>> ds.apply(lambda x: x+1)
#   a       b   c
-   -   -----   ------
0   1    1.00   Jim1
1   2    8.00   Jason1
2   3   15.00   John1
```

In the example below sum is not possible for a string so it is removed.

```
>>> ds.apply([rt.sum, rt.min, rt.max])
           a                   b                   c
#   Sum   Min   Max    Sum    Min     Max     Min    Max
-   ---   ---   ---   -----   ----   -----   -----   ----
0     3     0     2   21.00   0.00   14.00   Jason   John
```

**apply_cols**(*func_or_method_name*, **args*, *fill_value=None*, *unary=False*, *labels=False*, ***kwargs*)

Apply function (or named method) on each column. If results are all None (*=, +=, for example), None is returned; otherwise a Dataset of the return values will be returned (+, *, `abs`); in this case they are expected to be scalars or vectors of same length.

Constraints on first elem. of args (if unary is False, as for func being an arith op.). lhs can be:

1. a numeric scalar
2. a list of numeric scalars, length nrows (operating on each column)
3. an array of numeric scalars, length nrows (operating on each column)
4. a column vector of numeric scalars, shape (nrows, 1) (reshaped and operating on each column)
5. a Dataset of numeric scalars, shape (nrows, k) (operating on each matching column by name)
6. a Struct of (possibly mixed) (1), (2), (3), (4) (operating on each matching column by name)

**Parameters**

- **func_or_method_name** (*callable or name of method to be called on each column*) –
- **args** (*arguments passed to the func call.*) –
- **fill_value** – The fill value to use for columns with non-computable types.
  - None: return original column in result
  - alt_func (callable): force computation with alt_func
  - scalar: apply as uniform fill value
  - **dict / defaultdict: Mapping of colname->fill_value.**
    Specify per-column `fill_value` behavior. Column names can be mapped to one of the other value Columns whose names are missing from the mapping (or are mapped to `None`) will be dropped. Key-value pairs where the value is `None`, or an absent column name None, or an absent column name if not a `defaultdict` still means None (or absent if not a defaultdict) still means drop column and an alt_func still means force compute via alt_func.
- **unary** (*If False (default) then enforce shape constraints on first positional arg.*) –
- **labels** (*If False (default) then do not apply the function to any label columns.*) –
- **kwargs** (*all other kwargs are passed to func.*) –

**Return type**
*Dataset*, optional

### Examples

```
>>> ds = rt.Dataset({'A': rt.arange(3), 'B': rt.arange(3.0)})
>>> ds.A[2]=ds.A.inv
>>> ds.B[1]=np.nan
>>> ds
#     A      B
-   ---   ----
0     0   0.00
1     1    nan
2   Inv   2.00
```

```
>>> ds.apply_cols(rt.FastArray.fillna, 0)
>>> ds
#   A      B
-   -   ----
0   0   0.00
1   1   0.00
2   0   2.00
```

**apply_rows**(*pyfunc*, **args*, *otypes=None*, *doc=None*, *excluded=None*, *cache=False*, *signature=None*)

Will convert the dataset to a recordarray and then call np.vectorize

Applies a vectorized function which takes a nested sequence of objects or numpy arrays as inputs and returns an single or tuple of numpy array as output. The vectorized function evaluates `pyfunc` over suc-

cessive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

The data type of the output of `vectorized` is determined by calling the function with the first element of the input. This can be avoided by specifying the `otypes` argument.

**Parameters**
**pyfunc** (*callable*) – A python function or method.

### Example

```
>>> ds = rt.Dataset({'a':arange(3), 'b':arange(3.0), 'c':['Jim','Jason','John
↪']}, unicode=True)
>>> ds.apply_rows(lambda x: x[2] + str(x[1]))
rec.array(['Jim0.0', 'Jason1.0', 'John2.0'], dtype=<U8)
```

**apply_rows_numba**(**args*, *otype=None*, *myfunc='myfunc'*)

Prints to screen an example numba signature for the apply function. You can then copy this example to build your own numba function.

Can pass in multiple test arguments.

### Examples

```
>>> ds = rt.Dataset({'a':rt.arange(10), 'b': rt.arange(10)*2, 'c': rt.
↪arange(10)*3})
>>> ds.apply_rows_numba()
Copy the code snippet below and rename myfunc
---------------------------------------------
import numba
@numba.jit
def myfunc(data_out, a, b, c):
    for i in range(len(a)):
        data_out[i]=a[i]   #<-- put your code here

---------------------------------------------
Then call
data_out = rt.empty_like(ds.a)
myfunc(data_out, ds.a, ds.b, ds.c)
```

```
>>> import numba
>>> @numba.jit
... def myfunc(data_out, a, b, c):
...     for i in range(len(a)):
...         data_out[i]=a[i]+b[i]+c[i]
>>> data_out = rt.empty_like(ds.a)
>>> myfunc(data_out, ds.a, ds.b, ds.c)
>>> ds.data_out=data_out
>>> ds
#   a    b    c   data_out
-   -   --   --   --------
0   0    0    0          0
```

(continues on next page)

(continued from previous page)

```
1   1    2    3          6
2   2    4    6         12
```

**argmax**(*axis=0*, *as_dataset=True*, *fill_value=None*)

**argmin**(*axis=0*, *as_dataset=True*, *fill_value=None*)

**as_matrix**(*save_metadata=True*, *column_data={}*)

**as_pandas_df**()

This method is deprecated, please use riptable.Dataset.to_pandas.

Create a pandas DataFrame from this riptable.Dataset. Will attempt to preserve single-key categoricals, otherwise will appear as an index array. Any bytestrings will be converted to unicode.

**Return type**
pandas.DataFrame

**See also:**

`riptable.Dataset.to_pandas`, `riptable.Dataset.from_pandas`

**as_recordarray**(*allow_conversions=False*)

Convert Dataset to one array (record array).

DateTimeNano will be returned as datetime64[ns].

If allow_conversions = True, additional conversions will be performed: Date will be converted to datetime64[D] DateSpan will be converted to timedelta64[D] TimeSpan will be converted (truncated) to timedelta64[ns]

Other wrapped class arrays such as Categorical will lose their type.

**Parameters**
**allow_conversions** (*bool, default False*) – allow column type conversions to appropriate dtypes

### Examples

```
>>> ds = rt.Dataset({'a': rt.arange(3), 'b': rt.arange(3.0), 'c':['Jim','Jason
↪','John']})
>>> ds.as_recordarray()
rec.array([(0, 0., b'Jim'), (1, 1., b'Jason'), (2, 2., b'John')],
          dtype=[('a', '<i4'), ('b', '<f8'), ('c', 'S5')])
```

```
>>> ds.as_recordarray().c
array([b'Jim', b'Jason', b'John'], dtype='|S5')
```

```
>>> ds = rt.Dataset({'a': rt.DateTimeNano("20230301 14:05", from_tz='NYC'), 'b
↪': rt.Date("20210908"), 'c': rt.TimeSpan(-1.23)})
>>> ds.as_recordarray(allow_conversions=True)
rec.array([('2023-03-01T19:05:00.000000000', '2021-09-08', -1)],
        dtype=[('a', '<M8[ns]'), ('b', '<M8[D]'), ('c', '<m8[ns]')])
```

**See also:**

`numpy.core.records.array`

**as_struct**()

Convert a dataset to a struct.

If the dataset is only one row, the struct will be of scalars.

**Return type**
*Struct*

**asrows**(*as_type='Dataset'*, *dtype=None*)

Iterate over rows in any number of of ways, set as_type as appropriate.

When some columns are strings (unicode or byte) and as_type is 'array', best to set dtype=object.

**Parameters**

- **as_type** (*{'Dataset', 'Struct', 'dict', 'OrderedDict', 'namedtuple', 'tuple', 'list', 'array', 'iter'}*) – A string selector which determines return type of iteration, defaults to 'Dataset'.
- **dtype** (*str or np.dtype, optional*) – For `as_type='array'`; if set, force the numpy type of the returned array. Defaults to None.

**Return type**
iterator over selected type.

**astype**(*new_type*, *ignore_non_computable=True*)

Return a new *Dataset* with values converted to the specified data type.

This method ignores string and `Categorical` columns unless forced with `ignore_non_computable = False`. Do not do this unless you know they will convert nicely.

**Parameters**

- **new_type** (*str or Riptable dtype or NumPy dtype*) – The data type to convert values to.
- **ignore_non_computable** (*bool, default True*) – If True (the default), ignore string and `Categorical` values. Set to False to convert them.

**Returns**
A new *Dataset* with values converted to the specified data type.

**Return type**
*Dataset*

**See also:**

`FastArray.astype`

### Examples

```
>>> ds = rt.Dataset({'a': rt.arange(-2.0, 2.0), 'b': 2*['A', 'B'],
...                  'c': 2*[True, False]})
>>> ds
#       a   b       c
-   -----   -   -----
0   -2.00   A    True
1   -1.00   B   False
2    0.00   A    True
3    1.00   B   False
```

By default, string columns are ignored:

```
>>> ds.astype(int)
#    a   b   c
-   --   -   -
0   -2   A   1
1   -1   B   0
2    0   A   1
3    1   B   0
```

When converting numerical values to booleans, only 0 is False. All other numerical values are True.

```
>>> ds.astype(bool)
#       a   b       c
-   -----   -   -----
0    True   A    True
1    True   B   False
2   False   A    True
3    True   B   False
```

You can use `ignore_non_computable = False` to convert a string representation of a numerical value to a numerical type that doesn't truncate the value:

```
>>> ds = rt.Dataset({'str_floats': ['1.1', '2.2', '3.3']})
>>> ds.astype(float, ignore_non_computable = False)
#   str_floats
-   ----------
0         1.10
1         2.20
2         3.30
```

When you force a `Categorical` to be converted, it's replaced with a conversion of its underlying integer `FastArray`:

```
>>> ds = rt.Dataset({'c': rt.Cat(2*['3', '4'])})
>>> ds2 = ds.astype(float, ignore_non_computable = False)
#      c
-   ----
0   1.00
1   2.00
2   1.00
3   2.00
>>> ds2.c
FastArray([1., 2., 1., 2.])
```

**cat**(*cols*, ***kwargs*)

**Parameters**

- **cols** (*str or list of str*) – A single column name or list of names to indicate which columns to build the categorical from or a numpy array to build the categoricals from
- **kwargs** (*any valid keywords in the categorical constructor*) –

**Returns**

A categorical with dataset set to self for groupby operations.

**Return type**
*Categorical*

### Examples

```
>>> np.random.seed(12345)
>>> ds = rt.Dataset({'strcol': np.random.choice(['a','b','c'],4), 'numcol': rt.
↪arange(4)})
>>> ds
#   strcol   numcol
-   ------   ------
0   c             0
1   b             1
2   b             2
3   a             3
```

```
>>> ds.cat('strcol').sum()
*strcol   numcol
-------   ------
a              3
b              3
c              0
```

**cat2keys**(*cat_rows*, *cat_cols*, *filter=None*, *ordered=True*, *sort_gb=False*, *invalid=False*, *fuse=False*)

Creates a `Categorical` with two sets of keys which have all possible unique combinations.

**Parameters**

- **cat_rows** (*str or list of str*) – A single column name or list of names to indicate which columns to build the categorical from or a numpy array to build the categoricals from.
- **cat_cols** (*str or list of str*) – A single column name or list of names to indicate which columns to build the categorical from or a numpy array to build the categoricals from.
- **filter** (*ndarray of bools, optional*) – only valid when invalid is set to True
- **ordered** (*bool, default True*) – only applies when `key1` or `key2` is not a categorical
- **sort_gb** (*bool, default False*) – only applies when `key1` or `key2` is not a categorical
- **invalid** (*bool, default False*) – Specifies whether or not to insert the invalid when creating the n x m unique matrix.
- **fuse** (*bool, default False*) – When True, forces the resulting categorical to have 2 keys, one for rows, and one for columns.

**Returns**
A categorical with at least 2 keys dataset set to self for groupby operations.

**Return type**
*Categorical*

### Examples

```
>>> ds = rt.Dataset({_k: list(range(_i * 2, (_i + 1) * 2)) for _i, _k in␣
↪enumerate(["alpha", "beta", "gamma"])}); ds
#   alpha   beta   gamma
-   -----   ----   -----
0       0      2       4
1       1      3       5
[2 rows x 3 columns] total bytes: 24.0 B
>>> ds.cat2keys(['alpha', 'beta'], 'gamma').sum(rt.arange(len(ds)))
*alpha   *beta   *gamma   col_0
------   -----   ------   -----
     0       2        4       0
     1       3        4       0
     0       2        5       0
     1       3        5       1
```

[4 rows x 4 columns] total bytes: 80.0 B

**See also:**

`rt_numpy.cat2keys`, `rt_dataset.accum2`

**col_replace_all**(*newdict*, *check_exists=True*)

Replace the data for each item in the item dict. Original attributes will be retained. Useful for internal routines that need to swap out all columns quickly.

**Parameters**

- **newdict** (*dictionary of item names -> new item data (can also be a Dataset)*) –
- **check_exists** (*bool*) – if True, all newdict keys and old item keys will be compared to ensure a match

**computable**()

returns a dict of computable columns. does not include groupby keys

**classmethod concat_columns**(*dsets*, *do_copy*, *on_duplicate='raise'*, *on_mismatch='warn'*)

Stack columns from multiple *Dataset* objects horizontally (column-wise).

All *Dataset* columns must be the same length.

**Parameters**

- **cls** (*class*) – The class (*Dataset*).
- **dsets** (iterable of *Dataset* objects) – The *Dataset* objects to be concatenated.
- **do_copy** (*bool*) – When True, makes deep copies of the arrays. When False, shallow copies are made.
- **on_duplicate** (*{'raise', 'first', 'last'}, default 'raise'*) – Governs behavior in case of duplicate column names.
  - 'raise' (default): Raises a KeyError. Overrides all `on_mismatch` values.
  - 'first': Keeps the column data from the first duplicate column. Overridden by `on_mismatch = 'raise'`.

- 'last': Keeps the column data from the last duplicate column. Overridden by `on_mismatch = 'raise'`.

- **on_mismatch** (*{'warn', 'raise', 'ignore'}, default 'warn'*) – Governs how to address duplicate column names.
  - 'warn' (default): Issues a warning. Overridden by `on_duplicate = 'raise'`.
  - 'raise': Raises a RuntimeError. Overrides `on_duplicate = 'first'` and `on_duplicate = 'last'`. Overridden by `on_duplicate = 'raise'`.
  - 'ignore': No error or warning. Overridden by `on_duplicate = 'raise'`.

**Returns**
A new *Dataset* created from the concatenated columns of the input *Dataset* objects.

**Return type**
*Dataset*

**See also:**

*Dataset.concat_rows*
Vertically stack columns from multiple *Dataset* objects.

### Examples

Basic concatenation:

```
>>> ds1 = rt.Dataset({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
>>> ds2 = rt.Dataset({'C': ['C0', 'C1', 'C2'], 'D': ['D0', 'D1', 'D2']})
>>> rt.Dataset.concat_columns([ds1, ds2], do_copy = True)
#   A    B    C    D
-   --   --   --   --
0   A0   B0   C0   D0
1   A1   B1   C1   D1
2   A2   B2   C2   D2
```

With a duplicated column 'B' and `on_duplicate = 'last'`:

```
>>> ds1 = rt.Dataset({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
>>> ds2 = rt.Dataset({'C': ['C0', 'C1', 'C2'], 'B': ['B3', 'B4', 'B5']})
>>> ds3 = rt.Dataset({'D': ['D0', 'D1', 'D2'], 'B': ['B6', 'B7', 'B8']})
>>> rt.Dataset.concat_columns([ds1, ds2, ds3], do_copy = True,
...                           on_duplicate = 'last', on_mismatch = 'ignore')
#   A    B    C    D
-   --   --   --   --
0   A0   B6   C0   D0
1   A1   B7   C1   D1
2   A2   B8   C2   D2
```

With `on_mismatch = 'raise'`:

```
>>> rt.Dataset.concat_columns([ds1, ds2, ds3], do_copy = True,
...                           on_duplicate = 'last', on_mismatch = 'raise')
Traceback (most recent call last):
RuntimeError: concat_columns() duplicate column mismatch: {'B'}
```

**classmethod concat_rows**(*ds_list*, *destroy=False*)

Stack columns from multiple *Dataset* objects vertically (row-wise).

Columns must have the same name to be concatenated. If a *Dataset* is missing a column that appears in others, the gap is filled with the default invalid value for the existing column's data type (for example, `NaN` for floats).

`Categorical` objects are merged and stacked.

**Parameters**

- **cls** (*class*) – The class (*Dataset*).
- **ds_list** (iterable of *Dataset* objects) – The *Dataset* objects to be concatenated.
- **destroy** (*bool, default False*) – Set to True to destroy the input *Dataset* objects to save memory.

**Returns**

A new *Dataset* created from the concatenated rows of the input *Dataset* objects.

**Return type**

*Dataset*

**Warning:**

- Vertically stacking columns that have a general data type mismatch (for example, a string column and a float column) is not recommended. Currently, a run-time warning is issued; in future versions of Riptable, general dtype mismatches will not be allowed.
- *Dataset* columns with two dimensions are technically supported by Riptable, but not recommended. Concatenating *Dataset* objects with two-dimensional columns is possible, but not recommended because it may produce unexpected results.

**See also:**

*Dataset.concat_columns*

Horizontally stack columns from multiple *Dataset* objects.

### Examples

```
>>> ds1 = rt.Dataset({'A': ['A0', 'A1', 'A2'], 'B': ['B0', 'B1', 'B2']})
>>> ds2 = rt.Dataset({'A': ['A3', 'A4', 'A5'], 'B': ['B3', 'B4', 'B5']})
>>> ds1
#   A    B
-   --   --
0   A0   B0
1   A1   B1
2   A2   B2
>>> ds2
#   A    B
-   --   --
0   A3   B3
1   A4   B4
2   A5   B5
```

Basic concatenation:

```
>>> rt.Dataset.concat_rows([ds1, ds2])
#   A    B
-   --   --
0   A0   B0
1   A1   B1
2   A2   B2
3   A3   B3
4   A4   B4
5   A5   B5
```

When a column exists in one *Dataset* but is missing in another, the gap is filled with the default invalid value for the existing column.

```
>>> ds1 = rt.Dataset({'A': rt.arange(3)})
>>> ds2 = rt.Dataset({'A': rt.arange(3, 6), 'B': rt.arange(3, 6)})
>>> rt.Dataset.concat_rows([ds1, ds2])
#   A     B
-   -   ---
0   0   Inv
1   1   Inv
2   2   Inv
3   3     3
4   4     4
5   5     5
```

Concatenate two *Dataset* objects with `Categorical` columns:

```
>>> ds1 = rt.Dataset({'cat_col': rt.Categorical(['a','a','b','c','a']),
...                   'num_col': rt.arange(5)})
>>> ds2 = rt.Dataset({'cat_col': rt.Categorical(['b','b','a','c','d']),
...                   'num_col': rt.arange(5)})
>>> ds_concat = rt.Dataset.concat_rows([ds1, ds2])
>>> ds_concat
#   cat_col   num_col
-   -------   -------
0   a               0
1   a               1
2   b               2
3   c               3
4   a               4
5   b               0
6   b               1
7   a               2
8   c               3
9   d               4
```

The `Categorical` objects are merged:

```
>>> ds_concat.cat_col
Categorical([a, a, b, c, a, b, b, a, c, d]) Length: 10
    FastArray([1, 1, 2, 3, 1, 2, 2, 1, 3, 4], dtype=int8) Base Index: 1
    FastArray([b'a', b'b', b'c', b'd'], dtype='|S1') Unique count: 4
```

**copy**(*deep=True*)

Make a copy of the *Dataset*.

**Parameters**

**deep** (*bool, default True*) – Whether the underlying data should be copied. When `deep = True` (the default), changes to the copy do not modify the underlying data (and vice versa). When `deep = False`, the copy is shallow: Only references to the underlying data are copied, and any changes to the copy also modify the underlying data (and vice versa).

**Return type**

*Dataset*

### Examples

Create a *Dataset*:

```
>>> ds = rt.Dataset({'a': rt.arange(-3,3), 'b':3*['A', 'B'], 'c':3*[True,␣
↪False]})
>>> ds
#    a   b       c
-   --   -   -----
0   -3   A    True
1   -2   B   False
2   -1   A    True
3    0   B   False
4    1   A    True
5    2   B   False
```

When `deep = True` (the default), changes to the original `ds` do not modify the copy, `ds1`.

```
>>> ds1 = ds.copy()
>>> ds.a = ds.a + 1
>>> ds1
#    a   b       c
-   --   -   -----
0   -3   A    True
1   -2   B   False
2   -1   A    True
3    0   B   False
4    1   A    True
5    2   B   False
```

**count**(*axis=0*, *as_dataset=True*, *fill_value=len*)

See documentation of `reduce()`

**describe**(*q=None*, *fill_value=None*)

Generate descriptive statistics for a Dataset's numerical columns.

Descriptive statistics include those that summarize the central tendency, dispersion, and shape of a Dataset's distribution, excluding `NaN` values.

Columns remain stable, with a 'Stats' column added to provide labels for each statistical measure. Non-numerical columns are ignored. If the Dataset has no numerical columns, only the column of labels is returned.

**Parameters**

- **q** (*list of float, default [0.10, 0.25, 0.50, 0.75, 0.90]*) – The quantiles to calculate. All should fall between 0 and 1.
- **fill_value** (*int, float, or str, default None*) – Placeholder value for non-computable columns. Can be a single value, or a list or FastArray of values that is the same length as the Dataset.

**Returns**

A Dataset containing a label column and the calculated values for each numerical column, or filled values (if provided) for non-numerical columns.

**Return type**

*Dataset*

**Warning:** This routine can be expensive if the Dataset is large.

**See also:**

**FastArray.describe**
Generates descriptive statistics for a FastArray.

### Notes

Descriptive statistics provided:

| Stat | Description |
|---|---|
| Count | Total number of items |
| Valid | Total number of valid values |
| Nans | Total number of NaN values |
| Mean | Mean |
| Std | Standard deviation |
| Min | Minimum value |
| P10 | 10th percentile |
| P25 | 25th percentile |
| P50 | 50th percentile |
| P75 | 75th percentile |
| P90 | 90th percentile |
| Max | Maximum value |
| MeanM | Mean without top or bottom 10% |

**dhead**(*n=0*)

Displays the head of the Dataset. Compare with head() which returns a new Dataset.

**drop_duplicates**(*subset=None*, *keep='first'*, *inplace=False*)

Return Dataset with duplicate rows removed, optionally only considering certain columns

**Parameters**

- **subset** (*column label or sequence of labels, optional*) – Only consider certain columns for identifying duplicates, by default use all of the columns
- **keep** (*{'first', 'last', False}, default 'first'*) –

- `first` : Drop duplicates except for the first occurrence.
- `last` : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

- **inplace** (*boolean, default False*) – Whether to drop duplicates in place or to return a copy

**Returns**
**deduplicated**

**Return type**
*Dataset*

### Notes

If *keep* is 'last', the rows in the result will match pandas, but the order will be based on first occurrence of the unique key.

### Examples

```
>>> np.random.seed(12345)
>>> ds = rt.Dataset({
...     'strcol' : np.random.choice(['a','b','c','d'], 15),
...     'intcol' : np.random.randint(0, 3, 15),
...     'rand' : np.random.rand(15)
... })
>>> ds
 #   strcol   intcol   rand
--   ------   ------   ----
0   c             2   0.05
1   b             1   0.81
2   b             2   0.93
3   b             0   0.36
4   a             2   0.69
5   b             1   0.13
6   c             1   0.83
7   c             2   0.32
8   b             1   0.74
9   c             2   0.60
10   b            2   0.36
11   b            1   0.79
12   c            0   0.70
13   b            1   0.82
14   d            1   0.90

[15 rows x 3 columns] total bytes: 195.0 B
```

Keep only the row of the first occurrence:

```
>>> ds.drop_duplicates(['strcol','intcol'])
#   strcol   intcol   rand
-   ------   ------   ----
```

(continues on next page)

(continued from previous page)

```
0   c             2   0.05
1   b             1   0.81
2   b             2   0.93
3   b             0   0.36
4   a             2   0.69
5   c             1   0.83
6   c             0   0.70
7   d             1   0.90

[8 rows x 3 columns] total bytes: 104.0 B
```

Keep only the row of the last occurrence:

```
>>> ds.drop_duplicates(['strcol','intcol'], keep='last')
#   strcol   intcol   rand
-   ------   ------   ----
0   c             2   0.60
1   b             1   0.82
2   b             2   0.36
3   b             0   0.36
4   a             2   0.69
5   c             1   0.83
6   c             0   0.70
7   d             1   0.90

[8 rows x 3 columns] total bytes: 104.0 B
```

Keep only the rows which only occur once:

```
>>> ds.drop_duplicates(['strcol','intcol'], keep=False)
#   strcol   intcol   rand
-   ------   ------   ----
0   b             0   0.36
1   a             2   0.69
2   c             1   0.83
3   c             0   0.70
4   d             1   0.90

[5 rows x 3 columns] total bytes: 65.0 B
```

**dtail**(*n=0*)

Displays the tail of the Dataset. Compare with `tail()` which returns a new Dataset.

**duplicated**(*subset=None*, *keep='first'*)

Return a boolean FastArray set to True where duplicate rows exist, optionally only considering certain columns

**Parameters**

- **subset** (*str or list of str, optional*) – A column label or list of column labels to inspect for duplicate values. When `None`, all columns will be examined.
- **keep** (*{'first', 'last', False}, default 'first'*) –
  - `first` : keep duplicates except for the first occurrence.

- `last` : keep duplicates except for the last occurrence.
- False : set to True for all duplicates.

### Examples

```
>>> ds=rt.Dataset({'somenans': [0., 1., 2., rt.nan, 0., 5.], 's2': [0., 1., rt.
↪nan, rt.nan, 0., 5.]})
>>> ds
#   somenans     s2
-   --------   ----
0       0.00   0.00
1       1.00   1.00
2       2.00    nan
3        nan    nan
4       0.00   0.00
5       5.00   5.00
```

```
>>> ds.duplicated()
FastArray([False, False, False, False,  True, False])
```

### Notes

Consider using `rt.Grouping(subset).ifirstkey` as a fancy index to pull in unique rows.

**equals**(*other*, *axis=None*, *labels=False*, *exact=False*)

Test whether two Datasets contain the same elements in each column. NaNs in the same location are considered equal.

**Parameters**

- **other** (*Dataset or dict*) – another dataset or dict to compare to
- **axis** (*int, optional*) –
  - None: returns a True or False for all columns
  - 0 : to return a boolean result per column
  - 1 : to return an array of booleans per column
- **labels** (*bool*) – Indicates whether or not to include column labels in the comparison.
- **exact** (*bool*) – When True, the exact order of all columns (including labels) must match

**Returns**

Based on the value of `axis`, a boolean or Dataset containing the equality comparison results.

**Return type**

bool or *Dataset*

**See also:**

Dataset.crc, ==, >=, <=, >, <

### Examples

```
>>> ds = rt.Dataset({'somenans': [0., 1., 2., nan, 4., 5.]})
>>> ds2 = rt.Dataset({'somenans': [0., 1., nan, 3., 4., 5.]})
>>> ds.equals(ds)
True
```

```
>>> ds.equals(ds2, axis=0)
#   somenans
-   --------
0      False
```

```
>>> ds.equals(ds, axis=0)
#   somenans
-   --------
0       True
```

```
>>> ds.equals(ds2, axis=1)
#   somenans
-   --------
0       True
1       True
2      False
3      False
4       True
5       True
```

```
>>> ds.equals(ds2, axis=0, exact=True)
FastArray([False])
```

```
>>> ds.equals(ds, axis=0, exact=True)
FastArray([True])
```

```
>>> ds.equals(ds2, axis=1, exact=True)
FastArray([[ True],
           [ True],
           [False],
           [False],
           [ True],
           [ True]])
```

**fillna**(*value=None*, *method=None*, *inplace=False*, *limit=None*)

Replace NaN and invalid values with a specified value or nearby data.

Optionally, you can modify the original *Dataset* if it's not locked.

**Parameters**

- **value** (*scalar, default None*) – A value to replace all NaN and invalid values. Required if `method = None`. Note that this **cannot** be a `dict` yet. If a `method` is also provided, the `value` will be used to replace NaN and invalid values only where there's not a valid value to propagate forward or backward.

- **method** (*{None, 'backfill', 'bfill', 'pad', 'ffill'}, default None*) – Method to use to propagate valid values within each column.
  - backfill/bfill: Propagates the next encountered valid value backward. Calls `FastArray.fill_backward()`.
  - pad/ffill: Propagates the last encountered valid value forward. Calls `FastArray.fill_forward()`.
  - None: A replacement value is required if `method = None`. Calls `FastArray.replacena()`.

  If there's not a valid value to propagate forward or backward, the NaN or invalid value is not replaced unless you also specify a `value`.
- **inplace** (*bool, default False*) – If False, return a copy of the *Dataset*. If True, modify original column arrays. This will modify any other views on this object. This fails if the *Dataset* is locked.
- **limit** (*int, default None*) – If `method` is specified, this is the maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled.

**Returns**
The *Dataset* will be the same size and have the same dtypes as the original input.

**Return type**
*Dataset*

**See also:**

*riptable.rt_fastarraynumba.fill_forward*
Replace NaN and invalid values with the last valid value.

*riptable.rt_fastarraynumba.fill_backward*
Replace NaN and invalid values with the next valid value.

**riptable.fill_forward**
Replace NaN and invalid values with the last valid value.

**riptable.fill_backward**
Replace NaN and invalid values with the next valid value.

**FastArray.replacena**
Replace NaN and invalid values with a specified value.

**FastArray.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**Categorical.fill_forward**
Replace NaN and invalid values with the last valid group value.

**Categorical.fill_backward**
Replace NaN and invalid values with the next valid group value.

**GroupBy.fill_forward**
Replace NaN and invalid values with the last valid group value.

**GroupBy.fill_backward**
Replace NaN and invalid values with the next valid group value.

### Examples

Replace all NaN and invalid values with 0s.

```
>>> ds = rt.Dataset({'A': rt.arange(3), 'B': rt.arange(3.0)})
>>> ds.A[2]=ds.A.inv  # Replace with the invalid value for the column's dtype.
>>> ds.B[1]=rt.nan
>>> ds
#     A      B
-   ---   ----
0     0   0.00
1     1    nan
2   Inv   2.00
>>> ds.fillna(0)
#   A      B
-   -   ----
0   0   0.00
1   1   0.00
2   0   2.00
```

The following examples will use this *Dataset*:

```
>>> ds = rt.Dataset({'A':[rt.nan, 2, rt.nan, 0], 'B': [3, 4, 2, 1],
...                  'C':[rt.nan, rt.nan, rt.nan, 5], 'D':[rt.nan, 3, rt.nan,␣
↪4]})
>>> ds.B[2] = ds.B.inv  # Replace with the invalid value for the column's dtype.
>>> ds
#      A     B      C      D
-   ----   ---   ----   ----
0    nan     3    nan    nan
1   2.00     4    nan   3.00
2    nan   Inv    nan    nan
3   0.00     1   5.00   4.00
```

Propagate the last encountered valid value forward. Note that where there's no valid value to propagate, the NaN or invalid value isn't replaced.

```
>>> ds.fillna(method = 'ffill')
#      A   B      C      D
-   ----   -   ----   ----
0    nan   3    nan    nan
1   2.00   4    nan   3.00
2   2.00   4    nan   3.00
3   0.00   1   5.00   4.00
```

You can use the `value` parameter to specify a value to use where there's no valid value to propagate.

```
>>> ds.fillna(value = 10, method = 'ffill')
#       A   B       C       D
-   -----   -   -----   -----
0   10.00   3   10.00   10.00
1    2.00   4   10.00    3.00
2    2.00   4   10.00    3.00
3    0.00   1    5.00    4.00
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values.

```
>>> ds.fillna(method = 'bfill', limit = 1)
#      A   B      C      D
-   ----   -   ----   ----
0   2.00   3    nan   3.00
1   2.00   4    nan   3.00
2   0.00   1   5.00   4.00
3   0.00   1   5.00   4.00
```

**filter**(*rowfilter*, *inplace=False*)

Return a copy of the *Dataset* containing only the rows that meet the specified condition.

**Parameters**

- **rowfilter** (*array: fancy index or boolean mask*) – A fancy index specifies both the desired rows and their order in the returned *Dataset*. When a boolean mask is passed, only rows that meet the specified condition are in the returned *Dataset*.
- **inplace** (*bool, default False*) – When set to True, reduces memory overhead by modifying the original *Dataset* instead of making a copy.

**Returns**

A *Dataset* containing only the rows that meet the filter condition.

**Return type**

*Dataset*

### Notes

Making a copy of a large *Dataset* is expensive. Use `inplace=True` when possible.

If you want to perform an operation on a filtered column, get the column and then perform the operation using the `filter` keyword argument. For example, `ds.ColumnName.sum(filter=boolean_mask)`.

Alternatively, you can filter the column and then perform the operation. For example, `ds.ColumnName[boolean_mask].sum()`.

### Examples

Create a *Dataset*:

```
>>> ds = rt.Dataset({"a": rt.arange(-3, 3), "b": 3 * ['A', 'B'], "c": 3 *␣
↪[True, False]})
>>> ds
#    a   b       c
-   --   -   -----
0   -3   A    True
1   -2   B   False
2   -1   A    True
3    0   B   False
4    1   A    True
5    2   B   False

[6 rows x 3 columns] total bytes: 36.0 B
```

Filter using a fancy index:

```
>>> ds.filter([5, 0, 1])
#    a   b       c
-   --   -   -----
0    2   B   False
1   -3   A    True
2   -2   B   False

[3 rows x 3 columns] total bytes: 18.0 B
```

Filter using a condition that creates a boolean mask array:

```
>>> ds.filter(ds.b == "A")
#    a   b      c
-   --   -   ----
0   -3   A   True
1   -1   A   True
2    1   A   True

[3 rows x 3 columns] total bytes: 18.0 B
```

Filter a large *Dataset* using the least memory possible with `inplace=True`.

```
>>> ds = rt.Dataset({"a": rt.arange(10_000_000), "b": rt.arange(10_000_000.0)})
>>> f = rt.logical(rt.arange(10_000_000) % 2)
>>> ds.filter(f, inplace=True)
      #         a           b
-------   -------   ---------
      0         1        1.00
      1         3        3.00
      2         5        5.00
    ...       ...         ...
4999997   9999995   1.000e+07
4999998   9999997   1.000e+07
4999999   9999999   1.000e+07

[5000000 rows x 2 columns] total bytes: 57.2 MB
```

**footer_get_dict**(*labels=None*, *columns=None*)

Dictionary of footer rows, the latter in dictionary form.

**Parameters**

- **labels** (*list, optional*) – Footer rows to return values for. If not provided, all footer rows will be returned.
- **columns** (*list of str, optional*) – Columns to return footer values for. If not provided, all column footers will be returned.

### Examples

```
>>> ds = rt.Dataset({'colA': rt.arange(5), 'colB': rt.arange(5), 'colC': rt.
↪arange(5)})
>>> ds.footer_set_values('row1', {'colA':1, 'colC':2})
>>> ds.footer_get_dict()
{'row1': {'colA': 1, 'colC': 2}}
```

```
>>> ds.footer_get_dict(columns=['colC','colA'])
{'row1': [2, 1]}
```

```
>>> ds.footer_remove()
>>> ds.footer_get_dict()
{}
```

**Returns**

**footers** – Keys are footer row names. Values are dictionaries of column name and value pairs.

**Return type**

dictionary

**footer_get_values**(*labels=None*, *columns=None*, *fill_value=None*)

Dictionary of footer rows. Missing footer values will be returned as None.

**Parameters**

- **labels** (*list, optional*) – Footer rows to return values for. If not provided, all footer rows will be returned.
- **columns** (*list, optional*) – Columns to return footer values for. If not provided, all column footers will be returned.
- **fill_value** (*optional, default None*) – Value to use when no footer is found.

### Examples

```
>>> ds = rt.Dataset({'colA': rt.arange(5), 'colB': rt.arange(5), 'colC': rt.
↪arange(5)})
>>> ds.footer_set_values('row1', {'colA':1, 'colC':2})
>>> ds.footer_get_values()
{'row1': [1, None, 2]}
```

```
>>> ds.footer_get_values(columns=['colC','colA'])
{'row1': [2, 1]}
```

```
>>> ds.footer_remove()
>>> ds.footer_get_values()
{}
```

**Returns**

**footers** – Keys are footer row names. Values are lists of footer values or None, if missing.

**Return type**
dictionary

**footer_remove**(*labels=None*, *columns=None*)

Remove all or specific footers from all or specific columns.

**Parameters**

- **labels** (*string or list of strings, default None*) – If provided, remove only footers under these names.
- **columns** (*string or list of strings, default None*) – If provided, only remove (possibly specified) footers from these columns.

### Examples

```
>>> ds = rt.Dataset({'colA': rt.arange(3),'colB': rt.arange(3)*2})
>>> ds.footer_set_values('sum', {'colA':3, 'colB':6}
>>> ds.footer_set_values('mean', {'colA':1.0, 'colB':2.0})
>>> ds
   #   colA   colB
----   ----   ----
   0      0      0
   1      1      2
   2      2      4
----   ----   ----
 sum      3      6
mean   1.00   2.00
```

Remove single footer from single column

```
>>> ds.footer_remove('sum','colA')
>>> ds
   #   colA   colB
----   ----   ----
   0      0      0
   1      1      2
   2      2      4
----   ----   ----
 sum             6
mean   1.00   2.00
```

Remove single footer from all columns

```
>>> ds.footer_remove('mean')
>>> ds
  #   colA   colB
---   ----   ----
  0      0      0
  1      1      2
  2      2      4
---   ----   ----
sum             6
```

Remove all footers from all columns

```
>>> ds.footer_remove()
>>> ds
#   colA   colB
-   ----   ----
0      0      0
1      1      2
2      2      4
```

### Notes

Calling this method with no keywords will clear all footers from all columns.

**See also:**

*Dataset.footer_set_values*

**footer_set_values**(*label*, *footerdict*)

Assign footer values to specific columns.

**Parameters**

- **label** (*string*) – Name of existing or new footer row. This string will appear as a label on the left, below the right-most label key or row numbers.
- **footerdict** (*dictionary*) – Keys are valid column names (otherwise raises ValueError). Values are scalars. They will appear as a string with their default type formatting.

**Return type**

None

### Examples

```
>>> ds = rt.Dataset({'colA': rt.arange(3), 'colB': rt.arange(3)*2})
>>> ds.footer_set_values('sum', {'colA':3, 'colB':6})
>>> ds
  #   colA   colB
---   ----   ----
  0      0      0
  1      1      2
  2      2      4
---   ----   ----
sum      3      6
```

```
>>> ds.colC = rt.ones(3)
>>> ds.footer_set_values('mean', {'colC': 1.0})
>>> ds
   #   colA   colB   colC
----   ----   ----   ----
   0      0      0   1.00
   1      1      2   1.00
   2      2      4   1.00
----   ----   ----   ----
```

(continues on next page)

(continued from previous page)

```
 sum      3      6
mean                1.00
```

### Notes

- Not all footers need to be set. Missing footers will appear as blank in final display.
- Footers will appear in dataset slices as they do in the original dataset.
- If the footer is a column total, it may need to be recalculated.
- This routine can also be used to replace existing footers.

**See also:**

*Dataset.footer_remove*

**static from_arrow**(*tbl*, *zero_copy_only=True*, *writable=False*, *auto_widen=False*, *fill_value=None*)

Convert a pyarrow `Table` to a riptable *Dataset*.

**Parameters**

- **tbl** (*pyarrow.Table*) –
- **zero_copy_only** (*bool, default True*) – If True, an exception will be raised if the conversion to a `FastArray` would require copying the underlying data (e.g. in presence of nulls, or for non-primitive types).
- **writable** (*bool, default False*) – For a `FastArray` created with zero copy (view on the Arrow data), the resulting array is not writable (Arrow data is immutable). By setting this to True, a copy of the array is made to ensure it is writable.
- **auto_widen** (*bool, optional, default to False*) – When False (the default), if an arrow array contains a value which would be considered the 'invalid'/NA value for the equivalent dtype in a `FastArray`, raise an exception. When True, the converted array
- **fill_value** (*Mapping[str, int or float or str or bytes or bool], optional, defaults to None*) – Optional mapping providing non-default fill values to be used. May specify as many or as few columns as the caller likes. When None (or for any columns which don't have a fill value specified in the mapping) the riptable invalid value for the column (given it's dtype) will be used.

**Return type**

*Dataset*

### Notes

This function does not currently support pyarrow's nested Tables. A future version of riptable may support nested Datasets in the same way (where a Dataset contains a mixture of arrays/columns or nested Datasets having the same number of rows), which would make it trivial to support that conversion.

**classmethod from_jagged_dict**(*dct*, *fill_value=None*, *stacked=False*)

Creates a Dataset from a dict where each key represents a column name base and each value an iterable of 'rows'. Each row in the values iterable is, in turn, a scalar or an iterable of scalar values having variable length.

**Parameters**

- **dct** – a dictionary of columns that are to be formed into rows
- **fill_value** – value to fill missing values with, or if None, with the NODATA value of the type of the first value from the first row with values for the given key
- **stacked** (*bool*) – Whether to create stacked rows in the output when an input row in one of the input values objects contains an iterable.

**Returns**
A new Dataset.

**Return type**
*Dataset*

### Notes

For a given key, if each row in the corresponding values iterable is a scalar, a single column will be created with a column name equal to the key name.

If for a given key, a row in the corresponding values iterable is an iterable, the behavior is determined by the stacked parameter.

If stacked is False (the default), as many columns will be created as necessary to contain the maximum number of scalar values in the value rows. The column names will be the key name plus a zero based index. Any empty elements in a row will be filled with the specified fill_value, or if None, with a NODATA value of the type corresponding to the first value from the first row with values for the given key.

If stacked is True, one column will be created for each input key, and for each row of input values, a row will be created in the output for every combination of value elements from each column in the input row.

### Examples

```
>>> d = {'name': ['bob', 'mary', 'sue', 'john'],
...      'letters': [['A', 'B', 'C'], ['D'], ['E', 'F', 'G'], 'H']}
>>> ds1 = rt.Dataset.from_jagged_dict(d)
>>> nd = rt.INVALID_DICT[np.dtype(str).num]
>>> ds2 = rt.Dataset({'name': ['bob', 'mary', 'sue', 'john'],
...      'letters0': ['A','D','E','H'], 'letters1': ['B',nd,'F',nd],
...      'letters2': ['C',nd,'G',nd]})
>>> (ds1 == ds2).all(axis=None)
True
```

```
>>> ds3 = rt.Dataset.from_jagged_dict(d, stacked=True)
>>> ds4 = rt.Dataset({'name': ['bob', 'bob', 'bob', 'mary', 'sue', 'sue', 'sue
↪', 'john'],
...      'letters': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']})
>>> (ds3 == ds4).all(axis=None)
True
```

**classmethod from_jagged_rows**(*rows*, *column_name_base='C'*, *fill_value=None*)

Returns a Dataset from rows of different lengths. All columns in Dataset will be bytes or unicode. Bytes will be used if possible.

**Parameters**

- **rows** – list of numpy arrays, lists, scalars, or anything that can be turned into a numpy array.
- **column_name_base** (*str*) – columns will by default be numbered. this is an optional prefix which defaults to 'C'.
- **fill_value** (*str, optional*) – custom fill value for missing cells. will default to the invalid string

### Notes

*performance warning*: this routine iterates over rows in non-contiguous memory to fill in final column values. TODO: maybe build all final columns in the same array and fill in a snake-like manner like Accum2.

**classmethod from_pandas**(*df*, *tz='UTC'*, *preserve_index=None*)

Creates a riptable Dataset from a pandas DataFrame. Pandas categoricals and datetime arrays are converted to their riptable counterparts. Any timezone-unaware datetime arrays (or those using a timezone not recognized by riptable) are localized to the timezone specified by the tz parameter.

**Recognized pandas timezones:**
UTC, GMT, US/Eastern, and Europe/Dublin

**Parameters**

- **df** (*pandas.DataFrame*) – The pandas DataFrame to be converted
- **tz** (*string*) – A riptable-supported timezone ('UTC', 'NYC', 'DUBLIN', 'GMT') as fallback timezone.

**Return type**
riptable.Dataset

**See also:**

`riptable.Dataset.to_pandas`

**classmethod from_rows**(*rows_iter*, *column_names*)

Create a Dataset from an iterable of 'rows', each to be an iterable of scalar values, all having the same length, that being the length of column_names.

**Parameters**

- **rows_iter** (*iterable of iterable of scalars*) –
- **column_names** (*list of str*) – list of column names matching length of each row

**Returns**
A new Dataset

**Return type**
*Dataset*

### Examples

```
>>> ds1 = rt.Dataset.from_rows([[1, 11], [2, 12]], ['a', 'b'])
>>> ds2 = rt.Dataset({'a': [1, 2], 'b': [11, 12]})
>>> (ds1 == ds2).all(axis=None)
True
```

**classmethod from_tagged_rows**(*rows_iter*)

Create a Dataset from an iterable of 'rows', each to be a dict, Struct, or named_tuple of scalar values.

**Parameters**
**rows_iter** (*iterable of dict, Struct or named_tuple of scalars*) –

**Returns**
A new Dataset.

**Return type**
*Dataset*

### Notes

Still TODO: Handle case w/ not all rows having same keys. This is waiting on SafeArray and there are stop-gaps to use until that point.

### Examples

```
>>> ds1 = rt.Dataset.from_tagged_rows([{'a': 1, 'b': 11}, {'a': 2, 'b': 12}])
>>> ds2 = rt.Dataset({'a': [1, 2], 'b': [11, 12]})
>>> (ds1 == ds2).all(axis=None)
True
```

**gb**(*by*, ***kwargs*)

Equivalent to `groupby()`

**gbrows**(*strings=False*, *dtype=None*, ***kwargs*)

Create a GroupBy object based on "computable" rows or string rows.

**Parameters**

- **strings** (*bool*) – Defaults to False. Set to True to process strings.
- **dtype** (*str or numpy.dtype, optional*) – Defaults to None. When set, all columns will be cast to this dtype.
- **kwargs** – Any other kwargs will be passed to `groupby()`.

**Return type**
*GroupBy*

### Examples

```
>>> ds = rt.Dataset({'a': rt.arange(3), 'b': rt.arange(3.0), 'c':['Jim','Jason
↪','John']})
>>> ds.gbrows()
GroupBy Keys ['RowNum'] @ [2 x 3]
ikey:True  iFirstKey:False  iNextKey:False  nCountGroup:False _filter:False  _
↪return_all:False

*RowNum   Count
-------   -----
    0       2
    1       2
    2       2
```

```
>>> ds.gbrows().sum()
*RowNum    Row
-------   ----
    0   0.00
    1   2.00
    2   4.00

[3 rows x 2 columns] total bytes: 36.0 B
```

Example usage of the string-processing mode of `gbrows()`:

```
>>> ds.gbrows(strings=True)
GroupBy Keys ['RowNum'] @ [2 x 3]
ikey:True  iFirstKey:False  iNextKey:False  nCountGroup:False _filter:False  _
↪return_all:False

*RowNum   Count
-------   -----
    0       1
    1       1
    2       1
```

**gbu**(*by*, ***kwargs*)

Equivalent to `groupby()` with sort=False

**get_nrows**()

The number of elements in each column of the *Dataset*.

**Returns**

The number of elements in each column of the *Dataset*.

**Return type**

int

**See also:**

*Dataset.size*

The number of elements in the *Dataset* (nrows x ncols).

**Struct.get_ncols**

The number of items in a `Struct` or the number of elements in each row of a *Dataset*.

**Struct.shape**
A tuple containing the number of rows and columns in a Struct or *Dataset*.

### Examples

```
>>> ds = rt.Dataset({'A': [1.0, 2.0], 'B': [3, 4], 'C': ['c', 'c']})
>>> ds.get_nrows()
2
```

**get_row_sort_info**()

**get_sorted_col_data**(*col_name*)

Private method. :param col_name: :return: numpy array

**groupby**(*by*, ***kwargs*)

Returns an GroupBy object constructed from the dataset.

This function can accept any keyword arguments (in kwargs) allowed by the GroupBy constructor.

**Parameters**

- **by** (*str or list of str*) – The list of column names to group by
- **filter** (*ndarray of bool*) – Pass in a boolean array to filter data. If a key no longer exists after filtering it will not be displayed.
- **sort_display** (*bool*) – Defaults to True. set to False if you want to display data in the order of appearance.
- **lex** (*bool*) – When True, use a lexsort to the data.

**Return type**
*GroupBy*

### Examples

All calculations from GroupBy objects will return a Dataset. Operations can be called in the following ways:

Initialize dataset and groupby a single key:

```
>>> #TODO: Need to call np.random.seed(12345) here to deterministically init␣
↪the RNG used below
>>> d = {'strings':np.random.choice(['a','b','c','d','e'], 30)}
>>> for i in range(5): d['col'+str(i)] = np.random.rand(30)
>>> ds = rt.Dataset(d)
>>> gb = ds.groupby('strings')
```

Perform operation on all columns:

```
>>> gb.sum()
*strings   col0   col1   col2   col3   col4
--------   ----   ----   ----   ----   ----
a          2.67   3.35   3.74   3.46   4.20
b          1.36   1.53   2.59   1.24   0.73
c          3.91   2.00   2.76   2.62   2.10
```

(continues on next page)

(continued from previous page)

```
d          4.76   5.13   4.30   3.46   2.21
e          4.18   2.86   2.95   3.22   3.14
```

Perform operation on a single column:

```
>>> gb['col1'].mean()
*strings   col1
--------   ----
a          0.48
e          0.38
d          0.40
d          0.64
c          0.48
```

Perform operation on multiple columns:

```
>>> gb[['col1','col2','col4']].min()
*strings   col1   col2   col4
--------   ----   ----   ----
a          0.05   0.03   0.02
e          0.02   0.24   0.02
d          0.03   0.15   0.16
d          0.17   0.19   0.05
c          0.00   0.03   0.28
```

Perform specific operations on specific columns:

```
>>> gb.agg({'col1':['min','max'], 'col2':['sum','mean']})
              col1          col2
*strings    Min    Max    Sum   Mean
--------   ----   ----   ----   ----
a          0.05   0.92   3.74   0.53
b          0.02   0.72   2.59   0.65
c          0.03   0.73   2.76   0.55
d          0.17   0.96   4.30   0.54
e          0.00   0.82   2.95   0.49
```

GroupBy objects can also be grouped by multiple keys:

```
>>> gbmk = ds.groupby(['strings', 'col1'])
>>> gbmk
*strings   *col1   Count
--------   -----   -----
a           0.05       1
.           0.11       1
.           0.16       1
.           0.55       1
.           0.69       1
         ...     ...
e           0.33       1
.           0.36       1
.           0.68       1
```

(continues on next page)

(continued from previous page)

```
.           0.68       1
.           0.82       1
```

**head**(*n=20*)

Return the first `n` rows.

This function returns the first `n` rows of the Dataset, based on position. It's useful for spot-checking your data.

For negative values of `n`, this function returns all rows except the last `n` rows (equivalent to `ds[:-n, :]`).

**Parameters**
**n** (*int, default 20*) – Number of rows to select.

**Returns**
A view of the first `n` rows of the Dataset.

**Return type**
*Dataset*

**See also:**

*Dataset.tail*
Returns the last `n` rows of the Dataset.

*Dataset.sample*
Returns `N` randomly selected rows of the Dataset.

**classmethod hstack**(*ds_list*, *destroy=False*)

See *Dataset.concat_rows()*.

**imatrix_make**(*dtype=None*, *order='F'*, *colnames=None*, *cats=False*, *gb=False*, *inplace=True*, *retnames=False*)

**Parameters**

- **dtype** (*str or np.dtype, optional, default None*) – Defaults to None, can force a final dtype such as `np.float32`.
- **order** (*{'F', 'C'}*) – Defaults to 'F', can be 'C' also; when 'C' is used, `inplace` cannot be True since the shape will not match.
- **colnames** (*list of str, optional*) – Column names to turn into a 2d matrix. If None is passed, it will use all computable columns in the Dataset.
- **cats** (*bool, default False*) – If set to True will include categoricals.
- **gb** (*bool, default False*) – If set to True will include the groupby keys.
- **inplace** (*bool, default True*) – If set to True (default) will rearrange and stack the columns in the dataset to be part of the matrix. If set to False, the columns in the existing dataset will not be affected.
- **retnames** (*bool, default False*) – Defaults to False. If set to True will return the column names it used.

**Returns**

- **imatrix** (*np.ndarray*) – A 2D array (matrix) containing the data from this *Dataset* with the specified `order`.

- **colnames** (*list of str, optional*) – If `retnames` is True, a list of the column names included in the returned matrix; otherwise, this list is not returned.

### Examples

```
>>> arrsize=3
>>> ds=rt.Dataset({'time': rt.arange(arrsize * 1.0), 'data': rt.
↪arange(arrsize)})
>>> ds.imatrix_make(dtype=rt.int32)
FastArray([[0, 0],
           [1, 1],
           [2, 2]])
```

**imatrix_totals**(*colnames=None*, *name=None*)

**imatrix_xy**(*func*, *name=None*, *showfilter=True*)

**Parameters**

- **func** (*str or callable*) – function or method name of function
- **name** –
- **showfilter** (*bool*) –

**Return type**
X and Y axis calculations

**imatrix_y**(*func*, *name=None*)

**Parameters**

- **func** (*callable or str or list of callable*) – Function or method name of function.
- **name** (*str or list of str, optional*) –

**Returns**
Y axis calculations for the functions

**Return type**
*Dataset*

### Example

```
>>> ds = rt.Dataset({'a1': rt.arange(3)%2, 'b1': rt.arange(3)})
>>> ds.imatrix_y([np.sum, np.mean])
#   a1   b1   Sum   Mean
-   --   --   ---   ----
0    0    0     0   0.00
1    1    1     2   1.00
2    0    2     2   1.00
```

**isin**(*values*)

Call `isin()` for each column in the *Dataset*.

**Parameters**
**values** (*scalar or list or array_like*) – A list or single value to be searched for.

**Returns**
Dataset of boolean arrays with the same column headers as the original dataset. True indicates that the column element occurred in the provided values.

**Return type**
*Dataset*

### Notes

Note: different behavior than pandas DataFrames:

- Pandas handles object arrays, and will make the comparison for each element type in the provided list.
- Riptable favors bytestrings, and will make conversions from unicode/bytes to match for operations as necessary.
- We will also accept single scalars for values.

### Examples

```
>>> data = {'nums': rt.arange(5), 'strs': rt.FA(['a','b','c','d','e'],␣
↪unicode=True)}
>>> ds = rt.Dataset(data)
>>> ds.isin([2, 'b'])
#    nums    strs
-   -----   -----
0   False   False
1   False    True
2   False   False
3   False   False
4   False   False
```

```
>>> df = pd.DataFrame(data)
>>> df.isin([2, 'b'])
    nums   strs
0  False  False
1  False   True
2   True  False
3  False  False
4  False  False
```

**See also:**

pandas.DataFrame.isin

**iterrows**()

NOTE: This routine is slow

It returns a struct with scalar values for each row. It does not preserve dtypes.

Do not modify anything you are iterating over.

### Examples

```
>>> ds = rt.Dataset({'test': rt.arange(10)*3, 'test2': rt.arange(10.0)/2})
>>> temp=[*ds.iterrows()]
>>> temp[2]
(2,
 #   Name    Type      Size   0     1   2
 -   -----   -------   ----   ---   -   -
 0   test    int32     0      27
 1   test2   float64   0      4.5

 [2 columns])
```

**keep**(*func*, *rows=True*)

**func must be set. Examples of func include isfinite, isnan, lambda x: x==0**

- any column that contains all False after calling func will be removed.
- any row that contains all False after calling func will be removed if rows is True.

**Parameters**

- **func** (*callable*) – A function which accepts an array and returns a boolean mask of the same shape as the input.
- **rows** (*bool*) – If rows is True (the default), any rows which are all zeros or all nans will also be removed.

**Return type**
*Dataset*

### Example

```
>>> ds = rt.Dataset({'a': rt.arange(3), 'b': rt.arange(3.0)})
>>> ds.keep(lambda x: x > 1)
#   a      b
-   -   ----
2   2   2.00
```

```
>>> ds.keep(rt.isfinite)
#   a      b
-   -   ----
0   0   0.00
1   1   1.00
2   2   2.00
```

**classmethod load**(*path=''*, *share=None*, *decompress=True*, *info=False*, *include=None*, *filter=None*, *sections=None*, *threads=None*)

Load dataset from .sds file or shared memory.

**Parameters**

- **path** (*str*) – full path to load location + file name (if no .sds extension is included, it will be added)

- **share** (*str, optional*) – shared memory name. loader will check for dataset in shared memory first. if it's not there, the data (if file found on disk) will be loaded into the user's workspace AND shared memory. a sharename must be accompanied by a file name. (the rest of a full path will be trimmed off internally)
- **decompress** (*bool*) – **not implemented.** the internal .sds loader will detect if the file is compressed
- **info** (*bool*) – Defaults to False. If True, load information about the contained arrays instead of loading them from file.
- **include** (*sequence of str, optional*) – Defaults to None. If provided, only load certain columns from the dataset.
- **filter** (*np.ndarray of int or np.ndarray of bool, optional*) –
- **sections** (*sequence of str, optional*) –
- **threads** (*int, optional*) – Defaults to None. Request certain number of threads during load.

### Examples

```
>>> ds = rt.Dataset({'col_'+str(i):np.random.rand(5) for i in range(3)})
>>> ds.save('my_data')
>>> rt.Dataset.load('my_data')
#   col_0   col_1   col_2
-   -----   -----   -----
0    0.94    0.88    0.87
1    0.95    0.93    0.16
2    0.18    0.94    0.95
3    0.41    0.60    0.05
4    0.53    0.23    0.71
```

```
>>> ds = rt.Dataset.load('my_data', share='sharename')
>>> os.remove('my_data.sds')
>>> os.path.exists('my_data.sds')
False
```

```
>>> rt.Dataset.load('my_data', share='sharename')
#   col_0   col_1   col_2
-   -----   -----   -----
0    0.94    0.88    0.87
1    0.95    0.93    0.16
2    0.18    0.94    0.95
3    0.41    0.60    0.05
4    0.53    0.23    0.71
```

**mask_and_isfinite()**

Return a boolean array that's True for each *Dataset* row in which all values are finite, False otherwise.

A value is considered to be finite if it's not positive or negative infinity or a NaN (Not a Number).

This method applies AND to all columns using `riptable.isfinite()`.

**Returns**
A `FastArray` that's True for each *`Dataset`* row in which all values are finite, False otherwise.

**Return type**
`FastArray`

**See also:**

`riptable.isfinite`, `riptable.isnotfinite`, `riptable.isinf`, `riptable.isnotinf`, `FastArray.isfinite`, `FastArray.isnotfinite`, `FastArray.isinf`, `FastArray.isnotinf`

*`Dataset.mask_or_isfinite`*
Return a boolean array that's True for each *`Dataset`* row that has at least one finite value.

*`Dataset.mask_or_isinf`*
Return a boolean array that's True for each *`Dataset`* row that has at least one value that's positive or negative infinity.

*`Dataset.mask_and_isinf`*
Return a boolean array that's True for each *`Dataset`* row that contains all infinite values.

### Examples

```
>>> ds = rt.Dataset({'a': [1.0, 2.0, 3.0], 'b': [0, rt.nan, rt.inf]})
>>> ds
#      a      b
-   ----   ----
0   1.00   0.00
1   2.00    nan
2   3.00    inf
>>> ds.mask_and_isfinite()
FastArray([ True, False, False])
```

**`mask_and_isinf`()**

Return a boolean array that's True for each *`Dataset`* row in which all values are positive or negative infinity, False otherwise.

This method applies `AND` to all columns using `riptable.isinf()`.

**Returns**
A `FastArray` that's True for each *`Dataset`* row in which all values are positive or negative infinity, False otherwise.

**Return type**
`FastArray`

**See also:**

`riptable.isinf`, `riptable.isnotinf`, `riptable.isfinite`, `riptable.isnotfinite`, `FastArray.isinf`, `FastArray.isnotinf`, `FastArray.isfinite`, `FastArray.isnotfinite`

*`Dataset.mask_or_isinf`*
Return a boolean array that's True for each *`Dataset`* row that has at least one value that's positive or negative infinity.

*`Dataset.mask_or_isfinite`*
Return a boolean array that's True for each *`Dataset`* row that has at least one finite value.

*Dataset.mask_and_isfinite*
Return a boolean array that's True for each *Dataset* row that contains all finite values.

### Examples

```
>>> ds = rt.Dataset({'a': [1.0, rt.inf, 3.0], 'b': [rt.inf, -rt.inf, rt.nan]})
>>> ds
#      a      b
-   ----   ----
0   1.00    inf
1    inf   -inf
2   3.00    nan
>>> ds.mask_and_isinf()
FastArray([False,  True, False])
```

**mask_and_isnan**()

Return a boolean array that's True for each *Dataset* row in which every value is NaN, otherwise False.

This method applies AND to all columns using `riptable.isnan()`.

**Returns**
A FastArray that's True for each *Dataset* row that contains all NaNs, otherwise False.

**Return type**
FastArray

**See also:**

`riptable.isnan`

*Dataset.mask_or_isnan*
Return a boolean array that's True for each *Dataset* row that contains at least one NaN.

### Examples

```
>>> ds = rt.Dataset({'a': [1, 2, rt.nan], 'b': [0, rt.nan, rt.nan]})
>>> ds
#      a      b
-   ----   ----
0   1.00   0.00
1   2.00    nan
2    nan    nan
>>> ds.mask_and_isnan()
FastArray([False, False,  True])
```

**mask_or_isfinite**()

Return a boolean array that's True for each *Dataset* row that has at least one finite value, False otherwise.

A value is considered to be finite if it's not positive or negative infinity or a NaN (Not a Number).

This method applies OR to all columns using `riptable.isfinite()`.

**Returns**
A FastArray that's True for each *Dataset* row that has at least one finite value, False otherwise.

**Return type**
FastArray

**See also:**

riptable.isfinite, riptable.isnotfinite, riptable.isinf, riptable.isnotinf, FastArray.isfinite, FastArray.isnotfinite, FastArray.isinf, FastArray.isnotinf

*Dataset.mask_and_isfinite*
Return a boolean array that's True for each *Dataset* row that contains all finite values.

*Dataset.mask_or_isinf*
Return a boolean array that's True for each *Dataset* row that has at least one value that's positive or negative infinity.

*Dataset.mask_and_isinf*
Return a boolean array that's True for each *Dataset* row that contains all infinite values.

### Examples

```
>>> ds = rt.Dataset({'a': [1, 2, rt.inf], 'b': [0, rt.inf, rt.nan]})
>>> ds
#      a      b
-   ----   ----
0   1.00   0.00
1   2.00    inf
2    inf    nan
>>> ds.mask_or_isfinite()
FastArray([ True,  True, False])
```

**mask_or_isinf**()

Return a boolean array that's True for each *Dataset* row that has at least one value that's positive or negative infinity, False otherwise.

This method applies OR to all columns using riptable.isinf().

**Returns**
A FastArray that's True for each *Dataset* row that has at least one value that's positive or negative infinity, False otherwise.

**Return type**
FastArray

**See also:**

riptable.isinf, riptable.isnotinf, riptable.isfinite, riptable.isnotfinite, FastArray.isinf, FastArray.isnotinf, FastArray.isfinite, FastArray.isnotfinite

*Dataset.mask_and_isinf*
Return a boolean array that's True for each *Dataset* row that contains all infinite values.

*Dataset.mask_or_isfinite*
Return a boolean array that's True for each *Dataset* row that has at least one finite value.

*Dataset.mask_and_isfinite*
Return a boolean array that's True for each *Dataset* row that contains all finite values.

### Examples

```
>>> ds = rt.Dataset({'a': [1, 2, rt.inf], 'b': [0, rt.inf, rt.nan]})
>>> ds
#      a      b
-   ----   ----
0   1.00   0.00
1   2.00    inf
2    inf    nan
>>> ds.mask_or_isinf()
FastArray([False,  True,  True])
```

**mask_or_isnan**()

Return a boolean array that's True for each *Dataset* row that contains at least one NaN, otherwise False.

This method applies OR to all columns using `riptable.isnan()`.

**Returns**

A `FastArray` that's True for each *Dataset* row that contains at least one NaN, otherwise False.

**Return type**

`FastArray`

**See also:**

`riptable.isnan`

*Dataset.mask_and_isnan*

Return a boolean array that's True for each all-NaN *Dataset* row.

### Examples

```
>>> ds = rt.Dataset({'a': [1, 2, rt.nan], 'b': [0, rt.nan, rt.nan]})
>>> ds
#      a      b
-   ----   ----
0   1.00   0.00
1   2.00    nan
2    nan    nan
>>> ds.mask_or_isnan()
FastArray([False, True,  True])
```

**max**(*axis=0*, *as_dataset=True*, *fill_value=max*)

See documentation of `reduce()`

**mean**(*axis=0*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**median**(*axis=0*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**melt**(*id_vars=None*, *value_vars=None*, *var_name=None*, *value_name='value'*, *trim=False*)

"Unpivots" a Dataset from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a Dataset into a format where one or more columns are identifier variables (id_vars), while all other columns, considered measured variables (value_vars), are "unpivoted" to the row axis, leaving just two non-identifier columns, 'variable' and 'value'.

**Parameters**

- **id_vars** (*tuple, list, or ndarray, optional*) – Column(s) to use as identifier variables.
- **value_vars** (*tuple, list, or ndarray, optional*) – Column(s) to unpivot. If not specified, uses all columns that are not set as id_vars.
- **var_name** (*str, optional*) – Name to use for the 'variable' column. If None it uses 'variable'.
- **value_name** (*str*) – Name to use for the 'value' column. Defaults to 'value'.
- **trim** (*bool*) – defaults to False. Set to True to drop zeros or nan (trims a dataset)

### Notes

BUG: the current version does not handle categoricals correctly.

**merge**(*right*, *on=None*, *left_on=None*, *right_on=None*, *how='left'*, *suffixes=('_x', '_y')*, *indicator=False*, *columns_left=None*, *columns_right=None*, *verbose=False*, *hint_size=0*)

**merge2**(*right*, *on=None*, *left_on=None*, *right_on=None*, *how='left'*, *suffixes=None*, *copy=True*, *indicator=False*, *columns_left=None*, *columns_right=None*, *validate=None*, *keep=None*, *high_card=None*, *hint_size=None*)

**merge_asof**(*right*, *on=None*, *left_on=None*, *right_on=None*, *by=None*, *left_by=None*, *right_by=None*, *suffixes=None*, *copy=True*, *columns_left=None*, *columns_right=None*, *tolerance=None*, *allow_exact_matches=True*, *direction='backward'*, *action_on_unsorted='sort'*, *matched_on=False*, ***kwargs*)

**merge_lookup**(*right*, *on=None*, *left_on=None*, *right_on=None*, *require_match=False*, *suffix=None*, *copy=True*, *columns_left=None*, *columns_right=None*, *keep=None*, *inplace=False*, *high_card=None*, *hint_size=None*, *suffixes=None*)

Combine two *Dataset* objects by performing a database-style left-join operation on columns.

This method has an option to perform an in-place merge, in which columns from the right *Dataset* are added to the left *Dataset* (`self`).

Also note that this method has both `suffix` and `suffixes` as optional parameters. At most one can be specified; see usage details below.

**Parameters**

- **right** (*Dataset*) – The *Dataset* to merge with the left *Dataset* (`self`). If rows in `right` don't have matches in the left *Dataset* they will be discarded. If they match multiple rows in the left *Dataset* they will be duplicated appropriately. (All rows in the left *Dataset* are always preserved in a *merge_lookup*. If there's no matching key in `right`, an invalid value is used as a fill value.)
- **on** (*str or (str, str) or list of str or list of (str, str), optional*) – Names of columns (keys) to join on. If `on` isn't specified, `left_on` and `right_on` must be specified. Options for types:
  - Single string: Join on one column that has the same name in both *Dataset* objects.

  - List: A list of strings is treated as a multi-key in which all associated key column values in the left *Dataset* must have matches in `right`. The column names must be the same in both *Dataset* objects, unless they're in a tuple; see below.
  - Tuple: Use a tuple to specify key columns that have different names. For example, `("col_a", "col_b")` joins on `col_a` in the left *Dataset* and `col_b` in `right`. Both columns are in the returned *Dataset* unless you specify otherwise using `columns_left` or `columns_right`.

- **left_on** (*str or list of str, optional*) – Use instead of `on` to specify names of columns in the left *Dataset* to join on. A list of strings is treated as a multi-key in which all associated key column values in the left *Dataset* must have matches in `right`. If both `on` and `left_on` are specified, an error is raised.
- **right_on** (*str or list of str, optional*) – Use instead of `on` to specify names of columns in the right *Dataset* to join on. A list of strings is treated as a multi-key in which all associated key column values in `right` must have matches in the left *Dataset*. If both `on` and `right_on` are specified, an error is raised.
- **require_match** (bool, default `False`) – When `True`, all keys in the left *Dataset* are required to have a matching key in `right`, and an error is raised when this requirement is not met.
- **suffix** (*str, optional*) – Suffix to apply to overlapping non-key-column names in `right` that are included in the returned *Dataset*. Cannot be used with `suffixes`. If there are overlapping non-key-column names in the returned *Dataset* and `suffix` or `suffixes` isn't specified, an error is raised.
- **copy** (bool, default `True`) – Set to `False` to avoid copying data when possible. This can reduce memory usage, but be aware that data can be shared among the left *Dataset*, `right`, and the *Dataset* returned by this function.
- **columns_left** (*str or list of str, optional*) – Names of columns from the left *Dataset* to include in the merged *Dataset*. By default, all columns are included. When `inplace=True`, this can't be used; remove columns in a separate operation instead.
- **columns_right** (*str or list of str, optional*) – Names of columns from `right` to include in the merged *Dataset*. By default, all columns are included.
- **keep** (*{None, 'first', 'last'}, optional*) – When `right` has more than one match for a key in the left *Dataset*, only one can be used; this parameter indicates whether it should be the first or last match. By default (`keep=None`), an error is raised if there's more than one matching key value in `right`.
- **inplace** (bool, default `False`) – If `False` (the default), a new *Dataset* is returned. If `True`, the operation is performed in place (the data in `self` is modified). When `inplace=True`:
  - `suffixes` can't be used; use `suffix` instead.
  - `columns_left` can't be used; remove columns in a separate operation.
- **high_card** (*bool or (bool, bool), optional*) – Hint to the low-level grouping implementation that the key(s) of the left or right *Dataset* contain a high number of unique values (cardinality); the grouping logic *may* use this hint to select an algorithm that can provide better performance for such cases.
- **hint_size** (*int or (int, int), optional*) – An estimate of the number of unique keys used for the join. Used as a performance hint to the low-level grouping implementation. This hint is typically ignored when `high_card` is specified.

- **suffixes** (*tuple of (str, str), optional*) – Suffixes to apply to returned overlapping non-key-column names in the left and right *Dataset* objects, respectively. Cannot be used with `suffix` or with `inplace=True`. By default, an error is raised for any overlapping non-key columns that will be in the returned *Dataset*.

**Returns**

A merged *Dataset* that has the same number of rows as `self`. If `inplace=True`, `self` is modified and returned. Otherwise, a new *Dataset* is returned.

**Return type**

*Dataset*

**See also:**

**rt_merge.merge_lookup**
Merge two *Dataset* objects.

**rt_merge.merge_asof**
Merge two *Dataset* objects using the nearest key.

**rt_merge.merge2**
Merge two *Dataset* objects using various database-style joins.

**rt_merge.merge_indices**
Return the left and right indices created by the join engine.

***Dataset.merge2***
Merge two *Dataset* objects using various database-style joins.

***Dataset.merge_asof***
Merge two *Dataset* objects using the nearest key.

### Examples

A basic merge on a single column. In a *merge_lookup*, all rows in the left *Dataset* are in the resulting *Dataset*.

```
>>> ds_l = rt.Dataset({"Symbol": rt.FA(["GME", "AMZN", "TSLA", "SPY", "TSLA",
...                                     "AMZN", "GME", "SPY", "GME", "TSLA"])})
>>> ds_r = rt.Dataset({"Symbol": rt.FA(["TSLA", "GME", "AMZN", "SPY"]),
...                    "Trader": rt.FA(["Nate", "Elon", "Josh", "Dan"])})
>>> ds_l
#   Symbol
-   ------
0   GME
1   AMZN
2   TSLA
3   SPY
4   TSLA
5   AMZN
6   GME
7   SPY
8   GME
9   TSLA

[10 rows x 1 columns] total bytes: 40.0 B
>>> ds_r
```

(continues on next page)

(continued from previous page)

```
#   Symbol   Trader
-   ------   ------
0   TSLA     Nate
1   GME      Elon
2   AMZN     Josh
3   SPY      Dan

[4 rows x 2 columns] total bytes: 32.0 B
>>> ds_l.merge_lookup(ds_r, on="Symbol")
#   Symbol   Trader
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY      Dan
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY      Dan
8   GME      Elon
9   TSLA     Nate

[10 rows x 2 columns] total bytes: 80.0 B
```

If a key in the left *Dataset* has no match in the right *Dataset*, an invalid value is used as a fill value.

```
>>> ds2_l = rt.Dataset({"Symbol": rt.FA(["GME", "AMZN", "TSLA", "SPY", "TSLA",
...                                      "AMZN", "GME", "SPY", "GME", "TSLA"])})
>>> ds2_r = rt.Dataset({"Symbol": rt.FA(["TSLA", "GME", "AMZN"]),
...                     "Trader": rt.FA(["Nate", "Elon", "Josh"])})
>>> ds2_l.merge_lookup(ds2_r, on="Symbol")
#   Symbol   Trader
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY
8   GME      Elon
9   TSLA     Nate

[10 rows x 2 columns] total bytes: 80.0 B
```

When key columns have different names, use `left_on` and `right_on` to specify them:

```
>>> ds_r.col_rename("Symbol", "Primary_Symbol")
>>> ds_l.merge_lookup(ds_r, left_on="Symbol", right_on="Primary_Symbol",
...                   columns_right="Trader")
#   Symbol   Trader
```

(continues on next page)

(continued from previous page)

```
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY      Dan
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY      Dan
8   GME      Elon
9   TSLA     Nate

[10 rows x 2 columns] total bytes: 80.0 B
```

For non-key columns with the same name that will be returned, specify `suffixes`:

```
>>> # Add duplicate non-key columns.
>>> ds_l.Value = rt.FA([0.72, 0.85, 0.14, 0.55, 0.77, 0.65, 0.23, 0.15, 0.43,␣
↪0.25])
>>> ds_r.Value = rt.FA([0.28, 0.56, 0.89, 0.74])
>>> # You can also use a tuple to specify left and right key columns.
>>> ds_l.merge_lookup(ds_r, on=("Symbol", "Primary_Symbol"),
...                   suffixes=["_1", "_2"], columns_right=["Value", "Trader"])
#   Symbol   Value_1   Value_2   Trader
-   ------   -------   -------   ------
0   GME         0.72      0.56   Elon
1   AMZN        0.85      0.89   Josh
2   TSLA        0.14      0.28   Nate
3   SPY         0.55      0.74   Dan
4   TSLA        0.77      0.28   Nate
5   AMZN        0.65      0.89   Josh
6   GME         0.23      0.56   Elon
7   SPY         0.15      0.74   Dan
8   GME         0.43      0.56   Elon
9   TSLA        0.25      0.28   Nate

[10 rows x 4 columns] total bytes: 240.0 B
```

When `on` is a list, a multi-key join is performed. All keys must match in the right *Dataset*.

If a matching value for a key in the left *Dataset* isn't found in the right *Dataset*, the returned *Dataset* includes a row with the columns from the left *Dataset* but with NaN values in the columns from `right`.

```
>>> # Add associated Size values for multi-key join. Note that one
>>> # symbol-size pair in the left Dataset doesn't have a match in
>>> # the right Dataset.
>>> ds_l.Size = rt.FA([500, 150, 430, 225, 430, 320, 175, 620, 135, 260])
>>> ds_r.Size = rt.FA([430, 500, 150, 2250])
>>> # Pass a list of key columns that contains a tuple.
>>> ds_l.merge_lookup(ds_r, on=[("Symbol", "Primary_Symbol"), "Size"],
...                   suffixes=["_1", "_2"])
#   Size   Symbol   Value_1   Trader   Value_2
-   ----   ------   -------   ------   -------
```

(continues on next page)

(continued from previous page)

```
0    500   GME         0.72   Elon        0.56
1    150   AMZN        0.85   Josh        0.89
2    430   TSLA        0.14   Nate        0.28
3    225   SPY         0.55                nan
4    430   TSLA        0.77   Nate        0.28
5    320   AMZN        0.65                nan
6    175   GME         0.23                nan
7    620   SPY         0.15                nan
8    135   GME         0.43                nan
9    260   TSLA        0.25                nan

[10 rows x 5 columns] total bytes: 280.0 B
```

When the right *Dataset* has more than one matching key, use *keep* to specify which one to use:

```
>>> ds_l = rt.Dataset({"Symbol": rt.FA(["GME", "AMZN", "TSLA", "SPY", "TSLA",
...                                     "AMZN", "GME", "SPY", "GME", "TSLA"])})
>>> ds_r = rt.Dataset({"Symbol": rt.FA(["TSLA", "GME", "AMZN", "SPY", "SPY"]),
...                    "Trader": rt.FA(["Nate", "Elon", "Josh", "Dan", "Amy"])}
↪)
>>> ds_l.merge_lookup(ds_r, on="Symbol", keep="last")
#   Symbol   Trader
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY      Amy
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY      Amy
8   GME      Elon
9   TSLA     Nate

[10 rows x 2 columns] total bytes: 80.0 B
```

Invalid values are not treated as equal keys:

```
>>> ds1 = rt.Dataset({"Key": [1.0, rt.nan, 2.0], "Value1": ["a", "b", "c"]})
>>> ds2 = rt.Dataset({"Key": [1.0, 2.0, rt.nan], "Value2": [1, 2, 3]})
>>> ds1.merge_lookup(ds2, on="Key")
#    Key   Value1   Value2
-   ----   ------   ------
0   1.00   a             1
1    nan   b           Inv
2   2.00   c             2

[3 rows x 3 columns] total bytes: 72.0 B
```

**min**(*axis=0*, *as_dataset=True*, *fill_value=min*)

See documentation of `reduce()`

**nanargmax**(*axis=0*, *as_dataset=True*, *fill_value=None*)

**nanargmin**(*axis=0*, *as_dataset=True*, *fill_value=None*)

**nanmax**(*axis=0*, *as_dataset=True*, *fill_value=max*)

See documentation of `reduce()`

**nanmean**(*axis=0*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**nanmedian**(*axis=0*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**nanmin**(*axis=0*, *as_dataset=True*, *fill_value=min*)

See documentation of `reduce()`

**nanstd**(*axis=0*, *ddof=1*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**nansum**(*axis=0*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**nanvar**(*axis=0*, *ddof=1*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**noncomputable**()

returns a dict of noncomputable columns. includes groupby keys

**normalize_minmax**(*axis=0*, *as_dataset=True*, *fill_value=None*)

**normalize_zscore**(*axis=0*, *as_dataset=True*, *fill_value=None*)

**one_hot_encode**(*columns=None*, *exclude=None*)

Replaces categorical columns with one-hot-encoded columns for their categories. Original columns will be removed from the dataset.

Default is to encode all categorical columns. Otherwise, certain columns can be specified. Also an optional exclude list for convenience.

**Parameters**

- **columns** (*list of str, optional*) – specify columns to encode (if set, exclude param will be ignored)
- **exclude** (*str or list of str, optional*) – exclude certain columns from being encoded

**outliers**(*col_keep*)

return a dataset with the min/max outliers for each column

**pivot**(*labels=None*, *columns=None*, *values=None*, *ordered=True*, *lex=None*, *filter=None*)

Return reshaped Dataset or Multiset organized by labels / column values.

Uses unique values from specified `labels` / `columns` to form axes of the resulting Dataset. This function does not support data aggregation, multiple values will result in a Multiset in the columns.

**Parameters**

- **labels** (*str or list of str, optional*) – Column to use to make new labels. If None, uses existing labels.
- **columns** (*str*) – Column to use to make new columns.

- **values** (*str or list of str, optional*) – Column(s) to use for populating new values. If not specified, all remaining columns will be used and the result will have a Multiset.
- **ordered** (*bool, defaults to True*) –
- **lex** (*bool, defaults to None*) –
- **filter** (*ndarray of bool, optional*) –

**Return type**
*Dataset* or *Multiset*

**Raises**
**ValueError:** – When there are any `labels`, `columns` combinations with multiple values.

### Examples

```
>>> ds = rt.Dataset({'foo': ['one', 'one', 'one', 'two', 'two', 'two'],
...                  'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                  'baz': [1, 2, 3, 4, 5, 6],
...                  'zoo': ['x', 'y', 'z', 'q', 'w', 't']})
>>> ds
#   foo   bar   baz   zoo
-   ---   ---   ---   ---
0   one   A       1   x
1   one   B       2   y
2   one   C       3   z
3   two   A       4   q
4   two   B       5   w
5   two   C       6   t
```

```
>>> ds.pivot(labels='foo', columns='bar', values='baz')
foo   A   B   C
---  --  --  --
one   1   2   3
two   4   5   6
```

**putmask**(*mask*, *values*)

Call riptable `putmask` routine which is faster than `__setitem__` with bracket indexing.

**Parameters**

- **mask** (*ndarray of bools*) – boolean numpy array with a length equal to the number of rows in the dataset.
- **values** (*rt.Dataset or ndarray*) –
  - Dataset: Corresponding column values will be copied, must have same shape as calling dataset.
  - ndarray: Values will be copied to each column, must have length equal to calling dataset's nrows.

**Return type**
None

### Examples

```
>>> ds = rt.Dataset({'a': np.arange(-3,3), 'b':np.arange(6), 'c':np.arange(10,
↪70,10)})
>>> ds
#    a   b    c
-   --   -   --
0   -3   0   10
1   -2   1   20
2   -1   2   30
3    0   3   40
4    1   4   50
5    2   5   60
```

```
>>> ds1 = ds.copy()
>>> ds.putmask(ds.a < 0, np.arange(100,106))
>>> ds
#     a     b     c
-   ---   ---   ---
0   100   100   100
1   101   101   101
2   102   102   102
3     0     3    40
4     1     4    50
5     2     5    60
```

```
>>> ds.putmask(np.array([True, True, False, False, False, False]), ds1)
>>> ds
#     a     b     c
-   ---   ---   ---
0    -3     0    10
1    -2     1    20
2   102   102   102
3     0     3    40
4     1     4    50
5     2     5    60
```

**quantile**(*q=None*, *fill_value=None*)

**Parameters**

- **q** (*defaults to [0.50], list of quantiles*) –
- **fill_value** (*optional place-holder value for non-computable columns*) –

**Return type**
Dataset.

**reduce**(*func*, *axis=0*, *as_dataset=True*, *fill_value=None*, ***kwargs*)

Returns calculated reduction along axis.

---

**Note:** Behavior for `axis=None` differs from pandas!

The default `fill_value` is `None` (drop) to ensure the most sensible default behavior for `axis=None` and `axis=1`. As a thought problem, consider all three axis behaviors for func=sum or product.

---

**Parameters**

- **func** (*reduction function (e.g. numpy.sum, numpy.std, ...)*) –
- **axis** (*int, optional*) –
  - 0: reduce over columns, returning a Struct (or Dataset) of scalars. Reasonably cheap. String synonyms: `c`, `C`, `col`, `COL`, `column`, `COLUMN`.
  - 1: reduce over rows, returning an array of scalars. Could well be expensive/slow. String synonyms: `r`, `R`, `row`, `ROW`.
  - `None`: reduce over rows and columns, returning a scalar. Could well be very expensive/slow. String synonyms: `all`, `ALL`.
- **as_dataset** (*bool*) – When `axis` is 0, this flag specifies a Dataset should be returned instead of a Struct. Defaults to False.
- **fill_value** –
  - fill_value=None (default) -> drop all non-computable type columns from result
  - **fill_value=alt_func -> force computation with alt_func**
    (for axis=1 must work on indiv. elements)
  - fill_value=scalar -> apply as uniform fill value
  - **fill_value=dict (defaultdict) of colname->fill_value, where**
    None (or absent if not a defaultdict) still means drop column and an alt_func still means force compute via alt_func.
- **kwargs** – all other kwargs are passed to `func`

**Return type**
*Struct* or *Dataset* or array or scalar

**sample**(*N=10*, *filter=None*, *seed=None*)

Return a given number of randomly selected *Dataset* rows.

This function is useful for spot-checking your data, especially if the first or last rows aren't representative.

**Parameters**

- **N** (*int, default 10*) – Number of rows to select. The entire *Dataset* is returned if `N` is greater than the number of *Dataset* rows.
- **filter** (*array (bool or int), optional*) – A boolean mask or index array to filter values before selection. A boolean mask must have the same length as the columns of the original *Dataset*.
- **seed** (*int or other types, optional*) – A seed to initialize the random number generator. If one is not provided, the generator is initialized using random data from the OS. For details and other accepted types, see the `seed` parameter for `numpy.random.default_rng`.

**Returns**
A new *Dataset* containing the randomly selected rows.

**Return type**
*Dataset*

**See also:**

*Dataset.head*
Return the first rows of a *Dataset*.

*Dataset.tail*
Return the last rows of a *Dataset*.

*FastArray.sample*
Return a given number of randomly selected values from a *FastArray*.

### Examples

```
>>> ds = rt.Dataset({"A": rt.FA([0, 1, 2, 3, 4]),
...                  "B": rt.FA(["a", "b", "c", "d", "e"])})
>>> ds.sample(2)
#   A   B  # random
-   -   -
0   0   a
1   1   b

[2 rows x 2 columns] total bytes: 10.0 B
```

Filter with a boolean mask array:

```
>>> f = ds.A > 2
>>> ds.sample(2, filter=f)
#   A   B  # random
-   -   -
0   3   d
1   4   e

[2 rows x 2 columns] total bytes: 10.0 B
```

Filter with an index array:

```
>>> f = rt.FA([0, 1, 2])
>>> ds.sample(2, filter=f)
#   A   B  # random
-   -   -
0   0   a
1   2   c

[2 rows x 2 columns] total bytes: 10.0 B
```

**save**(*path=''*, *share=None*, *compress=True*, *overwrite=True*, *name=None*, *onefile=False*, *bandsize=None*, *append=None*, *complevel=None*)

Save a dataset to a single .sds file or shared memory.

**Parameters**

- **path** (*str or os.PathLike*) – full path to save location + file name (if no .sds extension is included, it will be added)

- **share** (*str, optional*) – Shared memory name. If set, dataset will be saved to shared memory and NOT to disk when shared memory is specified, a filename must be included in path. only this will be used, the rest of the path will be discarded.
- **compress** (*bool*) – Use compression when saving the file. Shared memory is always saved uncompressed.
- **overwrite** (*bool*) – Defaults to True. If False, prompt the user when overwriting an existing .sds file; mainly useful for Struct.save(), which may call Dataset.save() multiple times.
- **name** (*str, optional*) –
- **bandsize** (*int, optional*) – If set to an integer > 10000 it will compress column data every bandsize rows
- **append** (*str, optional*) – If set to a string it will append to the file with the section name.
- **complevel** (*int, optional*) – Compression level from 0 to 9. 2 (default) is average. 1 is faster, less compressed, 3 is slower, more compressed.

### Examples

```
>>> ds = rt.Dataset({'col_'+str(i):a rt.range(5) for i in range(3)})
>>> ds.save('my_data')
>>> os.path.exists('my_data.sds')
True
```

```
>>> ds.save('my_data', overwrite=False)
my_data.sds already exists and is a file. Overwrite? (y/n) n
No file was saved.
```

```
>>> ds.save('my_data', overwrite=True)
Overwriting file with my_data.sds
```

```
>>> ds.save('shareds1', share='sharename')
>>> os.path.exists('shareds1.sds')
False
```

**See also:**

*Dataset.load*, `Struct.save`, `Struct.load`, `load_sds`, `load_h5`

**show_all**(*max_cols=8*)

Display all rows and up to the specified number of columns.

**Parameters**

**max_cols** (*int*) – The maximum number of columns to display.

### Notes

**TODO: This method currently displays the data using 'print'; it should be deprecated or adapted** to use our normal display code so it works e.g. in a Jupyter notebook.

**sort_copy**(*by*, *ascending=True*, *kind='mergesort'*, *na_position='last'*)

Return a copy of the *Dataset* that's sorted by the specified columns.

The columns are sorted in the order given. The original *Dataset* is not modified.

**Parameters**

- **by** (*str or list of str*) – The column name or list of column names to sort by. The columns are sorted in the order given.
- **ascending** (*bool, default True*) – Whether the sort is ascending. When True (the default), the sort is ascending. When False, the sort is descending.
- **kind** (*str*) – **Not used.** The sorting algorithm used is 'mergesort'; user-provided values for this parameter are ignored.
- **na_position** (*str*) – **Not used.** If `ascending` is True (the default), NaN values are put last. If `ascending` is False, NaN values are put first. User-provided values for this parameter are ignored.

**Return type**

*Dataset*

**See also:**

*Dataset.sort_inplace*
Sort the *Dataset*, modifying the original data.

*Dataset.sort_view*
Sort the *Dataset* columns only when displayed.

### Examples

Create a *Dataset*:

```
>>> ds = rt.Dataset({'a': rt.arange(10), 'b':5*['A', 'B'], 'c':3*[10,20,
↪30]+[10]})
>>> ds
#   a   b    c
-   -   -   --
0   0   A   10
1   1   B   20
2   2   A   30
3   3   B   10
4   4   A   20
5   5   B   30
6   6   A   10
7   7   B   20
8   8   A   30
9   9   B   10
```

Sort column `b`, then column `c`:

```
>>> ds.sort_copy(['b','c'])
#   a   b    c
-   -   -   --
0   0   A   10
1   6   A   10
2   4   A   20
3   2   A   30
4   8   A   30
5   3   B   10
6   9   B   10
7   1   B   20
8   7   B   20
9   5   B   30
```

Sort column a in descending order:

```
>>> ds.sort_copy('a', ascending = False)
#   a   b    c
-   -   -   --
0   9   B   10
1   8   A   30
2   7   B   20
3   6   A   10
4   5   B   30
5   4   A   20
6   3   B   10
7   2   A   30
8   1   B   20
9   0   A   10
```

**sort_inplace**(*by*, *ascending=True*, *kind='mergesort'*, *na_position='last'*)

Return a *Dataset* with the specified columns sorted in place.

The columns are sorted in the order given. To preserve data alignment, this method modifies the order of all *Dataset* rows.

**Parameters**

- **by** (*str or list of str*) – The column name or list of column names to sort by. The columns are sorted in the order given.
- **ascending** (*bool, default True*) – Whether the sort is ascending. When True (the default), the sort is ascending. When False, the sort is descending.
- **kind** (*str*) – **Not used.** The sorting algorithm used is 'mergesort'; user-provided values for this parameter are ignored.
- **na_position** (*str*) – **Not used.** If ascending is True (the default), NaN values are put last. If ascending is False, NaN values are put first. User-provided values for this parameter are ignored.

**Returns**

The reference to the input *Dataset* is returned to allow for method chaining.

**Return type**

*Dataset*

**See also:**

*Dataset.sort_copy*
Returns a sorted copy of the *Dataset*.

*Dataset.sort_view*
Sorts the *Dataset* columns only when displayed.

### Examples

Create a *Dataset*:

```
>>> ds = rt.Dataset({'a': rt.arange(10), 'b':5*['A', 'B'], 'c':3*[10,20,
↪30]+[10]})
>>> ds
#   a   b    c
-   -   -   --
0   0   A   10
1   1   B   20
2   2   A   30
3   3   B   10
4   4   A   20
5   5   B   30
6   6   A   10
7   7   B   20
8   8   A   30
9   9   B   10
```

Sort column b, then column c:

```
>>> ds.sort_inplace(['b','c'])
#   a   b    c
-   -   -   --
0   0   A   10
1   6   A   10
2   4   A   20
3   2   A   30
4   8   A   30
5   3   B   10
6   9   B   10
7   1   B   20
8   7   B   20
9   5   B   30
```

Sort column a in descending order:

```
>>> ds.sort_inplace('a', ascending = False)
#   a   b    c
-   -   -   --
0   9   B   10
1   8   A   30
2   7   B   20
3   6   A   10
4   5   B   30
5   4   A   20
6   3   B   10
```

(continues on next page)

(continued from previous page)

```
7   2   A   30
8   1   B   20
9   0   A   10
```

**sort_view**(*by*, *ascending=True*, *kind='mergesort'*, *na_position='last'*)

Sort the specified columns only when displayed.

This routine is fast and does not change data underneath.

**Parameters**

- **by** (*string or list of strings*) – The column name or list of column names to sort by. The columns are sorted in the order given.
- **ascending** (*bool, default True*) – Whether the sort is ascending. When True (the default), the sort is ascending. When False, the sort is descending.
- **kind** (*str*) – **Not used.** The sorting algorithm used is 'mergesort'; user-provided values for this parameter are ignored.
- **na_position** (*str*) – **Not used.** If `ascending` is True (the default), NaN values are put last. If `ascending` is False, NaN values are put first. User-provided values for this parameter are ignored.

**Return type**

*Dataset*

**See also:**

***Dataset.sort_copy***

Return a sorted copy of the *Dataset*.

***Dataset.sort_inplace***

Sort the *Dataset*, modifying the original data.

### Examples

Create a *Dataset*:

```
>>> ds = rt.Dataset({'a': rt.arange(10), 'b':5*['A', 'B'], 'c':3*[10,20,
↪30]+[10]})
>>> ds
#   a   b    c
-   -   -   --
0   0   A   10
1   1   B   20
2   2   A   30
3   3   B   10
4   4   A   20
5   5   B   30
6   6   A   10
7   7   B   20
8   8   A   30
9   9   B   10
```

Sort column b, then column c:

```
>>> ds.sort_view(['b','c'])
#   a   b    c
-   -   -   --
0   0   A   10
1   6   A   10
2   4   A   20
3   2   A   30
4   8   A   30
5   3   B   10
6   9   B   10
7   1   B   20
8   7   B   20
9   5   B   30
```

Sort column `a` in descending order:

```
>>> ds.sort_view('a', ascending = False)
#   a   b    c
-   -   -   --
0   9   B   10
1   8   A   30
2   7   B   20
3   6   A   10
4   5   B   30
5   4   A   20
6   3   B   10
7   2   A   30
8   1   B   20
9   0   A   10
```

**sorts_off**()

Turns off all row/column sorts for display (happens when sort_view is called) If sort is cached, it will remain in cache in case sorts are toggled back on.

**Returns**
None

**sorts_on**()

Turns on all row/column sorts for display. False by default. sorts_view must have been called before

**Returns**
None

**std**(*axis=0*, *ddof=1*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**sum**(*axis=0*, *as_dataset=True*, *fill_value=None*)

See documentation of `reduce()`

**tail**(*n=20*)

Return the last `n` rows.

This function returns the last `n` rows of the Dataset, based on position. It's useful for spot-checking your data, especially after sorting or appending rows.

For negative values of `n`, this function returns all rows except the first `n` rows (equivalent to `ds[n:, :]`).

**Parameters**
**n** (*int, default 20*) – Number of rows to select.

**Returns**
A view of the last n rows of the Dataset.

**Return type**
*Dataset*

**See also:**

*Dataset.head*
Returns the first n rows of the Dataset.

*Dataset.sample*
Returns N randomly selected rows of the Dataset.

**to_arrow**(*, *preserve_fixed_bytes=False*, *empty_strings_to_null=True*)

Convert a riptable *Dataset* to a pyarrow Table.

**Parameters**

- **preserve_fixed_bytes** (*bool, optional, defaults to False*) – For FastArray columns which are ASCII string arrays (dtype.kind == 'S'), set this parameter to True to produce a fixed-length binary array instead of a variable-length string array.
- **empty_strings_to_null** (*bool, optional, defaults To True*) – For FastArray columns which are ASCII or Unicode string arrays, specify True for this parameter to convert empty strings to nulls in the output. riptable inconsistently recognizes the empty string as an 'invalid', so this parameter allows the caller to specify which interpretation they want.

**Return type**
pyarrow.Table

### Notes

**TODO: Maybe add a destroy bool parameter here to indicate the original arrays should be deleted** immediately after being converted to a pyarrow array? We'd need to handle the case where the pyarrow array object was created in "zero-copy" style and wraps our original array (vs. a new array having been allocated via pyarrow); in that case, it won't be safe to delete the original array. Or, maybe we just call 'del' anyway to decrement the object's refcount so it can be cleaned up sooner (if possible) vs. waiting for this whole method to complete and the GC and riptable "Recycler" to run?

**to_pandas**(*unicode=True*, *use_nullable=True*)

Create a pandas DataFrame from this riptable.Dataset. Will attempt to preserve single-key categoricals, otherwise will appear as an index array. Any byte strings will be converted to unicode unless unicode=False.

**Parameters**

- **unicode** (*bool*) – Set to False to keep byte strings as byte strings. Defaults to True.
- **use_nullable** (*bool*) – Whether to use pandas nullable integer dtype for integer columns (default: True).

**Return type**
pandas.DataFrame

**Raises**
NotImplementedError – If a CategoryMode is not handled for a given column.

### Notes

As of Pandas v1.1.0 pandas.Categorical does not handle riptable CategoryMode``s for ``Dictionary, MultiKey, nor IntEnum. Converting a Categorical of these category modes will result in loss of information and emit a warning. Although the column values will be respected, the underlying category codes will be remapped as a single key categorical.

**See also:**

riptable.Dataset.from_pandas

**tolist**()

Return list of lists of values, by rows.

**Return type**
list of lists.

**transpose**(*colnames=None*, *cats=False*, *gb=False*, *headername='Col'*)

Return a transposed version of the Dataset.

**Parameters**

- **colnames** (*list of str, optional*) – Set to list of colnames you want transposed; defaults to None, which means all columns are included.
- **cats** (*bool*) – Set to True to include Categoricals in transposition. Defaults to False.
- **gb** (*bool*) – Set to True to include groupby keys (labels) in transposition. Defaults to False.
- **headername** (*str*) – The name of the column which was once all the column names. Defaults to 'Col'.

**Returns**
A transposed version of this Dataset instance.

**Return type**
*Dataset*

**trim**(*func=None*, *zeros=True*, *nans=True*, *columns=True*, *rows=True*, *keep=False*, *ret_filters=False*)

Returns a Dataset with columns and/or rows removed that contain all zeros and/or nans. Whether to remove only zeros, only nans, or both zeros and nans is controlled by kwargs zeros and nans.

If columns is True (the default), any columns which are all zeros and/or nans will be removed.

If rows is True (the default), any rows which are all zeros and/or nans will be removed.

If func is set, it will bypass the zeros and nan check and instead call func.

- Any column that contains all True after calling func will be removed.
- Any row that contains all True after calling func will be removed if rows is True.

**Parameters**

- **func** – A function which inputs an array and returns a boolean mask.
- **zeros** (*bool*) – Defaults to True. Values must be non-zero.
- **nans** (*bool*) – Defaults to True. Values cannot be nan.

- **columns** (*bool*) – Defaults to True. Reduce columns if entire column filtered.
- **rows** (*bool*) – Defaults to True. Reduce rows if entire row filtered.
- **keep** (*bool*) – Defaults to False. When set to True, does the opposite.
- **ret_filters** (*bool*) – If True, return row and column filters based on the comparisons

**Return type**
*Dataset* or (*Dataset*, row_filter, col_filter)

#### Example

```
>>> ds = rt.Dataset({'a': rt.arange(3), 'b': rt.arange(3.0)})
>>> ds.trim()
#   a      b
-   -   ----
0   1   1.00
1   2   2.00
```

```
>>> ds.trim(lambda x: x > 1)
#   a      b
-   -   ----
0   0   0.00
1   1   1.00
```

```
>>> ds.trim(isfinite)
Dataset is empty (has no rows).
```

**var**(*axis=0*, *ddof=1*, *as_dataset=True*, *fill_value=None*)
See documentation of `reduce()`

### 2.2.11 riptable.rt_datetime

#### Classes

| | |
|---|---|
| *Date* | Date arrays have an underlying int32 array. The array values are number of days since January 1st. 1970. |
| *DateScalar* | Derived from np.int32 |
| *DateSpan* | DateSpan arrays have an underlying int32 array. The array values are in number of days. |
| *DateSpanScalar* | Derived from np.int32 |
| *DateTimeBase* | Base class for DateTimeNano and TimeSpan. |
| *DateTimeNano* | Date and timezone-aware time information, stored to nanosecond precision. |
| *DateTimeNanoScalar* | Derived from np.int64 |
| *TimeSpan* | Array of time delta in nanoseconds, held in float64. |
| *TimeSpanScalar* | Derived from np.float64 |

### Functions

| | |
|---|---|
| *DateTimeUTC*(arr[, to_tz, from_matlab, format, ...]) | Forces DateTimeNano `from_tz` keyword to 'UTC'. |
| *datestring_to_nano*(datestring[, time, from_tz, to_tz]) | Converts date string to DateTimeNano object (default midnight). |
| *datetimestring_to_nano*(dtstring[, from_tz, to_tz]) | Converts datetime string to DateTimeNano object. |
| *parse_epoch*(etime[, to_tz]) | Days since epoch and milliseconds since midnight from nanosecond timestamps. |
| *strptime_to_nano*(dtstrings, format[, from_tz, to_tz]) | Converts datetime string to DateTimeNano object with user-specified format. |
| *timestring_to_nano*(timestring[, date, from_tz, to_tz]) | Converts timestring to TimeSpan or DateTimeNano object. |

**class** riptable.rt_datetime.**Date**(*arr*, *from_matlab=False*, *format=None*)

Bases: `DateBase`, `TimeStampBase`

Date arrays have an underlying int32 array. The array values are number of days since January 1st. 1970. Can be initialized from integer date values, strings, or matlab ordinal dates.

**Parameters**

- **arr** (*array, categorical, list, or scalar*) –
- **from_matlab** (*indicates that values are from matlab datenum*) –
- **format** (*if initialized with string, specify a format string for strptime to parse date information*) – otherwise, will assume format is YYYYMMDD

#### Examples

From strings:

```
>>> datestrings = tile(np.array(['2018-02-01', '2018-03-01', '2018-04-01']), 3)
>>> Date(datestrings)
Date([2018-02-01, 2018-03-01, 2018-04-01, 2018-02-01, 2018-03-01, 2018-04-01, 2018-
↪02-01, 2018-03-01, 2018-04-01])
```

From riptable.Categorical (sometimes Matlab data comes in this way):

```
>>> c = Categorical(datestrings)
>>> c
Categorical([2018-02-01, 2018-03-01, 2018-04-01, 2018-02-01, 2018-03-01, 2018-04-01,
↪ 2018-02-01, 2018-03-01, 2018-04-01]) Length: 9
  FastArray([1, 2, 3, 1, 2, 3, 1, 2, 3], dtype=int8) Base Index: 1
  FastArray(['2018-02-01', '2018-03-01', '2018-04-01'], dtype='<U10') Unique count:␣
↪3
>>> d = Date(c)
>>> d
Date([2018-02-01, 2018-03-01, 2018-04-01, 2018-02-01, 2018-03-01, 2018-04-01, 2018-
↪02-01, 2018-03-01, 2018-04-01])
```

From Matlab datenum:

```
>>> d = FA([737061.0, 737062.0, 737063.0, 737064.0, 737065.0])
>>> Date(dates, from_matlab=True)
Date([2018-01-01, 2018-01-02, 2018-01-03, 2018-01-04, 2018-01-05])
```

From riptable DateTimeNano:

```
>>> dtn = DateTimeNano.random(5)
>>> dtn
DateTimeNano([20150318 13:28:01.853344227, 20150814 17:34:43.991344669, 19761204 04:
↪30:52.680683459, 20120524 06:44:13.482424912, 19830803 17:12:54.771824294])
>>> Date(dtn)
Date([2015-03-18, 2015-08-14, 1976-12-04, 2012-05-24, 1983-08-03])
```

**property _year_splits**

Midnght on Jan. 1st from 1970 - 2099 in utc nanoseconds.

**property _yearday_splits**

Midnight on the 1st of the month in dayssince the beginning of the year.

**property _yearday_splits_leap**

Midnight on the 1st of the month in days since the beginning of the year during a leap year.

**property day_of_month**

The day of the month of each *Date* element.

Days are represented as integers: 1 = Jan 1, 31 = Jan 31, etc.

NaN or invalid *Date* values return Riptable's int32 sentinel value (-MAXINT).

**Returns**

A FastArray of integers representing the day of the month of each *Date* element.

**Return type**

FastArray

**See also:**

*Date.day_of_year*, *Date.day_of_week*, *Date.year*, *Date.month*, *Date.monthyear*

### Examples

```
>>> d = rt.Date(['2019-01-01', '2020-02-29', '2021-12-31'])
>>> d.day_of_month
FastArray([ 1, 29, 31])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.day_of_month
FastArray([-2147483648, -2147483648,          31])
```

**property day_of_week**

The day of the week of each *Date* element.

Days are represented as integers: 0 = Monday, 1 = Tuesday, . . . , 6 = Sunday.

NaN or invalid *Date* values return Riptable's int32 sentinel value (-MAXINT).

**Returns**
A `FastArray` of integers representing the day of the week of each *Date* element.

**Return type**
`FastArray`

**See also:**

*Date.day_of_year*, *Date.day_of_month*, *Date.year*, *Date.month*, *Date.monthyear*

### Examples

```
>>> d = rt.Date(['2019-02-11', '2019-02-12', '2019-02-13',
...              '2019-02-14', '2019-02-15', '2019-02-16', '2019-02-17'])
>>> d.day_of_week
FastArray([0, 1, 2, 3, 4, 5, 6])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.day_of_week
FastArray([-2147483648, -2147483648,           2,           3,
            4,           5,           6])
```

**property day_of_year**

The day of the year of each *Date* element.

Days are represented as integers: 1 = Jan 1, 32 = Feb 1, etc.

NaN or invalid *Date* values return Riptable's int32 sentinel value (-MAXINT).

**Returns**
A `FastArray` of integers representing the day of the year of each *Date* element.

**Return type**
`FastArray`

**See also:**

*Date.day_of_month*, *Date.day_of_week*, *Date.year*, *Date.month*, *Date.monthyear*

### Examples

```
>>> d = rt.Date(['2019-01-01', '2020-02-29', '2021-12-31'])
>>> d.day_of_year
FastArray([  1,  60, 365])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.day_of_year
FastArray([-2147483648, -2147483648,         365])
```

**property is_leapyear**

Return a boolean array that's True for each *Date* element that's in a leap year, False otherwise.

NaN or invalid *Date* values return False.

**Returns**
A FastArray of booleans that's True for each *Date* element that's in a leap year, False otherwise.

**Return type**
FastArray

**See also:**

*Date.is_weekend*, *Date.is_weekday*

### Examples

```
>>> d = rt.Date(['1996-01-01', '2000-01-01', '2004-01-01', '2022-01-01'])
>>> d.is_leapyear
FastArray([ True,  True,  True, False])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.is_leapyear
FastArray([False, False,  True, False])
```

**property is_weekday**

Return a boolean array that's True for each *Date* element that's a weekday (Monday-Friday), False otherwise.

NaN or invalid *Date* values return False.

**Returns**
A FastArray of booleans that's True for each *Date* element that's a weekday, False otherwise.

**Return type**
FastArray

**See also:**

*Date.is_weekend*, *Date.is_leapyear*

### Examples

```
>>> d = rt.Date(['2019-02-11', '2019-02-12', '2019-02-13',
...              '2019-02-14', '2019-02-15', '2019-02-16', '2019-02-17'])
>>> d.is_weekday
FastArray([ True,  True,  True,  True,  True, False, False])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.is_weekday
FastArray([False, False,  True,  True,  True, False, False])
```

**property is_weekend**

Return a boolean array that's True for each *Date* element that's a Saturday or Sunday, False otherwise.

NaN or invalid *Date* values return False.

**Returns**
A FastArray of booleans that's True for each *Date* element that's a Saturday or Sunday, False otherwise.

**Return type**
FastArray

**See also:**

*Date.is_weekday*, *Date.is_leapyear*

### Examples

```
>>> d = rt.Date(['2019-02-09', '2019-02-10', '2019-02-11', '2019-02-12',
...              '2019-02-13', '2019-02-14', '2019-02-15', '2019-02-16', '2019-
↪02-17'])
>>> d.is_weekend
FastArray([ True,  True, False, False, False, False, False,  True,  True])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.is_weekend
FastArray([False, False, False, False, False, False, False,  True,  True])
```

**property month**

The month of each *Date* element.

Months are represented as integers: 1 = Jan, 2 = Feb, etc.

NaN or invalid *Date* values return Riptable's int32 sentinel value (-MAXINT).

**Returns**
A FastArray of integers representing the month of each *Date* element.

**Return type**
FastArray

**See also:**

*Date.monthyear*, *Date.year*, *Date.day_of_year*, *Date.day_of_month*, *Date.day_of_week*

### Examples

```
>>> d = rt.Date(['2016-02-01', '2017-03-01', '2018-04-01'])
>>> d.month
FastArray([2, 3, 4])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.month
FastArray([-2147483648, -2147483648,           4])
```

**property monthyear**

The month and year of each *Date* element.

Each month-year value is a byte string with a three-letter month abbreviation concatenated with a four-digit year.

NaN or invalid *Date* values return Riptable's int32 sentinel value (-MAXINT) as a byte string.

**Returns**
A `FastArray` of byte strings containing the month and year of each *Date* element.

**Return type**
`FastArray`

**See also:**

*Date.year*, *Date.month*, *Date.day_of_year*, *Date.day_of_month*, *Date.day_of_week*

### Examples

```
>>> d = rt.Date(['2000-02-29', '2018-12-25', '2019-03-18'])
>>> d.monthyear
FastArray([b'Feb2000', b'Dec2018', b'Mar2019'], dtype='|S14')
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.monthyear
FastArray([b'-2147483648', b'-2147483648', b'Mar2019'], dtype='|S14')
```

**property seconds_since_epoch**

Many existing python datetime routines expect seconds since epoch. This call is to eliminate "magic numbers" like 3600 from code.

**property start_of_month**

rtype: rt.Date array of first of self's month

**property start_of_week**

rtype: rt.Date array of previous Monday

**property year**

The year of each *Date* element.

Years are currently limited to 1970-2099. To expand the range, add to the UTC_1970_DAY_SPLITS table.

NaN or invalid *Date* values return Riptable's int32 sentinel value (-MAXINT).

**Returns**
A FastArray of integers representing the year of each *Date* element.

**Return type**
FastArray

**See also:**

*Date.month*, *Date.monthyear*, *Date.day_of_year*, *Date.day_of_month*, *Date.day_of_week*

### Examples

```
>>> d = rt.Date(['2016-02-01', '2017-02-01', '2018-02-01'])
>>> d.year
FastArray([2016, 2017, 2018])
```

With NaN and invalid values:

```
>>> d[0] = 0
>>> d[1] = d.inv
>>> d.year
FastArray([-2147483648, -2147483648,        2018])
```

**property yyyymmdd**

**MetaDefault**

**MetaVersion = 1**

**forbidden_mathops = ('__mul__', '__imul__')**

**abstract __abs__()**

**__add__**(*value*)

**Addition rules**

Date + Date = TypeError Date + DateTimeNano = TypeError Date + DateSpan = Date Date + TimeSpan = DateTimeNano

All other operands will be treated as DateSpan and return Date.

**abstract __and__**(*other*)

**__arrow_array__**(*type=None*)

Implementation of the __arrow_array__ protocol for conversion to a pyarrow array.

**Parameters**
**type** (*pyarrow.DataType, optional, defaults to None*) –

**Return type**
pyarrow.Array or pyarrow.ChunkedArray

### Notes

https://arrow.apache.org/docs/python/extending_types.html#controlling-conversion-to-pyarrow-array-with-the-arrow-array

**abstract __ceil__**()

**abstract __complex__**()

**__eq__**(*other*)
Return self==value.

**abstract __float__**()

**abstract __floor__**()

**__ge__**(*other*)
Return self>=value.

**__gt__**(*other*)
Return self>value.

**__iadd__**(*value*)

**abstract __iand__**(*other*)

**abstract __ifloordiv__**(*other*)

**abstract __ilshift__**(*other*)

**abstract __imatmul__**(*other*)

**abstract __imod__**(*other*)

**abstract __imul__**(*other*)

**abstract __int__**()

**abstract __invert__**()

**abstract __ior__**(*other*)

**abstract __ipow__**(*other*, *modulo=None*)

**abstract __irshift__**(*other*)

**__isub__**(*value*)

**abstract __itruediv__**(*other*)

**abstract __ixor__**(*other*)

**__le__**(*other*)
Return self<=value.

**abstract __lshift__**(*other*)

**__lt__**(*other*)
Return self<value.

**abstract __matmul__**(*other*)

**abstract __mul__**(*other*)

**__ne__**(*other*)

Return self!=value.

**abstract __neg__**()

**abstract __or__**(*other*)

**abstract __pos__**()

**abstract __pow__**(*other*, *modulo=None*)

**__radd__**(*value*)

**abstract __rand__**(*other*)

**abstract __rdivmod__**(*other*)

**abstract __rfloordiv__**(*other*)

**abstract __rlshift__**(*other*)

**abstract __rmatmul__**(*other*)

**abstract __rmod__**(*other*)

**abstract __rmul__**(*other*)

**abstract __ror__**(*other*)

**abstract __round__**(*ndigits=0*)

**abstract __rpow__**(*other*)

**abstract __rrshift__**(*other*)

**abstract __rshift__**(*other*)

**__rsub__**(*value*)

**abstract __rtruediv__**(*other*)

**abstract __rxor__**(*other*)

**__sub__**(*value*)

**Subtraction rules**

Date - Date = DateSpan Date - DateSpan = Date Date - DateTimeNano = TimeSpan Date - TimeSpan = DateTimeNano

All other operands will be treated as DateSpan and return Date.

**abstract __trunc__**()

**abstract __xor__**(*other*)

**_check_mathops**(*funcname*, *value*)

This gets called after a math operation has been performed on the Date's FastArray. Return type may differ based on operation. Preserves invalids from original input.

**Parameters**

- **funcname** (*name of ufunc*) –
- **value** (*original operand in math operation*) –
- **return_type** (*returns*) –
- **other_inv_mask** –

**_check_mathops_nano**(*funcname*, *value*, *other_inv_mask*, *return_type*, *caller*)

Operations with TimeSpan and DateTimeNano will flip to nano precision, or raise an error.

**Parameters**

- **funcname** (*name of ufunc*) –
- **value** (*original operand in math operation*) –
- **other_inv_mask** (*None, might be set in this routine*) –
- **return_type** (*None, might be set to TimeSpan or* DateTimeNano) –
- **caller** (*FastArray view of Date object.*) –

**classmethod _convert_datestring**(*arr*, *format=None*)

For construction from array of strings or categorical.

**classmethod _convert_matlab_days**(*arr*)

TODO: move this to a more generic superclass - almost exactly the same as DateTimeNano._convert_matlab_days

**Parameters**

- **arr** (*array of matlab datenums (1 is 1-Jan-0000)*) –
- **timezone** (*TimeZone object from DateTimeNano constructor*) –

Converts matlab datenums to an array of int64 containing utc nanoseconds.

**_date_compare_check**(*funcname*, *other*)

Funnel for all comparison operations. Helps Date interact with DateTimeNano, TimeSpan.

**static _from_arrow**(*arr*, *zero_copy_only=True*, *writable=False*)

Create a *Date* instance from a "date32" or "date64"-typed pyarrow.Array.

**Parameters**

- **arr** (*pyarrow.Array or pyarrow.ChunkedArray*) – Must be a "date32"- or "date64"-typed pyarrow array.
- **zero_copy_only** (*bool, optional, defaults to False*) –
- **writable** (*bool, optional, defaults to False*) –

**Return type**
*Date*

**classmethod _load_from_sds_meta_data**(*name*, *arr*, *cols*, *meta*)

Restore Date class after loading from .sds file.

**static _parse_item_format**(*itemformat*)

**diff**(*periods=1*)

**Return type**
*DateSpan*

**static display_convert_func**(*date_num*, *itemformat*)

**fill_invalid**(*shape=None*, *dtype=None*, *inplace=True*)

Replace all values of the input `FastArray` with an invalid value.

The invalid value used is determined by the input array's dtype or a user-specified dtype.

Warning: By default, this operation is in place.

**Parameters**

- **shape** (*int or sequence of int, optional*) – Shape of the new array, for example: `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str, optional*) – The desired dtype for the returned array.
- **inplace** (*bool, default True*) – If `True` (the default), modify original data. If `False`, return a copy of the array.

**Returns**
If `inplace=False`, a copy of the input `FastArray` is returned that has all values replaced with an invalid value. Otherwise, nothing is returned.

**Return type**
*FastArray*, optional

**See also:**

**FastArray.inv**
Return the invalid value for the input array's dtype.

**FastArray.copy_invalid**
Return a copy of a `FastArray` filled with the invalid value for the array's dtype.

### Examples

Replace an integer array's values with the invalid value for the array's dtype. By default, the returned array is the same size and dtype as the input array, and the operation is performed in place:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a.fill_invalid()
>>> a
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Replace a floating-point array's values with the invalid value for the `int32` dtype:

```
>>> a2 = rt.FA([0., 1., 2., 3., 4.])
>>> a2
FastArray([0., 1., 2., 3., 4.])
>>> a2.fill_invalid(dtype="int32", inplace=False)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Specify the size and dtype of the output array:

```
>>> a3 = rt.FA(["AMZN", "IBM", "MSFT", "AAPL"])
>>> a3
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a3.fill_invalid(2, dtype="bool", inplace=False)
FastArray([False, False])
```

**static format_date_num**(*date_num*, *itemformat*)

**get_classname**()

**get_scalar**(*scalarval*)

**classmethod hstack**(*dates*)

hstacks Date objects and returns a new Date object. Will be called by riptable.hstack() if the first item in the sequence is a Date object.

**Parameters**

**dates** (*list or tuple of Date objects*) –

```
>>> d1 = Date('2015-02-01')
>>> d2 = Date(['2016-02-01', '2017-02-01', '2018-02-01'])
>>> hstack([d1, d2])
Date([2015-02-01, 2016-02-01, 2017-02-01, 2018-02-01])
```

**isfinite**()

Return a boolean array that's True for each *Date* element that's not a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int32 sentinel value are considered to be NaN.

**Returns**

A FastArray of booleans that's True for each non-NaN element, False otherwise.

**Return type**

FastArray

**See also:**

*Date.isnan*, *DateTimeNano.isnan*, *DateTimeNano.isnotnan*, riptable.isnan, riptable.isnotnan, riptable.isnanorzero, FastArray.isnan, FastArray.isnotnan, FastArray.notna, FastArray.isnanorzero, Categorical.isnan, Categorical.isnotnan, Categorical.notna

**Dataset.mask_or_isnan**

Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**

Return a boolean array that's True for each all-NaN Dataset row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> d = rt.Date.range('20190201', days = 3, step = 2)
>>> d[0] = 0
>>> d[1] = d.inv
>>> d
Date(['Inv', 'Inv', '2019-02-05'])
>>> d.isfinite()
FastArray([False, False,  True])
```

**isnan**()

Return a boolean array that's True for each *Date* element that's a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int32 sentinel value are considered to be NaN.

**Returns**
A `FastArray` of booleans that's True for each NaN element, False otherwise.

**Return type**
`FastArray`

**See also:**

*Date.isnotnan*, *DateTimeNano.isnan*, *DateTimeNano.isnotnan*, `riptable.isnan`, `riptable.isnotnan`, `riptable.isnanorzero`, `FastArray.isnan`, `FastArray.isnotnan`, `FastArray.notna`, `FastArray.isnanorzero`, `Categorical.isnan`, `Categorical.isnotnan`, `Categorical.notna`

**Dataset.mask_or_isnan**
Return a boolean array that's True for each `Dataset` row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN `Dataset` row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> d = rt.Date.range('20190201', days = 3, step = 2)
>>> d[0] = 0
>>> d[1] = d.inv
>>> d
Date(['Inv', 'Inv', '2019-02-05'])
>>> d.isnan()
FastArray([ True,  True, False])
```

**isnotfinite**()

Return a boolean array that's True for each *Date* element that's a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int32 sentinel value are considered to be NaN.

**Returns**
A `FastArray` of booleans that's True for each NaN element, False otherwise.

**Return type**
`FastArray`

**See also:**

*Date.isnotnan*, *DateTimeNano.isnan*, *DateTimeNano.isnotnan*, `riptable.isnan`, `riptable.isnotnan`, `riptable.isnanorzero`, `FastArray.isnan`, `FastArray.isnotnan`, `FastArray.notna`, `FastArray.isnanorzero`, `Categorical.isnan`, `Categorical.isnotnan`, `Categorical.notna`

**Dataset.mask_or_isnan**
Return a boolean array that's True for each `Dataset` row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN `Dataset` row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> d = rt.Date.range('20190201', days = 3, step = 2)
>>> d[0] = 0
>>> d[1] = d.inv
>>> d
Date(['Inv', 'Inv', '2019-02-05'])
>>> d.isnotfinite()
FastArray([ True,  True, False])
```

**isnotnan()**

Return a boolean array that's True for each *Date* element that's not a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int32 sentinel value are considered to be NaN.

**Returns**
A `FastArray` of booleans that's True for each non-NaN element, False otherwise.

**Return type**
`FastArray`

**See also:**

*Date.isnan*, *DateTimeNano.isnan*, *DateTimeNano.isnotnan*, `riptable.isnan`, `riptable.isnotnan`, `riptable.isnanorzero`, `FastArray.isnan`, `FastArray.isnotnan`, `FastArray.notna`, `FastArray.isnanorzero`, `Categorical.isnan`, `Categorical.isnotnan`, `Categorical.notna`

**Dataset.mask_or_isnan**
Return a boolean array that's True for each `Dataset` row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN `Dataset` row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> d = rt.Date.range('20190201', days = 3, step = 2)
>>> d[0] = 0
>>> d[1] = d.inv
>>> d
Date(['Inv', 'Inv', '2019-02-05'])
>>> d.isnotnan()
FastArray([False, False,  True])
```

**classmethod range**(*start*, *end=None*, *days=None*, *step=1*, *format=None*, *closed=None*)

Return a *Date* object of dates within a given interval, spaced by `step`.

Note: Either `end` or `days` must be provided, but providing both results in unexpected behavior. In future versions, an error will be raised.

**Parameters**

- **start** (*int or str*) – Start date as an integer (YYYYMMDD) or string. If the string is not in 'YYYYMMDD' format, `format` is required.
- **end** (*int or str, optional*) – End date as an integer (YYYYMMDD) or string. If the string is not in 'YYYYMMDD' format, `format` is required. If `end` is not provided, the number of dates to generate must be specified with `days`.
- **days** (*int, optional*) – Instead of using `end`, use `days` to specify the number of dates to generate. Required if `end` isn't provided. Providing both `end` and `days` results in unexpected behavior.
- **step** (*int, default 1*) – The number of days between generated dates.
- **format** (*str, optional*) – For a string `start` or `end` value, one or more format codes supported by the `datetime.strptime()` function of the standard Python distribution. For codes, see strftime() and strptime() Behavior. The format code is used to parse the string representation and convert it to a *Date* element.
- **closed** (*{None, 'left', 'right'}, default None*) – Determines whether the `start` and `end` dates are included in the result. Applies only when `start` and `end` are specified and `step=1`.
  - `left`: Start date is included, end date is excluded.
  - `right`: End date is included, start date is excluded.
  - None (the default): Both the start and end dates are included.

**Returns**

A *Date* object of dates within a given interval, spaced by `step`.

**Return type**

*Date*

**See also:**

*DateTimeNano.random*

Return an array of randomly generated *DateTimeNano* values.

**riptable.arange**
Return an array of evenly spaced values within a specified interval.

### Examples

With integer `start` and `end` dates:

```
>>> rt.Date.range(20230101, 20230105)
Date(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05'])
```

With string `start` and `end` dates, and a format code:

```
>>> rt.Date.range('01 January, 2023', '05 January, 2023', format='%d %B, %Y')
Date(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05'])
```

If `end` isn't specified, `days` is required:

```
>>> rt.Date.range(20230101, days=5)
Date(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05'])
```

Changing the `step`:

```
>>> rt.Date.range(20230101, 20230105, step=2)
Date(['2023-01-01', '2023-01-03'])
```

A left-inclusive, right-exclusive range:

```
>>> rt.Date.range(20230101, 20230105, closed='left')
Date(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04'])
```

**strftime**(*format*, *dtype='O'*)

Convert each *Date* element to a formatted string representation.

**Parameters**

- **format** (*str*) – One or more format codes supported by the `datetime.date.strftime()` function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.
- **dtype** (*{"O", "S", "U"}, default "O"*) – The data type of the returned array elements:
  - "O": object string
  - "S": byte string
  - "U": unicode string

**Returns**
An `ndarray` of strings.

**Return type**
ndarray

**See also:**

*DateScalar.strftime*, *DateTimeNano.strftime*, *DateTimeNanoScalar.strftime*, *TimeSpan.strftime*, *TimeSpanScalar.strftime*

### Notes

This routine has not been sped up yet. It's also not NaN-aware: NaNs are converted to the timestamp of the epoch (01-01-1970), then formatted.

### Examples

```
>>> d = rt.Date(['20210101', '20210519', '20220308'])
>>> d.strftime('%D')
array(['01/01/21', '05/19/21', '03/08/22'], dtype=object)
```

**to_arrow**(*type=None*, ***, *preserve_fixed_bytes=False*, *empty_strings_to_null=True*)

Convert this *Date* to a pyarrow.Array.

**Parameters**

- **type** (*pyarrow.DataType, optional, defaults to None*) – Unused.
- **preserve_fixed_bytes** (*bool, optional, defaults to False*) – Unused.
- **empty_strings_to_null** (*bool, optional, defaults To True*) – Unused.

**Return type**

pyarrow.Array or pyarrow.ChunkedArray

**class** riptable.rt_datetime.**DateScalar**(***kwargs*)

Bases: numpy.int32

Derived from np.int32 days since unix epoch in 1970 TODO: need to inherit math functions

**property _fa**

**property _np**

**__slots__ = '_display_length'**

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**get_classname**()

**get_item_format**()

**repeat**(*repeats*, *axis=None*)

**strftime**(*format*)

Convert a *DateScalar* to a formatted string representation.

**Parameters**

**format** (*str*) – One or more format codes supported by the datetime.date.strftime() function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.

**Returns**

A string representation of the reformatted *DateScalar*.

**Return type**
str

**See also:**

*Date.strftime*, *DateTimeNano.strftime*, *DateTimeNanoScalar.strftime*, *TimeSpan.strftime*, *TimeSpanScalar.strftime*

### Notes

This routine has not been sped up yet. It also raises an error on NaNs.

### Examples

```
>>> d = rt.Date(['20210101', '20210519', '20220308'])
>>> d[0].strftime('%D')
'01/01/21'
```

**tile**(*repeats*)

**class** riptable.rt_datetime.**DateSpan**(*arr*, *unit=None*)

Bases: DateBase

DateSpan arrays have an underlying int32 array. The array values are in number of days. These are created as the result of certain math operations on Date objects.

**Parameters**

- **arr** (*numeric array, list, or scalar*) –
- **unit** (*can set units to 'd' (day) or 'w' (week)*) –

**property format_long**

**property format_short**

**MetaDefault**

**MetaVersion = 1**

**NAN_DATE**

**forbidden_mathops = ()**

**__add__**(*value*)

**__eq__**(*other*)
Return self==value.

**__ge__**(*other*)
Return self>=value.

**__gt__**(*other*)
Return self>value.

**__iadd__**(*value*)

**__isub__**(*value*)

__le__(*other*)

Return self<=value.

__lt__(*other*)

Return self<value.

__ne__(*other*)

Return self!=value.

__sub__(*value*)

_check_mathops(*funcname*, *value*)

This gets called after a math operation has been performed on the Date's FastArray. Return type may differ based on operation. Preserves invalids from original input.

**Parameters**

- **funcname** (*name of ufunc*) –
- **value** (*original operand in math operation*) –
- **return_type** (*returns*) –
- **other_inv_mask** –

_check_mathops_nano(*funcname*, *value*, *other_inv_mask*, *return_type*, *caller*)

Operations with TimeSpan and DateTimeNano will flip to nano precision, or raise an error.

**Parameters**

- **funcname** (*name of ufunc*) –
- **value** (*original operand in math operation*) –
- **other_inv_mask** (*None, might be set in this routine*) –
- **return_type** (*None, might be set to TimeSpan or* DateTimeNano) –
- **caller** (*FastArray view of Date object.*) –

_datespan_compare_check(*funcname*, *other*)

Funnel for all comparison operations. Helps Date interact with DateTimeNano, TimeSpan.

classmethod _load_from_sds_meta_data(*name*, *arr*, *cols*, *meta*)

Restore Date class after loading from .sds file.

static display_convert_func(*date_num*, *itemformat*)

Called by main rt_display() routine to format items in array correctly in Dataset display. Also called by DateSpan's __str__() and __repr__().

fill_invalid(*shape=None*, *dtype=None*, *inplace=True*)

Replace all values of the input `FastArray` with an invalid value.

The invalid value used is determined by the input array's dtype or a user-specified dtype.

Warning: By default, this operation is in place.

**Parameters**

- **shape** (*int or sequence of int, optional*) – Shape of the new array, for example: `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str, optional*) – The desired dtype for the returned array.

- **inplace** (*bool, default True*) – If `True` (the default), modify original data. If `False`, return a copy of the array.

**Returns**

If `inplace=False`, a copy of the input `FastArray` is returned that has all values replaced with an invalid value. Otherwise, nothing is returned.

**Return type**

*FastArray*, optional

**See also:**

**`FastArray.inv`**
Return the invalid value for the input array's dtype.

**`FastArray.copy_invalid`**
Return a copy of a `FastArray` filled with the invalid value for the array's dtype.

### Examples

Replace an integer array's values with the invalid value for the array's dtype. By default, the returned array is the same size and dtype as the input array, and the operation is performed in place:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a.fill_invalid()
>>> a
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Replace a floating-point array's values with the invalid value for the `int32` dtype:

```
>>> a2 = rt.FA([0., 1., 2., 3., 4.])
>>> a2
FastArray([0., 1., 2., 3., 4.])
>>> a2.fill_invalid(dtype="int32", inplace=False)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Specify the size and dtype of the output array:

```
>>> a3 = rt.FA(["AMZN", "IBM", "MSFT", "AAPL"])
>>> a3
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a3.fill_invalid(2, dtype="bool", inplace=False)
FastArray([False, False])
```

**`static format_date_span`**(*date_span*, *itemformat*)

Turn a single value in the DateSpan array into a string for display.

**`get_classname`**()

**`get_scalar`**(*scalarval*)

**classmethod hstack**(*dates*)

hstacks DateSpan objects and returns a new DateSpan object. Will be called by riptable.hstack() if the first item in the sequence is a DateSpan object.

**Parameters**

**dates** (*list or tuple of DateSpan objects*) –

```
>>> d1 = Date('2015-02-01')
>>> d2 = Date(['2016-02-01', '2017-02-01', '2018-02-01'])
>>> hstack([d1, d2])
Date([2015-02-01, 2016-02-01, 2017-02-01, 2018-02-01])
```

**strftime**(*format*, *dtype='O'*)

Convert each *DateSpan* element to a formatted string representation.

Deprecated since version 1.3: *DateSpan.strftime* is deprecated will be removed in the future.

Note that because each *DateSpan* element is converted to a timestamp relative to the epoch before it is formatted (for example, a *DateSpan* of "2 days" is converted to 01-03-1970), you may need to adjust the data before calling this method.

Negative *DateSpan* values (for example, "-10 days") can't be formatted with this method.

**Parameters**

- **format** (*str*) – One or more format codes supported by the `datetime.date.strftime()` function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.
- **dtype** (*{"O", "S", "U"}, default "O"*) – The data type of the returned array elements.
  - "O": object string
  - "S": byte string
  - "U": unicode string

**Returns**

An `ndarray` of strings.

**Return type**

`ndarray`

**See also:**

*DateScalar.strftime*, *Date.strftime*, *DateSpan.strftime*, *DateTimeNano.strftime*, *DateTimeNanoScalar.strftime*, *TimeSpan.strftime*, *TimeSpanScalar.strftime*

### Notes

This routine has not been sped up yet. It's also not NaN-aware: NaNs are converted to the timestamp of the epoch (01-01-1970), then formatted.

### Examples

```
>>> d = rt.Date(['20210101', '20210519', '20220308'])
>>> ds = d - rt.Date('20201201')
>>> ds
DateSpan(['31 days', '169 days', '462 days'])
>>> ds.strftime('%D')
array(['02/01/70', '06/19/70', '04/08/71'], dtype=object)
```

**class** riptable.rt_datetime.**DateSpanScalar**(***kwargs*)

Bases: numpy.int32

Derived from np.int32 Number of days between two dates

**property _fa**

**property _np**

**NAN_DATESPANSCALAR**

**__slots__ = '_display_length'**

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**get_classname**()

**get_item_format**()

**isfinite**()

**isnan**()

**isnotfinite**()

**isnotnan**()

**repeat**(*repeats*, *axis=None*)

**tile**(*repeats*)

**class** riptable.rt_datetime.**DateTimeBase**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: *riptable.rt_fastarray.FastArray*

Base class for DateTimeNano and TimeSpan. Both of these subclasses have times with nanosecond precision.

**property _fa**

**property display_length**

**DEFAULT_FORMATTER**

**NAN_TIME = 0**

**PRECISION = 9**

**__array_finalize__**(*obj*)

Finalizes self from other, called as part of ndarray.__new__()

**__getitem__**(*fld*)

riptable has special routines to handle array input in the indexer. Everything else will go to numpy getitem.

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**static _add_nano_ext**(*utcnano*, *timestr*)

**_as_meta_data**(*name=None*)

**_build_sds_meta_data**(*name*, ***kwargs*)

Build meta data for DateTimeNano

**_build_string**()

**_funnel_mathops**(*funcname*, *value*)

Wrapper for all math operations on Date and DateSpan.

Both subclasses need to take over: _check_mathops_nano() _check_mathops()

maybe… still testing _build_mathops_result()

Easier to catch forbidden operations here.

**_math_error_string**(*value*, *operator*, *reverse=False*)

**abstract _meta_dict**(*name=None*)

**copy**(*order='K'*)

Return a copy of the input `FastArray`.

**Parameters**

**order** (*{'K', 'C', 'F', 'A'}, default 'K'*) – Controls the memory layout of the copy: 'K' means match the layout of the input array as closely as possible; 'C' means row-based (C-style) order; 'F' means column-based (Fortran-style) order; 'A' means 'F' if the input array is formatted as 'F', 'C' if not.

**Returns**

A copy of the input `FastArray`.

**Return type**

*FastArray*

**See also:**

*Categorical.copy*

Return a copy of the input *Categorical*.

*Dataset.copy*

Return a copy of the input *Dataset*.

*Struct.copy*

Return a copy of the input *Struct*.

### Examples

Copy a FastArray:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a2 = a.copy()
>>> a2
FastArray([1, 2, 3, 4, 5])
>>> a2 is a
False  # The copy is a separate object.
```

**abstract display_item**(*utcnano*)

**get_classname**()

**class** riptable.rt_datetime.**DateTimeNano**(*arr*, *from_matlab=False*, *from_tz=None*, *to_tz=None*, *format=None*, *start_date=None*, *gmt=None*)

Bases: *DateTimeBase*, TimeStampBase, DateTimeCommon

Date and timezone-aware time information, stored to nanosecond precision.

*DateTimeNano* arrays have an underlying int64 array representing the number of nanoseconds since the Unix epoch (00:00:00 UTC on 01-01-1970). Dates before the Unix epoch are invalid.

In most cases, *DateTimeNano* objects default to display in Eastern/NYC time, accounting for Daylight Saving Time. The exception is when arr is an array of *Date* objects, in which case the default display timezone is UTC.

**Parameters**

- **arr** (array of int, str, *Date*, *TimeSpan*, datetime, numpy.datetime64) – Datetimes to store in the *DateTimeNano* array.
  - Integers represent nanoseconds since the Unix epoch (00:00:00 UTC on 01-01-1970).
  - Datetime strings can generally be in YYYYMMDD HH:MM:SS.fffffffff format without format codes needing to be specified. Bytestrings, unicode strings, and strings in ISO 8601 format are supported. If your strings are in another format (for example, MMDDYY), specify it with format. Other notes for string input:
    - from_tz is required.
    - If start_date is provided, strings are parsed as *TimeSpan* objects before start_date is applied. See how this affects output in the Examples section below.
    - For NumPy vs. Riptable string parsing differences, see the Notes section below.
  - For *Date* objects, both from_tz and to_tz are "UTC" by default.
  - For *TimeSpan* objects, start_date needs to be specified.
  - Using the *DateTimeNano* constructor is recommended for *Date* + *TimeSpan* operations.
  - numpy.datetime64 values are converted to nanoseconds.
- **from_tz** (*str*) – The timezone the data in arr is stored in. Required if the *DateTimeNano* is created from strings, and recommended in other cases to ensure expected results. The default from_tz is "UTC" for all arr types except strings, for which a from_tz must be specified.

Timezones supported (Daylight Saving Time is accounted for):

- "America/New_York"
- "Australia/Sydney"
- "Europe/Dublin"
- "DUBLIN": alias for "Europe/Dublin"
- "GMT": Greenwich Mean Time
- "NYC": US/Eastern
- "UTC": (not a timezone, but accepted as an alias for GMT)

- **to_tz** (*str*) – The timezone the data is displayed in. If `arr` is *Date* objects, the default `to_tz` is "UTC". For other `arr` types, the default `to_tz` is "NYC".
- **from_matlab** (*bool, default False*) – When set to `True`, indicates that `arr` contains Matlab datenums (the number of days since 0-Jan-0000). Because Matlab datenums may also include a fraction of a day, be sure to specify `from_tz` for accurate time data.
- **format** (*str*) – Specify a format for string `arr` input. For format codes, see the Python strptime cheatsheet. This parameter is ignored for non-string `arr` input.
- **start_date** (`str` or array of *Date*) –
  - Required if constructing a *DateTimeNano* from a *TimeSpan*.
  - If `arr` is strings, the values in `arr` are parsed as *TimeSpan* objects before `start_date` is applied. See how this affects output in the Examples section below. Otherwise, `start_date` is added (as nanos) to dates in `arr`.
  - If `start_date` is a string, use YYYYMMDD format.
  - If `start_date` is a *Date* array, it is broadcast to `arr` if possible; otherwise an error is raised.
  - A `start_date` before the Unix epoch is converted to the Unix epoch.

### Notes

- The constructor does not attempt to preserve NaN times from Python `datetime` objects.
- If the integer data in a *DateTimeNano* object is extracted, it is in the `from_tz` timezone. To initialize another *DateTimeNano* with the same underlying array, use the same `from_tz`.
- *DateTimeNano* objects have no knowledge of timezones. All timezone operations are handled by the `TimeZone` class.

Math Operations

The following math operations can be performed:

| |
|---|
| Date + TimeSpan = DateTimeNano |
| Date - DateTimeNano = TimeSpan |
| Date - TimeSpan = DateTimeNano |
| DateTimeNano - DateTimeNano = TimeSpan |
| DateTimeNano - Date = TimeSpan |
| DateTimeNano - TimeSpan = DateTimeNano |
| DateTimeNano + TimeSpan = DateTimeNano |

String Parsing Differences Between NumPy and Riptable

- Riptable *DateTimeNano* string parsing is generally more forgiving than NumPy's numpy.datetime64 array parsing.
- In some cases where NumPy raises an error, Riptable returns an object.
- The lower limit for *DateTimeNano* string parsing is Unix epoch time.
- You can always guarantee that Riptable and NumPy get the same results by using the full ISO 8601 datetime format (YYYY-MM-DDTHH:MM:SS.ffffffff).

Riptable parses strings without leading zeros:

```
>>> import numpy as np
>>> rt.DateTimeNano(["2018-1-1"], from_tz="NYC")
DateTimeNano(['20180101 00:00:00.000000000'], to_tz='NYC')
>>> np.array(["2018-1-1"], dtype="datetime64[ns]")
ValueError: Error parsing datetime string "2018-1-1" at position 5
```

Riptable handles extra trailing spaces; NumPy incorrectly treats them as a timezone whose parsing will be deprecated soon:

```
>>> rt.DateTimeNano(["2018-10-11 10:11:00.123           "], from_tz="NYC")
DateTimeNano(['20181011 10:11:00.123000000'], to_tz='NYC')
>>> np.array(["2018-10-11 10:11:00.123           "], dtype="datetime64[ns]")
DeprecationWarning: parsing timezone aware datetimes is deprecated; this will
raise an error in the future
array(['2018-10-11T10:11:00.123000000'], dtype='datetime64[ns]')
```

Riptable correctly parses dates without delimiters:

```
>>> rt.DateTimeNano(["20181231"], from_tz="NYC")
DateTimeNano(['20181231 00:00:00.000000000'], to_tz='NYC')
>>> np.array(["20181231"], dtype="datetime64[ns]")
array(['1840-08-31T19:51:12.568664064'], dtype='datetime64[ns]')
```

To ensure that Riptable and NumPy get the same results, use the full ISO 8601 datetime format:

```
>>> rt.DateTimeNano(["2018-12-31T12:34:56.789123456"], from_tz="NYC")
DateTimeNano(['20181231 12:34:56.789123456'], to_tz='NYC')
>>> np.array(["2018-12-31T12:34:56.789123456"], dtype="datetime64[ns]")
array(['2018-12-31T12:34:56.789123456'], dtype='datetime64[ns]')
```

**See also:**

*DateTimeNano.info*
: See timezone info for a *DateTimeNano* object.

*Date*
: Riptable's *Date* class.

*DateSpan*
: Riptable's *DateSpan* class.

*TimeSpan*
: Riptable's *TimeSpan* class.

*TimeZone*
Riptable's *TimeZone* class.

### Examples

Create a *DateTimeNano* from an integer representing the nanoseconds since 00:00:00 UTC on 01-01-1970:

```
>>> rt.DateTimeNano([1514828730123456000], from_tz="UTC")
DateTimeNano(['20180101 12:45:30.123456000'], to_tz='NYC')
```

From a datetime string in NYC time:

```
>>> rt.DateTimeNano(["2018-01-01 12:45:30.123456000"], from_tz="NYC")
DateTimeNano(['20180101 12:45:30.123456000'], to_tz='NYC')
```

From `numpy.datetime64` array (note that NumPy has less precision):

```
>>> dt = np.array(["2018-11-02 09:30:00.002201", "2018-11-02 09:30:00.004212"],␣
↪dtype="datetime64[ns]")
>>> rt.DateTimeNano(dt, from_tz="NYC")
DateTimeNano(['20181102 09:30:00.002201000', '20181102 09:30:00.004212000'], to_tz=
↪'NYC')
```

If your datetime strings are nonstandard, specify the format using `format` with Python strptime codes.

```
>>> rt.DateTimeNano(["12/31/19 08:05:01", "6/30/19 14:20:35"], format="%m/%d/%y %H:
↪%M:%S", from_tz="NYC")
DateTimeNano(['20191231 08:05:01.000000000', '20190630 14:20:35.000000000'], to_tz=
↪'NYC')
```

Convert Matlab datenums:

```
>>> rt.DateTimeNano([737426, 738251.75], from_matlab=True, from_tz="NYC")
DateTimeNano(['20190101 00:00:00.000000000', '20210405 18:00:00.000000000'], to_tz=
↪'NYC')
```

Note that if you create a *DateTimeNano* by adding a *Date* and a *TimeSpan* without using the *DateTimeNano* constructor, `from_tz` and `to_tz` will be "GMT":

```
>>> d = rt.Date("20230305")
>>> ts = rt.TimeSpan("05:00")
>>> dtn = d + ts
>>> dtn.info()
DateTimeNano(['20230305 05:00:00.000000000'], to_tz='GMT')
Displaying in timezone: GMT
Origin: GMT
Offset: 0 hours
```

Create a *DateTimeNano* from a list of Python `datetime` objects:

```
>>> from datetime import datetime as dt
>>> pdt = [dt(2018, 7, 2, 14, 30), dt(2019, 6, 8, 8, 30)]
>>> rt.DateTimeNano(pdt)
UserWarning: FastArray contains an unsupported type 'object'.  Problems may occur.
```

(continues on next page)

(continued from previous page)

```
Consider categoricals.
  warnings.warn(warning_string)
DateTimeNano(['20180702 10:30:00.000000000', '20190608 04:30:00.000000000', to_tz=
↪'NYC')
```

If you specify a `start_date` with an `arr` of strings, the strings are parsed as *TimeSpan* objects before `start_date` is applied. Note the first two examples in `arr` result in NaN TimeSpans, which are silently treated as zeros:

```
>>> arr = ["20180205", "20180205 14:30", "14:30"]
>>> rt.DateTimeNano(arr, from_tz="UTC", to_tz="UTC", start_date="20230601")
DateTimeNano(['20230601 00:00:00.000000000', '20230601 00:00:00.000000000',
↪'20230601 14:30:00.000000000'], to_tz='UTC')
```

*GetNanoTime* gets the current Unix epoch time:

```
>>> rt.DateTimeNano([rt.GetNanoTime()], from_tz="UTC")
DateTimeNano(['20230615 18:36:58.378020700'], to_tz='NYC')
```

**property display_length**

**FrequencyStrings**

**MetaDefault**

**MetaVersion = 0**

**_INVALID_FREQ_ERROR = 'Invalid frequency: {}'**

**__abs__**()

**__add__**(*other*, *inplace=False*)

**abstract __and__**(*other*)

**__arrow_array__**(*type=None*)

Implementation of the `__arrow_array__` protocol for conversion to a pyarrow array.

**Parameters**

**type** (*pyarrow.DataType, optional, defaults to None*) –

**Return type**

pyarrow.Array or pyarrow.ChunkedArray

### Notes

https://arrow.apache.org/docs/python/extending_types.html#controlling-conversion-to-pyarrow-array-with-the-arrow-array

**abstract __ceil__**()

**abstract __complex__**()

**__eq__**(*other*)

Return self==value.

**abstract __float__**()

**abstract __floor__**()

**__floordiv__**(*value*)

**__ge__**(*other*)

Return self>=value.

**__gt__**(*other*)

Return self>value.

**__iadd__**(*other*)

**abstract __iand__**(*other*)

**abstract __ifloordiv__**(*other*)

**abstract __ilshift__**(*other*)

**abstract __imatmul__**(*other*)

**abstract __imod__**(*other*)

**abstract __imul__**(*other*)

**abstract __int__**()

**abstract __invert__**()

**abstract __ior__**(*other*)

**abstract __ipow__**(*other*, *modulo=None*)

**abstract __irshift__**(*other*)

**__isub__**(*other*)

**abstract __itruediv__**(*other*)

**abstract __ixor__**(*other*)

**__le__**(*other*)

Return self<=value.

**abstract __lshift__**(*other*)

**__lt__**(*other*)

Return self<value.

**abstract __matmul__**(*other*)

**__mul__**(*value*)

**__ne__**(*other*)

Return self!=value.

**abstract __neg__**()

**abstract __or__**(*other*)

**abstract __pos__**()

**abstract __pow__**(*other*, *modulo=None*)

**__radd__**(*other*)

**abstract __rand__**(*other*)

**abstract __rdivmod__**(*other*)

**__repr__**(*verbose=False*)

Return repr(self).

**abstract __rfloordiv__**(*other*)

**abstract __rlshift__**(*other*)

**abstract __rmatmul__**(*other*)

**abstract __rmod__**(*other*)

**abstract __rmul__**(*other*)

**abstract __ror__**(*other*)

**abstract __round__**(*ndigits=0*)

**abstract __rpow__**(*other*)

**abstract __rrshift__**(*other*)

**abstract __rshift__**(*other*)

**__rsub__**(*other*)

**abstract __rtruediv__**(*other*)

**abstract __rxor__**(*other*)

**__sub__**(*other*, *inplace=False*)

**__truediv__**(*value*)

**abstract __trunc__**()

**abstract __xor__**(*other*)

**classmethod _convert_matlab_days**(*arr*, *timezone*)

**Parameters**

- **arr** (*array of matlab datenums (1 is 1-Jan-0000)*) –
- **timezone** (*TimeZone object from DateTimeNano constructor*) –

Converts matlab datenums to an array of int64 containing utc nanoseconds.

**_datetimenano_compare_check**(*funcname*, *other*)

**static _from_arrow**(*arr*, *zero_copy_only=True*, *writable=False*)

Create a *DateTimeNano* instance from a "timestamp"-typed pyarrow.Array.

**Parameters**

- **arr** (*pyarrow.Array or pyarrow.ChunkedArray*) – Must be a "timestamp"-typed pyarrow array.
- **zero_copy_only** (*bool, optional, defaults to False*) –
- **writable** (*bool, optional, defaults to False*) –

**Return type**
*DateTimeNano*

**classmethod _from_meta_data**(*arrdict*, *arrflags*, *meta*)

**_guard_math_op**(*value*, *op_name*)

**classmethod _load_from_sds_meta_data**(*name*, *arr*, *cols*, *meta*, *tups=None*)

---

**Note:** This will be changed to a private method with a different name as it only pertains to the SDS file format.

Load DateTimeNano from an SDS file as the correct class. Restore formatting if different than default.

---

**Parameters**

- **name** (*item's name in the calling container, or the classname 'DateTimeNano' by default*) –
- **arr** (*underlying integer FastArray in UTC nanoseconds*) –
- **cols** (*empty list (not used for this class)*) –
- **meta** (*meta data generated by build_meeta_data() routine*) –
- **tups** (*empty list (not used for this class)*) –
- **object.** (*returns reconstructed DateTimeNano*) –

**_meta_dict**(*name=None*)

Meta dictionary for _build_sds_meta_data, _as_meta_data

**classmethod _parse_item_format**(*itemformat*)

Translate a value in the DisplayLength enum into a time format string

**classmethod _random**(*sz*, *to_tz='NYC'*, *from_tz='NYC'*, *inv=None*, *start=None*, *end=None*)

Internal routine for random(), random_invalid()

**astimezone**(*tz*)

Returns a new DateTimeNano object in a different displayed timezone. The new object holds a reference to the same underlying array.

**Parameters**
**tz** (*str*) – Abbreviated name of desired timezone. See rt.TimeZone.valid_timezones

**Returns**
**obj**

**Return type**
*DateTimeNano*

### Notes

Unlike Python's datetime.datetime.astimezone(), accepts strings, not timezone objects.

**cut_time**(*buckets*, *start_time=None*, *end_time=None*, *add_pre_bucket=False*, *add_post_bucket=False*, *label='left'*, *label_fmt=None*, *nyc=False*)

Analogous to rt.cut() but for times. We ignore the date part and cut based on time of day component only.

**Parameters**

- **buckets** (*int or rt.TimeSpan or a list of for custom buckets*) – Specify your bucket size or buckets. Supply either an int for the common use case of equally sized minute buckets or a custom list

  **Acceptable lists formats:**
  [(h, m, s, ms)] - it'll assume fields are 0 if length is less than 4
- **start_time** (*optional if buckets is explicitly supplied, (h, m) or (h, m, s) or (h, m , s, ms) tuple*) – left end point of first bucket, this type may change in future
- **end_time** – see start_time, right end point of last bucket
- **add_pre_bucket** (*bool*) – add a pre-open bucket or treat as invalid ?
- **add_post_bucket** (*bool*) – add a after close bucket or treat as invalid ?
- **label** (*optional str*) – "left": for left end points "right": for right end points
- **label_fmt** (*optional str*) – strftime format for label
- **nyc** (*bool, default is False*) – convenience shortcut to default to NYC start and end time, ignored if buckets explicitly supplied

**Return type**
rt.Categorical

**See also:**

inspired

### Examples

TODO - sanitize - add cut_time examples See the version history for structure of older examples.

**diff**(*periods=1*)

**Return type**
*TimeSpan*

**diff**(*periods=1*)

Calculate the n-th discrete difference.

**Parameters**
**periods** (*int, optional*) – The number of times values are differenced. If zero, the input is returned as-is.

**Returns**
**obj**

**Return type**
*TimeSpan*

**static display_convert_func**(*utcnano*, *itemformat*)

Convert a utc nanosecond timestamp to a string for display.

**Parameters**

- **utcnano** (*int*) – Timestamp in nanoseconds, a single value from a DateTimeNano array
- **itemformat** (obj:ItemFormat) – Style object retrieved from display callback.

**Returns**
Timestamp as string.

**Return type**
str

**See also:**

*DateTimeNano.display_query_properties*, *riptable.Utils.rt_display_properties*

**display_item**(*utcnano*)

Convert a utc nanosecond timestamp to a string for array repr.

**Parameters**
**utcnano** (*int*) – Timestamp in nanoseconds, a single value from a DateTimeNano array

**Returns**
Timestamp as string.

**Return type**
str

**display_query_properties**()

Call back for display functions to get the formatting function and style for timestrings. Each instance knows how to format its time strings. The formatter is specified in TIME_FORMATS The length property of item_format stores the index into TIME_FORMATS for the display_convert_func

**Returns**

- **obj** (ItemFormat) – See riptable.Utils.rt_display_properties
- *function* – Callback function for formatting the timestring

**fill_invalid**(*shape=None*, *dtype=None*, *inplace=True*)

Replace all values of the input FastArray with an invalid value.

The invalid value used is determined by the input array's dtype or a user-specified dtype.

Warning: By default, this operation is in place.

**Parameters**

- **shape** (*int or sequence of int, optional*) – Shape of the new array, for example: (2, 3) or 2. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str, optional*) – The desired dtype for the returned array.
- **inplace** (*bool, default True*) – If True (the default), modify original data. If False, return a copy of the array.

**Returns**
If inplace=False, a copy of the input FastArray is returned that has all values replaced with an invalid value. Otherwise, nothing is returned.

**Return type**
*FastArray*, optional

**See also:**

**FastArray.inv**
Return the invalid value for the input array's dtype.

**FastArray.copy_invalid**
Return a copy of a FastArray filled with the invalid value for the array's dtype.

### Examples

Replace an integer array's values with the invalid value for the array's dtype. By default, the returned array is the same size and dtype as the input array, and the operation is performed in place:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a.fill_invalid()
>>> a
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Replace a floating-point array's values with the invalid value for the int32 dtype:

```
>>> a2 = rt.FA([0., 1., 2., 3., 4.])
>>> a2
FastArray([0., 1., 2., 3., 4.])
>>> a2.fill_invalid(dtype="int32", inplace=False)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Specify the size and dtype of the output array:

```
>>> a3 = rt.FA(["AMZN", "IBM", "MSFT", "AAPL"])
>>> a3
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a3.fill_invalid(2, dtype="bool", inplace=False)
FastArray([False, False])
```

**static format_nano_time**(*utcnano*, *itemformat*)

Convert a utc nanosecond timestamp to a string for display.

**Parameters**

- **utcnano** (*int*) – Timestamp in nanoseconds, a single value from a DateTimeNano array
- **itemformat** (obj:ItemFormat) – Style object retrieved from display callback.

**Returns**
Timestamp as string.

**Return type**
str

### Notes

Uses Python's datetime module for final string conversion.

**get_classname**()

Return object's class name for array repr.

**Returns**
**obj** – Object's class name.

**Return type**
str

**get_scalar**(*scalarval*)

**classmethod hstack**(*dtlist*)

Performs an hstack on a list of DateTimeNano objects. All items in list must have their display set to the same timezone.

**Parameters**
**dtlist** (obj:list of obj:*DateTimeNano*) – DateTimeNano objects to be stacked.

### Examples

```
>>> dtn1 = DateTimeNano(['2019-01-01', '2019-01-02'], from_tz='NYC')
>>> dtn2 = DateTimeNano(['2019-01-03', '2019-01-04'], from_tz='NYC')
>>> DateTimeNano.hstack([dtn1, dtn2])
DateTimeNano([20190101 00:00:00.000000000, 20190102 00:00:00.000000000,␣
↪20190103 00:00:00.000000000, 20190104 00:00:00.000000000])
```

**Returns**
**obj**

**Return type**
*DateTimeNano*

**info**()

**Returns**
Verbose array repr with timezone information.

**Return type**
str

**isfinite**()

Return a boolean array that's True for each *DateTimeNano* element that's not a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int64 sentinel value are considered to be NaN.

**Returns**
A FastArray of booleans that's True for each non-NaN element, False otherwise.

**Return type**
FastArray

**See also:**

*DateTimeNano.isnan*, *Date.isnan*, *Date.isnotnan*, riptable.isnan, riptable.isnotnan, riptable.isnanorzero, FastArray.isnan, FastArray.isnotnan, FastArray.notna, FastArray.isnanorzero, Categorical.isnan, Categorical.isnotnan, Categorical.notna

**Dataset.mask_or_isnan**
Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN Dataset row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17',
...                        '20210713 02:44:19'], from_tz = 'NYC')
>>> dtn[0] = 0
>>> dtn[1] = dtn.inv
>>> dtn
DateTimeNano(['Inv', 'Inv', '20210712 22:44:19.000000000'], to_tz='NYC')
>>> dtn.isfinite()
FastArray([False, False,  True])
```

**isnan**()
Return a boolean array that's True for each *DateTimeNano* element that's a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int64 sentinel value are considered to be NaN.

**Returns**
A FastArray of booleans that's True for each NaN element, False otherwise.

**Return type**
FastArray

**See also:**

*DateTimeNano.isnotnan*, *Date.isnan*, *Date.isnotnan*, riptable.isnan, riptable.isnotnan, riptable.isnanorzero, FastArray.isnan, FastArray.isnotnan, FastArray.notna, FastArray.isnanorzero, Categorical.isnan, Categorical.isnotnan, Categorical.notna

**Dataset.mask_or_isnan**
Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN Dataset row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17',
...                        '20210713 02:44:19'], from_tz = 'NYC')
>>> dtn[0] = 0
>>> dtn[1] = dtn.inv
>>> dtn
DateTimeNano(['Inv', 'Inv', '20210712 22:44:19.000000000'], to_tz='NYC')
>>> dtn.isnan()
FastArray([ True,  True, False])
```

**isnotfinite**()

Return a boolean array that's True for each *DateTimeNano* element that's a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int64 sentinel value are considered to be NaN.

**Returns**
A `FastArray` of booleans that's True for each NaN element, False otherwise.

**Return type**
`FastArray`

**See also:**

*DateTimeNano.isnotnan*, *Date.isnan*, *Date.isnotnan*, `riptable.isnan`, `riptable.isnotnan`, `riptable.isnanorzero`, `FastArray.isnan`, `FastArray.isnotnan`, `FastArray.notna`, `FastArray.isnanorzero`, `Categorical.isnan`, `Categorical.isnotnan`, `Categorical.notna`

**Dataset.mask_or_isnan**
Return a boolean array that's True for each `Dataset` row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN `Dataset` row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17',
...                        '20210713 02:44:19'], from_tz = 'NYC')
>>> dtn[0] = 0
>>> dtn[1] = dtn.inv
>>> dtn
DateTimeNano(['Inv', 'Inv', '20210712 22:44:19.000000000'], to_tz='NYC')
>>> dtn.isnotfinite()
FastArray([ True,  True, False])
```

**isnotnan**()

Return a boolean array that's True for each *DateTimeNano* element that's not a NaN (Not a Number), False otherwise.

Both the DateTime NaN (0) and Riptable's int64 sentinel value are considered to be NaN.

**Returns**
A FastArray of booleans that's True for each non-NaN element, False otherwise.

**Return type**
FastArray

**See also:**

*DateTimeNano.isnan*, *Date.isnan*, *Date.isnotnan*, riptable.isnan, riptable.isnotnan, riptable.isnanorzero, FastArray.isnan, FastArray.isnotnan, FastArray.notna, FastArray.isnanorzero, Categorical.isnan, Categorical.isnotnan, Categorical.notna

**Dataset.mask_or_isnan**
Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN Dataset row.

### Notes

Riptable currently uses 0 for the DateTime NaN value. This constant is held in the *DateTimeBase* class.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17',
...                        '20210713 02:44:19'], from_tz = 'NYC')
>>> dtn[0] = 0
>>> dtn[1] = dtn.inv
>>> dtn
DateTimeNano(['Inv', 'Inv', '20210712 22:44:19.000000000'], to_tz='NYC')
>>> dtn.isnotnan()
FastArray([False, False,  True])
```

**max**(***kwargs*)

The latest *DateTimeNano* in an array.

**Returns**
A *DateTimeNano* array containing the latest *DateTimeNano* from the input array.

**Return type**
*DateTimeNano*

**See also:**

*DateTimeNano.min*, Date.min, Date.max, DateSpan.min, Datespan.max

### Notes

This returns an array, not a scalar. However, broadcasting rules will apply to operations with it.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17'],
...                       from_tz='NYC', to_tz='NYC')
>>> dtn.max()
DateTimeNano(['20210101 09:31:15.000000000'], to_tz='NYC')
```

**min**(***kwargs*)

The earliest *DateTimeNano* in an array.

Note that until a reported bug is fixed, this method is not NaN-aware.

**Returns**

A *DateTimeNano* array containing the earliest *DateTimeNano* from the input array.

**Return type**

*DateTimeNano*

**See also:**

*DateTimeNano.max*, Date.min, Date.max, DateSpan.min, Datespan.max

### Notes

This returns an array, not a scalar. However, broadcasting rules will apply to operations with it.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17'],
...                       from_tz='NYC', to_tz='NYC')
>>> dtn.min()
DateTimeNano(['20210101 09:31:15.000000000'], to_tz='NYC')
```

**classmethod newclassfrominstance**(*instance*, *origin*)

Restore timezone/length info.

**classmethod random**(*sz*, *to_tz='NYC'*, *from_tz='NYC'*, *inv=None*, *start=None*, *end=None*)

Return an array of randomly generated *DateTimeNano* values.

If start and end are not provided, years range from 1971 to 2020.

**Parameters**

- **sz** (*int*) – The length of the generated array.
- **to_tz** (*str, default 'NYC'*) – The timezone for display. For valid timezone options, see *TimeZone.valid_timezones*.
- **from_tz** (*str, default 'NYC'*) – The timezone of origin. For valid timezone options, see *TimeZone.valid_timezones*.
- **inv** (*array of bool, optional*) – Where True, an invalid *DateTimeNano* is in the returned array.

- **start** (*int, optional*) – The start year for the range. If no end year is provided, all times are within the start year.
- **end** (*int, optional*) – The end year for the range. Used only if `start` is provided.

**Returns**
A *DateTimeNano* with randomly generated values.

**Return type**
*DateTimeNano*

**See also:**

*DateTimeNano.random_invalid*
Return a randomly generated *DateTimeNano* array with randomly placed invalid values.

*Date.range*
Return a *Date* object of dates within a given interval, spaced by `step`.

`riptable.arange`
Return an array of evenly spaced values within a given interval.

### Examples

```
>>> rt.DateTimeNano.random(3)
DateTimeNano(['19980912 15:31:08.025189457', '19931121 15:48:32.855425859',␣
↪'19930915 14:58:31.376750294'])  # random
```

If `start` is provided but `end` is not, all times are within the start year:

```
>>> rt.DateTimeNano.random(3, start=2015)
DateTimeNano(['20151011 12:15:45.588049363', '20150207 14:54:33.649991888',
↪'20150131 18:58:13.543792210'], to_tz='NYC')  # random
```

With an `inv` mask. Where True, an invalid *DateTimeNano* is in the returned array:

```
>>> i = rt.FastArray([True, False, True])
>>> rt.DateTimeNano.random(3, inv=i)
DateTimeNano(['Inv', '19930915 02:39:29.621051630', 'Inv'], to_tz='NYC')
```

**classmethod random_invalid**(*sz*, *to_tz='NYC'*, *from_tz='NYC'*, *start=None*, *end=None*)

Return a randomly generated *DateTimeNano* object with randomly placed invalid values.

This method is the same as *DateTimeNano.random*, except that a mask is randomly generated to place the invalid values.

If `start` and `end` are not provided, years for valid *DateTimeNano* values range from 1971 to 2020.

**Parameters**

- **sz** (*int*) – The length of the generated array.
- **to_tz** (*str, default 'NYC'*) – The timezone for display. For valid timezone options, see *TimeZone.valid_timezones*.
- **from_tz** (*str, default 'NYC'*) – The timezone of origin. For valid timezone options, see *TimeZone.valid_timezones*.
- **start** (*int, optional*) – The start year for the range. If no end year is provided, all times are within the start year.

- **end** (*int, optional*) – The end year for the range. Used only if `start` is provided.

**Returns**
A *DateTimeNano* with randomly placed invalid values.

**Return type**
*DateTimeNano*

**See also:**

*DateTimeNano.random*
Return an array of randomly generated *DateTimeNano* values.

*Date.range*
Return a *Date* object of dates within a given interval, spaced by `step`.

**riptable.arange**
Return an array of evenly spaced values within a specified interval.

### Examples

```
>>> rt.DateTimeNano.random_invalid(3)
DateTimeNano(['Inv', '19830405 15:24:01.815771855', 'Inv'], to_tz='NYC')  #␣
↪random
```

**resample**(*rule*, *dropna=False*)

Convenience method for frequency conversion and resampling of DateTimeNano arrays.

**Parameters**

- **rule** (*string*) – The offset string or object representing target conversion. Can also begin the string with a number e.g. '3H' Currently supported: H hour T, min minute S second L, ms millisecond U, us microsecond N, ns nanosecond
- **dropna** (*bool, default False*) – If True, returns a DateTimeNano the same length as caller, with all values rounded to specified frequency. If False, returns a DateTimeNano range from caller's min to max with values at every specified frequency.

### Examples

```
>>> dtn = DateTimeNano(['2015-04-15 14:26:54.735321368',
                        '2015-04-20 07:30:00.858219615',
                        '2015-04-23 13:15:24.526871083',
                        '2015-04-21 02:25:11.768548100',
                        '2015-04-24 07:47:54.737776979',
                        '2015-04-10 23:59:59.376589955'],
                             from_tz='UTC', to_tz='UTC')
>>> dtn.resample('L', dropna=True)
DateTimeNano(['20150415 14:26:54.735000000', '20150420 07:30:00.858000000',
↪'20150423 13:15:24.526000000', '20150421 02:25:11.768000000', '20150424 07:
↪47:54.737000000', '20150410 23:59:59.376000000'], to_tz='UTC')
```

```
>>> dtn = DateTimeNano(['20190417 17:47:00.000001',
                        '20190417 17:47:00.000003',
                        '20190417 17:47:00.000005'],
```

(continues on next page)

(continued from previous page)

```
                                        from_tz='NYC')
>>> dtn.resample('1us')
DateTimeNano(['20190417 17:47:00.000001000', '20190417 17:47:00.000002000',
↪'20190417 17:47:00.000003000', '20190417 17:47:00.000004000', '20190417 17:
↪47:00.000005000'], to_tz='NYC')
```

**Returns**
**dtn**

**Return type**
*DateTimeNano*

**set_timezone**(*tz*)

Changes the timezone that the times are displayed in. Different lookup array will be used for daylight savings fixups. Does not modify the underlying array.

**Parameters**
**tz** (*str*) – Abbreviated name of desired timezone. See rt.TimeZone.valid_timezones

### Examples

Normal: >>> dtn = DateTimeNano(['2019-01-07 10:36'], from_tz='NYC', to_tz='NYC') >>> dtn DateTimeNano([20190107 10:36:00.000000000]) >>> dtn.set_timezone('DUBLIN') >>> dtn DateTimeNano([20190107 15:36:00.000000000])

NYC is in daylight savings time, Dublin is not: >>> dtn = DateTimeNano(['2019-03-15 10:36'], from_tz='NYC', to_tz='NYC') >>> dtn DateTimeNano([20190315 10:36:00.000000000]) >>> dtn.set_timezone('DUBLIN') >>> dtn DateTimeNano([20190315 14:36:00.000000000])

**shift**(*periods=1*)

Modeled on pandas.shift. Values in the array will be shifted to the right if periods is positive, to the left if negative. Spaces at either end will be filled with invalid. If abs(periods) >= the length of the array, the result will be full of invalid.

**Parameters**
**periods** (*int*) – Number of periods to move, can be positive or negative

**Returns**
**obj**

**Return type**
*DateTimeNano*

**strftime**(*format*, *dtype='O'*)

Convert each *DateTimeNano* element to a formatted string representation.

**Parameters**

- **format** (*str*) – One or more format codes supported by the `datetime.datetime.strftime()` function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.
- **dtype** (*{"O", "S", "U"}, default "O"*) – The data type of the returned array:
  - "O": object string
  - "S": byte string

– "U": unicode string

**Returns**
An ndarray of strings.

**Return type**
ndarray

**See also:**

*DateTimeNanoScalar.strftime*, *Date.strftime*, *DateScalar.strftime*, *TimeSpan.strftime*, *TimeSpanScalar.strftime*

### Notes

This routine has not been sped up yet. It also raises an error on NaNs.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17'], from_tz=
↪'NYC')
>>> dtn
DateTimeNano(['20210101 09:31:15.000000000', '20210519 05:21:17.000000000'],␣
↪to_tz='NYC')
>>> dtn.strftime('%c')
array(['Fri Jan  1 09:31:15 2021', 'Wed May 19 05:21:17 2021'], dtype=object)
```

**to_arrow**(*type=None*, ***, *preserve_fixed_bytes=False*, *empty_strings_to_null=True*)

Convert this *DateTimeNano* to a pyarrow.Array.

**Parameters**

- **type** (*pyarrow.DataType, optional, defaults to None*) – Unused.
- **preserve_fixed_bytes** (*bool, optional, defaults to False*) – Unused.
- **empty_strings_to_null** (*bool, optional, defaults To True*) – Unused.

**Return type**
pyarrow.Array or pyarrow.ChunkedArray

**to_iso**()

Generates a FastArray of ISO-8601 timestamp bytestrings. The string will match the time +/- timezone offset displayed in the output of the DateTimeNano object.

### Examples

```
>>> dtn = DateTimeNano(['2019-01-22 12:34'],from_tz='NYC')
>>> dtn
DateTimeNano([20190122 12:34:00.000000000])
>>> dtn.to_iso()
FastArray([b'2019-01-22T12:34:00.000000000'], dtype='|S48')
```

```
>>> dtn = DateTimeNano(['2019-01-22'],from_tz='GMT',to_tz='NYC')
>>> dtn
DateTimeNano([20190121 19:00:00.000000000])
>>> dtn.to_iso()
FastArray([b'2019-01-21T19:00:00.000000000'], dtype='|S48')
```

**Returns**
**obj**

**Return type**
FastArray

class riptable.rt_datetime.**DateTimeNanoScalar**(***kwargs*)

Bases: numpy.int64, DateTimeCommon, TimeStampBase

Derived from np.int64 NOTE: np.int64 is a SLOT wrapper and does not have a __dict__ Number of nanoseconds since unix epoch 1970 in UTC

**property _fa**

**property _np**

**__slots__ = ('_display_length', '_timezone')**

**__add__**(*value*)

**__repr__**()
Return repr(self).

**__str__**()
Return str(self).

**__sub__**(*value*)

**get_classname**()

**get_item_format**()

**isfinite**()

**isnan**()

**isnotfinite**()

**isnotnan**()

**repeat**(*repeats*, *axis=None*)

**strftime**(*format*)
Convert a *DateTimeNanoScalar* to a formatted string representation.

**Parameters**
**format** (*str*) – One or more format codes supported by the datetime.datetime.strftime() function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.

**Returns**
A string representation of the reformatted *DateTimeNanoScalar*.

**Return type**
str

**See also:**

*DateTimeNano.strftime*, *Date.strftime*, *DateScalar.strftime*, *TimeSpan.strftime*, *TimeSpanScalar.strftime*

### Notes

This routine has not been sped up yet. It also raises an error on NaNs.

### Examples

```
>>> dtn = rt.DateTimeNano(['20210101 09:31:15', '20210519 05:21:17'], from_tz=
↪'NYC')
>>> dtn
DateTimeNano(['20210101 09:31:15.000000000', '20210519 05:21:17.000000000'],␣
↪to_tz='NYC')
>>> dtn[0].strftime('%c')
'Fri Jan  1 09:31:15 2021'
```

**tile**(*repeats*)

**class** riptable.rt_datetime.**TimeSpan**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: TimeSpanBase, *DateTimeBase*

Array of time delta in nanoseconds, held in float64.

**Parameters**

- **values** (*numeric or string array or scalar*) – If string, interpreted as HH:MM:SS.fffffff ( seconds/second fractions optional ) If numeric, interpreted as nanoseconds, unless unit provided. single number or array / list of numbers (unless unit is specified, will assume nanoseconds)
- **unit** (*str, optional, default 'ns'*) – Precision of data in the constructor. All will be converted to nanoseconds. Valid units: 'Y', 'W', 'D', 'h', 'm', 's', 'ms', 'us', 'ns'

### Examples

From single string: >>> dts = TimeSpan('12:34') >>> dts TimeSpan([12:34:00.000000000])

From milliseconds since midnight: >>> dts = TimeSpan(FA([34500000., 36500000., 38500000.,]), unit='ms') >>> dts TimeSpan([09:35:00.000000000, 10:08:20.000000000, 10:41:40.000000000])

From the result of DateTimeNano subtraction: >>> dtn1 = DateTimeNano(['2018-01-01 09:35:00'], from_tz='NYC') >>> dtn2 = DateTimeNano(['2018-01-01 07:15:00'], from_tz='NYC') >>> dtn1 - dtn2 TimeSpan([02:20:00.000000000])

Certain DateTimeNano properties can return a TimeSpan: >>> dtn = DateTimeNano(['2018-01-01 09:35:00'], from_tz='NYC') >>> dtn.hour_span TimeSpan([09:35:00.000000000])

Can be added to DateTimeNano objects: >>> dtn = DateTimeNano(['2018-01-01 09:35:00'], from_tz='NYC') >>> ts = TimeSpan(FA([8400000000000.0])) >>> dtn + ts DateTimeNano([20180101 11:55:00.000000000])

Can be multiplied / divided by scalars: >>> ts = TimeSpan(FA([8400000000000.0])) >>> ts TimeSpan([02:20:00.000000000]) >>> ts / 2 TimeSpan([01:10:00.000000000]) >>> ts * 5.6 TimeSpan([13:04:00.000000000])

**__arrow_array__**(*type=None*)

Implementation of the `__arrow_array__` protocol for conversion to a pyarrow array.

**Parameters**
**type** (*pyarrow.DataType, optional, defaults to None*) –

**Return type**
pyarrow.Array or pyarrow.ChunkedArray

### Notes

https://arrow.apache.org/docs/python/extending_types.html#controlling-conversion-to-pyarrow-array-with-the-arrow-array

**static _from_arrow**(*arr*, *zero_copy_only=True*, *writable=False*)

Create a *TimeSpan* instance from a "duration"-typed `pyarrow.Array`.

**Parameters**

- **arr** (*pyarrow.Array or pyarrow.ChunkedArray*) – Must be a "duration"-typed pyarrow array.
- **zero_copy_only** (*bool, optional, defaults to False*) –
- **writable** (*bool, optional, defaults to False*) –

**Return type**
*TimeSpan*

**classmethod _from_meta_data**(*arrdict*, *arrflags*, *meta*)

**classmethod _load_from_sds_meta_data**(*name*, *arr*, *cols*, *meta*, *tups=None*)

Load DateTimeNano from an SDS file as the correct class. Restore formatting if different than default.

**_meta_dict**(*name=None*)

**fill_invalid**(*shape=None*, *dtype=None*, *inplace=True*)

Replace all values of the input `FastArray` with an invalid value.

The invalid value used is determined by the input array's dtype or a user-specified dtype.

Warning: By default, this operation is in place.

**Parameters**

- **shape** (*int or sequence of int, optional*) – Shape of the new array, for example: `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str, optional*) – The desired dtype for the returned array.
- **inplace** (*bool, default True*) – If `True` (the default), modify original data. If `False`, return a copy of the array.

**Returns**
If `inplace=False`, a copy of the input `FastArray` is returned that has all values replaced with an invalid value. Otherwise, nothing is returned.

**Return type**
*FastArray*, optional

**See also:**

**FastArray.inv**
Return the invalid value for the input array's dtype.

**FastArray.copy_invalid**
Return a copy of a FastArray filled with the invalid value for the array's dtype.

### Examples

Replace an integer array's values with the invalid value for the array's dtype. By default, the returned array is the same size and dtype as the input array, and the operation is performed in place:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a.fill_invalid()
>>> a
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Replace a floating-point array's values with the invalid value for the int32 dtype:

```
>>> a2 = rt.FA([0., 1., 2., 3., 4.])
>>> a2
FastArray([0., 1., 2., 3., 4.])
>>> a2.fill_invalid(dtype="int32", inplace=False)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Specify the size and dtype of the output array:

```
>>> a3 = rt.FA(["AMZN", "IBM", "MSFT", "AAPL"])
>>> a3
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a3.fill_invalid(2, dtype="bool", inplace=False)
FastArray([False, False])
```

**get_classname**()

**get_scalar**(*scalarval*)

**classmethod hstack**(*tspans*)
TODO: maybe add type checking? This is a very simple class, rewrap the hstack result in class.

**classmethod newclassfrominstance**(*instance*, *origin*)

**strftime**(*format*, *dtype='U'*)
Convert each *TimeSpan* element to a formatted string representation.

**Parameters**

- **format** (*str*) – One or more format codes supported by the datetime.datetime.strftime() function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.
- **dtype** (*{"U", "S", "O"}, default "U"*) – The data type of the returned array:
  - "U": unicode string
  - "S": byte string
  - "O": object string

**Returns**
An ndarray of strings.

**Return type**
ndarray

**See also:**

*TimeSpanScalar.strftime*, *Date.strftime*, *DateScalar.strftime*, *DateTimeNano.strftime*, *DateTimeNanoScalar.strftime*

### Notes

This routine has not been sped up yet. It also raises an error on NaNs.

### Examples

```
>>> ts = rt.TimeSpan(['09:00', '10:45', '02:30'])
>>> ts
TimeSpan(['09:00:00.000000000', '10:45:00.000000000', '02:30:00.000000000'])
>>> ts.strftime('%X')
array(['09:00:00', '10:45:00', '02:30:00'], dtype='<U8')
```

**to_arrow**(*type=None*, ***, *preserve_fixed_bytes=False*, *empty_strings_to_null=True*)

Convert this *TimeSpan* to a pyarrow.Array.

**Parameters**

- **type** (*pyarrow.DataType, optional, defaults to None*) – Unused.
- **preserve_fixed_bytes** (*bool, optional, defaults to False*) – Unused.
- **empty_strings_to_null** (*bool, optional, defaults To True*) – Unused.

**Return type**
pyarrow.Array or pyarrow.ChunkedArray

**class** riptable.rt_datetime.**TimeSpanScalar**(***kwargs*)

Bases: numpy.float64, TimeSpanBase

Derived from np.float64 ******** not implemented Holds single float values for TimeSpan arrays. These will be returned from operations that currently return a TimeSpan of a single item.

**property _fa**

**property _np**

__slots__ = '_display_length'

__abs__()
abs(self)

__add__(*value*)
Return self+value.

__eq__(*other*)
Return self==value.

__floordiv__(*value*)
Return self//value.

__mul__(*value*)
Return self*value.

__neg__()
-self

__pos__()
+self

__radd__(*value*)
Return value+self.

__repr__()
Return repr(self).

__rmul__(*value*)
Return value*self.

__rsub__(*value*)
Return value-self.

__str__()
Return str(self).

__sub__(*value*)
Return self-value.

__truediv__(*value*)
Return self/value.

abs()

get_classname()

get_item_format()

isfinite()

isnan()

isnotfinite()

isnotnan()

repeat(*repeats*, *axis=None*)

**strftime**(*format*)

Convert a *TimeSpanScalar* to a formatted string representation.

**Parameters**

**format** (*str*) – One or more format codes supported by the datetime.datetime.strftime() function of the standard Python distribution. For codes, see strftime() and strptime() Behavior.

**Returns**

A string representation of the reformatted *TimeSpanScalar*.

**Return type**

str

**See also:**

*TimeSpan.strftime*, *Date.strftime*, *DateScalar.strftime*, *DateTimeNano.strftime*, *DateTimeNanoScalar.strftime*

### Notes

This routine has not been sped up yet. It also raises an error on NaNs.

### Examples

```
>>> ts = rt.TimeSpan(['09:00', '10:45', '02:30'])
>>> ts
TimeSpan(['09:00:00.000000000', '10:45:00.000000000', '02:30:00.000000000'])
>>> ts[0].strftime('%X')
'09:00:00'
```

**tile**(*repeats*)

riptable.rt_datetime.**DateTimeUTC**(*arr*, *to_tz='NYC'*, *from_matlab=False*, *format=None*, *start_date=None*, *gmt=None*)

Forces DateTimeNano from_tz keyword to 'UTC'. For more see DateTimeNano.

riptable.rt_datetime.**datestring_to_nano**(*datestring*, *time=None*, *from_tz=None*, *to_tz='NYC'*)

Converts date string to DateTimeNano object (default midnight). By default, the timestrings are assumed to be in Eastern Time. If they are already in UTC time, set gmt=True.

**Parameters**

- **datestring** (*array of datestrings in format YYYY-MM-DD or YYYYMMDD (bytestrings/unicode supported)*) –
- **time** (*a single string or array of strings in the format HH:MM:SS.ffffff (bytestrings/unicode supported)*) –
- **from_tz** (*a string for the timezone of origin: 'NYC', 'GMT', 'DUBLIN', etc.*) –
- **to_tz** (*a string for the timezone that the time will be displayed in*) –
- **DateTimenano** (*returns*) –
- **Also** (*See*) –

### Examples

Date only:

```
>>> dates = FA(['2018-01-01', '2018-01-02', '2018-01-03'])
>>> datestring_to_nano(dates, from_tz='NYC')
DateTimeNano([20180101 00:00:00.000000000, 20180102 00:00:00.000000000, 20180103 00:
↪00:00.000000000])
```

With time:

```
>>> dates = FA(['2018-01-01', '2018-01-02', '2018-01-03'])
>>> datestring_to_nano(dates, time='9:30:00', from_tz='NYC')
DateTimeNano([20180101 09:30:00.000000000, 20180102 09:30:00.000000000, 20180103 09:
↪30:00.000000000])
```

riptable.rt_datetime.**datetimestring_to_nano**(*dtstring*, *from_tz=None*, *to_tz='NYC'*)

Converts datetime string to DateTimeNano object. By default, the timestrings are assumed to be in Eastern Time. If they are already in UTC time, set gmt=True.

**Parameters**

- **dtstring** (*array of timestrings in format YYYY-MM-DD HH:MM:SS, YYYYMMDD HH:MM:SS.ffffff, etc. (bytestrings/unicode supported)*) –
- **from_tz** (*a string for the timezone of origin: 'NYC', 'GMT', 'DUBLIN', etc.*) –
- **to_tz** (*a string for the timezone that the time will be displayed in*) –
- **DateTimeNano** (*returns*) –
- **Also** (*See*) –

### Examples

```
>>> dts = FA(['2012-12-12 12:34:56.001002', '20130303 1:14:15', '2008-07-06 15:14:13
↪'])
>>> datetimestring_to_nano(dts, from_tz='NYC')
DateTimeNano([20121212 12:34:56.001002000, 20130303 01:14:15.000000000, 20080706 15:
↪14:13.000000000])
```

riptable.rt_datetime.**parse_epoch**(*etime*, *to_tz='NYC'*)

Days since epoch and milliseconds since midnight from nanosecond timestamps.

**Parameters**

- **etime** (*array-like*) – UTC nanoseconds.
- **to_tz** (*str, default 'NYC'*) – TimeZone short string - see riptable.rt_timezone. This routine didn't used to take a timezone, so it defaults to the previous setting.
- **loader.** (*Used in the phonyx data*) –

**Returns**

- **days** (*array (int32)*) – Days since epoch.

- **millis** (*array (float64)*) – Milliseconds since midnight.

riptable.rt_datetime.**strptime_to_nano**(*dtstrings*, *format*, *from_tz=None*, *to_tz='NYC'*)

Converts datetime string to DateTimeNano object with user-specified format.

**Parameters**

- **dtstrings** (*array of timestrings*) –
- **format** (*timestring format*) – Currently supports the following escape codes:

  **Date**

  - %y Year without century as zero-padded decimal number.
  - %Y Year with century as decimal number.
  - %m Month as a decimal number (with or without zero-padding).
  - %B Full month name: [‘January’, ‘February’, ‘March’, ‘April’, ‘May’, ‘June’, ‘July’, ‘August’, ‘September’, ‘October’, ‘November’, ‘December’]
  - %b Abbreviated month name: [‘Jan’, ‘Feb’, ‘Mar’, ‘Apr’, ‘May’, ‘Jun’, ‘Jul’, ‘Aug’, ‘Sep’, ‘Oct’, ‘Nov’, ‘Dec’]
  - %d Day of the month as a decimal number (with or without zero-padding).

  **Time**

  - %H Hour (24-hour clock) as a decimal number (with or without zero-padding). (Note: if a %p formatter is present, this will be interpretted as a 12-hour clock hour)
  - %I Hour (12-hour clock) as a decimal number (with or without zero-padding). (Note: unlike %H, must be 1-12)
  - %p Locale’s equivalent of either AM or PM.
  - %M Minute as a decimal number (with or without zero-padding).
  - %S Second as a decimal number (with or without zero-padding).
- **from_tz** (*str*) – The timezone of origin: ‘NYC’, ‘GMT’, ‘DUBLIN’, etc.
- **to_tz** (*str*) – The timezone that the time will be displayed in.

### Notes

Works best with timestrings that include a date:

- If no year is present in the string, an invalid time will be returned for all values.
- If no form of year/month/day is present, values will yield a time in 1970.

Consider using *timestring_to_nano()*, which also will accept one datestring for all times.

If the timestring ends in a ‘.’, the following numbers will be parsed as a second fraction. This happens automatically, no escape character is required in the format string.

If no time escape characters are present, will return midnight at all date values. If formatted correctly, consider using *datestring_to_nano()*.

### Examples

Date, with/without padding:

```
>>> dt = FastArray(['02/01/1992', '2/1/1992'])
>>> fmt = '%m/%d/%Y'
>>> strptime_to_nano(dt, fmt, from_tz='NYC')
DateTimeNano([19920201 00:00:00.000000000, 19920201 00:00:00.000000000])
```

Date + 24-hour clock:

```
>>> dt = FastArray(['02/01/1992 7:48:30', '2/1/1992 19:48:30'])
>>> fmt = '%m/%d/%Y %H:%M:%S'
>>> strptime_to_nano(dt, fmt, from_tz='NYC')
DateTimeNano([19920201 07:48:30.000000000, 19920201 19:48:30.000000000])
```

Date + 12-hour clock + am/pm:

```
>>> dt = FastArray(['02/01/1992 7:48:30 AM', '2/1/1992 7:48:30 PM'])
>>> fmt = '%m/%d/%Y %I:%M:%S %p'
>>> strptime_to_nano(dt, fmt, from_tz='NYC')
DateTimeNano([19920201 07:48:30.000000000, 19920201 19:48:30.000000000])
```

Date + time + second fraction:

```
>>> dt = FastArray(['02/01/1992 7:48:30.123456789', '2/1/1992 15:48:30.000000006'])
>>> fmt = '%m/%d/%Y %H:%M:%S'
>>> strptime_to_nano(dt, fmt, from_tz='NYC')
DateTimeNano([19920201 07:48:30.123456789, 19920201 15:48:30.000000006])
```

riptable.rt_datetime.**timestring_to_nano**(*timestring*, *date=None*, *from_tz=None*, *to_tz='NYC'*)

Converts timestring to TimeSpan or DateTimeNano object. By default, the timestrings are assumed to be in Eastern Time. If they are already in UTC time, set gmt=True. If a date is specified, a DateTimeNano object will be returned. If a date is not specified, a TimeSpan will be returned.

**Parameters**

- **timestring** (*array of timestrings in format HH:MM:SS, H:MM:SS, HH:MM:SS.ffffff (bytestrings/unicode supported)*) –
- **date** (*a single string or array of date strings in format YYYY-MM-DD (bytestrings/unicode supported)*) –
- **from_tz** (*a string for the timezone of origin: 'NYC', 'GMT', 'DUBLIN', etc.*) –
- **to_tz** (*a string for the timezone that the time will be displayed in*) –
- **DateTimeNano** (*returns TimeSpan or*) –
- **Also** (*See*) –

#### Examples

Return TimeSpan:

```
>>> ts = FA(['1:23:45', '12:34:56.000100', '       14:00:00'])
>>> timestring_to_nano(ts, from_tz='NYC')
TimeSpan([01:23:45.000000000, 12:34:56.000100000, 14:00:00.000000000])
```

With single date string:

```
>>> ts = FA(['1:23:45', '12:34:56', '23:22:21'])
>>> timestring_to_nano(ts, date='2018-02-01', from_tz='NYC')
DateTimeNano([20180201 01:23:45.000000000, 20180201 12:34:56.000000000, 20180201 23:
↪22:21.000000000])
```

Multiple date strings:

```
>>> ts = FA(['1:23:45', '12:34:56', '23:22:21'])
>>> dts = FA(['2018-02-01', '2018-02-07', '2018-05-12'])
>>> timestring_to_nano(ts, date=dts, from_tz='NYC')
DateTimeNano([20180201 01:23:45.000000000, 20180207 12:34:56.000000000, 20180512 23:
↪22:21.000000000])
```

### 2.2.12 riptable.rt_display

#### Classes

| | |
|---|---|
| *DisplayCell* | Wrapper around a string for styled console or html display. |
| *DisplayColumn* | New display column class for holding final display data. |
| *DisplayDetect* | |
| *DisplayString* | |
| *DisplayTable* | |
| *DisplayText* | Only uses two colors: green and purple OR cyan and blue |

**class** riptable.rt_display.**DisplayCell**(*string*, *value=None*, *color=None*, *html=False*, *colspan=None*, *align=None*)

Wrapper around a string for styled console or html display. Original value can also be stored for future column or table-wide math operations.

**color_mode_dict**

**darkbg_styles**

**lightbg_styles**

**no_styles**

__len__()

__repr__()
Return repr(self).

__str__()
Return str(self).

display(*html=False*, *plain=False*)

paint_cell()

class riptable.rt_display.DisplayColumn(*data*, *row_break=None*, *color=None*, *header=''*, *align=DisplayJustification.Right*, *html=False*, *itemformat=None*, *footer=None*)

New display column class for holding final display data.

**Responsible for:**
-adding a row break character to each array at the appropriate spot -painting display cells -adding padding for console -replacing whitespace with   for HTML -orchestrating math operations that span the whole column

property data
does not check for any formatting that has been applied. This routine is to assist in modifying repeated values in multikey groupby operations.

**Type**
Return string data array. Note

property display_width
When columns are being fit during display, need to account for padding between them, otherwise overflow will occur and table will break.

property footer

property header

css_color_classes

css_decoration_classes

css_justification_classes

__getitem__(*index*)

__repr__()
Return repr(self).

__setitem__(*index*, *value*)

__str__()
Return str(self).

_build_end(*end*, *final_color=None*, *plain=False*, *justification=None*)

add_footer(*footer*)

align_column(*align*, *col_slice=slice(None, None, None)*)

**static align_console_string**(*string*, *width*, *align=DisplayJustification.Right*)

Pad string for correct alignment in console table columns.

**static align_html**(*align=DisplayJustification.Right*)

**build_footer**(*final_color=None*, *plain=False*)

**build_header**(*final_color=None*, *plain=False*, *align=None*)

**build_summary**()

**paint_column**(*color*, *col_slice=None*, *badrows=None*)

Called when a column needs to be colored For instance the left side may be row numbered

**paint_highlightmax**()

**not implemented**, maybe for future Dataset.style() call Will paint max value of each numeric column gold.

**paint_posneg**()

***not implemented**, maybe for future Dataset.style() call Will paint positive values green, negative values red for all numeric columns in IPython console.

**plain_string_list**()

**static style_classes_html**(*itemformat=None*)

Calls routines to add a CSS class for every styling option in the ItemFormat object

**style_column**(*style*, *col_slice=None*, *badrows=None*)

This will replace paint_column as a single styling call for all column properties.

**styled_string_list**()

**static text_decoration_html**(*style=None*)

**class** riptable.rt_display.**DisplayDetect**

Bases: object

**ColorMode**

**ForceRepr = False**

**Mode = 0**

**static get_display_mode**()

**class** riptable.rt_display.**DisplayString**(*string*)

Bases: object

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**_repr_html_**()

**class** riptable.rt_display.**DisplayTable**(*attribs=None*)

Bases: object

**DebugMode = False**

**FORCE_REPR = False**

**INVALID_DATA**

**TestFooter = False**

**console_x_offset = 3**

**options**

**add_required_columns**(*header_names*, *table_data*, *footers*, *masked=False*, *gbkeys=None*, *transpose=False*, *color=None*, *style=None*)

header_names : list of string header names (not tuples) table_data : list of arrays footers : list of footer rows (lists of ColHeader tuples) masked : flag to indicate that the column has already been trimmed (build column does not need to apply a row mask) gbkeys : dictionary of groupby keys - columns need to be painted differently

**all_columns_console**(*console_width*, *left_offset*, *headers*, *columns*)

**all_columns_console_multiline**(*console_width*, *left_offset*, *headers*, *columns*)

**not implemented** only supports two-line headers

**build_column**(*header*, *column*, *masked=False*, *footer=None*, *style=None*)

All DisplayColumns built for final display will funnel through this function. Any row breaks will be added here if necessary.

**build_final_ends**(*plain=False*, *badcols=None*, *badrows=None*)

**build_final_headers_console**(*plain=False*)

**specifically for multi-line** Translates the tables header tuples into console strings with spaces for padding Note: this routine is very similar to build_final_headers_html. Keeping them separate for readability.

**build_final_headers_html**(*plain=False*)

Translates the tables header tuples into HTML tags. Note: this routine is very similar to build_final_headers_console. Keeping them separate for readability.

**build_result_table**(*header_tups*, *main_data*, *nrows*, *footer_tups=None*, *keys=None*, *sortkeys=None*, *from_str=False*, *sorted_row_idx=None*, *transpose_on=False*, *row_numbers=None*, *right_cols=None*, *badrows=None*, *badcols=None*, *styles=None*, *callback=None*)

Step 1: save all parameters into self namespace, as build_result_table is broken down into several functions. Step 2: if set_view has been called, only display the specified columns. if sort_values has been called, move those columns to the front. Step 3: build a row mask. if the table is too large to display, pull the first and last rows for display. if a sorted index is present, apply it. Step 4: measure the table. groupby key columns will always be included. fit as many columns as possible into the console. if the display is for html, defaults have been set to a hard-coded console width. other console width is detected upon each display. if there are too many columns to display, a column break will be set. Step 5: build the table. the result table is broken down into three parts: headers, left side, and main table. the headers are column names combined with left headers, or numbers if the table is transposed. the left side is row numbers, row labels, or groupby keys. the main table is first and last columns that would fit in the display. use the DisplayColumn class to organize the data for future styling. If the table is abbreviated, include a row break in each column. Step 6: style the table. html_on will let DisplayColumn and DisplayCell know how to "paint" the individual cells. Step 7: if the header has multiple lines and/or needs to be transposed, fix it up now. Step 8: transpose the table for final display. we build the table by column, but it needs to be displayed by row. if the table should be transposed, don't rotate it - clean up the headers. Step 9: pass the table string to our console or html routine for final output.

**TODO:** reduce the measuring and building to one pass over the data. currently rendering time is not an issue. ~15ms

**build_result_table_new**(*header_tups*, *main_data*, *nrows*, *keys=None*, *sortkeys=None*, *from_str=False*, *sorted_row_idx=None*, *transpose_on=False*, *row_numbers=None*, *right_cols=None*, *footer_tups=None*, *badcols=None*, *badrows=None*, *styles=None*, *callback=None*)

**callback: func, default None**
callback to signature

**build_row_mask**(*head*, *tail*, *total*)

**build_transposed_columns**(*columns*)

Transposed column data needs to be constructed differently. Widths will be calculated as a maximum items in multiple arrays. At the end of the table's construction, it will remain as a list of rows.

**classmethod console_detect_settings**()

For debugging console display.

**static display_detect**()

Call to redetect the display mode. This is useful when launching a qtconsole from jupyter lab.

**static display_html**(*html=None*)

**Parameters**
**html** (*defaults to None. Set to True to force html.*) – set to None to return the current mode.

**static display_precision**(*precision=2*)

**Parameters**
**precision** (*defaults to 2. How many places after the decimal to display.*) – set to None to return the current precision.

### Examples

rt.display_precision(4)

**static display_rows**(*rows=None*)

**Parameters**

- **rows** (*defaults to None. How many top and bottom rows to display in a Dataset.*) – set to None to return the current rows.
- **Display.options.HEAD_ROWS/TAIL_ROWS** (*Contolled by*) –

**See also:**

Display.options.TAIL_ROWS, Display.options.HEAD_ROWS

### Examples

rt.display_rows(20)

**static display_threshold**(*threshold=6*)

**Parameters**
**precision** (*defaults to 6. How many powers of 10 before flipping to scientific notation.*) – set to None to return the current threshold.

### Notes

E_THRESHOLD = 6 # power of 10 at which the float flips to scientific notation 10**+/- E_PRECISION = 3 # number of digits to display to the right of the decimal (sci notation)

### Examples

rt.display_threshold(6)

**fit_max_columns**(*headers*, *columns*, *total_width*, *console_width*, *footers=None*)

The display will attempt to fit as many columns as possible into the console. HTML display has been assigned a default value for self._console_x (see DisplayTable.__init__)

If the user changes their self.options.COL_ALL to True, all columns will be displayed on the same line. **Note:** this will break console display for large tables and should only be used in jupyter lab now.

**in progress** If the user requested all columns to be shown - regardless of width, the display will split them up into separate views with the maximum columns per line.

**fix_multiline_footers**(*plain=False*, *badcols=None*, *badrows=None*)

**fix_multiline_headers**()

Fixes multi-line headers if a column break was present. cell_spans in ColHeader might need to be changed. Need use cases for more than two lines, but the same loop should work.

**fix_repeated_keys**(*columns*, *repeat_string='.'*)

Display a different string when the first column of a multikey groupby is repeated. TODO: add support for the same behavior with repeated keys in multiple columns.

**footers_to_string**(*footer_row*)

Takes row of footer tuples and turns into string list. For adding/styling multiline footers.

**get_bad_color**()

put in the bad_col dictionary

**get_sort_col_idx**(*col_names*)

**class** riptable.rt_display.**DisplayText**(*text*)

Bases: object

Only uses two colors: green and purple OR cyan and blue For HTML

ds = rt.Dataset({'test': rt.arange(10)}) schema = {'Description': 'This is a structure', 'Steward': 'Nick'} ds.apply_schema(schema) ds.info()

**ESC = '\x1b['**

**HEADER_DARK = '1;36m'**

**HEADER_LIGHT = '1;34m'**

**RESET = '\x1b[00m'**

**TITLE_DARK = '1;32m'**

**TITLE_LIGHT = '1;35m'**

**__repr__**()
Return repr(self).

**__str__**()
Return str(self).

**static _as_if_dark**()

**static _format**(*txt*, *fmt*)

**static _header_color**()

**_repr_html_**()

**static _title_color**()

**static header_format**(*txt*)

**static title_format**(*txt*)

### 2.2.13 riptable.rt_ema

### 2.2.14 riptable.rt_enum

#### Classes

| | |
|---|---|
| *DATETIME_TYPES* | Enum where members are also (and must be) ints |
| *DS_DISPLAY_TYPES* | Enum where members are also (and must be) ints |
| *DisplayArrayTypes* | Enum where members are also (and must be) ints |
| *DisplayColumnColors* | Enum where members are also (and must be) ints |
| *DisplayDetectModes* | Enum where members are also (and must be) ints |
| *DisplayJustification* | Enum where members are also (and must be) ints |
| *DisplayLength* | Enum where members are also (and must be) ints |
| *GB_FUNCTIONS* | Enum where members are also (and must be) ints |
| *MATH_OPERATION* | MATH_OPERATION is the encoding of the Riptable implemented mathematical operations. |
| *NumpyCharTypes* | |
| *REDUCE_FUNCTIONS* | Enum where members are also (and must be) ints |
| *ROLLING_FUNCTIONS* | Enum where members are also (and must be) ints |
| *SD_TYPES* | Enum where members are also (and must be) ints |
| *SM_DTYPES* | Enum where members are also (and must be) ints |
| *TIMEWINDOW_FUNCTIONS* | Enum where members are also (and must be) ints |
| *TypeRegister* | When special classes are loaded, they register with this class to avoid cyclical dependencies |

### Attributes

| |
|---|
| *ColHeader* |
| *INVALID_DICT* |
| *INVALID_DICT* |

**class** riptable.rt_enum.**DATETIME_TYPES**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

**ORDINAL_DATE = 1**

**class** riptable.rt_enum.**DS_DISPLAY_TYPES**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

**HTML = 1**

**REPR = 2**

**STR = 3**

**class** riptable.rt_enum.**DisplayArrayTypes**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

**Bool = 0**

**Bytes = 3**

**Categorical = 4**

**DateTime = 6**

**DateTimeBase = 7**

**DateTimeNano = 9**

**Float = 2**

**Integer = 1**

**Record = 11**

**String = 5**

**TimeSpan = 10**

**class** riptable.rt_enum.**DisplayColumnColors**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Accum2t = 8

BGColor = 14

DarkBlue = 13

Default = 0

FGColor = 15

GrayItalic = 12

Groupby = 3

Multiset_col_a = 6

Multiset_col_b = 7

Multiset_head_a = 4

Multiset_head_b = 5

Pink = 10

Purple = 9

Red = 11

Rownum = 1

Sort = 2

class riptable.rt_enum.DisplayDetectModes

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Console = 3

HTML = 5

Ipython = 2

Jupyter = 1

class riptable.rt_enum.DisplayJustification

Bases: enum.IntEnum

Enum where members are also (and must be) ints

Center = 3

Left = 1

Right = 2

Undefined = 0

class riptable.rt_enum.DisplayLength

Bases: enum.IntEnum

Enum where members are also (and must be) ints

**Long = 3**

**Medium = 2**

**Short = 1**

**Undefined = 0**

**class** riptable.rt_enum.**GB_FUNCTIONS**

Bases: enum.IntEnum

Enum where members are also (and must be) ints

**GB_CUMMAX = 308**

**GB_CUMMIN = 309**

**GB_CUMNANMAX = 306**

**GB_CUMNANMIN = 307**

**GB_CUMPROD = 302**

**GB_CUMSUM = 300**

**GB_EMADECAY = 301**

**GB_EMANORMAL = 304**

**GB_EMAWEIGHTED = 305**

**GB_FINDNTH = 303**

**GB_FIRST = 100**

**GB_LAST = 102**

**GB_MAX = 3**

**GB_MEAN = 1**

**GB_MEDIAN = 103**

**GB_MIN = 2**

**GB_MODE = 104**

**GB_NANMAX = 53**

**GB_NANMEAN = 51**

**GB_NANMIN = 52**

**GB_NANSTD = 55**

**GB_NANSUM = 50**

**GB_NANVAR = 54**

**GB_NTH = 101**

GB_QUANTILE_MULT = 106

GB_ROLLING_COUNT = 204

GB_ROLLING_DIFF = 202

GB_ROLLING_MEAN = 205

GB_ROLLING_NANMEAN = 206

GB_ROLLING_NANSUM = 201

GB_ROLLING_QUANTILE = 207

GB_ROLLING_SHIFT = 203

GB_ROLLING_SUM = 200

GB_STD = 5

GB_SUM = 0

GB_TRIMBR = 105

GB_VAR = 4

**class** riptable.rt_enum.**MATH_OPERATION**

Bases: enum.IntEnum

MATH_OPERATION is the encoding of the Riptable implemented mathematical operations.

ABS = 201

ADD = 1

BITWISE_AND = 503

BITWISE_ANDNOT = 506

BITWISE_LSHIFT = 501

BITWISE_NOT = 218

BITWISE_NOTAND = 507

BITWISE_OR = 505

BITWISE_RSHIFT = 502

BITWISE_XOR = 504

BITWISE_XOR_SPECIAL = 550

CBRT = 308

CEIL = 206

CMP_EQ = 401

CMP_GT = 404

CMP_GTE = 406

CMP_LT = 403

CMP_LTE = 405

CMP_NE = 402

DIV = 101

EXP = 216

EXP2 = 217

EXPM1 = 305

FABS = 203

FLOOR = 205

FLOORDIV = 10

FMOD = 13

INVERT = 204

ISFINITE = 605

ISINF = 603

ISNAN = 604

ISNANORZERO = 611

ISNORMAL = 606

ISNOTFINITE = 609

ISNOTINF = 607

ISNOTNAN = 608

ISNOTNORMAL = 610

LOG = 302

LOG10 = 304

LOG1P = 306

LOG2 = 303

LOGICAL_AND = 407

LOGICAL_NOT = 601

LOGICAL_OR = 409

LOGICAL_XOR = 408

MAX = 7

MIN = 6

MOD = 5

MUL = 3

NANMAX = 9

NANMIN = 8

NEG = 202

NEGATIVE = 212

POSITIVE = 213

POWER = 11

RECIPROCAL = 309

REMAINDER = 12

RINT = 215

ROUND = 208

SIGN = 214

SIGNBIT = 612

SQRT = 301

SQUARE = 307

SUB = 2

SUBDATES = 103

SUBDATETIMES = 102

TRUNC = 207

class riptable.rt_enum.NumpyCharTypes

All = '?bhilqpBHILQPefdgFDGSUVOMm'

AllFloat = 'efdgFDG'

AllInteger = 'bBhHiIlLqQpP'

Character = 'c'

Complex = 'FDG'

Computable = 'fdgbBhHiIlLqQpP'

Datetime = 'Mm'

Float = 'efdg'

Float64 = 'dg'

Integer = 'bhilqp'

Noncomputable = 'SeFDGUVOMm'

SignedInteger64 = 'qp'

Supported = '?fdgbBhHiIlLqQpPSUV'

SupportedAlternate = '?fdgbBhHiIlLqQpPSU'

SupportedFloat = 'fdg'

UnsignedInteger = 'BHILQP'

UnsignedInteger64 = 'QP'

Unsupported = 'eFDGVOMm'

class riptable.rt_enum.REDUCE_FUNCTIONS

Bases: enum.IntEnum

Enum where members are also (and must be) ints

REDUCE_ALL = 209

REDUCE_ANY = 208

REDUCE_ARGMAX = 206

REDUCE_ARGMIN = 204

REDUCE_MAX = 202

REDUCE_MEAN = 102

REDUCE_MIN = 200

REDUCE_NANARGMAX = 207

REDUCE_NANARGMIN = 205

REDUCE_NANMAX = 203

REDUCE_NANMEAN = 103

REDUCE_NANMIN = 201

REDUCE_NANSTD = 109

REDUCE_NANSUM = 1

REDUCE_NANVAR = 107

REDUCE_STD = 108

REDUCE_SUM = 0

REDUCE_VAR = 106

class riptable.rt_enum.ROLLING_FUNCTIONS

Bases: enum.IntEnum

Enum where members are also (and must be) ints

ROLLING_MEAN = 102

ROLLING_NANMEAN = 103

ROLLING_NANSTD = 109

ROLLING_NANSUM = 1

ROLLING_NANVAR = 107

ROLLING_QUANTILE = 104

ROLLING_STD = 108

ROLLING_SUM = 0

ROLLING_VAR = 106

class riptable.rt_enum.SD_TYPES

Bases: enum.IntEnum

Enum where members are also (and must be) ints

SD_CELL = 8

SD_CHAR = 6

SD_CLASS = 3

SD_DATASET = 2

SD_FUNCTIONH = 1

SD_LOGICAL = 7

SD_NUMERIC = 9

SD_NUMPY = 21

SD_PANDAS = 20

SD_SCALAR = 5

SD_STRUCT = 4

SD_UNKNOWN = 0

SD_VECTOR = 10

class riptable.rt_enum.SM_DTYPES

Bases: enum.IntEnum

Enum where members are also (and must be) ints

DT_BOOL = 1

DT_BYTE = 2

DT_BYTES = 18

DT_CHARARRAY = 24

DT_DATETIME64 = 21

DT_FLOAT16 = 11

DT_FLOAT32 = 12

DT_FLOAT64 = 13

DT_HALF = 23

DT_INT16 = 4

DT_INT32 = 5

DT_INT64 = 6

DT_INT8 = 3

DT_INVALID = 0

DT_NPVOID = 20

DT_OBJECT = 17

DT_TIMEDELTA64 = 22

DT_UINT16 = 8

DT_UINT32 = 9

DT_UINT64 = 10

DT_UINT8 = 7

DT_UNICODE = 19

class riptable.rt_enum.TIMEWINDOW_FUNCTIONS

Bases: enum.IntEnum

Enum where members are also (and must be) ints

TIMEWINDOW_PROD = 1

TIMEWINDOW_SUM = 0

class riptable.rt_enum.TypeRegister

When special classes are loaded, they register with this class to avoid cyclical dependencies

Accum2: ClassVar[Type[TypeRegister.Accum2]]

Calendar: ClassVar[Type[TypeRegister.Calendar]]

Categorical: ClassVar[Type[TypeRegister.Categorical]]

Categories: ClassVar[Type[TypeRegister.Categories]]

Dataset: ClassVar[Type[TypeRegister.Dataset]]

Date: ClassVar[Type[TypeRegister.Date]]

DateBase: ClassVar[Type[TypeRegister.DateBase]]

DateSpan: ClassVar[Type[TypeRegister.DateSpan]]

DateTimeBase: ClassVar[Type[TypeRegister.DateTimeBase]]

DateTimeNano: ClassVar[Type[TypeRegister.DateTimeNano]]

DisplayAttributes: ClassVar[Type[TypeRegister.DisplayAttributes]]

DisplayDetect: ClassVar[Type[TypeRegister.DisplayDetect]]

DisplayOptions: ClassVar[Type[TypeRegister.DisplayOptions]]

DisplayString: ClassVar[Type[TypeRegister.DisplayString]]

DisplayTable: ClassVar[Type[TypeRegister.DisplayTable]]

DisplayText: ClassVar[Type[TypeRegister.DisplayText]]

FastArray: ClassVar[Type[TypeRegister.FastArray]]

GroupBy: ClassVar[Type[TypeRegister.GroupBy]]

Grouping: ClassVar[Type[TypeRegister.Grouping]]

MathLedger: ClassVar[Type[TypeRegister.MathLedger]]

Multiset: ClassVar[Type[TypeRegister.Multiset]]

PDataset: ClassVar[Type[TypeRegister.PDataset]]

SharedMemory: ClassVar[Type[TypeRegister.SharedMemory]]

SortCache: ClassVar[Type[TypeRegister.SortCache]]

Struct: ClassVar[Type[TypeRegister.Struct]]

TimeSpan: ClassVar[Type[TypeRegister.TimeSpan]]

TimeZone: ClassVar[Type[TypeRegister.TimeZone]]

classmethod as_meta_data(*obj*)

classmethod from_meta_data(*itemdict=None*, *flags=None*, *meta=''*)

classmethod is_array_subclass(*arr*)

Certain routines can be sped up by skipping the logic before falling back on a numpy call. Note: this is different than using python's issubclass(), which returns True if the classes are the same. Returns True if the item is an instance of a FastArray or numpy array subclass.

classmethod is_binned_array(*arr*)

Use this instead of checking isinstance(item, TypeRegister.Categorical). For other binned types in the future.

Called by: Dataset.melt() -re-expands Dataset.from_jagged_rows() -re-expands GroupBy.__init__ -calls grouping, gb_keychain properties to borrow bins

**classmethod is_binned_type**(*arrtype*)

Check the type rather than the instance. See also is_binned_array()

Called by: rt_utils._multistack_items()

**classmethod is_computable**(*other*)

**classmethod is_datelike**(*arr*)

return True if it is a date or time

**classmethod is_spanlike**(*arr*)

return True if it is a datespan or timespan

**classmethod is_string_or_object**(*arr*)

**classmethod newclassfrominstance**(*instance*, *origin*)

After slicing or an array routine, return a new instance of a FastArray subclass. If the array was not a subclass, instance is unchanged.

**Parameters**

- **instance** (*ndarray*) – Array generated from operation.
- **origin** (*ndarray*) – Array, possibly a FastArray subclass.

**Returns**

**instance** – Array of the same class as origin if the origin class has a newclassfrominstance defined.

**Return type**

ndarray

**classmethod validate_registry**()

riptable.rt_enum.**ColHeader**

riptable.rt_enum.**INVALID_DICT: Mapping[int, Any]**

riptable.rt_enum.**INVALID_DICT: Mapping[int, Any]**

### 2.2.15 riptable.rt_fastarray

#### Classes

| | |
|---|---|
| *FastArray* | A *FastArray* is a 1-dimensional array of items that are the same data type. |
| *Ledger* | |
| *Recycle* | |
| *Threading* | |

class riptable.rt_fastarray.**FastArray**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: numpy.ndarray

A *FastArray* is a 1-dimensional array of items that are the same data type.

Because it's a subclass of NumPy's numpy.ndarray, all ndarray functions and attributes can be used with *FastArray* objects. However, Riptable optimizes many of NumPy's functions to make them faster and more memory-efficient. Riptable has also added some methods.

*FastArray* objects with more than 1 dimension are not supported.

See NumPy's docs for details on all ndarray methods and attributes.

**Parameters**

- **arr** (*array, iterable, or scalar value*) – Contains data to be stored in the *FastArray*.
- ****kwargs** – Additional keyword arguments to be passed to the function.

## Notes

To improve performance, *FastArray* objects take over some of NumPy's universal functions (ufuncs), use array recycling and multiple threads, and pass certain method calls to Bottleneck.

Note that whenever Riptable has implemented its own version of an existing NumPy method, a call to the NumPy method results in a call to the optimized Riptable version instead. We encourage users to directly call the Riptable method in order to avoid any confusion as to what method is actually being called.

See the list of NumPy Methods Optimized by Riptable for FastArrays.

## Examples

### Construct a FastArray

Pass a list to the constructor:

```
>>> rt.FastArray([1, 2, 3, 4, 5])
FastArray([1, 2, 3, 4, 5])
```

```
>>> #NOTE: rt.FA also works.
>>> rt.FA([1.0, 2.0, 3.0, 4.0, 5.0])
FastArray([1., 2., 3., 4., 5.])
```

Or use a utility function:

```
>>> rt.full(10, 0.7)
FastArray([0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7, 0.7])
```

```
>>> rt.arange(10)
FastArray([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You can optionally specify a data type:

```
>>> x = rt.FastArray([3, 6, 10],  dtype = rt.float64)
>>> x, x.dtype
(FastArray([ 3.,  6., 10.]), dtype('float64'))
```

```
>>> # Using a string shortcut:
>>> x = rt.FastArray([3,6,10],  dtype = 'float64')
>>> x, x.dtype
(FastArray([ 3.,  6., 10.]), dtype('float64'))
```

By default, characters are stored as byte strings. When `unicode=True`, the *FastArray* allows Unicode characters.

```
>>> rt.FA(list('abc'), unicode=True)
FastArray(['a', 'b', 'c'], dtype='<U1')
```

To convert an existing NumPy array, use the *FastArray* constructor.

```
>>> np_arr = np.array([1, 2, 3])
>>> rt.FA(np_arr)
FastArray([1, 2, 3])
```

To view the NumPy array as a *FastArray* (which is slightly less expensive than using the constructor), use the `view` method.

```
>>> fa = np_arr.view(FA)
>>> fa
FastArray([1, 2, 3])
```

To view it as a NumPy array again:

```
>>> fa.view(np.ndarray)
array([1, 2, 3])
```

```
>>> # Alternatively:
>>> fa._np
array([1, 2, 3])
```

**Get a Subset of a FastArray**

You can use standard Python slicing notation or fancy indexing to access a subset of a *FastArray*.

```
>>> # Create a FastArray:
>>> array = rt.arange(8)**2
>>> array
FastArray([0, 1, 4, 9, 16, 25, 36, 49])
>>> # Use Python slicing to get elements 2, 3, and 4:
>>> array[2:5]
FastArray([4, 9, 16])
```

```
>>> # Use fancy indexing to get elements 2, 4, and 1 (in that order):
>>> array[[2, 4, 1]]
FastArray([4, 16, 1])
```

For more details, see the examples for 1-dimensional arrays in NumPy's docs: Indexing on ndarrays.

Note that slicing creates a view of the array and does not copy the underlying data; modifying the slice modifies the original array. Fancy indexing creates a copy of the extracted data; modifying this array does not modify the original array.

You can also pass a Boolean mask array.

```
>>> # Create a Boolean mask:
>>> evenMask = (array % 2 == 0)
>>> evenMask
FastArray([True, False, True, False, True, False, True, False])
>>> # Index using the Boolean mask:
>>> array[evenMask]
FastArray([0, 4, 16, 36])
```

**How to Subclass FastArray**

Include the required class definition:

```
>>> class TestSubclass(FastArray):
...     def __new__(cls, arr, **args):
...         # Before this call, arr needs to be a np.ndarray instance.
...         return arr.view(cls)
...     def __init__(self, arr, **args):
...         pass
```

If the subclass is computable, you might define your own math operations. In these operations, you might define what the subclass can be computed with. For examples of new definitions, see the `DateTimeNano` class.

Common operations to hook are comparisons (`__eq__()`, `__ne__()`, `__gt__()`, `__lt__()`, `__le__()`, `__ge__()`) and basic math functions (`__add__()`, `__sub__()`, `__mul__()`, etc.).

Bracket indexing operations are very common. If the subclass needs to set or return a value other than that in the underlying array, you need to take over *`__getitem__()`* or *`__setitem__()`*.

Indexing is also used in display. For regular console/notebook display, you need to take over:

- `__repr__()`
- `__str__()`
- `_repr_html_()` (for JupyterLab and Jupyter notebooks)

If the array is being displayed in a `Dataset` and you require certain formatting, you need to define two more methods:

**`display_query_properties()`**
: Returns an `ItemFormat` object (see `rt.Utils.rt_display_properties`)

**`display_convert_func()`**
: The conversion function returned by `display_query_properties()` must return a string. Each item being displayed, the result of `__getitem__()` at a single index, will go through this function individually, accompanied by an `ItemFormat` object.

Many Riptable operations need to return arrays of the same class they received. To ensure that your subclass will retain its special properties, you need to take over `newclassfrominstance()`. Failure to take this over will often result in an object with uninitialized variables.

*`copy()`* is another method that is called generically in Riptable routines, and needs to be taken over to retain subclass properties.

For a view of the underlying *`FastArray`*, you can use the `_fa` property.

class _ArrayFunctionHelper

Array function helper is responsible maintaining the array function protocol array implementations in the form of the following API:

- get_array_function: given the Numpy function, returns overridden array function
- get_array_function_type_compatibility_check: given the Numpy function, returns overridden array function type compatibility check
- register_array_function: a function decorator whose argument is the Numpy function to override and the function that will override it
- register_array_function_type_compatibility: similar to register_array_function, but guards against incompatible array function protocol type arguments for the given Numpy function
- deregister: deregistration of the Numpy function and type compatibility override
- deregister_array_function_type_compatibility: deregistration of Numpy function type compatibility override

HANDLED_FUNCTIONS: Dict[callable, callable]

Dictionary of Numpy API function with overridden functions.

HANDLED_TYPE_COMPATIBILITY_CHECK: Dict[callable, callable]

Dictionary of type compatibility functions per each Numpy API overridden function.

classmethod deregister(*np_function*)

classmethod deregister_array_function(*np_function*)

Deregistration of the Numpy function and type compatibility override.

**Parameters**
**np_function** (*callable*) – The overridden Numpy array function.

classmethod deregister_array_function_type_compatibility(*np_function*)

Deregistration of the Numpy function and type compatibility override.

**Parameters**
**np_function** (*callable*) – The overridden Numpy array function.

classmethod get_array_function(*np_function*)

Given the Numpy function, returns overridden array function if implemented, otherwise None.

**Parameters**
**np_function** (*callable*) – The overridden Numpy array function.

**Returns**
The overridden function as a callable or None if it's not implemented.

**Return type**
callable, optional

classmethod get_array_function_type_compatibility_check(*np_function*)

Given the Numpy function, returns the corresponding array function type compatibility callable, otherwise None.

**Parameters**
**np_function** (*callable*) – The overridden Numpy array function.

**Returns**
The overridden type compatibility function as a callable or None if it's not implemented.

**Return type**
callable, optional

**classmethod register_array_function**(*np_function*)

A function decorator whose argument is the Numpy function to override and the function that will override it. This registers the `np_function` with the function that it decorates.

**Parameters**
**np_function** (*callable*) – The overridden Numpy array function.

**Returns**
The decorator that registers `np_function` with the decorated function.

**Return type**
callable

**classmethod register_array_function_type_compatibility**(*np_function*)

This registers the type compatibility check for the `np_function` with the function that it decorates.

**Parameters**
**np_function** (*callable*) – The overridden Numpy array function.

**Returns**
The decorator that registers the type compatibility check for the `np_function` with the decorated function.

**Return type**
callable

**property _np: numpy.ndarray**

Return a NumPy array view of the input *FastArray*.

**Returns**
A NumPy array view of the input *FastArray*.

**Return type**
numpy.ndarray

**See also:**

**numpy.ndarray.view**
Can be used to view a NumPy array as a *FastArray*.

### Examples

Return a NumPy array view for an integer *FastArray*:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a._np
array([1, 2, 3, 4, 5])
```

Changes to the view are reflected in the original *FastArray*:

```
>>> npview = a._np
>>> npview[2] = 10
>>> a
FastArray([ 1,  2, 10,  4,  5])
```

To view a NumPy array as a *FastArray*, you can use numpy.ndarray.view:

```
>>> npview.view(rt.FastArray)
FastArray([1, 2, 10, 4, 5])
```

**property crc: int**

Calculate the 32-bit CRC of the data in this array using the Castagnoli polynomial (CRC32C).

This function does not consider the array's shape or strides when calculating the CRC, it simply calculates the CRC value over the entire buffer described by the array.

### Examples

can be used to compare two arrays for structural equality >>> a = arange(100) >>> b = arange(100.0) >>> a.crc == b.crc False

**property doc**

Return the *Doc* object for the input *FastArray*.

If no *Doc* object exists, return None.

**Returns**
The *Doc* object for the input *FastArray*. If no *Doc* object exists, return None.

**Return type**
*Doc*

**See also:**

***FastArray.info***
Return a description of the input array's contents.

***apply_schema***
Set *Doc* object values.

### Examples

No *Doc* object exists:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> print(a.doc)
None
```

Apply a schema and return the *Doc* object:

```
>>> schema = {"Description": "This is an array", "Steward": "Brian", "Type":
↪"int32"}
>>> a.apply_schema(schema)
{}
>>> a.doc
```

(continues on next page)

(continued from previous page)

```
Description: This is an array
Steward: Brian
Type: int32
```

Return specific *Doc* object information:

```
>>> a.doc._type
'int32'
>>> a.doc._descrip
'This is an array'
>>> a.doc._steward
'Brian'
>>> print(a.doc._detail)
None
```

**property inv: Any**

Return the invalid value for the input array's data type.

**Returns**

The invalid value for the input array's dtype. For example, `int8` returns -128, `uint8` returns 255, and `bool_` returns `False`.

**Return type**

Any

**See also:**

***FastArray.copy_invalid***

Return a copy of a *FastArray* filled with the invalid value for the array's dtype.

***FastArray.fill_invalid***

Replace the values of a *FastArray* with the invalid value for the array's dtype.

***INVALID_DICT***

A mapping of invalid values to dtypes.

### Examples

Return the invalid value for an integer array:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a.inv
-2147483648
```

Return the invalid value for a floating-point array:

```
>>> a2 = rt.FA([0., 1., 2., 3., 4.])
>>> a2
FastArray([0., 1., 2., 3., 4.])
>>> a2.inv
nan
```

Return the invalid value for a string array:

```
>>> a3 = rt.FA(["AMZN", "IBM", "MSFT", "AAPL"])
>>> a3
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a3.inv
b''
```

**property numbastring**

converts byte string and unicode strings to a 2dimensional array so that numba can process it correctly

### Examples

```
>>> @numba.jit(nopython=True)
... def numba_str(txt):
...     x=0
...     for i in range(txt.shape[0]):
...         if (txt[i,0]==116 and  # 't'
...             txt[i,1]==101 and  # 'e'
...             txt[i,2]==120 and  # 'x'
...             txt[i,3]==116):    # 't'
...             x += 1
...     return x
>>>
>>> x=FastArray(['some','text','this','is'])
>>> numba_str(x.view(np.uint8).reshape((len(x), x.itemsize)))
>>> numba_str(x.numbastring)
```

**CompressPickle = True**

**FasterUFunc = True**

**MAX_DISPLAY_LEN = 10**

**NEW_ARRAY_FUNCTION_ENABLED = False**

Enable implementation of array function protocol (default False).

**NoTolerance = False**

**Recycle = True**

**SafeConversions = True**

**Verbose = 1**

**WarningDict**

**WarningLevel = 1**

**_reduce_op_identity_value: Mapping[*riptable.rt_enum.REDUCE_FUNCTIONS*, Any]**

**add**

**div**

**floordiv**

**mod**

**mul**

**pow**

**sub**

**static _FastFunctionsOff**()

**static _FastFunctionsOn**()

**static _GCNOW**(*timeout=0*)

Pass the garbage collector timeout value to cleanup. Passing 0 will force an immediate garbage collection.

**Return type**
Dictionary of memory heuristics including 'TotalDeleted'

**static _GCSET**(*timeout=100*)

Pass the garbage collector timeout value to expire The timeout value is roughly in 2/5 secs A value of 100 is usually about 40 seconds

**Return type**
Previous timespan

**static _LCLEAR**()

Clear all the entries in the math ledger

**static _LDUMP**(*dataset=True*)

Print out the math ledger

**static _LDUMPF**(*filename*)

Save the math ledger to a file

**static _LOFF**()

Turn the math ledger off

**static _LON**()

Turn the math ledger on to record all array math routines

**static _OFF**()

disable intercepting of array ufunc

**static _ON**()

enable intercepting array ufunc

**static _RDUMP**()

Displays to server's stdout

**Return type**
Total size of items not in use

**static _ROFF**(*quiet=False*)

Turn off recycling.

**Parameters**
**quiet** (*bool, optional*) –

**Return type**
True if recycling was previously on, else False

**static _RON**(*quiet=False*)

Turn on recycling.

**Parameters**
**quiet** (*bool, optional*) –

**Return type**
True if recycling was previously on, else False

**static _TOFF**()

**static _TON**()

**static _V0**()

**static _V1**()

**static _V2**()

**__array_finalize__**(*obj*)

Finalizes self from other, called as part of ndarray.__new__()

**__array_function__**(*func*, *types*, *args*, *kwargs*)

**__array_ufunc__**(*ufunc*, *method*, **inputs*, ***kwargs*)

The FastArray universal function (or ufunc) override offers multithreaded C/C++ implementation at the RiptideCPP layer.

**When FastArray receives a ufunc callable it will attempt to handle it in priority order:**

1. considering `FastArray` `FastFunction` is enabled, ufunc is handled by an explicit ufunc override, otherwise
2. ufunc is handled at the Riptable / Numpy API overrides level, otherwise
3. ufunc is handled at the Numpy API level.

Given a combination of `ufunc`, `inputs`, and `kwargs`, if neither of the aforementioned cases support this then a warning is emitted.

**The following references to supported ufuncs are grouped by method type.**

- For `method` type `reduce`, see `gReduceUFuncs`.
- For `method` type `__call__`, see `gBinaryUFuncs`, `gBinaryLogicalUFuncs`, `gBinaryBitwiseUFuncs`, and `gUnaryUFuncs`.
- For `method` type `at` return `None`.

If `out` argument is specified, then an extra array copy is performed on the result of the ufunc computation.

If a `dtype` keyword is specified, all efforts are made to respect the `dtype` on the result of the computation.

**Parameters**

- **ufunc** (*callable*) – The ufunc object that was called.
- **method** (*str*) – A string indicating which Ufunc method was called (one of "__call__", "reduce", "reduceat", "accumulate", "outer", "inner").
- **inputs** – A tuple of the input arguments to the ufunc.
- **kwargs** – A dictionary containing the optional input arguments of the ufunc. If given, any out arguments, both positional and keyword, are passed as a tuple in kwargs.

**Return type**
The method should return either the result of the operation, or NotImplemented if the operation requested is not implemented.

### Notes

The current implementation does not support the following keyword arguments: `casting`, `sig`, `signature`, and `core_signature`.

It has partial support for keyword arguments: *where*, `axis`, and `axes`, if they match the default values.

If FastArray's `WarningLevel` is enabled, then warnings will be emitted if any of unsupported or partially supported keyword arguments are passed.

TODO document custom up casting rules.

**See also:**

**For**

- https://numpy.org/doc/stable/reference/arrays.classes.html#numpy.class.__array_ufunc__ - https://numpy.org/doc/stable/reference/ufuncs.html

`Note`, `the`, `None`

**__arrow_array__**(*type=None*)

Implementation of the `__arrow_array__` protocol for conversion to a pyarrow array.

**Parameters**
**type** (*pyarrow.DataType, optional, defaults to None*) –

**Return type**
pyarrow.Array or pyarrow.ChunkedArray

### Notes

https://arrow.apache.org/docs/python/extending_types.html#controlling-conversion-to-pyarrow-array-with-the-arrow-array

**__eq__**(*other*)

Return self==value.

**__ge__**(*other*)

Return self>=value.

**__getitem__**(*fld*)

riptable has special routines to handle array input in the indexer. Everything else will go to numpy getitem.

**__gt__**(*other*)

Return self>value.

**__le__**(*other*)

Return self<=value.

**__lt__**(*other*)

Return self<value.

**__ne__**(*other*)

Return self!=value.

**__reduce__**()

Used for pickling. For just a FastArray we pass back the view of the np.ndarray, which then knows how to pickle itself. NOTE: I think there is a faster way.. possible returning a byte string.

**__setitem__**(*fld*, *value*)

Used on the left hand side of arr[fld] = value

This routine tries to convert invalid dtypes to that invalids are preserved when setting The mbset portion of this is no written (which will not raise an indexerror on out of bounds)

**Parameters**

- **fld** (*scalar, boolean, fancy index mask, slice, sequence, or list*) –
- **value** (*scalar, sequence or dataset value as follows*) – sequence can be list, tuple, np.ndarray, FastArray

**Raises**

**IndexError** –

**static _argmax**(*a*, *axis=None*, *out=None*)

**static _argmin**(*a*, *axis=None*, *out=None*)

**classmethod _check_ndim**(*instance*)

Iterates through dimensions of an array, counting how many dimensions have values greater than 1. Problems may occure with multidimensional FastArrays, and the user will be warned.

**_compare_check**(*func*, *other*)

**static _empty_like**(*array*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

**_fa_filter_wrapper**(*myFunc*, *filter=None*, *dtype=None*)

**_fa_keyword_wrapper**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, *ddof=None*, ***kwargs*)

**_fill_invalid_internal**(*shape=None*, *dtype=None*, *inplace=True*, *fill_val=None*)

**static _from_arrow**(*arr*, *zero_copy_only=True*, *writable=False*, *auto_widen=False*)

Convert a pyarrow `Array` to a riptable *FastArray*.

**Parameters**

- **arr** (*pyarrow.Array or pyarrow.ChunkedArray*) –
- **zero_copy_only** (*bool, default True*) – If True, an exception will be raised if the conversion to a *FastArray* would require copying the underlying data (e.g. in presence of nulls, or for non-primitive types).
- **writable** (*bool, default False*) – For a *FastArray* created with zero copy (view on the Arrow data), the resulting array is not writable (Arrow data is immutable). By setting this to True, a copy of the array is made to ensure it is writable.
- **auto_widen** (*bool, optional, default to False*) – When False (the default), if an arrow array contains a value which would be considered the 'invalid'/NA value for the equivalent dtype in a *FastArray*, raise an exception because direct conversion would be lossy / change the semantic meaning of the data. When True, the converted array will be widened (if possible) to the next-largest dtype to ensure the data will be interpreted in the same way.

**Return type**
*FastArray*

**_internal_self_compare**(*math_op*, *periods=1*, *fancy=False*)
internal routine used for differs and transitions

**_is_not_supported**(*arr*)
returns True if a numpy array is not FastArray internally supported

**_kwarg_check**(**args*, ***kwargs*)

**_legacy_array_function**(*func*, *types*, *args*, *kwargs*)
Called before array_ufunc. Does not get called for every function np.isnan/trunc/true_divide for instance.

**static _max**(*a*, *axis=None*, *out=None*, *keepdims=None*, *initial=None*, *where=None*)

**static _mean**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*)

**static _min**(*a*, *axis=None*, *out=None*, *keepdims=None*, *initial=None*, *where=None*)

**static _nanargmax**(*a*, *axis=None*)

**static _nanargmin**(*a*, *axis=None*)

**static _nanmax**(*a*, *axis=None*, *out=None*, *keepdims=None*)

**static _nanmean**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*)

**static _nanmin**(*a*, *axis=None*, *out=None*, *keepdims=None*)

**static _nanstd**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=None*, *keepdims=None*)

**static _nansum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*)

**static _nanvar**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=None*, *keepdims=None*)

**_new_array_function**(*func*, *types*, *args*, *kwargs*)
FastArray implementation of the array function protocol.

**Parameters**

- **func** (*callable*) – An callable exposed by NumPy's public API, which was called in the form `func(*args, **kwargs)`.
- **types** (*tuple*) – A tuple of unique argument types from the original NumPy function call that implement `__array_function__`.
- **args** (*tuple*) – The tuple of arguments that will be passed to `func`.
- **kwargs** (*dict*) – The dictionary of keyword arguments that will be passed to `func`.

**Raises**
**TypeError** – If `func` is not overridden by a corresponding riptable array function then a TypeError is raised.

### Notes

This array function implementation requires each class, such as FastArray and any other derived class, to implement their own version of the Numpy array function API. In the event these array functions defer to the inheriting class they will need to either re-wrap the results in the correct type or raise exception if a particular operation is not well-defined nor meaningful for the derived class. If an array function, which is also a universal function, is not overridden as an array function, but defined as a ufunc then it will not be called unless it is registered with the array function helper since array function protocol takes priority over the universal function protocol.

Reference: NEP 18 Array Function Protocol

**classmethod _possibly_warn**(*warning_string*)

**classmethod _py_number_to_np_dtype**(*val*, *dtype*)

Convert a python type to numpy dtype. Only handles integers.

**_reduce_check**(*reduceFunc*, *npFunc*, **args*, ***kwargs*)

Arg2: npFunc pass in None if no numpy equivalent function

**static _round_**(*a*, *decimals=None*, *out=None*)

**static _std**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=None*, *keepdims=None*)

**static _sum**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=None*, *initial=None*, *where=None*)

**_unary_op**(*funcnum*, *fancy=False*)

**static _var**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=None*, *keepdims=None*)

**_view_internal**(*type=None*)

FastArray subclasses need to take this over if they want to make a shallow copy of a fastarray instead of viewing themselves as a fastarray (which drops their other properties). Taking over view directly may have a lot of unintended consequences.

**abs**(***kwargs*)

**apply**(*pyfunc*, **args*, *otypes=None*, *doc=None*, *excluded=None*, *cache=False*, *signature=None*)

Generalized function class. see: np.vectorize

Creates and then applies a vectorized function which takes a nested sequence of objects or numpy arrays as inputs and returns an single or tuple of numpy array as output. The vectorized function evaluates `pyfunc` over successive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

The data type of the output of `vectorized` is determined by calling the function with the first element of the input. This can be avoided by specifying the `otypes` argument.

**Parameters**

- **pyfunc** (*callable*) – A python function or method.
- **otypes** (*str or list of dtypes, optional*) – The output data type. It must be specified as either a string of typecode characters or a list of data type specifiers. There should be one data type specifier for each output.
- **doc** (*str, optional*) – The docstring for the function. If `None`, the docstring will be the `pyfunc.__doc__`.

- **excluded** (*set, optional*) – Set of strings or integers representing the positional or keyword arguments for which the function will not be vectorized. These will be passed directly to `pyfunc` unmodified.

  New in version 1.7.0.

- **cache** (*bool, optional*) – If `True`, then cache the first function call that determines the number of outputs if `otypes` is not provided.

  New in version 1.7.0.

- **signature** (*string, optional*) – Generalized universal function signature, e.g., `(m,n),(n)->(m)` for vectorized matrix-vector multiplication. If provided, `pyfunc` will be called with (and expected to return) arrays with shapes given by the size of corresponding core dimensions. By default, `pyfunc` is assumed to take scalars as input and output.

  New in version 1.12.0.

**Returns**

**vectorized** – Vectorized function.

**Return type**

callable

**See also:**

*FastArray.apply_numba*, *FastArray.apply_pandas*

### Examples

```
>>> def myfunc(a, b):
...     "Return a-b if a>b, otherwise return a+b"
...     if a > b:
...         return a - b
...     else:
...         return a + b
>>>
>>> a=arange(10)
>>> b=arange(10)+1
>>> a.apply(myfunc,b)
FastArray([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

Example with one input array

```
>>> def square(x):
...     return x**2
>>>
>>> a=arange(10)
>>> a.apply(square)
FastArray([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Example with lambda

```
>>> a=arange(10)
>>> a.apply(lambda x: x**2)
FastArray([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Example with numba

```
>>> from numba import jit
>>> @jit
... def squareit(x):
...     return x**2
>>> a.apply(squareit)
FastArray([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Examples to use existing builtin oct function but change the output from string, to unicode, to object

```
>>> a=arange(10)
>>> a.apply(oct, otypes=['S'])
FastArray([b'0o0', b'0o1', b'0o2', b'0o3', b'0o4', b'0o5', b'0o6', b'0o7', b
↪'0o10', b'0o11'], dtype='|S4')
```

```
>>> a=arange(10)
>>> a.apply(oct, otypes=['U'])
FastArray(['0o0', '0o1', '0o2', '0o3', '0o4', '0o5', '0o6', '0o7', '0o10',
↪'0o11'], dtype='<U4')
```

```
>>> a=arange(10)
>>> a.apply(oct, otypes=['O'])
FastArray(['0o0', '0o1', '0o2', '0o3', '0o4', '0o5', '0o6', '0o7', '0o10',
↪'0o11'], dtype=object)
```

**apply_numba**(**args*, *otype=None*, *myfunc='myfunc'*, *name=None*)

Print to screen an example numba signature for the array.

You can then copy this example to build your own numba function.

**Parameters**

- ***args** – Test arguments
- **otype** (*str, default None*) – A different output data type
- **myfunc** (*str, default 'myfunc'*) – A string to call the function
- **name** (*str, default None*) – A string to name the array

### Examples

```
>>> import numba
>>> @numba.guvectorize(['void(int64[:], int64[:])'], '(n)->(n)')
... def squarev(x,out):
...     for i in range(len(x)):
...         out[i]=x[i]**2
...
>>> a=arange(1_000_000).astype(np.int64)
>>> squarev(a)
FastArray([           0,            1,            4, ..., 999994000009,
           999996000004, 999998000001], dtype=int64)
```

**apply_pandas**(*func*, *convert_dtype=True*, *args=()*, ***kwds*)

Invoke function on values of FastArray. Can be ufunc (a NumPy function that applies to the entire FastArray) or a Python function that only works on single values

**Parameters**

- **func** (*function*) –
- **convert_dtype** (*boolean, default True*) – Try to find better dtype for elementwise function results. If False, leave as dtype=object
- **args** (*tuple*) – Positional arguments to pass to function in addition to the value
- **function** (*Additional keyword arguments will be passed as keywords to the*) –

**Returns**

**y**

**Return type**

*FastArray* or Dataset if func returns a FastArray

**See also:**

***FastArray.map***

For element-wise operations

**FastArray.agg**

only perform aggregating type operations

**FastArray.transform**

only perform transforming type operations

### Examples

Create a FastArray with typical summer temperatures for each city.

```
>>> fa = rt.FastArray([20, 21, 12], index=['London', 'New York','Helsinki'])
>>> fa
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```
>>> def square(x):
...     return x**2
>>> fa.apply(square)
London      400
New York    441
Helsinki    144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```
>>> fa.apply(lambda x: x**2)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```
>>> def subtract_custom_value(x, custom_value):
...     return x-custom_value
>>> fa.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```

Define a custom function that takes keyword arguments and pass these arguments to `apply`.

```
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
...         x+=kwargs[month]
...     return x
>>> fa.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library.

```
>>> fa.apply(np.log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

**apply_schema**(*schema*)

Apply a schema containing descriptive information to the FastArray

**Parameters**
**schema** – dict

**Returns**
dictionary of deviations from the schema

**argmax**(***kwargs*)

**argmin**(***kwargs*)

**argpartition2**(**args*, ***kwargs*)

**astype**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Return a *FastArray* with values converted to the specified data type.

Check your results when you convert missing values. Sentinel values are preserved when Riptable handles the conversion. However, in some cases the array is sent to NumPy for conversion and results may not be what you expect.

For parameter descriptions, see `numpy.ndarray.astype()`. Note that until a reported bug is fixed, the `casting` parameter is ignored when Riptable handles the conversion.

**Returns**
A *FastArray* with values converted to the specified data type.

**Return type**
*FastArray*

**See also:**

`Dataset.astype`

### Examples

```
>>> a = rt.FastArray([1.7, 2.0, 3.0])
>>> a.astype(int)
FastArray([1, 2, 3])
```

Convert a `NaN` to an `int` sentinel and back:

```
>>> a = rt.FastArray([rt.nan, 1.0, 2.0])
>>> a_int = a.astype(int)
>>> a_int
FastArray([-2147483648,           1,           2])
>>> a_int.astype(float)
FastArray([nan,  1.,  2.])
```

**between**(*low*, *high*, *include_low=True*, *include_high=False*)

Return a boolean *FastArray* indicating which input values are in a specified interval.

**Parameters**

- **low** (*scalar or array*) – Lower bound for the interval. If an array, it must be the same size as `self`, and comparisons are done elementwise.
- **high** (*scalar or array*) – Upper bound for the interval. If an array, it must be the same size as `self`, and comparisons are done elementwise.
- **include_low** (bool, default `True`) – Specifies whether `low` is included when performing comparisons.
- **include_high** (bool, default `False`) – Specifies whether `high` is included when performing comparisons.

**Returns**
A boolean *FastArray* indicating which input values are in a specified interval.

**Return type**
*FastArray*

### Examples

Specify an interval using scalars:

```
>>> a = rt.FA([9, 2, 3, 5, 8, 9, 1, 4, 6])
>>> a.between(5, 9, include_low=False)  # Exclude 5 (left endpoint).
FastArray([False, False, False, False,  True, False, False, False,  True])
```

Specify an interval using arrays:

```
>>> a2 = rt.FA([1, 2, 3, 4, 5])
>>> a2.between([1, 3, 5, 5, 5], [2, 4, 6, 6, 6])
FastArray([ True, False, False, False,  True])
```

Specify an interval mixing scalar and array bounds:

```
>>> a3 = rt.FA([1, 2, 3, 4, 5])
>>> a3.between(2, [2, 4, 6, 6, 6])
FastArray([False,  True,  True,  True,  True])
```

**clip_lower**(*a_min*, ***kwargs*)

**clip_upper**(*a_max*, ***kwargs*)

**copy**(*order='K'*)

Return a copy of the input *FastArray*.

**Parameters**

**order** (*{'K', 'C', 'F', 'A'}, default 'K'*) – Controls the memory layout of the copy: 'K' means match the layout of the input array as closely as possible; 'C' means row-based (C-style) order; 'F' means column-based (Fortran-style) order; 'A' means 'F' if the input array is formatted as 'F', 'C' if not.

**Returns**

A copy of the input *FastArray*.

**Return type**

*FastArray*

**See also:**

*Categorical.copy*

Return a copy of the input *Categorical*.

*Dataset.copy*

Return a copy of the input *Dataset*.

*Struct.copy*

Return a copy of the input *Struct*.

### Examples

Copy a *FastArray*:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a2 = a.copy()
>>> a2
FastArray([1, 2, 3, 4, 5])
>>> a2 is a
False  # The copy is a separate object.
```

**copy_invalid**()

Return a copy of a *FastArray* filled with the invalid value for the array's data type.

**Returns**

A copy of the input array, filled with the invalid value for the array's dtype.

**Return type**

*FastArray*

**See also:**

***FastArray.inv***

Return the invalid value for the input array's dtype.

***FastArray.fill_invalid***

Replace the values of a *FastArray* with the invalid value for the array's dtype.

### Examples

Copy an integer array and replace with invalids:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a2 = a.copy_invalid()
>>> a2
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
>>> a
FastArray([1, 2, 3, 4, 5])  # a is unchanged.
```

Copy a floating-point array and replace with invalids:

```
>>> a3 = rt.FA([0., 1., 2., 3., 4.])
>>> a3
FastArray([0., 1., 2., 3., 4.])
>>> a3.copy_invalid()
FastArray([nan, nan, nan, nan, nan])
```

Copy a string array and replace with invalids:

```
>>> a4 = rt.FA(['AMZN', 'IBM', 'MSFT', 'AAPL'])
>>> a4
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a4.copy_invalid()
FastArray([b'', b'', b'', b''], dtype='|S4')  # Invalid string value is an␣
↪empty string.
```

**count**(*sorted=True*, *filter=None*)

The count of each unique value.

This returns the same information that `.unique(return_counts = True)` does, except in a `Dataset` instead of a tuple.

**Parameters**

- **sorted** (*bool, default True*) – When True (the default), unique values are returned in sorted order. Set to False to return them in order of first appearance.
- **filter** (*ndarray of bool, default None*) – If provided, any False values will be ignored in the calculation.

**Returns**

A `Dataset` containing the unique values and their counts.

**Return type**

*Dataset*

**See also:**

*FastArray.unique*

### Examples

```
>>> a = rt.FastArray([0, 2, 1, 3, 3, 2, 2])
>>> a.count()
*Unique   Count
-------   -----
      0       1
      1       1
      2       3
      3       2
```

With `sorted = False`:

```
>>> a.count(sorted = False)
*Unique   Count
-------   -----
      0       1
      2       3
      1       1
      3       2
```

**diff**(*periods=1*)

Compute the differences between adjacent elements of a *FastArray*.

Spaces at either end are filled with invalid values based on the input array's dtype. If a calculated difference isn't supported by the dtype, it is displayed as a NaN or rollover value. For example, negative differences

in a `uint8` array are displayed as 255. To resolve this, you can explicitly upcast to the next larger signed `int` dtype before calculating the differences.

**Parameters**

**periods** (*int, default 1*) – Number of element positions to shift right (if positive) or left (if negative) before subtracting. Raises an error if set to 0.

**Returns**

An equivalent-length array containing the differences between input array elements that are adjacent or separated by a specified period. Spaces at either end are filled with invalids based on the input array's dtype.

**Return type**

*FastArray*

**See also:**

***FastArray.shift***

Shift an array's elements right or left.

### Examples

Calculate differences using the `periods=1` default (array elements one position to the right):

```
>>> a=rt.FA([0, 2, 4, 8, 16, 32])
>>> a
FastArray([ 0,  2,  4,  8, 16, 32])
>>> a.diff()
FastArray([-2147483648,           2,           2,           4,
                    8,          16])
```

Calculate differences using array elements two positions to the left:

```
>>> a.diff(-2)
FastArray([         -4,          -6,         -12,         -24,
           -2147483648, -2147483648])
```

Specify a `periods` value that is greater than the array length:

```
>>> a.diff(10)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648, -2147483648])
```

**differs**(*periods=1*, *fancy=False*)

Identify array values that are the same as adjacent values.

Returns either a boolean *FastArray*, where `True` indicates equivalent values, or a fancy index *FastArray* containing the indices of equivalent values.

**Parameters**

- **periods** (*int, default 1*) – The number of array element positions to look behind (positive number) or look ahead (negative number) for comparison.
- **fancy** (*bool, default False*) – If `False` (the default), returns a boolean array. If `True`, returns a fancy index array.

**Returns**
A boolean or fancy index array that identifies equivalent elements in the input array.

**Return type**
*FastArray*

**See also:**

*FastArray.transitions*
Identify nonequivalent items in the input array and return a boolean or fancy index array.

### Examples

Return a boolean array using the `periods=1` default value (look behind one element position for comparisons):

```
>>> a = rt.FA([1, 2, 2, 3, 2, 4, 5, 6, 2, 2, 5])
>>> a
FastArray([1, 2, 2, 3, 2, 4, 5, 6, 2, 2, 5])
>>> a.differs()
FastArray([False, False,  True, False, False, False, False, False, False,
            True, False])
```

Return a boolean array and look ahead three element positions for comparisons:

```
>>> a.differs(periods=-3)
FastArray([False,  True, False, False, False, False, False, False, False,
           False, False])
```

Return a fancy index array using the `periods=1` default value (look behind one element position for comparisons):

```
>>> a.differs(fancy=True)
FastArray([2, 9])
```

Set `periods` to a number larger than the length of the input array:

```
>>> a.differs(periods=15)
FastArray([False, False, False, False, False, False, False, False, False,
           False, False])
```

**display_query_properties**()

Returns an ItemFormat object and a function for converting the FastArrays items to strings. Basic types: Bool, Int, Float, Bytes, String all have default formats / conversion functions. (see Utils.rt_display_properties)

If a new type is a subclass of FastArray and needs to be displayed in format different from its underlying type, it will need to take over this routine.

**duplicated**(*keep='first'*, *high_unique=False*)

Return a boolean *FastArray* indicating `True` for duplicate items in the input array.

**Parameters**

- **keep** (*{'first', 'last', 'False'}, default 'first'*) –
  - 'first' : Mark each duplicate as `True` except for the first occurrence.

- 'last' : Mark each duplicate as `True` except for the last occurrence.
- 'False' : Mark all duplicates as `True`.

- **high_unique** (bool, default `False` (hashing)) – Controls whether the function uses hashing- or sorting-based logic to find unique values in the input array. If your data has a high proportion of unique values, set to `True` for faster performance.

**Returns**
A boolean *FastArray* indicating `True` for duplicate items in the input array.

**Return type**
*FastArray*

**See also:**

*FastArray.nunique*
Return the number of unique values in an array.

*Dataset.duplicated*
Return a boolean *FastArray* indicating `True` for duplicate rows.

### Examples

Exclude the first occurrence of each duplicate (use the default `keep` value):

```
>>> a = rt.FA([1, 2, 3, 4, 2, 7, 8, 8, 3])
>>> a
FastArray([1, 2, 3, 4, 2, 7, 8, 8, 3])
>>> a.duplicated()
FastArray([False, False, False, False,  True, False, False,  True,  True])
```

Mark all duplicates:

```
>>> a.duplicated(keep=False)
FastArray([False,  True,  True, False,  True, False,  True,  True,  True])
```

**eq**(*other*)

**fill_invalid**(*shape=None*, *dtype=None*, *inplace=True*)

Replace all values of the input *FastArray* with an invalid value.

The invalid value used is determined by the input array's dtype or a user-specified dtype.

Warning: By default, this operation is in place.

**Parameters**

- **shape** (*int or sequence of int, optional*) – Shape of the new array, for example: `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str, optional*) – The desired dtype for the returned array.
- **inplace** (*bool, default True*) – If `True` (the default), modify original data. If `False`, return a copy of the array.

**Returns**
If `inplace=False`, a copy of the input *FastArray* is returned that has all values replaced with an invalid value. Otherwise, nothing is returned.

**Return type**
*FastArray*, optional

**See also:**

*FastArray.inv*
Return the invalid value for the input array's dtype.

*FastArray.copy_invalid*
Return a copy of a *FastArray* filled with the invalid value for the array's dtype.

### Examples

Replace an integer array's values with the invalid value for the array's dtype. By default, the returned array is the same size and dtype as the input array, and the operation is performed in place:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a
FastArray([1, 2, 3, 4, 5])
>>> a.fill_invalid()
>>> a
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Replace a floating-point array's values with the invalid value for the `int32` dtype:

```
>>> a2 = rt.FA([0., 1., 2., 3., 4.])
>>> a2
FastArray([0., 1., 2., 3., 4.])
>>> a2.fill_invalid(dtype="int32", inplace=False)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648])
```

Specify the size and dtype of the output array:

```
>>> a3 = rt.FA(["AMZN", "IBM", "MSFT", "AAPL"])
>>> a3
FastArray([b'AMZN', b'IBM', b'MSFT', b'AAPL'], dtype='|S4')
>>> a3.fill_invalid(2, dtype="bool", inplace=False)
FastArray([False, False])
```

**fillna**(*value=None*, *method=None*, *inplace=False*, *limit=None*)

Replace NaN and invalid values with a specified value or nearby data.

Optionally, you can modify the original *FastArray* if it's not locked.

**Parameters**

- **value** (*scalar or array, default None*) – A value or an array of values to replace all NaN and invalid values. A `value` is required if `method = None`. An array can be used only when `method = None`. If an array is used, the number of values in the array must equal the number of NaN and invalid values.
- **method** (*{None, 'backfill', 'bfill', 'pad', 'ffill'}, default None*) – Method to use to propagate valid values.

- backfill/bfill: Propagates the next encountered valid value backward. Calls `FastArray.fill_backward()`.
- pad/ffill: Propagates the last encountered valid value forward. Calls `FastArray.fill_forward()`.
- None: A replacement value is required if `method = None`. Calls *FastArray.replacena()*.

If there's not a valid value to propagate forward or backward, the NaN or invalid value is not replaced unless you also specify a `value`.

- **inplace** (*bool, default False*) – If False, return a copy of the *FastArray*. If True, modify original data. This will modify any other views on this object. This fails if the *FastArray* is locked.
- **limit** (*int, default None*) – If `method` is specified, this is the maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled.

**Returns**
The *FastArray* will be the same size and dtype as the original array.

**Return type**
*FastArray*

**See also:**

*riptable.rt_fastarraynumba.fill_forward*
Replace NaN and invalid values with the last valid value.

*riptable.rt_fastarraynumba.fill_backward*
Replace NaN and invalid values with the next valid value.

**riptable.fill_forward**
Replace NaN and invalid values with the last valid value.

**riptable.fill_backward**
Replace NaN and invalid values with the next valid value.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

*FastArray.replacena*
Replace NaN and invalid values with a specified value.

**Categorical.fill_forward**
Replace NaN and invalid values with the last valid group value.

**Categorical.fill_backward**
Replace NaN and invalid values with the next valid group value.

**GroupBy.fill_forward**
Replace NaN and invalid values with the last valid group value.

**GroupBy.fill_backward**
Replace NaN and invalid values with the next valid group value.

### Examples

Replace all NaN values with 0s:

```
>>> a = rt.FastArray([rt.nan, 1.0, rt.nan, rt.nan, rt.nan, 5.0])
>>> a.fillna(0)
FastArray([0., 1., 0., 0., 0., 5.])
```

Replace all invalid values with 0s:

```
>>> b = rt.FastArray([0, 1, 2, 3, 4, 5])
>>> b[0:3] = b.inv
>>> b.fillna(0)
FastArray([0, 0, 0, 3, 4, 5])
```

Replace each instance of NaN with a different value:

```
>>> a.fillna([0, 2, 3, 4])
FastArray([0., 1., 2., 3., 4., 5.])
```

Propagate the last encountered valid value forward. Note that where there's no valid value to propagate, the NaN or invalid value isn't replaced.

```
>>> a.fillna(method = 'ffill')
FastArray([nan,  1.,  1.,  1.,  1.,  5.])
```

You can use the `value` parameter to specify a value to use where there's no valid value to propagate.

```
>>> a.fillna(value = 0, method = 'ffill')
FastArray([0., 1., 1., 1., 1., 5.])
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values.

```
>>> a.fillna(method = 'bfill', limit = 1)
FastArray([ 1.,  1., nan, nan,  5.,  5.])
```

**filter**(*filter*)

Return a copy of the *FastArray* containing only the elements that meet the specified condition.

**Parameters**

**filter** (*array: fancy index or Boolean mask*) – A fancy index specifies both the desired elements and their order in the returned *FastArray*. When a Boolean mask is passed, only rows that meet the specified condition are in the returned *FastArray*.

**Return type**

*FastArray*

### Notes

If you want to perform an operation on a filtered FastArray, it's more efficient to perform the operation using the `filter` keyword argument. For example, `my_fa.sum(filter = boolean_mask)`.

### Examples

Create a *FastArray*:

```
>>> fa = rt.FastArray(np.linspace(0, 1, 11))
>>> fa
FastArray([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Filter using a fancy index:

```
>>> fa.filter([5, 0, 1])
FastArray([0.5, 0. , 0.1])
```

Filter using a condition that creates a Boolean mask array:

```
>>> fa.filter(fa > 0.75)
FastArray([0.8, 0.9, 1. ])
```

**static from_arrow**(*arr*, *zero_copy_only=True*, *writable=False*, *auto_widen=False*)

Convert a pyarrow `Array` to a riptable *FastArray*.

**Parameters**

- **arr** (*pyarrow.Array or pyarrow.ChunkedArray*) –
- **zero_copy_only** (*bool, default True*) – If True, an exception will be raised if the conversion to a *FastArray* would require copying the underlying data (e.g. in presence of nulls, or for non-primitive types).
- **writable** (*bool, default False*) – For a *FastArray* created with zero copy (view on the Arrow data), the resulting array is not writable (Arrow data is immutable). By setting this to True, a copy of the array is made to ensure it is writable.
- **auto_widen** (*bool, optional, default to False*) – When False (the default), if an arrow array contains a value which would be considered the 'invalid'/NA value for the equivalent dtype in a *FastArray*, raise an exception because direct conversion would be lossy / change the semantic meaning of the data. When True, the converted array will be widened (if possible) to the next-largest dtype to ensure the data will be interpreted in the same way.

**Return type**

*FastArray*

**ge**(*other*)

**get_name**()

Get the name that's assigned to a *FastArray*.

When a *FastArray* object is created, it has no name. It can be assigned a name via *set_name*. For details, see *FastArray.set_name()*.

**Returns**

The assigned name, or None if the array has not been named.

**Return type**
str or None

**See also:**

*FastArray.set_name*

### Examples

Assign the *FastArray* a name using *FastArray.set_name()*:

```
>>> a = rt.arange(5)
>>> a.set_name('FA Name')
FastArray([0, 1, 2, 3, 4])
```

Get the name:

```
>>> a.get_name()
'FA Name'
```

**gt**(*other*)

**info**(***kwargs*)

Return a description of the input array's contents.

This information is set using *FastArray.apply_schema* and includes the steward and dtype.

**Parameters**
****kwargs** (*optional*) – Keyword arguments passed to *rt_meta.info()*.

**Returns**
A description of the input array's contents.

**Return type**
*rt_meta.Info*

**See also:**

***FastArray.doc***
Return the *Doc* object for the input *FastArray*.

***Categorical.info***
Display a description of the input *Categorical*.

***Struct.info***
Return an object containing a description of the input structure's contents.

### Examples

Return the description of the input array's contents:

```
>>> a = rt.FA([1, 2, 3, 4, 5])
>>> a.info()
Description: <no description>
Steward: <no steward>
Type: int32
```

Apply a schema and return the description of the input array's contents:

```
>>> schema = {"Description": "This is an array", "Steward": "Brian"}
>>> a.apply_schema(schema)
{}
>>> a.info()
Description: This is an array
Steward: Brian
Type: int32
```

Return the description of the input array's contents with a title:

```
>>> a.info(title="Test")
Test
====
Description: This is an array
Steward: Brian
Type: int32
```

**iscomputable**()

**isfinite**(*fancy=False*)

Return a boolean array that's True for each finite *FastArray* element, False otherwise.

A value is considered to be finite if it's not positive or negative infinity or a NaN (Not a Number).

**Parameters**

**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (finite) values.

**Returns**

An array or booleans or indices.

**Return type**

*FastArray*

**See also:**

*FastArray.isnotfinite*, riptable.isfinite, riptable.isnotfinite, riptable.isinf, riptable.isnotinf, *FastArray.isinf*, *FastArray.isnotinf*

**Dataset.mask_or_isfinite**

Return a boolean array that's True for each Dataset row that has at least one finite value.

**Dataset.mask_and_isfinite**

Return a boolean array that's True for each Dataset row that contains all finite values.

**Dataset.mask_or_isinf**

Return a boolean array that's True for each Dataset row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**

Return a boolean array that's True for each Dataset row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> a.isfinite()
FastArray([False, False, False,  True])
```

With `fancy = True`:

```
>>> a.isfinite(fancy = True)
FastArray([3])
```

**isin**(*test_elements*, *, *assume_unique=False*, *invert=False*)

Calculates `self in test_elements`, broadcasting over `self` only. Returns a boolean array of the same shape as `self` that is True where an element of `self` is in `test_elements` and False otherwise.

**Parameters**

- **test_elements** (*array_like*) – The values against which to test each value of `element`. This argument is flattened if it is an array or array_like. See notes for behavior with non-array-like parameters.
- **assume_unique** (*bool, optional*) – If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.
- **invert** (*bool, optional*) – If True, the values in the returned array are inverted, as if calculating `element not in test_elements`. Default is False. `np.isin(a, b, invert=True)` is equivalent to (but faster than) `np.invert(np.isin(a, b))`.

**Returns**

- **isin** (*ndarray, bool*) – Has the same shape as `element`. The values `element[isin]` are in `test_elements`.
- **Note** (*behavior differs from pandas*)
- *- Riptable favors bytestrings, and will make conversions from unicode/bytes to match for operations as necessary.*
- *- We will also accept single scalars for values.*
- *- Pandas series will return another series - we have no series, and will return a FastArray*

### Examples

```
>>> from riptable import *
>>> a = FA(['a','b','c','d','e'], unicode=False)
>>> a.isin(['a','b'])
FastArray([ True,  True, False, False, False])
>>> a.isin('a')
FastArray([ True,  False, False, False, False])
>>> a.isin({'b'})
FastArray([ False, True, False, False, False])
```

**isinf**(*fancy=False*)

Return a boolean array that's True for each *FastArray* element that's positive or negative infinity, False otherwise.

**Parameters**
**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (infinite) values.

**Returns**
An array or booleans or indices.

**Return type**
*FastArray*

**See also:**

*FastArray.isnotinf*, *FastArray.isfinite*, *FastArray.isnotfinite*, riptable.isinf, riptable.isnotinf, riptable.isfinite, riptable.isnotfinite

**Dataset.mask_or_isfinite**
Return a boolean array that's True for each Dataset row that has at least one finite value.

**Dataset.mask_and_isfinite**
Return a boolean array that's True for each Dataset row that contains all finite values.

**Dataset.mask_or_isinf**
Return a boolean array that's True for each Dataset row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**
Return a boolean array that's True for each Dataset row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> a.isinf()
FastArray([ True,  True, False, False])
```

With fancy = True:

```
>>> a.isinf(fancy = True)
FastArray([0, 1])
```

**isna**()
isnan is mapped directly to isnan() Categoricals and DateTime take over isnan. FastArray handles sentinels.

```
>>> a=arange(100.0)
>>> a[5]=np.nan
>>> a[87]=np.nan
>>> sum(a.isna())
2
>>> sum(a.astype(np.int32).isna())
2
```

**isnan**(*fancy=False*)
Return a boolean array that's True for each element that's a NaN (Not a Number), False otherwise.

**Parameters**
**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (NaN) values.

**Returns**
A *FastArray* of booleans or indices.

**Return type**
*FastArray*

**See also:**

*FastArray.isnotnan*, *FastArray.notna*, *FastArray.isnanorzero*, riptable.isnan, riptable.isnotnan, riptable.isnanorzero, Categorical.isnan, Categorical.isnotnan, Categorical.notna, Date.isnan, Date.isnotnan, DateTimeNano.isnan, DateTimeNano.isnotnan

**Dataset.mask_or_isnan**
Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN Dataset row.

### Examples

```
>>> a = rt.FastArray([rt.nan, rt.nan, rt.inf, 3])
>>> a.isnan()
FastArray([ True,  True, False, False])
```

With fancy = True:

```
>>> a.isnan(fancy = True)
FastArray([0, 1])
```

**isnanorzero**(*fancy=False*)

Return a boolean array that's True for each element that's a NaN (Not a Number) or zero, False otherwise.

**Parameters**
**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (NaN or zero) values.

**Returns**
A *FastArray* of booleans or indices.

**Return type**
*FastArray*

**See also:**

riptable.isnanorzero, riptable.isnan, riptable.isnotnan, *FastArray.isnan*, *FastArray.isnotnan*, Categorical.isnan, Categorical.isnotnan, Date.isnan, Date.isnotnan, DateTimeNano.isnan, DateTimeNano.isnotnan

**Dataset.mask_or_isnan**
Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN Dataset row.

### Examples

```
>>> a = rt.FastArray([0, rt.nan, rt.inf, 3])
>>> a.isnanorzero()
FastArray([ True,  True, False, False])
```

With `fancy = True`:

```
>>> a.isnanorzero(fancy = True)
FastArray([0, 1])
```

**isnormal**(*fancy=False*)

**isnotfinite**(*fancy=False*)

Return a boolean array that's True for each non-finite *FastArray* element, False otherwise.

A value is considered to be finite if it's not positive or negative infinity or a NaN (Not a Number).

**Parameters**

**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (non-finite) values.

**Returns**

An array or booleans or indices.

**Return type**

*FastArray*

**See also:**

*FastArray.isfinite*, riptable.isfinite, riptable.isnotfinite, riptable.isinf, riptable.isnotinf, *FastArray.isinf*, *FastArray.isnotinf*

**Dataset.mask_or_isfinite**

Return a boolean array that's True for each `Dataset` row that has at least one finite value.

**Dataset.mask_and_isfinite**

Return a boolean array that's True for each `Dataset` row that contains all finite values.

**Dataset.mask_or_isinf**

Return a boolean array that's True for each `Dataset` row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**

Return a boolean array that's True for each `Dataset` row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> a.isnotfinite()
FastArray([ True,  True,  True, False])
```

With `fancy = True`:

```
>>> a.isnotfinite(fancy = True)
FastArray([0, 1, 2])
```

**isnotinf**(*fancy=False*)

Return a boolean array that's True for each *FastArray* element that's not positive or negative infinity, False otherwise.

**Parameters**
**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (non-infinite) values.

**Returns**
An array or booleans or indices.

**Return type**
*FastArray*

**See also:**

*FastArray.isinf*, riptable.isnotinf, riptable.isinf, riptable.isfinite, riptable.isnotfinite, *FastArray.isfinite*, *FastArray.isnotfinite*

**Dataset.mask_or_isfinite**
Return a boolean array that's True for each Dataset row that has at least one finite value.

**Dataset.mask_and_isfinite**
Return a boolean array that's True for each Dataset row that contains all finite values.

**Dataset.mask_or_isinf**
Return a boolean array that's True for each Dataset row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**
Return a boolean array that's True for each Dataset row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> a.isnotinf()
FastArray([False, False,  True,  True])
```

With fancy = True:

```
>>> a.isnotinf(fancy = True)
FastArray([2, 3])
```

**isnotnan**(*fancy=False*)

Return a boolean array that's True for each element that's not a NaN (Not a Number), False otherwise.

**Parameters**
**fancy** (*bool, default False*) – Set to True to instead return the indices of the True (non-NaN) values.

**Returns**
A *FastArray* of booleans or indices.

**Return type**
*FastArray*

**See also:**

*FastArray.isnan*, *FastArray.notna*, *FastArray.isnanorzero*, riptable.isnan, riptable.isnotnan, riptable.isnanorzero, Categorical.isnan, Categorical.isnotnan,

Categorical.notna, Date.isnan, Date.isnotnan, DateTimeNano.isnan, DateTimeNano.isnotnan

**Dataset.mask_or_isnan**
Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN Dataset row.

### Examples

```
>>> a = rt.FastArray([rt.nan, rt.inf, 2])
>>> a.isnotnan()
FastArray([False,  True,  True])
```

With fancy = True:

```
>>> a.isnotnan(fancy = True)
FastArray([1, 2])
```

**isnotnormal**(*fancy=False*)

**issorted**()

Return True if the array is sorted, False otherwise.

NaNs at the end of an array are considered sorted.

Calls riptable.issorted().

**Returns**
True if the array is sorted, False otherwise.

**Return type**
bool

**See also:**

riptable.issorted

### Examples

```
>>> a = rt.FastArray(['a', 'b', 'c'])
>>> a.issorted()
True
```

```
>>> a = rt.FastArray([1.0, 2.0, 3.0, rt.nan])
>>> rt.issorted(a)
True
```

```
>>> a = rt.FastArray(['a', 'c', 'b'])
>>> a.issorted()
False
```

**le**(*other*)

**lt**(*other*)

**map**(*npdict*)

### Notes

Uses ismember and can handle large dictionaries

### Examples

```
>>> a=arange(3)
>>> a.map({1: 'a', 2:'b', 3:'c'})
FastArray(['', 'a', 'b'], dtype='<U1')
>>> a=arange(3)+1
>>> a.map({1: 'a', 2:'b', 3:'c'})
FastArray(['a', 'b', 'c'], dtype='<U1')
```

**map_old**(*npdict*)

### Example

```
>>> d = {1:10, 2:20}
>>> dat['c'] = dat.a.map(d)
>>> print(dat)
   a  b   cb   c
0  1  0  0.0  10
1  1  1  1.0  10
2  1  2  3.0  10
3  2  3  5.0  20
4  2  4  7.0  20
5  2  5  9.0  20
```

**mean**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, ***kwargs*)

Compute the arithmetic mean of the values in the first argument.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the mean calculation. If the filter is uniformly `False`, *mean* returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.mean(dtype = my_type)` is equivalent to `my_type(x.mean())`.

**Returns**

The mean of the values.

**Return type**

scalar

**See also:**

`numpy.mean`

*FastArray.nanmean*
Computes the mean of *FastArray* values, ignoring NaNs.

**Dataset.mean**
Computes the mean of numerical Dataset columns.

**GroupByOps.mean**
Computes the mean of each group. Used by Categorical objects.

### Notes

The dtype keyword for *FastArray.mean* specifies the data type of the result. This differs from numpy.mean, where it specifies the data type used to compute the mean.

**Notes on Using NumPy Parameters**

Using either of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (numpy.mean). However, until a reported bug is fixed, if you also include the dtype parameter it will be applied to the result, not used to compute the mean as it is in numpy.mean.

Also note that if you use either of the following NumPy parameters and also include a *filter* keyword argument (which numpy.mean does not accept), Riptable's implementation of *mean* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**
[None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

**keepdims**
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original input array.

If the default value is passed, then keepdims will not be passed through to the *mean* method of sub-classes of ndarray, however any non-default value will be. If the sub-class's method does not implement keepdims, any exceptions will be raised.

### Examples

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> a.mean()
4.0
```

With a dtype specified:

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> a.mean(dtype = rt.int32)
4
```

With a filter:

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> b = rt.FastArray([False, True, False, True])
>>> a.mean(filter = b)
5.0
```

**median**(***kwargs*)

**move_argmax**(**args*, ***kwargs*)

**move_argmin**(**args*, ***kwargs*)

**move_max**(**args*, ***kwargs*)

**move_mean**(**args*, ***kwargs*)

**move_median**(**args*, ***kwargs*)

**move_min**(**args*, ***kwargs*)

**move_rank**(**args*, ***kwargs*)

**move_std**(**args*, ***kwargs*)

**move_sum**(**args*, ***kwargs*)

**move_var**(**args*, ***kwargs*)

**nanargmax**(***kwargs*)

**nanargmin**(***kwargs*)

**nanmax**(***kwargs*)

**nanmean**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, ***kwargs*)

Compute the arithmetic mean of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `0.0` is returned.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the mean calculation. If the filter is uniformly `False`, *nanmean* returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.nanmean(dtype = my_type)` is equivalent to `my_type(x.nanmean())`.

**Returns**

The mean of the values.

**Return type**

scalar

**See also:**

`numpy.nanmean`

***FastArray.mean***

Computes the mean of *FastArray* values.

**Dataset.nanmean**

Computes the mean of numerical `Dataset` columns, ignoring NaNs.

**GroupByOps.nanmean**

Computes the mean of each group, ignoring NaNs. Used by `Categorical` objects.

### Notes

The `dtype` keyword for *`FastArray.nanmean`* specifies the data type of the result. This differs from `numpy.nanmean`, where it specifies the data type used to compute the mean.

**Notes on Using NumPy Parameters**

Using either of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (`numpy.nanmean`). However, until a reported bug is fixed, if you also include the `dtype` parameter it will be applied to the result, not used to compute the mean as it is in `numpy.nanmean`.

Also note that if you use either of the following NumPy parameters and also include a *`filter`* keyword argument (which `numpy.nanmean` does not accept), Riptable's implementation of *`nanmean`* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**
: [{int, tuple of int, None}, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

**keepdims**
: [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original input array.

  If the value is anything but the default, then `keepdims` will be passed through to the *`mean`* or `sum` methods of sub-classes of `ndarray`. If the sub-classes' methods do not implement `keepdims`, any exceptions will be raised.

### Examples

```
>>> a = rt.FastArray([1, 3, 5, rt.nan])
>>> a.nanmean()
3.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 3, 5, rt.nan])
>>> a.nanmean(dtype = rt.int32)
3
```

With a filter:

```
>>> a = rt.FastArray([1, 3, 5, rt.nan])
>>> b = rt.FastArray([False, True, True, True])
>>> a.nanmean(filter = b)
4.0
```

**nanmedian**(***kwargs*)

**nanmin**(***kwargs*)

**nanpercentile**(***kwargs*)

**nanquantile**(***kwargs*)

**nanrankdata**(**args*, ***kwargs*)

**nanstd**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, *ddof=None*, ***kwargs*)

Compute the standard deviation of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `NaN` is returned.

Riptable uses the convention that `ddof = 1`, meaning the standard deviation of `[x_1, ..., x_n]` is defined by `std = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the standard deviation calculation. If the filter is uniformly `False`, *nanstd* returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.nanstd(dtype = my_type)` is equivalent to `my_type(x.nanstd())`.

**Returns**

The standard deviation of the values.

**Return type**

scalar

**See also:**

`numpy.nanstd`

*FastArray.std*

Computes the standard deviation of *FastArray* values.

`Dataset.nanstd`

Computes the standard deviation of numerical `Dataset` columns, ignoring NaNs.

`GroupByOps.nanstd`

Computes the standard deviation of each group, ignoring NaNs. Used by `Categorical` objects.

## Notes

The `dtype` keyword for *FastArray.nanstd* specifies the data type of the result. This differs from `numpy.nanstd`, where it specifies the data type used to compute the standard deviation.

**Notes on Using NumPy Parameters**

Using any of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (`numpy.nanstd`). However, until a reported bug is fixed, if you also include the `dtype` parameter it will be applied to the result, not used to compute the variance as it is in `numpy.nanstd`.

Also note that if you use any of the following NumPy parameters and also include a *filter* keyword argument (which `numpy.nanstd` does not accept), Riptable's implementation of *nanstd* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original input array.

If this value is anything but the default it is passed through as-is to the relevant functions of the sub-classes. If these functions do not have a `keepdims` kwarg, a RuntimeError will be raised.

**ddof**
[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero for the NumPy implementation, versus one for the Riptable implementation.

### Examples

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> a.nanstd()
1.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> a.nanstd(dtype = rt.int32)
1
```

With filter:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> b = rt.FastArray([False, True, True, True])
>>> a.nanstd(filter = b)
0.7071067811865476
```

**nansum**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, ***kwargs*)

Compute the sum of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `0.0` is returned.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the sum calculation. If the filter is uniformly `False`, *nansum* returns `0.0`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.nansum(dtype = my_type)` is equivalent to `my_type(x.nansum())`.

**Returns**
The sum of the values.

**Return type**
scalar

**See also:**

`numpy.nansum`

**Dataset.nansum**
Sums the values of numerical `Dataset` columns, ignoring NaNs.

**GroupByOps.nansum**
Sums the values of each group, ignoring NaNs. Used by `Categorical` objects.

### Notes

The `dtype` keyword for *`FastArray.nansum`* specifies the data type of the result. This differs from `numpy.nansum`, where it specifies the data type used to compute the sum.

**Notes on Using NumPy Parameters**

Using either of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (`numpy.nansum`). However, until a reported bug is fixed, if you also include the `dtype` parameter it will be applied to the result, not used to compute the sum as it is in `numpy.nansum`.

Also note that if you use either of the following NumPy parameters and also include a *`filter`* keyword argument (which `numpy.nansum` does not accept), Riptable's implementation of *`nansum`* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**
: [{int, tuple of int, None}, optional] Axis or axes along which the sum is computed. The default is to compute the sum of the flattened array.

**keepdims**
: [bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original input array.

  If the value is anything but the default, then `keepdims` will be passed through to the *`mean`* or `sum` methods of sub-classes of `ndarray`. If the sub-classes' methods do not implement `keepdims`, any exceptions will be raised.

### Examples

```
>>> a = rt.FastArray([1, 3, 5, 7, rt.nan])
>>> a.nansum()
16.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1.0, 3.0, 5.0, 7.0, rt.nan])
>>> a.nansum(dtype = rt.int32)
16
```

With a filter:

```
>>> a = rt.FastArray([1, 3, 5, 7, rt.nan])
>>> b = rt.FastArray([False, True, False, True, True])
>>> a.nansum(filter = b)
10.0
```

**nanvar**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, *ddof=None*, ***kwargs*)

Compute the variance of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `NaN` is returned.

Riptable uses the convention that `ddof = 1`, meaning the variance of `[x_1, ..., x_n]` is defined by `var = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the variance calculation. If the filter is uniformly `False`, *nanvar* returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.nanvar(dtype = my_type)` is equivalent to `my_type(x.nanvar())`.

**Returns**
The variance of the values.

**Return type**
scalar

**See also:**

`numpy.nanvar`

***FastArray.var***
Computes the variance of *FastArray* values.

**`Dataset.nanvar`**
Computes the variance of numerical `Dataset` columns, ignoring NaNs.

**`GroupByOps.nanvar`**
Computes the variance of each group, ignoring NaNs. Used by `Categorical` objects.

## Notes

The `dtype` keyword for *FastArray.nanvar* specifies the data type of the result. This differs from `numpy.nanvar`, where it specifies the data type used to compute the variance.

**Notes on Using NumPy Parameters**

Using any of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (`numpy.nanvar`). However, until a reported bug is fixed, if you also include the `dtype` parameter it will be applied to the result, not used to compute the variance as it is in `numpy.nanvar`.

Also note that if you use any of the following NumPy parameters and also include a *filter* keyword argument (which `numpy.nanvar` does not accept), Riptable's implementation of *nanvar* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**
[{int, tuple of int, None}, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

**keepdims**
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original input array.

**ddof**
[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of non-NaN elements. By default `ddof` is zero for the NumPy implementation, versus one for the Riptable implementation.

### Examples

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> a.nanvar()
1.0
```

With a dtype specified:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> a.nanvar(dtype = rt.int32)
1
```

With a filter:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> b = rt.FastArray([False, True, True, True])
>>> a.nanvar(filter = b)
0.5
```

**ne**(*other*)

**normalize_minmax**()

**normalize_zscore**()

**notna**()

notna is mapped directly to isnotnan() Categoricals and DateTime take over isnotnan. FastArray handles sentinels.

```
>>> a=arange(100.0)
>>> a[5]=np.nan
>>> a[87]=np.nan
>>> sum(a.notna())
98
>>> sum(a.astype(np.int32).notna())
98
```

**nunique**()

Return the number of unique values in the input *FastArray*.

Does not include NaN or sentinel values.

**Returns**

Number of unique values in the input *FastArray*, excluding NaN and sentinel values.

**Return type**

int

**See also:**

*FastArray.duplicated*

Return a boolean *FastArray* indicating duplicate values.

*Categorical.nunique*

Return the number of unique values in the *Categorical*.

### Examples

Retrieve the number of unique values in a floating-point *FastArray*:

```
>>> a = rt.FastArray([1., 2., 3., 1., 2., 3.])
>>> a
FastArray([1., 2., 3., 1., 2., 3.])
>>> a.nunique()
3
```

Retrieve the number of unique values in a floating-point *FastArray* with a NaN value:

```
>>> a2 = rt.FastArray([1., 2., 3., 1., 2., 3., rt.nan])
>>> a2
FastArray([ 1.,  2.,  3.,  1.,  2.,  3., nan])
>>> a2.nunique()  # The NaN value is not included.
3
```

Retrieve the number of unique values in an unsigned integer *FastArray* with a sentinel value:

```
>>> a3 = rt.FastArray([255, 2, 3, 2, 3], dtype="uint8")
>>> a3
FastArray([255,   2,   3,   2,   3], dtype=uint8)
>>> a3.nunique()  # The sentinel value is not included.
2
```

**partition2**(**args*, ***kwargs*)

**percentile**(***kwargs*)

**push**(**args*, ***kwargs*)

**quantile**(***kwargs*)

**rankdata**(**args*, ***kwargs*)

**classmethod register_function**(*name*, *func*)

Used to register functions to FastArray. Used by rt_fastarraynumba

**repeat**(*repeats*, *axis=None*)

See `riptable.repeat`.

**replace**(*old*, *new*)

**replacena**(*value*, *inplace=False*)

Return a *FastArray* with all NaN and invalid values set to the specified value.

Optionally, you can modify the original *FastArray* if it's not locked.

**Parameters**

- **value** (*scalar or array*) – A value or an array of values to replace all NaN and invalid values. If an array, the number of values must equal the number of NaN and invalid values.
- **inplace** (*bool, default False*) – If False, return a copy of the *FastArray*. If True, modify the original. This will modify any other views on this object. This fails if the *FastArray* is locked.

**Returns**
The *FastArray* will be the same size and dtype as the original array. Returns None if `inplace = True`.

**Return type**
*FastArray* or None

**See also:**

*FastArray.fillna*
Replace NaN and invalid values with a specified value or nearby data.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**Categorical.fill_forward**
Replace NaN and invalid values with the last valid group value.

**Categorical.fill_backward**
Replace NaN and invalid values with the next valid group value.

**GroupBy.fill_forward**
Replace NaN and invalid values with the last valid group value.

**GroupBy.fill_backward**
Replace NaN and invalid values with the next valid group value.

### Examples

Replace all instances of NaN with a single value:

```
>>> a = rt.FastArray([rt.nan, 1.0, rt.nan, 3.0])
>>> a.replacena(0)
FastArray([0., 1., 0., 3.])
```

Replace all invalid values with 0s:

```
>>> b = rt.FastArray([0, 1, 2, 3, 4, 5])
>>> b[0:3] = b.inv
>>> b.replacena(0)
FastArray([0, 0, 0, 3, 4, 5])
```

Replace each instance of NaN with a different value:

```
>>> a.replacena([0, 2])
FastArray([0., 1., 2., 3.])
```

**reshape**(**args*, ***kwargs*)

**rolling_mean**(*window=3*)

**rolling_nanmean**(*window=3*)

**rolling_nanstd**(*window=3*)

**rolling_nansum**(*window=3*)

**rolling_nanvar**(*window=3*)

**rolling_quantile**(*q*, *window=3*)

**rolling_std**(*window=3*)

**rolling_sum**(*window=3*)

**rolling_var**(*window=3*)

**sample**(*N=10*, *filter=None*, *seed=None*)

Return a given number of randomly selected values from a *FastArray*.

**Parameters**

- **N** (*int, default 10*) – Number of values to select. The entire array is returned if `N` is greater than the size of the array.
- **filter** (*array (bool or int), optional*) – A boolean mask or index array to filter values before selection. A boolean mask must have the same length as the original *FastArray*.
- **seed** (*int or other types, optional*) – A seed to initialize the random number generator. If one is not provided, the generator is initialized using random data from the OS. For details and other accepted types, see the `seed` parameter for `numpy.random.default_rng`.

**Returns**

A new *FastArray* containing the randomly selected values.

**Return type**

*FastArray*

**See also:**

*Dataset.sample*

Return a specified number of randomly selected rows from a *Dataset*.

### Examples

No sample size specified:

```
>>> a = rt.FA([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
>>> a.sample()  # 10 randomly selected values returned.
FastArray([ 1,  2,  3,  4,  5,  6,  7,  9, 10, 11])  # Random
```

Sample 3 values:

```
>>> a = rt.FA([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
>>> a.sample(3)
FastArray([1, 4, 9])  # Random
```

Specify a sample size larger than the array:

```
>>> a2 = rt.FA([1, 2, 3, 4, 5])
>>> a2.sample(100)  # The entire array is returned.
FastArray([1, 2, 3, 4, 5])
```

Specify an index array for filtering:

```
>>> a3 = rt.FA(['TSLA','AMZN','IBM', 'SPY', 'GME', 'AAPL', 'FB', 'GOOG',
...             'MSFT', 'UBER'])  # Create sample data.
>>> filter = rt.FA([0, 1, 3, 7])  # Specify indices of a3 to take the sample␣
↪from.
>>> a3.sample(2, filter)
FastArray([b'TSLA', b'GOOG'], dtype='|S4')  # Random
```

Specify a boolean mask array for filtering:

```
>>> a3.sample(8, filter=rt.FA(a3 != 'SPY'))
FastArray([b'TSLA', b'IBM', b'GME', b'AAPL', b'FB', b'GOOG', b'MSFT',
           b'UBER'], dtype='|S4')  # Random
```

**save**(*filepath*, *share=None*, *compress=True*, *overwrite=True*, *name=None*)

Save a *FastArray* to an .sds file.

**Parameters**

- **filepath** (*str or os.PathLike*) – Path for the .sds file. If there's a trailing slash, `filepath` is treated as a path to a directory and you also need to specify `name`. Alternatively, you can include a file name (with or without the .sds extension) at the end of `filepath` (with no trailing slash), and an .sds file with that name is created. Directories that don't yet exist are created.
- **share** (*str, optional*) – If specified, the *FastArray* is saved to shared memory (NOT to disk) and path information from `filepath` is discarded. A `name` value must be provided. When shared memory is used, data is not compressed. Note that shared memory functions are not currently supported on Windows.
- **compress** (*bool, default True*) – When `True` (the default), compression is used when writing to the .sds file. Otherwise, no compression is used. (If shared memory is used, data is always saved uncompressed.)
- **overwrite** (*bool, default True*) – When `True` (the default), the user is not prompted to specify whether or not to overwrite an existing .sds file. When set to `False`, a prompt is displayed.
- **name** (*str, optional*) – Name for the .sds file. The .sds extension is not required. Note that if `name` is provided, `filepath` is treated as a path to a directory, even if `filepath` has no trailing slash.

**Return type**

An .sds file containing the *FastArray*.

**See also:**

*save_sds*

Save `Dataset` objects and arrays into a single .sds file.

*load_sds*

Load an .sds file.

### Examples

Include a file name in the path:

```
>>> a = rt.FA([0, 1, 2, 3, 4])
FastArray([0, 1, 2, 3, 4])
>>> a.save("C://junk//saved_file")
>>> os.listdir("C://junk")
['saved_file.sds']
```

When `name` is specified, `filepath` is treated as a path to a directory:

```
>>> a.save("C://junk//saved_file", name="fa")
>>> os.listdir("C://junk//saved_file")
['fa.sds']
```

Display a prompt before overwriting an existing file:

```
>>> a.save("C://junk//saved_file", overwrite=False)
C://junk//saved_file.sds already exists. Overwrite? (y/n) n
No file was saved.
```

**searchsorted**(*v*, *side='left'*, *sorter=None*)

**set_name**(*name*)

Assign a name to a *FastArray*.

A *FastArray* is a wrapper around a NumPy `ndarray`. When a *FastArray* is created, it has no name. You can assign it a name using *set_name*.

**Interactions with Dataset Objects**

When an unnamed *FastArray* is added to a `Dataset`:

- The *FastArray* inherits the name of the `Dataset` column.
- Calling `fa.set_name` or `ds.col.set_name`, or changing the displayed column name via `ds.col_rename`, changes the name assigned to the *FastArray*.
  - Note that calling `fa.set_name` or `ds.col.set_name` doesn't change the displayed column name.

When a named *FastArray* is added to a `Dataset`:

- A new *FastArray* instance is created that inherits the `Dataset` column name.
- Calling `ds.col.set_name` or changing the displayed column name via `ds.col_rename` changes the new instance's name.
- Calling *set_name* on the original *FastArray* instance changes only that instance's name.

In both cases, the NumPy array underlying the *FastArray* is shared – changes to its values appear in the `Dataset` column, and vice-versa.

**Interactions with FastArray Objects**

- When a *FastArray* is created as a view of another, named *FastArray*, the new *FastArray* instance inherits the name from the original *FastArray*.
- Whether the original *FastArray* is named or unnamed, calling *set_name* on either *FastArray* does not change the name of the other *FastArray*.

**Parameters**
**name** (*str*) – The name to assign to the *FastArray*.

**Returns**
The *FastArray* is returned. The name can be accessed using *FastArray.get_name()*.

**Return type**
*FastArray*

**See also:**

*FastArray.get_name*

### Examples

```
>>> a = rt.arange(5)
>>> a.set_name('FA Name')
FastArray([0, 1, 2, 3, 4])
```

You can get the name using *FastArray.get_name()*:

```
>>> a.get_name()
'FA Name'
```

When an unnamed *FastArray* is added to a `Dataset` column, the *FastArray* inherits the name of the column.

```
>>> a = rt.FastArray([1, 2, 3])
>>> ds = rt.Dataset()
>>> ds.Column_Name = a
>>> a.get_name()
'Column_Name'
```

Calling `ds.col.set_name` changes the name assigned to the *FastArray* (but not the displayed column name).

```
>>> ds.Column_Name.set_name('New Name')
>>> a.get_name()
'New Name'
>>> ds
#   Column_Name
-   -----------
0             1
1             2
2             3
```

When a named *FastArray* is added to a `Dataset` column, a new *FastArray* instance is created that inherits the column name. The original instance is not renamed.

```
>>> a = rt.FastArray([1, 2, 3])
>>> a.set_name('FA Name')
>>> ds = rt.Dataset()
>>> ds.Column_Name = a
>>> ds.Column_Name.get_name()
```

(continues on next page)

(continued from previous page)

```
'Column_Name'
>>> a.get_name()
'FA Name'
```

Changing the displayed column name affects the name of the new instance, but not the name of the original *FastArray*.

```
>>> ds.col_rename('Column_Name', 'New_Column')
>>> ds.New_Column.get_name()
'New_Column'
>>> a.get_name()
'FA Name'
```

**shift**(*periods=1*, *invalid=None*)

Shift an array's elements right or left.

Newly empty elements at either end (resulting from the shift) are filled with the invalid value for the input array's data type.

**Parameters**

**periods** (*int, default 1*) – Number of element positions to shift right (if positive) or left (if negative).

**Returns**

A shifted *FastArray*. Newly empty elements are filled with the invalid values for the input array's data type.

**Return type**

*FastArray*

**See also:**

*FastArray.diff*

Return a *FastArray* containing the differences between adjacent input array values.

*Categorical.shift*

Shift values in the *Categorical* by a specified number of periods.

### Examples

Shift array elements one position to the right:

```
>>> a = rt.FA([0, 2, 4, 8, 16, 32])
>>> a
FastArray([ 0,  2,  4,  8, 16, 32])
>>> a.shift()
FastArray([-2147483648,           0,           2,           4,
                    8,          16])
```

Shift array elements two positions to the left:

```
>>> a.shift(-2)
>>> FastArray([          4,           8,          16,          32,
           -2147483648, -2147483648])
```

Specify a shift value greater than the array length:

```
>>> a.shift(10)
FastArray([-2147483648, -2147483648, -2147483648, -2147483648,
           -2147483648, -2147483648])
```

**sign**(***kwargs*)

**squeeze**(**args*, ***kwargs*)

**statx**()

**std**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, *ddof=None*, ***kwargs*)

Compute the standard deviation of the values in the first argument.

Riptable uses the convention that `ddof = 1`, meaning the standard deviation of `[x_1, ..., x_n]` is defined by `std = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the standard deviation calculation. If the filter is uniformly `False`, *std* returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.std(dtype = my_type)` is equivalent to `my_type(x.std())`.

**Returns**

The standard deviation of the values.

**Return type**

scalar

**See also:**

`numpy.std`

***FastArray.nanstd***

Computes the standard deviation of *FastArray* values, ignoring NaNs.

**Dataset.std**

Computes the standard deviation of numerical `Dataset` columns.

**GroupByOps.std**

Computes the standard deviation of each group. Used by `Categorical` objects.

## Notes

The `dtype` keyword for *FastArray.std* specifies the data type of the result. This differs from `numpy.std`, where it specifies the data type used to compute the standard deviation.

**Notes on Using NumPy Parameters**

Using any of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (`numpy.std`). However, until a reported bug is fixed, if you also include the `dtype` parameter it will be applied to the result, not used to compute the variance as it is in `numpy.std`.

Also note that if you use any of the following NumPy parameters and also include a *filter* keyword argument (which numpy.std does not accept), Riptable's implementation of *std* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**
[None or int or tuple of ints, optional] Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a standard deviation is performed over multiple axes, instead of a single axis or all the axes as before.

**keepdims**
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be passed through to the *std* method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement `keepdims`, any exceptions will be raised.

**ddof**
[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero for the NumPy implementation, versus one for the Riptable implementation.

### Examples

```
>>> a = rt.FastArray([1, 2, 3])
>>> a.std()
1.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 2, 3])
>>> a.std(dtype = rt.int32)
1
```

With a filter:

```
>>> a = rt.FastArray([1, 2, 3])
>>> b = rt.FA([False, True, True])
>>> a.std(filter = b)
0.7071067811865476
```

**`str()`**

Casts an array of byte strings or unicode as `FAString`.

Enables a variety of useful string manipulation methods.

**Return type**
*FAString*

**Raises**
`TypeError` – If the FastArray is of dtype other than byte string or unicode

**See also:**

`np.chararray`, `np.char`, `rt.FAString.apply`

### Examples

```
>>> s=FA(['this','that','test ']*100_000)
>>> s.str.upper
FastArray([b'THIS', b'THAT', b'TEST ', ..., b'THIS', b'THAT', b'TEST '],
          dtype='|S5')
```

```
>>> s.str.lower
FastArray([b'this', b'that', b'test ', ..., b'this', b'that', b'test '],
          dtype='|S5')
```

```
>>> s.str.removetrailing()
FastArray([b'this', b'that', b'test', ..., b'this', b'that', b'test'],
          dtype='|S5')
```

**str_append**(*other*)

**tile**(*reps*)

See `riptable.tile`.

**timewindow_prod**(*time_array*, *time_dist*)

The input array must be int64 and sorted with ever increasing values. Multiplies up the values for a given time window.

**Parameters**

- **time_array** (*sorted integer array of timestamps*) –
- **time_dist** (*integer value of the time window size*) –

### Examples

```
>>> a=rt.arange(10, dtype=rt.int64)
>>> a.timewindow_prod(a,5)
FastArray([    0,     0,     0,     0,     0,     0,   720,  5040, 20160,␣
↪60480], dtype=int64)
```

**timewindow_sum**(*time_array*, *time_dist*)

The input array must be int64 and sorted with ever increasing values. Sums up the values for a given time window.

**Parameters**

- **time_array** (*sorted integer array of timestamps*) –
- **time_dist** (*integer value of the time window size*) –

### Examples

```
>>> a=rt.arange(10, dtype=rt.int64)
>>> a.timewindow_sum(a,5)
FastArray([ 0,  1,  3,  6, 10, 15, 21, 27, 33, 39], dtype=int64)
```

**to_arrow**(*type=None*, ***, *preserve_fixed_bytes=False*, *empty_strings_to_null=True*)

Convert this *FastArray* to a pyarrow.Array.

**Parameters**

- **type** (*pyarrow.DataType, optional, defaults to None*) –
- **preserve_fixed_bytes** (*bool, optional, defaults to False*) – If this *FastArray* is an ASCII string array (dtype.kind == 'S'), set this parameter to True to produce a fixed-length binary array instead of a variable-length string array.
- **empty_strings_to_null** (*bool, optional, defaults To True*) – If this *FastArray* is an ASCII or Unicode string array, specify True for this parameter to convert empty strings to nulls in the output. riptable inconsistently recognizes the empty string as an 'invalid', so this parameter allows the caller to specify which interpretation they want.

**Return type**

pyarrow.Array or pyarrow.ChunkedArray

### Notes

**TODO: Add bool parameter which directs the conversion to choose the most-compact output type possible?**

This would be relevant to indices of categorical/dictionary-encoded arrays, but could also make sense for regular FastArray types (e.g. to use an int8 instead of an int32 when it'd be a lossless conversion).

**transitions**(*periods=1*, *fancy=False*)

Returns a boolean array. The boolean array is set to True when the previous item in the array does not equal the current. Use -1 instead of 1 if you want True set when the next item in the array does not equal the previous. See also: `differs`

**Parameters**

- **periods** (*int*) – The number of elements to look ahead (or behind), defaults to 1
- **fancy** (*bool*) – Indicates whether to return a fancy_index instead of a boolean array, defaults to False.

**Returns**

boolean `FastArray`, or fancyIndex (see: `fancy` kwarg)

```
>>> a = FastArray([0,1,2,3,3,3,4])
>>> a.transitions(periods=1)
FastArray([False, True, True, True, False, False, True])
```

```
>>> a.transitions(periods=2)
FastArray([False, False, True, True, True, False, True])
```

```
>>> a.transitions(periods=-1)
FastArray([ True, True, True, False, False, True, False])
```

**trunc**(***kwargs*)

**unique**(*return_index=False*, *return_inverse=False*, *return_counts=False*, *sorted=True*, *lex=False*, *dtype=None*, *filter=None*, ***kwargs*)

Find the unique elements of an array or the unique combinations of elements with corresponding indices in multiple arrays.

See `riptable.unique()` for full documentation.

**var**(*filter=None*, *dtype=None*, *axis=None*, *keepdims=None*, *ddof=None*, ***kwargs*)

Compute the variance of the values in the first argument.

Riptable uses the convention that `ddof = 1`, meaning the variance of `[x_1, ..., x_n]` is defined by `var = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the variance calculation. If the filter is uniformly `False`, *var* returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a *FastArray* `x`, `x.var(dtype = my_type)` is equivalent to `my_type(x.var())`.

**Returns**
The variance of the values.

**Return type**
scalar

**See also:**

`numpy.var`

*FastArray.nanvar*
Computes the variance of *FastArray* values, ignoring NaNs.

`Dataset.var`
Computes the variance of numerical `Dataset` columns.

`GroupByOps.var`
Computes the variance of each group. Used by `Categorical` objects.

## Notes

The `dtype` keyword for *FastArray.var* specifies the data type of the result. This differs from `numpy.var`, where it specifies the data type used to compute the variance.

**Notes on Using NumPy Parameters**

Using any of the following NumPy parameters will cause Riptable to switch to the NumPy implementation of this method (`numpy.var`). However, until a reported bug is fixed, if you also include the `dtype` parameter it will be applied to the result, not used to compute the variance as it is in `numpy.var`.

Also note that if you use any of the following NumPy parameters and also include a *filter* keyword argument (which `numpy.var` does not accept), Riptable's implementation of *var* will be used with the filter argument and the NumPy parameters will be ignored.

**axis**
[None or int or tuple of ints, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

**keepdims**
[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be passed through to the *var* method of sub-classes of `ndarray`, however any non-default value will be. If the sub-classes' method does not implement `keepdims`, any exceptions will be raised.

**ddof**
[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default `ddof` is zero for the NumPy implementation, versus one for the Riptable implementation.

### Examples

```
>>> a = rt.FastArray([1, 2, 3])
>>> a.var()
1.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 2, 3])
>>> a.var(dtype = rt.int32)
1
```

With a filter:

```
>>> a = rt.FastArray([1, 2, 3])
>>> b = rt.FastArray([False, True, True])
>>> a.var(filter = b)
0.5
```

**where**(*condition*, *y=np.nan*)

Return a new *FastArray* in which values are replaced where a given condition is False.

To also provide a value for where the condition is True, use `riptable.where()`.

**Parameters**

- **condition** (*bool or array of bool*) – Where the condition is True, keep the original value. Where False, replace with `y` (if `y` is a scalar) or the corresponding value from `y` (if `y` is an array). If `condition` is an array or a a comparison that returns an array, the array must be the same length as the calling *FastArray*.
- **y** (*scalar, array, or callable, default np.nan*) – The value to use where `condition` is False. If `y` is an array or a callable that returns an array, it must be the same length as the calling *FastArray*. The value of `y` that corresponds to the False value is used.

**Returns**
A new *FastArray* with values replaced where `condition` is False.

**Return type**
*FastArray*

**See also:**

**riptable.where**
Replace values depending on whether a given condition is True or False.

### Examples

`condition` is a comparison that creates an array of booleans, and `y` is a scalar:

```
>>> a = rt.FastArray(rt.arange(5))
>>> a
FastArray([0, 1, 2, 3, 4])
>>> a.where(a > 2, 100)
FastArray([100, 100, 100,   3,   4])
```

`condition` and `y` are same-length arrays:

```
>>> condition = rt.FastArray([True, True, False, False, False])
>>> y = rt.FastArray([100, 200, 300, 400, 500])
>>> a.where(condition, y)
FastArray([  0,   1, 300, 400, 500])
```

**class** riptable.rt_fastarray.**Ledger**

**static clear**()
Clear all the entries in the math ledger

**static dump**(*dataset=True*)
Print out the math ledger

**static off**()
Turn the math ledger off

**static on**()
Turn the math ledger on to record all array math routines

**static to_file**(*filename*)
Save the math ledger to a file

**class** riptable.rt_fastarray.**Recycle**

**static now**(*timeout=0*)
Pass the garbage collector timeout value to cleanup. Also calls the python garbage collector.

**Parameters**
**timeout** (*default to 0. 0 will not set a timeout*) –

**Return type**
total arrays deleted

**static off**()

**static on**()
Turn riptable recycling on. Used only when riptable recycling was turned off.

### Example

a=arange(1_000_00) Recycle.off() %timeit a=a + 1 Recycle.on() %timeit a=a + 1

**static timeout**(*timeout=100*)

Pass the garbage collector timeout value to expire. The timeout value is roughly in 2/5 secs. A value of 100 is usually about 40 seconds. If an array has not been reused by the timeout, it is permanently deleted.

**Return type**
previous timespan

class riptable.rt_fastarray.**Threading**

**static off**()

Turn riptable threading off. Useful for when the system has other processes using other threads or to limit threading resources.

### Example

a=rt.arange(1_000_00) Threading.off() %time a+=1 Threading.on() %time a+=1

**Return type**
Previously whether threading was on or not. 0 or 1. 0=threading was off before.

**static on**()

Turn riptable threading on. Used only when riptable threading was turned off.

### Example

a=rt.arange(1_000_00) Threading.off() %time a+=1 Threading.on() %time a+=1

**Return type**
Previously whether threading was on or not. 0 or 1. 0=threading was off before.

**static threads**(*threadcount*)

Set how many worker threads riptable can use. Often defaults to 12 and cannot be set below 1 or > 31.

To turn riptable threading off completely use Threading.off() Useful for when the system has other processes using other threads or to limit threading resources.

### Example

Threading.threads(8)

**Return type**
number of threads previously used

### 2.2.16 riptable.rt_fastarraynumba

#### Functions

| | |
|---|---|
| *fill_backward*(arr[, fill_val, inplace, limit]) | Replace NaN and invalid array values by propagating the next encountered valid value |
| *fill_forward*(arr[, fill_val, inplace, limit]) | Replace NaN and invalid array values by propagating the last encountered valid value |

riptable.rt_fastarraynumba.**fill_backward**(*arr*, *fill_val=None*, *inplace=False*, *limit=0*)

Replace NaN and invalid array values by propagating the next encountered valid value backward.

Note that this method can be called either as a `FastArray` method (*riptable.rt_fastarraynumba.fill_backward()*) or a function (`riptable.fill_backward()`) that takes an array or `FastArray` as input. The function returns either an array or a `FastArray`, depending on the original input.

**Parameters**

- **fill_val** (*scalar, default None*) – The value to use where there is no valid value to propagate backward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid value to propagate backward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object.
- **limit** (*int, default 0*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.

**Returns**

The `FastArray` will be the same size and have the same dtype as the original input.

**Return type**

`FastArray`

**See also:**

*riptable.rt_fastarraynumba.fill_forward*
Replace NaN and invalid values with the last valid value.

*riptable.rt_fastarraynumba.fill_backward*
Replace NaN and invalid values with the next valid value.

**riptable.fill_backward**
Replace NaN and invalid values with the next valid value.

**FastArray.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**FastArray.replacena**
Replace NaN and invalid values with a specified value.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**Categorical.fill_backward**
Replace NaN and invalid values with the next valid group value.

**GroupBy.fill_backward**
Replace NaN and invalid values with the next valid group value.

### Examples

Use a `fill_val` to replace values where there's no valid value to propagate backward:

```
>>> a = rt.FastArray([0.0, rt.nan, rt.nan, rt.nan, 4.0, rt.nan])
>>> a.fill_backward(fill_val = 0)
FastArray([0., 4., 4., 4., 4., 0.])
```

Using `riptable.fill_backward()`:

```
>>> a = rt.FastArray([0.0, rt.nan, rt.nan, rt.nan, 4.0, rt.nan])
>>> rt.fill_backward(a, fill_val = 0)
FastArray([0., 4., 4., 4., 4., 0.])
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values:

```
>>> a.fill_backward(limit = 1)
FastArray([ 0., nan, nan,  4.,  4., nan])
```

riptable.rt_fastarraynumba.**fill_forward**(*arr*, *fill_val=None*, *inplace=False*, *limit=0*)

Replace NaN and invalid array values by propagating the last encountered valid value forward.

Note that this method can be called either as a `FastArray` method (*riptable.rt_fastarraynumba.fill_forward()*) or a function (`riptable.fill_forward()`) that takes an array or `FastArray` as input. The function returns either an array or a `FastArray`, depending on the original input.

**Parameters**

- **fill_val** (*scalar, default None*) – The value to use where there is no valid value to propagate forward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid value to propagate forward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object.
- **limit** (*int, default 0*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.

**Returns**
The `FastArray` will be the same size and have the same dtype as the original input.

**Return type**
`FastArray`

**See also:**

*riptable.rt_fastarraynumba.fill_backward*
Replace NaN and invalid values with the next valid value.

*riptable.rt_fastarraynumba.fill_forward*
Replace NaN and invalid values with the last valid value.

**riptable.fill_forward**
Replace NaN and invalid values with the last valid value.

**FastArray.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**FastArray.replacena**
Replace NaN and invalid values with a specified value.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**Categorical.fill_forward**
Replace NaN and invalid values with the last valid group value.

**GroupBy.fill_forward**
Replace NaN and invalid values with the last valid group value.

### Examples

Use a `fill_val` to replace values where there's no valid value to propagate forward:

```
>>> a = rt.FastArray([rt.nan, 1.0, rt.nan, rt.nan, rt.nan, 5.0])
>>> a.fill_forward(fill_val = 0)
FastArray([0., 1., 1., 1., 1., 5.])
```

Using `riptable.fill_forward()`:

```
>>> a = rt.FastArray([0.0, rt.nan, rt.nan, rt.nan, 4.0, rt.nan])
>>> rt.fill_forward(a, fill_val = 0)
FastArray([0., 0., 0., 0., 4., 4.])
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values:

```
>>> a.fill_forward(limit = 1)
FastArray([nan,  1.,  1., nan, nan,  5.])
```

## 2.2.17 riptable.rt_groupby

### Classes

*GroupBy*

**param dataset**
The dataset object

**class** riptable.rt_groupby.**GroupBy**(*dataset*, *keys=None*, *filter=None*, *ordered=None*, *sort_display=None*, *return_all=False*, *hint_size=0*, *lex=None*, *rec=False*, *totals=False*, *copy=False*, *cutoffs=None*, *verbose=False*, ***kwargs*)

Bases: *riptable.rt_groupbyops.GroupByOps*

**Parameters**

- **dataset** (Dataset) – The dataset object
- **keys** (*list. List of column names to groupby*) –
- **filter** (*None. Boolean mask array applied as filter before grouping*) –

- **return_all** (*bool. Default to False. When set to True will return all*) – the dataset columns for every operation.
- **hint_size** (*int. Hint size for the hash (optional)*) –
- **sort_display** (*bool. Default to True. Indicates*) –
- **lex** (*bool*) – Defaults to False. When True uses a lexsort to find the groups (otherwise uses a hash).
- **totals** (*bool*) –

**property gb_keychain**

**property gbkeys**

dictionary of numpy arrays binned from

**property ifirstkey**

**property ilastkey**

**property isortrows**

sorted index or None

**property transform**

The property transform sets a flag so that the next reduce function called after transform, will repopulate the original array with the reduced value.

### Example

```
>>> ds.groupby(['side', 'venue']).transform.sum()
```

**DebugMode = False**

**TestCatGb = True**

**__getattr__**(*name*)

__getattr__ is hit when '.' is used to trim a single column.

### Examples

```
>>> ds = Dataset({'col_'+str(i): np.random.rand(5) for i in range(5)})
>>> ds.keycol = FA(['a','a','b','c','a'])
>>> ds.gb('keycol').col_4.mean()
*keycol   col_4
-------   -----
a          0.73
b          0.03
c          0.76
```

**__getitem__**(*fld*)

**__iter__**()

Generates tuples of key, value pairs. Keys are key values for single key, or tuples of key values for multikey. Values are datasets containing all rows from data in group for that key.

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**_build_string**()

**_calculate_all**(*funcNum*, **args*, *func_param=0*, ***kwargs*)

Generate a GroupByKeys object if necessary and ask for the result of a calculation from the grouping object. Returns: a grouped by dataset with the result from the calculation

**_getitem**(*fld*)

Called by __getitem__ and __getattr__. Uses the field to index into the stored dataset. Often used to limit the data the groupby operation is being performed on. Returns a shallow copy of the groupby object.

This routine gets hit during the following common code pattern:

```
>>> ds = Dataset({'col_'+str(i): np.random.rand(5) for i in range(5)})
>>> ds.keycol = FA(['a','a','b','c','a'])
>>> ds.gb('keycol')[['col_1', 'col_2']].sum()
*keycol   col_1   col_2
-------   -----   -----
a          1.92    0.89
b          0.70    0.46
c          0.07    0.42
```

```
>>> ds.gb('keycol').col_4.mean()
*keycol   col_4
-------   -----
a          0.73
b          0.03
c          0.76
```

**_grouping_data_as_dict**(*ds*)

**_pop_gb_data**(*calledfrom*, *userfunc*, **args*, ***kwargs*)

GroupBy holds on to its dataset. There may be no additional data provided.

**add_totals**(*gb_ds*)

**as_categorical**()

Returns a categorical using the same binning information as the GroupBy object (no addtl. hash required). New categorical will not share a grouping object with this groupby object, but will share a reference to the iKey. Categorical operation results will be sorted or unsorted depending on if 'gb' or 'gbu' called this.

**backfill**(*limit=0*, *fill_val=None*, *inplace=False*)

Backward fill the values

**Parameters**

**limit** (*integer, optional*) – limit of how many values to fill

**See also:**

*fill_forward*, *fill_backward*, fill_invalid

**copy**(*deep=True*)

Called from getitem when user follows gb with []

**count**(***kwargs*)

Compute count of group

**abstract expanding**(***kwargs*)

**fill_backward**(*limit=0*, *fill_val=None*, *inplace=False*)

Replace NaN and invalid array values by propagating the next encountered valid group value backward.

**Parameters**

- **limit** (*int, default 0*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.
- **fill_val** (*scalar, default None*) – The value to use where there is no valid group value to propagate backward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate backward.
- ****kwargs** – Additional keyword arguments.

**Returns**

The returned `Dataset` contains the input `Dataset` object's numerical columns.

**Return type**

`Dataset`

**See also:**

*GroupBy.fill_forward*

Replace NaN and invalid array values with the last valid group value.

**Categorical.fill_backward**

Replace NaN and invalid array values with the next valid group value.

**riptable.fill_backward**

Replace NaN and invalid values with the next valid value.

**Dataset.fillna**

Replace NaN and invalid values with a specified value or nearby data.

**FastArray.fillna**

Replace NaN and invalid values with a specified value or nearby data.

**FastArray.replacena**

Replace NaN and invalid values with a specified value.

### Examples

```
>>> ds = rt.Dataset({'Key_col' : ['A', 'B', 'A', 'B', 'A', 'B'],
...                  'Vals' : [rt.nan, rt.nan, 2, 3, 4, 5]})
>>> ds.gb('Key_col').fill_backward()
#   Vals
-   ----
0   2.00
1   3.00
```

(continues on next page)

(continued from previous page)

```
2   2.00
3   3.00
4   4.00
5   5.00
```

Use a `fill_val` to replace values where there's no valid group value to propagate backward:

```
>>> ds.Vals = rt.FastArray([0, 1, 2, 3, rt.nan, rt.nan])
>>> ds.gb('Key_col').fill_backward(fill_val = 0)
#   Vals
-   ----
0   0.00
1   1.00
2   2.00
3   3.00
4   0.00
5   0.00
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values in a group:

```
>>> ds.Vals = rt.FastArray([rt.nan, rt.nan, rt.nan, rt.nan, 4, 5])
>>> ds.gb('Key_col').fill_backward(limit = 1)
#   Vals
-   ----
0    nan
1    nan
2   4.00
3   5.00
4   4.00
5   5.00
```

**fill_forward**(*limit=0*, *fill_val=None*, *inplace=False*)

Replace NaN and invalid array values by propagating the last encountered valid group value forward.

**Parameters**

- **limit** (*int, default 0*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.
- **fill_val** (*scalar, default None*) – The value to use where there is no valid group value to propagate forward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate forward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object. This fails if the array is locked.

**Returns**

The returned `Dataset` contains the input `Dataset` object's numerical columns.

**Return type**

`Dataset`

**See also:**

*GroupBy.fill_backward*
Replace NaN and invalid array values with the next valid group value.

**Categorical.fill_forward**
Replace NaN and invalid array values with the last valid group value.

**riptable.fill_forward**
Replace NaN and invalid values with the last valid value.

**Dataset.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**FastArray.fillna**
Replace NaN and invalid values with a specified value or nearby data.

**FastArray.replacena**
Replace NaN and invalid values with a specified value.

### Examples

```
>>> ds = rt.Dataset({'Key_col' : ['A', 'B', 'A', 'B', 'A', 'B'],
...                  'Vals' : [0, 1, 2, 3, rt.nan, rt.nan]})
>>> ds.gb('Key_col').fill_forward()
#   Vals
-   ----
0   0.00
1   1.00
2   2.00
3   3.00
4   2.00
5   3.00
```

Use a `fill_val` to replace values where there's no valid group value to propagate forward:

```
>>> ds.Vals = rt.FastArray([rt.nan, rt.nan, 2, 3, 4, 5])
>>> ds.gb('Key_col').fill_forward(fill_val = 0)
#   Vals
-   ----
0   0.00
1   0.00
2   2.00
3   3.00
4   4.00
5   5.00
```

Replace only the first NaN or invalid value in any consecutive series of NaN or invalid values in a group:

```
>>> ds.Vals = rt.FastArray([0, 1, rt.nan, rt.nan, rt.nan, rt.nan])
>>> ds.gb('Key_col').fill_forward(limit = 1)
#   Vals
-   ----
0   0.00
1   1.00
2   0.00
3   1.00
```

(continues on next page)

(continued from previous page)

```
4    nan
5    nan
```

**get_group**(*category*, ***kwargs*)

The name of the group to get as a Dataset.

**Parameters**

**category** (*string or tuple*) – A value from the column used to construct the GroupBy, or if multiple columns were used, a tuple of the multiple columns.

**Return type**

*Dataset*

### Example

```
>>> ds.groupby('symbol').get_group('AAPL')
```

**nth**(*n=1*)

Select the nth row from each group.

**Parameters**

**n** (*int*) – A single nth value for the row

### Examples

```
>>> ds = rt.Dataset({'A': [1, 1, 2, 1, 2],
...                  'B': [np.nan, 2, 3, 4, 5]})
>>> g = ds.groupby('A')
>>> g.nth(0)
*A      B
--   ----
1    nan
2   3.00

[2 rows x 2 columns] total bytes: 32.0 B
```

```
>>> g.nth(1)
*A      B
--   ----
1  2.00
2  5.00

[2 rows x 2 columns] total bytes: 32.0 B
```

```
>>> g.nth(-1)
*A      B
--   ----
1  4.00
2  5.00

[2 rows x 2 columns] total bytes: 32.0 B
```

**pad**(*limit=0*, *fill_val=None*, *inplace=False*)

Forward fill the values

**Parameters**

**limit** (*integer, optional*) – limit of how many values to fill

**See also:**

*fill_forward*, *fill_backward*, fill_invalid

**abstract stack**(***kwargs*)

**abstract unstack**(***kwargs*)

### 2.2.18 riptable.rt_groupbykeys

#### Classes

| | |
|---|---|
| *GroupByKeys* | Handles masking, appending invalid, and sorting of key columns for a groupby operation. |

**class** riptable.rt_groupbykeys.**GroupByKeys**(*grouping_dict*, *ifirstkey=None*, *isortrows=None*, *sort_display=False*, *pre_sorted=False*, *prebinned=False*)

Handles masking, appending invalid, and sorting of key columns for a groupby operation.

**Parameters**

- **grouping_dict** (*dict*) – Non-unique or unique key columns.
- **ifirstkey** (*array, optional*) – If set, first occurence to generate unique values from non-unique keys. Sometimes these are lazily evaluated and do not correspond to the grouping dict held. See the prebinned keyword.
- **isortrows** (*array, optional*) – A sorted index for the unique array. May be calculated later on, after grouping_dict is reduced to unique values.
- **sort_display** (*bool, default False*) – If True, unique keys in result of operation will be sorted. Otherwise, will appear unsorted.
- **pre_sorted** (*bool, default False*) – Unique grouping_dict is already in a sorted order, do not apply / calculate *isortrows*, even if sort on is True.
- **prebinned** (*bool, default False*) – If True, grouping_dict contains unique values. ifirstkey will not be stored. If False, grouping_dict contains non-unique values, the default if constructed by a GroupBy object.

#### Notes

**Constructor**

GroupByKeys has two main ways of initialization:

1. From a non-unique grouping_dict and ifirstkey (fancy index to unique values). The object will hold on to both, and lazily generate the groupby keys as necessary.
2. From already binned gbkeys (unique values). Most categoricals will initialize GroupByKeys this way. Because Categoricals are sometimes naturally sorted, they may set the pre_sorted keyword to True.

If sort_display is True and the keys are not already sorted, the gbkeys will be sorted in-place the first time a groupby calculation is made. After being sorted, the internal _sort_applied flag will be set. Despite the keys being sorted, the sort might still need to be applied to the data columns of the groupby calculation's result.

**Lazy Evaluation**

- If isortrows is not provided and the gbkeys are not pre-sorted, a lexsort will be performed, and the keys will be sorted inplace.
- If gbkeys are requested with a filter bin, a new bin will be permanently prepended to each key array. After the filtered bin is added, the gbkeys will still default to return arrays without the filter (a view of the held arrays offset by 1).
- Multikey labels are a list of strings of the tuples that will appear when a multikey is displayed. They will also add a filtered tuple as necessary, and default to a reduced view after the addition - just like the gbkeys. Multikey labels will not be generated until they are requested for display (because they are constructed in a python loop, generating these is expensive).

**property gbkeys**

Generates groupby keys if necessary. Returns groupby keys.

**property gbkeys_filtered**

Adds a filter to the gbkeys, or returns the already filtered gbkeys.

**property isortrows**

Generates isortrows (index to sort groupby keys). Possibly performs a lexsort.

**property multikey**

Returns True if GroupByKeys object is holding multiple columns in _gbkeys

**property multikey_labels**

**property multikey_labels_filtered**

**property singlekey**

Returns True if GroupByKeys object is holding a single column in _gbkeys

**property sort_gb_data**

If a sort has been applied to the gbkeys, they do not need to be sorted, however the data resulting from a groupby calculation is naturally unsorted and will still need a sort applied.

**property unique_count**

Returns number of unique groupby keys - lazily evaluated and stored.

**__getitem__**(*index*)

**__repr__**()

Return repr(self).

**__str__**()

Return str(self).

**_build_string**()

**_get_filter_bin_name**(*arr*)

**_get_index_from_tuple**(*tup*)

If the GroupByKeys object is holding a multikey dictionary, it can be indexed by a tuple. This internal routine (called by get_index_from_bin/__getitem__) will return the bin index of matching multikey entries or -1 if not found. Any string/bytes values will be fixed to match the string/bytes column.

_insert_filter_bin()

_insert_filter_label()

_make_isortrows()

_pull_from_ifirstkey()

_trim_keys(*keys*)

Return a trimmed view of the keys so the filtered bin is not included. Also trims list of multikey labels

copy(*deep=False*)

Creates a deep or shallow copy of the grouping

get_bin(*index*)

**Parameters**
**index** – int or list of integers

**Return result_bins**
matching bins for provided indices or an empty list

get_bin_from_index(*index*)

**Parameters**
**index** – int or list of integers

**Return result_bins**
matching bins for provided indices or an empty list

get_index_from_bin(*bin*)

**Parameters**
**bin** – a tuple of multiple keys or a single key (will be converted to tuple)

**Return index**
the bin index, or -1 if not found.

keys(*sort=None*, *showfilter=False*)

Return unique keys, possibly apply a sort and add a filter bin.

labels(*showfilter=False*)

Generates list of tuples from multikey columns.

unique_unsorted()

Pull the unique keys unsorted, using iFirstKey or the prebinned uniques.

unsort()

Sets the internal _sort_display flag to False. Will warn the user if the groupby keys are already sorted or were pre-sorted when GroupByKeys were constructed.

### 2.2.19 riptable.rt_groupbynumba

#### Classes

| | |
|---|---|
| *GroupbyNumba* | Holds all the functions for groupby |

**class** riptable.rt_groupbynumba.**GroupbyNumba**

Bases: *riptable.rt_groupbyops.GroupByOps*

Holds all the functions for groupby

Only used when inherited

Child class must set self.grouping and self._dataset Child class must also override methods; count, _calculate_all, and the property; gb_keychain

**CORE_COUNT = 12**

**static _nb_fill_backend**(*iGroup*, *iFirstGroup*, *nCountGroup*, *binLow*, *binHigh*, *data*, *ret*, *fill_val*, *limit*, *direction*)

Numba backend implementation for grouped fill_forward and fill_backward for all aplicable dtypes.

**Parameters**

- **iGroup** (*np.ndarray*) – Arrays from a groupby object's 'get_groupings' method
- **iFirstGroup** (*np.ndarray*) – Arrays from a groupby object's 'get_groupings' method
- **nCountGroup** (*np.ndarray*) – Arrays from a groupby object's 'get_groupings' method
- **binLow** (*int*) – Indexes corresponding to the first and the last groups in iFirstGroup and nCountGroup
- **binHigh** (*int*) – Indexes corresponding to the first and the last groups in iFirstGroup and nCountGroup
- **data** (*array*) – The original data to be opperated on
- **ret** (*array*) – An empty array the same size as 'data' which will contain the processed data. Must be `None` for inplace operation
- **fill_val** (*parameters for nb_fill_forward/nb_fill_backward*) – The value to use where there is no valid group value to propagate forward/backward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate forward/backward.
- **limit** (*parameters for nb_fill_forward/nb_fill_backward*) – The value to use where there is no valid group value to propagate forward/backward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate forward/backward.
- **direction** (*int (-1 or 1)*) – direction = 1 corresponds to fill_forward, -1 corresponds to fill_backward

**_nb_groupbycalculateall**(*ikey*, *unique_rows*, *funcList*, *binLowList*, *binHighList*, *func_param*)

_nb_groupbycalculateallpack(*ikey*, *iGroup*, *iFirstGroup*, *nCountGroup*, *unique_rows*, *funcList*, *binLowList*, *binHighList*, *inplace*, *func_param*)

_numbaEMA(*iFirstGroup*, *nCountGroup*, *binLow*, *binHigh*, *data*, *ret*, *time*, *decayRate*)

_numbaEMA2(*iFirstGroup*, *nCountGroup*, *data*, *ret*, *time*, *decayRate*)

For each group defined by the grouping arguments, sets 'ret' to a true EMA of the 'data' argument using the time argument as the time and the 'decayRate' as the decay rate.

**Parameters**

- **iGroup** (*from a groupby object's 'get_groupings' method*) –
- **iFirstGroup** (*from a groupby object's 'get_groupings' method*) –
- **nCountGroup** (*from a groupby object's 'get_groupings' method*) –
- **data** (*the original data to be opperated on*) –
- **ret** (*a blank array the same size as 'data' which will return the processed data*) –
- **time** (*a list of times associated to the rows of data*) –
- **decayRate** (*the decay rate (e based)*) –
- **TODO** (*Error checking.*) –

_numbaFillBackward(*iFirstGroup*, *nCountGroup*, *data*, *ret*)

propogate backward non-NaN values within a group, overwriting NaN values. TODO: better documentation

_numbaFillForward(*iFirstGroup*, *nCountGroup*, *data*, *ret*)

propogate forward non-NaN values within a group, overwriting NaN values. TODO: better documentation

_numbaTrim(*iFirstGroup*, *nCountGroup*, *data*, *ret*, *x*, *y*)

For each group defined by the grouping arguments, sets 'ret' to be a copy of the 'data' with elements below the 'x'th percentile or above the 'y'th percentile of the group set to nan.

**Parameters**

- **iGroup** (*from a groupby object's 'get_groupings' method*) –
- **iFirstGroup** (*from a groupby object's 'get_groupings' method*) –
- **nCountGroup** (*from a groupby object's 'get_groupings' method*) –
- **data** (*the original data to be opperated on*) –
- **ret** (*a blank array the same size as 'data' which will return the processed data*) –
- **x** (*the lower percentile bound*) –
- **y** (*the upper percentile bound*) –

static _numba_fill_direction(*direction*, *iGroup*, *iFirstGroup*, *nCountGroup*, *binLow*, *binHigh*, *data*, *ret*, *fill_val*, *limit*)

_numbamin(*unique_rows*, *binLow*, *binHigh*, *data*, *ret*)

_numbasum(*unique_rows*, *binLow*, *binHigh*, *data*, *ret*)

**grpFillBackward**()

propogate backward non-NaN values within a group, overwriting NaN values. TODO: better documentation

**grpFillForward**()

propogate forward non-NaN values within a group, overwriting NaN values. TODO: better documentation

**grpFillForwardBackward**()

propogate forward, then backward, non-NaN values within a group, overwriting NaN values. TODO: better documentation

**grpTrim**(*x*, *y*)

For each column, for each group, determine the x'th and y'th percentile of the data and set data below the x'th percentile or above the y'th percentile to nan.

**Parameters**

- **grp** (*a groupby object*) –
- **x** (*lower percentile*) –
- **y** (*uppper percentile*) –

**Returns**

- *A dataset with the values outside the given percentiles set to np.nan*
- **TODO** (*Test column types to make sure that the numba code will work nicely*)

**nb_ema**(**args*, *time=None*, *decay_rate=None*, ***kwargs*)

**Parameters**

- **time** (*an array of times (often in nanoseconds) associated to the rows of data*) –
- **decayRate** (*the scalar decay rate (e based)*) –

**nb_fill_backward**(**args*, *fill_val*, *limit=0*, *inplace=False*)

Replace NaN and invalid array values by propagating the next encountered valid group value backward.

Optionally, you can modify the original array if it's not locked.

**Parameters**

- ***args** (*array or list of arrays*) – The array or arrays that contain NaN or invalid values you want to replace.
- **limit** (*int, default 0 (disabled)*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.
- **fill_val** (*scalar, default None*) – The value to use where there is no valid group value to propagate backward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate backward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object. This fails if the array is locked.

**Returns**

The dataset (categorical) will be the same size and have the same dtypes as the original input.

**Return type**
Dataset-like object

**nb_fill_forward**(**args*, *limit=0*, *fill_val=None*, *inplace=False*)

Replace NaN and invalid array values by propagating the last encountered valid group value forward.

Optionally, you can modify the original array if it's not locked.

**Parameters**

- ***args** (*array or list of arrays*) – The array or arrays that contain NaN or invalid values you want to replace.
- **limit** (*int, default 0 (disabled)*) – The maximium number of consecutive NaN or invalid values to fill. If there is a gap with more than this number of consecutive NaN or invalid values, the gap will be only partially filled. If no `limit` is specified, all consecutive NaN and invalid values are replaced.
- **fill_val** (*scalar, default None*) – The value to use where there is no valid group value to propagate forward. If `fill_val` is not specified, NaN and invalid values aren't replaced where there is no valid group value to propagate forward.
- **inplace** (*bool, default False*) – If False, return a copy of the array. If True, modify original data. This will modify any other views on this object. This fails if the array is locked.

**Returns**
The dataset (categorical) will be the same size and have the same dtypes as the original input.

**Return type**
Dataset-like object

**nb_min**(**args*, ***kwargs*)

Compute sum of group

**nb_sum**(**args*, ***kwargs*)

Compute sum of group

**nb_sum_punt_test**(**args*, ***kwargs*)

Compute sum of group

### 2.2.20 riptable.rt_groupbyops

#### Classes

| | |
|---|---|
| *GroupByOps* | Holds all the functions for groupby |

**class** riptable.rt_groupbyops.**GroupByOps**

Bases: abc.ABC

Holds all the functions for groupby

Only used when inherited

Child class must set self.grouping and self._dataset Child class must also override methods; count, _calculate_all, and the property; gb_keychain

**property first_bool**

Return a boolean mask of the first occurrence.

### Examples

```
>>> c = rt.Cat(['this','this','that','that','this'])
>>> c.first_bool
FastArray([ True, False,  True, False, False])
```

**property first_fancy**

Return a fancy index mask of the first occurrence

### Notes

NOTE: not optimized for groupby which has grouping.ikey always set NOTE: categorical needs to lazy evaluate ikey

### Examples

```
>>> c = rt.Cat(['b','b','a','a','b'])
>>> c.first_fancy
FastArray([0, 2])
```

```
>>> c=Cat(['b','b','a','a','b'], ordered=False)
>>> c.first_fancy
FastArray([2, 0])
```

**abstract property gb_keychain: *riptable.rt_groupbykeys.GroupByKeys***

**property groups**

Returns a dictionary of unique key values -> their fancy indices of occurrence in the original data.

**property last_bool**

Return a boolean mask of the last occurrence.

### Examples

```
>>> c = rt.Cat(['this','this','that','that','this'])
>>> c.last_bool
FastArray([ False, False,  False, True, True])
```

**property last_fancy**

Return a fancy index mask of the last occurrence

### Notes

NOTE: not optimized for groupby which has grouping.ikey always set NOTE: categorical needs to lazy evaluate ikey

### Examples

```
>>> c = rt.Cat(['b','b','a','a','b'])
>>> c.last_fancy
FastArray([3, 4])
```

```
>>> c=Cat(['b','b','a','a','b'], ordered=False)
>>> c.last_fancy
FastArray([4, 3])
```

**AggNames**

**DebugMode = False**

**NumpyAggNames**

**QUANTILE_MULTIPLIER = 1000000000.0**

**_USE_FAST_COUNT_UNIQUES = True**

**_dataset: *riptable.rt_dataset.Dataset* | None**

**grouping: *riptable.rt_grouping.Grouping***

**abstract _calculate_all**(*funcNum*, **args*, *func_param=0*, *gbkeys=None*, *isortrows=None*, ***kwargs*)

**_dict_val_at_index**(*index*)

Returns the value of the group label for a given index. A single-key grouping will return a single value. A multi-key grouping will return a tuple of values.

**_ema_op**(*function*, **args*, *time=None*, *decay_rate=1.0*, *filter=None*, *reset_filter=None*, ***kwargs*)

Ema base function for time based ema functions

Formula:

**grp loops over each item in a groupby group**

**i loops over eachitem in the original dataset**

Output[i] = <some formula>

**Parameters**

- **time** (*float or int array used to calculate time difference*) –
- **decay_rate** (*see formula, used a half life*) –
- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –

**Return type**

Dataset same rows as original dataset

**_gb_keyword_wrapper**(*filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

**static _gb_quantile_name**(*q*, *is_nan_function*)

Returns a correct name of a quantile function given q and nan-flag

**_get_agg_func**(*item*)

Translates user input into name and method for groupby aggregations.

**Parameters**

**item** (*str or function*) – String or supported numpy math function. See GroupByOps.AggNames.

**Returns**

- **name** (*str*) – Lowercase name for aggregation function.
- **func** (*function*) – GroupByOps method.

**_iter_internal**(*dataset=None*)

Generates pairs of labels and the stored dataset sliced by their fancy indices. Right now, this is only called by categorical. Groupby has a faster way of return dataset slices.

**_iter_internal_contiguous**()

Sorts the data by group to create contiguous memory. Returns key + dataset view of key's rows for each group.

**_keys_as_list**()

**_nth**(**args*, *n=1*, ***kwargs*)

**_pop_gb_data**(*calledfrom*, *userfunc*, **args*, ***kwargs*)

Pop the groupby data from the args and keyword args, possibly combining. Avoid repeating this step when the data doesn't change.

**Parameters**

- **calledfrom** (*{'apply_reduce', 'apply_nonreduce', 'apply', 'agg'}*) –
- **userfunc** (*callable or int (function number)*) –

**Returns**

- *4 return values*
- *any user arguments*
- *the kwargs (with 'dataset' removed)*
- *the dictionary of numpy arrays to operarte on*
- **tups** (*0 or 1 or 2 depending on whether the first argument was a tuple of arrays*)

**See also:**

*GroupByOps.agg*

**_possibly_transform**(*gb_ds*, *label_keys=None*, ***kwargs*)

Called after a reduce operation to possibly re-expand back. Check transform flag.

**_prepare_gb_data**(*calledfrom*, *userfunc*, **args*, *dataset=None*, ***kwargs*)

**Parameters**

- **calledfrom** (*'Accum2', 'Categorical','GroupBy','apply_reduce', 'apply_nonreduce','apply','agg'*) –
- **userfunc** (*a callable function or a function number*) –
- **allowed)** (*args or dataset must be present (both also*) – if just args: make a dictionary from that if just dataset: make dictionary if both: make a new dataset, then make a dictionary from that if neither: error

  from Grouping, normally just a dataset from Categorical, normally just args (but user can use kwarg 'dataset' to supply one)
- **Grouping** (*This routine normalizes input from*) –
- **Accum2** –
- **Categorical** –
- **dataset.** (*GroupBy defaults to use the _dataset variable that it sets after being constructed from a*) –
- **methods.** (*no input data is required for the calculation*) –
- **for** (*Accum2 and Categorical can also set _dataset just like Groupby. See Dataset.accum2 and Dataset.cat*) –
- **examples.** –
- **set** (*If a _dataset has been*) –
- **methods.** –
- **arrays** (*internal function to parse argument and search for numpy*) –

**Returns**

- *a dictionary of arrays to be used as input to many groupby algorithms*
- *user_args if any (the first argument might be removed)*
- **tups** (*0 or or 2. Will be set to T> 0 if the first argument is a tuple*)

**Raises**

ValueError –

**_quantile**(**args*, *q=None*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *is_nan_function=None*, *is_percentile=None*, ***kwargs*)

Internal function for all (nan)quantile/percentile/median operations.

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **q** (*float, list of floats*) – Quantile(s) or percentile(s) to compute
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.

- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **is_nan_function** (*bool*) – Indicates if this was called a nan-version of a function.
- **is_percentile** (*bool*) – Indicates if this was called a (nan)percentile.

**static _quantile_funcParam_from_q**(*q*, *is_nan_function*)

Returns a funcParam to be passed to a cpp level. Multiplier is needed because functions only take interger funcParams See GroupByBase::AccumQuantile1e9Mult function in riptide_cpp/src/GroupBy.cpp

**static _quantile_q_from_funcParam**(*funcParam*)

Decodes a quantile q and a nan-flag from funcParam used for cpp level.

**agg**(*func=None*, **args*, *dataset=None*, ***kwargs*)

**Parameters**

**func** (*callable, string, dictionary, or list of string/callables*) – Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns**

**aggregated**

**Return type**

*Multiset*

### Notes

Numpy functions mean/median/prod/sum/std/var are special cased so the default behavior is applying the function along axis=0

### Examples

Aggregate these functions across all columns

```
>>> gb.agg(['sum', 'min'])
            A         B         C
sum -0.182253 -0.614014 -2.909534
min -1.916563 -1.460076 -1.568297
```

Different aggregations per column

```
>>> gb.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
            A         B
max       NaN  1.514318
min -1.916563 -1.460076
sum -0.182253       NaN
```

```
>>> gb.agg({'C': np.sum, 'D': lambda x: np.std(x,ddof=1)})
```

**aggregate**(*func*)

**apply**(*userfunc*, **args*, *dataset=None*, *label_keys=None*, ***kwargs*)

GroupByOps:apply calls Grouping:apply

**Parameters**

- **userfunc** (*callable*) – userfunction to call
- **dataset** (*None*) –
- **label_keys** (*None*) –

**apply_nonreduce**(*userfunc*, **args*, *dataset=None*, *label_keys=None*, *func_param=None*, *dtype=None*, ***kwargs*)

GroupByOps:apply_nonreduce calls Grouping:apply_reduce

**Parameters**

- **userfunc** (*callable*) – A callable that takes a contiguous array as its first argument, and returns a scalar. In addition the callable may take positional and keyword arguments.
- **args** – used to pass in columnar data from other datasets
- **dataset** (*None*) – User may pass in an entire dataset to compute.
- **label_keys** (*None.*) – Not supported, will use the existing groupby keys as labels.
- **dtype** (*str or np.dtype, optional*) – Change to a numpy dtype to return an array with that dtype. Defaults to None.
- **kwargs** – Optional positional and keyword arguments to pass to `userfunc`

### Notes

Grouping apply_reduce (for Categorical, groupby, accum2)

For every column of data to be computed, the userfunc will be called back per group as a single array. The order of the groups is either:

- Order of first apperance (when coming from a hash)
- Lexigraphical order (when lex=True or a Categorical with ordered=True)

The function passed to apply must take an array as its first argument and return back a single scalar value.

### Examples

From a Dataset groupby:

```
>>> ds.gb(['Symbol'])['TradeSize'].apply_reduce(np.sum)
```

From an existing categorical:

```
>>> ds.Symbol.apply_reduce(np.sum, ds.TradeSize)
```

Create your own with forced dtype:

```
>>> def mycumprodsum(arr):
>>>     return arr.cumprod().sum()
>>> ds.Symbol.apply_reduce(mycumprodsum, ds.TradeSize, dtype=np.float32)
```

**apply_reduce**(*userfunc*, **args*, *dataset=None*, *label_keys=None*, *nokeys=False*, *func_param=None*, *dtype=None*, *transform=False*, ***kwargs*)

GroupByOps:apply_reduce calls Grouping:apply_reduce

**Parameters**

- **userfunc** (*callable*) – A callable that takes a contiguous array as its first argument, and returns a scalar In addition the callable may take positional and keyword arguments.
- **args** – Used to pass in columnar data from other datasets
- **dataset** (*None*) – User may pass in an entire dataset to compute.
- **label_keys** (*None*) – Not supported, will use the existing groupby keys as labels.
- **func_param** (*tuple, optional*) – Set to a tuple to pass as arguments to the routine.
- **dtype** (*str or np.dtype, optional*) – Change to a numpy dtype to return an array with that dtype. Defaults to None.
- **transform** (*bool*) – Set to True to re-expand the results of the calculation. Defaults to False.
- **filter** –
- **kwargs** – Optional positional and keyword arguments to pass to `userfunc`

### Notes

Grouping apply_reduce (for Categorical, groupby, accum2)

For every column of data to be computed, the userfunc will be called back per group as a single array. The order of the groups is either:

- Order of first appearance (when coming from a hash)
- Lexigraphical order (when `lex=True` or a Categorical with ordered=True)

The function passed to apply must take an array as its first argument and return back a single scalar value.

### Examples

From a Dataset groupby:

```
>>> ds.gb(['Symbol'])['TradeSize'].apply_reduce(np.sum)
```

From an existing categorical:

```
>>> ds.Symbol.apply_reduce(np.sum, ds.TradeSize)
```

Create your own with forced dtype:

```
>>> def mycumprodsum(arr):
...     return arr.cumprod().sum()
>>> ds.Symbol.apply_reduce(mycumprodsum, ds.TradeSize, dtype=np.float32)
```

**as_filter**(*index*)

return an index filter for a given unique key

**static contains_np_arrays**(*container*)

Check to see if all items in a list-like container are numpy arrays.

**abstract count**()

Compute count of group

**count_uniques**(**args*, ***kwargs*)

Compute unique count of group

**Return type**
Dataset with grouped key plus the unique count for each column by group.

### Examples

```
>>> N = 17; np.random.seed(1)
>>> ds =Dataset(
        dict(
            Symbol = Cat(np.random.choice(['SPY','IBM'], N)),
            Exchange = Cat(np.random.choice(['AMEX','NYSE'], N)),
            TradeSize = np.random.choice([1,5,10], N),
            TradePrice = np.random.choice([1.1,2.2,3.3], N),
            ))
>>> ds.cat(['Symbol','Exchange']).count_uniques()
```

(continues on next page)

(continued from previous page)

```
*Symbol   *Exchange   TradeSize   TradePrice
-------   ---------   ---------   ----------
IBM       NYSE              2            2
.         AMEX              2            3
SPY       AMEX              3            2
.         NYSE              1            2
```

**cumcount**(**args*, *ascending=True*, ***kwargs*)

rolling count for each group Number each item in each group from 0 to the length of that group - 1.

**Parameters**

**ascending** (*bool, default True*) –

**Returns**

- *A single array, same size as the original grouping dict/categorical.*
- *If a filter was applied, integer sentinels will appear in those slots.*

**cummax**(**args*, *filter=None*, *reset_filter=None*, *skipna=True*, ***kwargs*)

Cumulative nanmax for each group

**Parameters**

- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –
- **skipna** (*boolean, default True*) – Exclude nan/invalid values.

**Return type**

Dataset same rows as original dataset

**cummin**(**args*, *filter=None*, *reset_filter=None*, *skipna=True*, ***kwargs*)

Cumulative nanmin for each group

**Parameters**

- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –
- **skipna** (*boolean, default True*) – Exclude nan/invalid values.

**Return type**

Dataset same rows as original dataset

**cumprod**(**args*, *filter=None*, *reset_filter=None*, ***kwargs*)

Cumulative product for each group

**Parameters**

- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –

**Return type**

Dataset same rows as original dataset

**cumsum**(**args*, *filter=None*, *reset_filter=None*, ***kwargs*)

Cumulative sum for each group

**Parameters**

- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –

**Return type**
Dataset same rows as original dataset

**abstract describe**(***kwargs*)

**diff**(*period=1*, ***kwargs*)

rolling diff for each group

**Parameters**
**period** (*optional, period size, defaults to 1*) –

**Return type**
Dataset same rows as original dataset

**ema_decay**(**args*, *time=None*, *decay_rate=None*, *filter=None*, *reset_filter=None*, ***kwargs*)

Ema decay for each group

Formula:

**grp loops over each item in a groupby group**

**i loops over eachitem in the original dataset**
Output[i] = Column[i] + LastEma[grp] * exp(-decay_rate * (Time[i] - LastTime[grp])); LastEma[grp] = Output[i] LastTime[grp] = Time[i]

**Parameters**

- **time** (*float or int array used to calculate time difference*) –
- **decay_rate** (*see formula, used a half life*) –
- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –

**Return type**
Dataset same rows as original dataset

### Example

```
>>> aapl
#    delta     sym       org    time
-   ------     ----   ------   -----
0    -3.11     AAPL    -3.11   25.65
1   210.54     AAPL   210.54   38.37
2    49.97     AAPL    42.11   41.66
```

```
>>> np.log(2)/(1e3*100)
6.9314718055994526e-06
```

```
>>> aapl.groupby('sym')['delta'].ema_decay(time=aapl.time, decay_rate=np.
↪log(2)/(1e3*100))[0]
FastArray([ -3.11271882, 207.42784495, 257.39155897])
```

**ema_normal**(**args*, *time=None*, *decay_rate=None*, *filter=None*, *reset_filter=None*, ***kwargs*)

Ema decay for each group

Formula:

**grp loops over each item in a groupby group**

**i loops over eachitem in the original dataset**

decayedWeight = exp(-decayRate * (Time[i] - LastTime[grp])); LastEma[grp] = Column[i] * (1 - decayedWeight) + LastEma[grp] * decayedWeight Output[i] = LastEma[grp] LastTime[grp] = Time[i]

**Parameters**

- **time** (*float or int array used to calculate time difference*) –
- **decay_rate** (*see formula, used a half life (defaults to 1.0)*) –
- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –

**Return type**

Dataset same rows as original dataset

### Example

```
>>> ds = rt.Dataset({'test': rt.arange(10), 'group2': rt.arange(10) % 3})
>>> ds.normal = ds.gb('group2')['test'].ema_normal(decay_rate=1.0, time = rt.
↪arange(10))['test']
>>> ds.weighted = ds.gb('group2')['test'].ema_weighted(decay_rate=0.5)['test']
>>> ds
#   test   group2   normal   weighted
-   ----   ------   ------   --------
0      0        0     0.00       0.00
1      1        1     1.00       1.00
2      2        2     2.00       2.00
3      3        0     2.85       1.50
4      4        1     3.85       2.50
5      5        2     4.85       3.50
6      6        0     5.84       3.75
7      7        1     6.84       4.75
8      8        2     7.84       5.75
9      9        0     8.84       6.38
```

**See also:**

*ema_weighted*, *ema_decay*

**ema_weighted**(**args*, *decay_rate=None*, *filter=None*, *reset_filter=None*, ***kwargs*)

Ema decay for each group with constant decay value (no time parameter)

Formula:

**grp loops over each item in a groupby group**

**i loops over eachitem in the original dataset**

LastEma[grp] = Column[i] * (1 - decay_rate) + LastEma[grp] * decay_rate Output[i] = LastEma[grp]

**Parameters**

- **time** (*<not used>*) –
- **decay_rate** (*see formula, used a half life*) –
- **filter** (*optional, boolean mask array of included*) –
- **reset_filter** (*optional, boolean mask array*) –

**Return type**
Dataset same rows as original dataset

### Example

```
>>> ds = rt.Dataset({'test': rt.arange(10), 'group2': rt.arange(10) % 3})
>>> ds.normal = ds.gb('group2')['test'].ema_normal(decay_rate=1.0, time=rt.
↪arange(10))['test']
>>> ds.weighted = ds.gb('group2')['test'].ema_weighted(decay_rate=0.5)['test']
>>> ds
#   test   group2   normal   weighted
-   ----   ------   ------   --------
0      0        0     0.00       0.00
1      1        1     1.00       1.00
2      2        2     2.00       2.00
3      3        0     2.85       1.50
4      4        1     3.85       2.50
5      5        2     4.85       3.50
6      6        0     5.84       3.75
7      7        1     6.84       4.75
8      8        2     7.84       5.75
9      9        0     8.84       6.38
```

**See also:**

*ema_normal*, *ema_decay*

**findnth**(**args*, *filter=None*, ***kwargs*)

FindNth

**Parameters**

- **filter** (*optional, boolean mask array of included*) –
- **bin** (*TAKES NO ARGUMENTS -- operates on*) –

**Return type**
Dataset same rows as original dataset

**first**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

First value in the group

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.

- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**get_groupings**(*filter=None*)

**Parameters**
**filter** (*ndarray of bools, optional*) – pass in a boolean filter

**Returns**
iGroup - the fancy indices for all groups, sorted by group. see iFirstGroup and nCountGroup for how to walk this. iFirstGroup - first index for each group in the igroup array. the first index is invalid nCountGroup - count for each unique group. the first count in this array is the invalid count.

**Return type**
dict containing ndarrays calculated in `pack_by_group()`.

**classmethod get_header_names**(*columns*, *default='col_'*)

**abstract head**(*n=5*, ***kwargs*)

Returns first n rows of each group.

Essentially equivalent to `.apply(lambda x: x.head(n))`, except ignores `as_index` flag.

### Examples

```
>>> df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])
>>> df.groupby('A', as_index=False).head(1)
   A  B
0  1  2
2  5  6
```

```
>>> df.groupby('A').head(1)
   A  B
0  1  2
2  5  6
```

**iter_groups**()

Very similar to the 'groups' property, but uses a generator instead of building the entire dictionary. Returned pairs will be group label value (or tuple of multikey group label values) –> fancy index for that group (base-0).

**key_from_bin**(*bin*)

Returns the value of the group label for a given index. (uses zero-based indexing) A single-key grouping will return a single value. A multi-key grouping will return a tuple of values.

**last**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Last value in the group

**max**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute max of group

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**mean**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute mean of groups

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.

- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**median**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, ***kwargs*)

Compute median of groups For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.

**min**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute min of group

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**mode**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute mode of groups (auto handles nan)

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.

- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**nanmax**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute max of group, excluding missing values

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**nanmean**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute mean of group, excluding missing values

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.

- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**nanmedian**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, ***kwargs*)

Compute median of group, excluding missing values For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.

**nanmin**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute min of group, excluding missing values

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**nanpercentile**(**args*, *q*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, ***kwargs*)

Compute percentile of groups, excluding missing values For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **q** (*float/int or list of floats/ints*) – Percentile(s) to compute. Must be value(s) between 0 and 100
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.

- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.

**nanquantile**(**args*, *q=None*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, ***kwargs*)

Compute quantile of groups, excluding missing values For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **q** (*float, list of floats*) – Quantile(s) to compute. Must be value(s) between 0. and 1.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **is_nan_function** (*bool*) – Not recommended for use. Indicates if this is a nan-version of a function.

**nanstd**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute standard deviation of groups, excluding missing values

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**nansum**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute sum of group, excluding missing values

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.

- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**nanvar**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute variance of groups, excluding missing values

For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (`Dataset`, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**abstract ngroup**(*ascending=True*, ***kwargs*)

Number each group from 0 to the number of groups - 1. This is the enumerative complement of cumcount. Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

**Parameters**

**ascending** (*bool, default True*) – If False, number in reverse, from number of group - 1 to 0.

### Examples

```
>>> df = pd.DataFrame({"A": list("aaabba")})
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
```

```
>>> df.groupby('A').ngroup()
0    0
1    0
2    0
3    1
4    1
5    0
dtype: int64
```

```
>>> df.groupby('A').ngroup(ascending=False)
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
```

```
>>> df.groupby(["A", [1,1,2,3,2,1]]).ngroup()
0    0
1    0
2    1
3    3
4    2
5    0
dtype: int64
```

**See also:**

*cumcount*
: Number the rows in each group.

**static np_quantile_mult**(*a*, *funcParam*)

Applies a correct numpy function for aggregation, used in accum2 Takes funcParam as an argument

**abstract nth**(**args*, ***kwargs*)

**null**(*showfilter=False*)

Performs a reduced no-op. No operation is performed.

**Parameters**
**showfilter** (*bool, False*) –

**Return type**
Dataset with grouping keys. No operation is performed.

### Examples

```
>>> rt.Cat(np.random.choice(['SPY','IBM'], 100)).null(showfilter=True)
```

**abstract ohlc**(***kwargs*)

Compute sum of values, excluding missing values For multiple groupings, the result index will be a MultiIndex

**percentile**(**args*, *q*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, ***kwargs*)

Compute percentile of groups. Returns nan for data that contains nans. For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **q** (*float/int or list of floats/ints*) – Percentile(s) to compute. Must be value(s) between 0 and 100
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.

**quantile**(**args*, *q*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, ***kwargs*)

Compute quantile of groups. Returns nan for data that contains nans. For multiple groupings, the result will be a MultiSet

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.

- **q** (*float or list of floats*) – Quantile(s) to compute. Must be value(s) between 0. and 1.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.

**static quantile_name_from_param**(*funcParam*)

Returns a correct name of a quantile function given funParam, used in accum2

**abstract rank**(*method='average'*, *ascending=True*, *na_option='keep'*, *pct=False*, *axis=0*, ***kwargs*)

Provides the rank of values within each group

**Parameters**

- **method** (*{'keep', 'top', 'bottom'}, default 'keep'*) –
  - average: average rank of group
  - min: lowest rank in group
  - max: highest rank in group
  - first: ranks assigned in order they appear in the array
  - dense: like 'min', but rank always increases by 1 between groups
- **method** –
  - keep: leave NA values where they are
  - top: smallest rank if ascending
  - bottom: smallest rank if descending
- **ascending** (*boolean, default True*) – False for ranks by high (1) to low (N)
- **pct** (*boolean, default False*) – Compute percentage rank of data within each group

**Return type**

DataFrame with ranking of values within each group

**classmethod register_functions**(*functable*)

Registration should follow the NUMBA_REVERSE_TABLE layout at the bottom of rt_groupbynumba.py If we register again, the last to register will be executed. NUMBA_REVERSE_TABLE[i + GB_FUNC_NUMBA]={'name': k, 'packing': v[0], 'func_front': v[1], 'func_back': v[2], 'func_gb':v[3], 'func_dtype': v[4], 'return_full': v[5]}

**abstract resample**(*rule*, **args*, ***kwargs*)

Provide resampling when using a TimeGrouper Return a new grouper with our resampler appended

**rolling_count**(**args*, *window=3*, ***kwargs*)

rolling count for each group

**Parameters**
**window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**rolling_diff**(**args*, *window=1*, ***kwargs*)

rolling diff for each group

**Parameters**
**window** (*optional, window size, defaults to 1*) –

**Return type**
Dataset same rows as original dataset

**rolling_mean**(**args*, *window=3*, ***kwargs*)

rolling mean for each group

**Parameters**
**window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**rolling_median**(**args*, *window=3*, ***kwargs*)

rolling nan median for each group

**Parameters**
**window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**rolling_nanmean**(**args*, *window=3*, ***kwargs*)

rolling nan mean for each group

**Parameters**
**window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**rolling_nansum**(**args*, *window=3*, ***kwargs*)

rolling nan sum for each group

**Parameters**
**window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**rolling_quantile**(**args*, *q*, *window=3*, ***kwargs*)

rolling nan quantile for each group

**Parameters**

- **q** (*float, quantile to compute*) –
- **window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**rolling_shift**(**args*, *window=1*, ***kwargs*)

rolling shift for each group

**Parameters**

- **window** (*optional, window size, defaults to 1*) –
- **negative** (*windows can be*) –

**Return type**
Dataset same rows as original dataset

**rolling_sum**(**args*, *window=3*, ***kwargs*)

rolling sum for each group

**Parameters**
**window** (*optional, window size, defaults to 3*) –

**Return type**
Dataset same rows as original dataset

**abstract sem**(***kwargs*)

Compute standard error of the mean of groups For multiple groupings, the result index will be a MultiIndex

**Parameters**
**ddof** (*integer, default 1*) – degrees of freedom

**shift**(*window=1*, ***kwargs*)

Shift each group by periods observations :param window: :type window: integer, default 1 number of periods to shift :param periods: :type periods: optional support, same as window

**std**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute standard deviation of groups

For multiple groupings, the result will be a MultiSet

**Parameters**

- **ddof** (*integer, default 1*) – degrees of freedom
- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.

- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**sum**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute sum of group

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work Accum2 not supported.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**abstract tail**(*n=5*, ***kwargs*)

Returns last n rows of each group Essentially equivalent to `.apply(lambda x: x.tail(n))`, except ignores `as_index` flag.

### Examples

```
>>> df = pd.DataFrame([['a', 1], ['a', 2], ['b', 1], ['b', 2]], columns=['A',
↪'B'])
>>> df.groupby('A').tail(1)
   A  B
1  a  2
3  b  2
```

```
>>> df.groupby('A').head(1)
   A  B
0  a  1
2  b  1
```

**trimbr**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute trimmed mean br of groups (auto handles nan)

**Parameters**

- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (*Dataset, optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

**var**(**args*, *filter=None*, *transform=False*, *showfilter=False*, *col_idx=None*, *dataset=None*, *return_all=False*, *computable=True*, *accum2=False*, *func_param=0*, ***kwargs*)

Compute variance of groups

For multiple groupings, the result will be a MultiSet

**Parameters**

- **ddof** (*integer, default 1*) – degrees of freedom
- ***args** – Elements to apply the GroupBy Operation to. Typically a FastArray or Dataset.
- **filter** (*array of bool, optional*) – Elements to include in the GroupBy Operation.
- **transform** (*bool*) – If transform = True, the output will have the same shape as `args`. If transform = False, the output will typically have the same shape as the categorical.
- **showfilter** (*bool*) – If showfilter is True, there will be an extra row in the output representing the GroupBy Operation applied to all those elements that were filtered out.
- **col_idx** (*str, list of str, optional*) – If the input is a Dataset, col_idx specifies which columns to keep.
- **dataset** (Dataset, *optional*) – If a dataset is specified, the GroupBy Operation will also be applied to the dataset. If there is an `args` argument and dataset is specified then the result will be appended to the dataset.
- **return_all** (*bool*) – If return_all is True, will return all columns, even those where the GroupBy Operation does not make sense. If return_all is False, it will not return columns it cannot apply the GroupBy to. Does not work with Accum2.
- **computable** (*bool*) – If computable is True, will not try to apply the GroupBy Operation to non-computable datatypes.
- **accum2** (*bool*) – Not recommended for use. If accum2 is True, the result is returned as a dictionary.
- **func_param** – Not recommended for use.

### 2.2.21 riptable.rt_grouping

#### Classes

| | |
|---|---|
| *Grouping* | Every GroupBy and Categorical object holds a grouping in self.grouping; |

### Functions

| | |
|---|---|
| *combine2groups*(group_row, group_col[, filter, showfilter]) | The group_row unique keys are used in the grouping_dict returned. |
| *hstack_groupings*(ikey, uniques[, i_cutoffs, ...]) | For hstacking Categoricals or fixing indices in a categorical from a stacked .sds load |
| *hstack_test*(arr_list) | |
| *merge_cats*(indices, listcats[, idx_cutoffs, ...]) | For hstacking Categoricals possibly from a stacked .sds load. |

class riptable.rt_grouping.**Grouping**(*grouping*, *categories=None*, *ordered=None*, *dtype=None*, *base_index=1*, *sort_display=False*, *filter=None*, *lex=False*, *rec=False*, *categorical=False*, *cutoffs=None*, *next=False*, *unicode=False*, *name=None*, *hint_size=0*, *hash_mode=2*, *_trusted=False*, *verbose=False*)

Every GroupBy and Categorical object holds a grouping in self.grouping; this class informs the groupby algorithms how to group the data.

**Stage 1**

Initializing from a GroupBy object or unbinned Categorical object:

- grouping_dict: dictionary of non-unique key columns (hash will be performed)
- iKey: array size is same as multikey, the unique key for which this row in multikey belongs
- iFirstKey: array size is same as unique keys, index into the first row for that unique key
- iNextKey:array size is same as multikey, index to the next row that hashed to same value
- nCountGroup: array size is same as unique keys, for each unique item, how many values

Initializing from a pre-binned Categorical object:

- grouping_dict: dictionary of pre-binned columns (no hash performed)
- iKey: array size is same as Categorical's underlying index array - often uses the same array.
- unique_count: unique number of items in the categorical.

**Stage 2**

- iGroup: unique keys are grouped together
- iFirstGroup: index into first row for the group
- nCountGroup: number of items in the group

**Performing calculations**

(See *Grouping._calculate_all*)

**Parameters**

- **origdict** (*) –
- **funcNum** (*) –
- **func_param** (*) –
- **table)** (*1. Check the keywords for "invalid" (wether or not an invalid bin will be included in the result*) –

- **it.** (*2. Check the keywords for a filter and store*) –
- **packing** (** _groupbycalculateallpack - for level 2 functions that require*) –

  **pack_by_group(filter=None, mustrepack=False)**

  1. If the grouping object has already been packed and no filter is present, return.
  2. If a filter is present, discard any existing iNextKey and combine the filter with the iKey.
  3. Call the groupbypack routine -> sends info to CPP.
  4. iGroup, iFirstGroup, nCountGroup are returned and stored.

- **pack_by_group.** (*call*) –

  **pack_by_group(filter=None, mustrepack=False)**

  1. If the grouping object has already been packed and no filter is present, return.
  2. If a filter is present, discard any existing iNextKey and combine the filter with the iKey.
  3. Call the groupbypack routine -> sends info to CPP.
  4. iGroup, iFirstGroup, nCountGroup are returned and stored.

- **calculation.** (*4. Prepare the origdict for*) –
- **``_get_calculate_dict(origdict** –
  1. Check for "col_idx" in keywords (used for agg function mapping certain operations to certain columns)
  2. The grouping object has a _grouping_dict (keys). If these columns are in origdict, they are removed.
  3. Most operations cannot be performed on strings or string-based categoricals. Remove columns of those types.
  4. Return the cleaned up dictionary, and a list of its columns. (npdict, values)
- **funcNum** –
  1. Check for "col_idx" in keywords (used for agg function mapping certain operations to certain columns)
  2. The grouping object has a _grouping_dict (keys). If these columns are in origdict, they are removed.
  3. Most operations cannot be performed on strings or string-based categoricals. Remove columns of those types.
  4. Return the cleaned up dictionary, and a list of its columns. (npdict, values)
- **func=None** –
  1. Check for "col_idx" in keywords (used for agg function mapping certain operations to certain columns)
  2. The grouping object has a _grouping_dict (keys). If these columns are in origdict, they are removed.
  3. Most operations cannot be performed on strings or string-based categoricals. Remove columns of those types.

4. Return the cleaned up dictionary, and a list of its columns. (npdict, values)

- **func_param=0)``** –
  1. Check for "col_idx" in keywords (used for agg function mapping certain operations to certain columns)
  2. The grouping object has a _grouping_dict (keys). If these columns are in origdict, they are removed.
  3. Most operations cannot be performed on strings or string-based categoricals. Remove columns of those types.
  4. Return the cleaned up dictionary, and a list of its columns. (npdict, values)
- **operation.** (*5. Perform the*) –
- **cumsum** (** rc.EmaAll32 - for*) –
- **cumprod** –
- **ema_decay** –
- **etc.** –
- **exists)** (** _groupbycalculateall - for basic functions that don't require packing (combine filter if*) –
- **packing** –
- **sorted.** (*accum_tuple is a series of columns after the operation. The data has not been*) –
- **requested** (*accum_tuple has an invalid item at [0] for each column. If no invalid was*) –
- **off.** (*trim it*) –
- **list.** (*Store the columns in a*) –
- **accum2** (*If the function was called from*) –
- **here.** (*return*) –
- **columns.** (*7. Make a dataset from the dictionary of calculated*) –
- **_make_accum_dataset** –
  1. Make a dictionary from the list of calculated columns. Use the names from npdict (see step 4)
  2. If nothing was calculated for the column, the value will be None. Remove it.
  3. If the column was a categorical, the calculate dict only has its indices. Pull the categories from the original dictionary and build a new categorical (shallow copy)
- **columns.** –
- **_return_dataset** –
  1. If the function is in cumsum, cumprod, ema_decay, etc. no groupby keys will appear (set to None)
  2. If the function is count, it will have a single column (Count) - build a dataset from this.
  3. Initialize an empty diciontary (newdict).

4. Iterate over the column names in the *original* dictionary and copy them to the newdict. accumdict only contains columns that were operated on. If the return_all flag was set to True, these columns still need to be included.
5. If the function is in cumsum, cumprod, ema_decay, etc. no sort will be applied, no labels (gbkeys) will be tagged
6. Otherwise, apply a sort (default for GroupBy) to each column with isortrows (from the GroupByKeys object). Tag all label columns in final dataset.

- **dataset** (*8. Return the*) –

**property _anydict**

Either the _grouping_dict or _grouping_unique_dict. Only be used for names, array datatypes. Will check for and return _grouping_dict first.

**property all_unique: bool**

Indicates whether all keys/groups occur exactly once.

**property base_index: int**

The starting index from which keys (valid groups) are numbered. Always equal to 0 or 1.

**property catinstance**

Integer array for constructing Categorical or Categorical-like array.

**Returns**
**instance_array** – If base index is 1, returns the ikey. If base index is 0, stores and returns ikey - 1. If in enum mode, returns integers from _grouping_dict.

**Return type**
FastArray

**property gbkeys: Mapping[str, numpy.ndarray]**

**property ifirstgroup**

Returns a sister array used with *ncountgroup* and *igroup*.

**Returns**
**ifirstgroup**

**Return type**
np.ndarray of int

**See also:**

*igroup*, *ncountgroup*, *igroupreverse*

**property ifirstkey**

returns the row locations of the first member of the group

**Returns**
**ifirstkey**

**Return type**
np.ndarray of int

**property igroup**

returns a fancy index that when applied will make all the groups contiguous (packed together)

**Returns**
**igroup**

**Return type**
np.ndarray of int

**See also:**

*ifirstgroup*, *ncountgroup*, *igroupreverse*

**property igroupreverse**

Returns the fancy index to reverse the shuffle from *igroup*.

**Returns**
**igroupreverse**

**Return type**
np.ndarray of int

**See also:**

*igroup*

**property ikey**

Returns a 1-based integer array with the bin number for each row.

Bin 0 is reserved for filtered out rows. This property will return +1 for base-0 grouping.

**Returns**
**ikey**

**Return type**
np.ndarray of int

**property ilastkey**

returns the row locations of the last member of the group

**Returns**
**ilastkey**

**Return type**
np.ndarray of int

**property inextkey**

returns the row locations of the next member of the group (or invalid int).

**Returns**
**inextkey**

**Return type**
np.ndarray of int

**property iprevkey**

returns the row locations of the previous member of the group (or invalid int)

**Returns**
**iprevkey**

**Return type**
np.ndarray of int

**property iscategorical: bool**

True if only uniques are being held - no reference to original data.

property isdirty: bool

bool, default False If True, it's possible that not all of the values in between 0 and the unique count appear in the iKey. Number of unique occurring values may be different than number of possible unique values. e.g. after slicing a Categorical.

**Type**
isdirty

property isdisplaysorted: bool

property isenum: bool

property ismultikey: bool

True if unique dict holds multiple arrays. False if unique dict holds single array or in enum mode.

property isordered: bool

property isortrows

property issinglekey: bool

True if unique dict holds single array. False if unique dict hodls multiple arrays or in enum mode.

property ncountgroup

returns a sister array used with *ifirstgroup* and *igroup*.

**Returns**
**ncountgroup**

**Return type**
np.ndarray of int

**See also:**

*igroup*, *ifirstgroup*, *igroupreverse*

property ncountkey

**returns: ncountkey – An array with the number of unique counts per key**
Does include the zero bin

**Return type**
np.ndarray of int

property packed: bool

The grouping operation has performed an operation that requires packing e.g. `median()` If *packed*, `iGroup`, `iFirstGroup`, and `nCountGroup` have been generated.

property unique_count: int

Number of unique groups.

property uniquedict: Mapping[str, numpy.ndarray]

Dictionary of key names -> array(s) of unique categories.

`GroupBy` will pull values from non-unique dictionary using `iFirstKey`. `Categorical` already holds a unique dictionary. Enums will pull with `iFirstKey`, and return unique strings after translating integer codes.

**Returns**
Dictionary of key names -> array(s) of unique categories.

**Return type**
dict

### Notes

No sort is applied here.

**property uniquelist**

See Grouping.uniquedict Sets FastArray names as key names.

**DebugMode = False**

**GroupingInit**

**REGISTERED_REVERSE_TABLES = []**

**__getitem__**(*fld*)

Perform an indexing / slice operation on iKey, _catinstance, _grouping_dict if they have been set.

**Parameters**
**fld** (*integer (single item) raise error*) – slice integer array (fancy index) boolean array (true/false mask) string, list of strings: raise error

**Returns**
**newgroup** – A copy of the grouping object with a reindexed iKey. The dirty flag in the result will be set to True. A single scalar value (for enum/singlekey grouping) A tuple of scalar values (for multikey grouping)

**Return type**
*Grouping* or scalar or tuple

**__repr__**()

Return repr(self).

**_build_unique_dict**(*grouping*)

Pull values from the non-unique grouping dict using the `iFirstKey` index. If enumstring is True, translate enum codes to their strings.

**_calculate_all**(*origdict*, *funcNum*, *func_param=0*, *keychain=None*, *user_args=()*, *tups=0*, *accum2=False*, *return_all=False*, ***kwargs*)

All groupby calculations from GroupBy, Categorical, Accum2, and some groupbyops will be enter through this method.

**Parameters**

- **orgidict** –
- **funcNum** –
- **func_param** (*int*) – parameters from GroupByOps (often simple scalars)
- **keychain** – option groupby keys to apply to the final dataset at end
- **user_args** – A tuple of None or more arguments to pass to user_function. `user_args` only exists for apply* related function calls
- **tups** (*int, 0*) – Defaults to 0. 1 if user functions had tuples () indicating to pass in all arrays. `tups` is only > 0 for apply* related function calls where the first parameter was (arr1, ..)
- **accum2** (*bool*) –

- **return_all** (*bool*) –
- **showfilter** (*bool, optional*) – If set will calculate contents in the 0 bin.

**See also:**

*Grouping*

**_empty_allowed**(*funcNum*)

Operations like cumcount do not need an origdict to calculate. Calculations are made only on binned columns. Might be more later, so keep here.

**_finalize_dataset**(*accumdict*, *keychain*, *gbkeys*, *transform=False*, *showfilter=False*, *addkeys=False*, ***kwargs*)

possibly transform? TODO: move to here possibly reattach keys possibly sort

**Parameters**

- **accumdict** (*dict or Dataset*) –
- **keychain** –
- **None** (*gbkeys may be passed as*) –

**_from_categories**(*grouping*, *categories*, *arr_len*, *base_index*, *filter*, *dtype*, *ordered*, *_trusted*)

Initialize a Grouping object from pre-defined uniques.

**Parameters**

- **grouping** (*dict of single array*) – Pre-defined iKey or non-unique values.
- **categories** (*dict of arrays*) – Pre-defined dictionary of unique categories or enum mapping (not implemented)
- **arr_len** (*int*) – Length of arrays in `categories` dict.
- **filter** (*boolean array*) – Pre-filter the same length as the non-unique values.
- **_trusted** (*bool*) – If True, data will not be validated with min / max check.

**Returns**

- **ikey** (*ndarray of ints*) – Base 0 or Base 1 ikey
- **ordered_flag** (*bool*) – Flag indicating whether the categories were/are ordered. This is the `ordered` flag just being passed through.

**_get_calculate_dict**(*origdict*, *funcNum*, *func=None*, *return_all=False*, *computable=True*, *func_param=0*, ***kwargs*)

Builds a dictionary to perform the groupby calculation on.

If string/string-like columns cannot be computed, they will not be included. If specific columns have been specified (in col_idx, see GroupBy.agg), only they will be included.

**Returns**

- **npdict** (*dict*) – Final dictionary for calculation.
- **values** (*list*) – List of columns in `npdict`. (NOTE: this is repetitive as npdict has these values also.)

**static _hstack**(*glist*, *_trusted=False*, *base_index=1*, *ordered=False*, *destroy=False*)

'hstack' operation for Grouping instances.

**Parameters**

- **glist** (*list of Grouping*) – A list of Grouping objects.
- **_trusted** (*bool*) – Indicates whether we need to validate the data in the supplied Grouping instances for consistency / correctness before using it. In certain cases, the caller knows the data is safe to use directly (e.g. because they've just created it), so the validation can be skipped.
- **base_index** (*int*) – The base index to use for the resulting Categorical.
- **ordered** (*bool*) – Indicates whether the resulting Categorical will be an 'ordered' Categorical (sometimes called an 'Ordinal').
- **destroy** (*bool*) – This parameter is unused.

**Return type**
*Grouping*

**_make_accum_dataset**(*origdict*, *npdict*, *accum*, *funcNum*, *return_all=False*, *keychain=None*, ***kwargs*)

Returns a Dataset

**_make_enumikey**(*list_values*, *filter=None*)

internal routine to lazy generate ikey for enum if a filter is passed on init, have to generate upfront

will generate ikey, ifirstkey, unique_count also

**_make_isortrows**(*gbkeys*)

Sort a single or multikey dictionary of unique values. Return the sorted index.

**_return_dataset**(*origdict*, *accumdict*, *func_num*, *return_all=False*, *col_idx=None*, *keychain=None*, ***kwargs*)

**_set_anydict**(*d*)

Replace the dict returned by _anydict Will check for and set _grouping_dict first.

**_set_newinstance**(*newinstance*)

**apply**(*origdict*, *userfunc*, **args*, *tups=0*, *filter=None*, *label_keys=None*, *return_all=False*, ***kwargs*)

Grouping apply (for Categorical, groupby, accum2) Apply function userfunc group-wise and combine the results together. The userfunc will be called back per group. The order of the groups is either:

- Order of first apperance (when coming from a hash)
- Lexigraphical order (when lex=True or a Categorical with ordered=True)

If a group from a categorical has no rows (an empty group), then a dataset with one row of invalids (as a place holder) will be used and the userfunc will be called.

The function passed to apply must take a Dataset as its first argument and return one of the following:

- a Dataset (with one or more rows returned)
- a dictionary of name:array pairs
- a single array

The set of returned columns must be consistent for each input (group) dataset. `apply` will then take care of combining the results back together into a Dataset with the groupby key(s) in the initial column(s). `apply` is therefore a highly flexible grouping method.

While `apply` is a very flexible method, its downside is that using it can be quite a bit slower than using more specific methods. riptable offers a wide range of methods that will be much faster than using `apply` for their specific purposes, so try to use them before reaching for `apply`.

**Parameters**

- **userfunc** (*callable*) – A callable that takes a Dataset as its first argument, and returns a Dataset, dict, or single array. In addition the callable may take positional and keyword arguments.
- **args** (*tuple and dict*) – Optional positional and keyword arguments to pass to userfunc
- **kwargs** (*tuple and dict*) – Optional positional and keyword arguments to pass to userfunc
- **possible)** (*Returns (2)* –
- ------- –
- **dataset)** (*Dataset that is grouped by (reduced from original)* –
- **by)** (Dataset *of original length (not grouped)* –

### Examples

```
>>> ds = rt.Dataset({'A': 'a a b'.split(), 'B': [1,2,3], 'C': [4,6, 5]})
>>> g = rt.GroupBy(ds, 'A')
```

From ds above we can see that g has two groups, a and b. Calling apply in various ways, we can get different grouping results.

Example 1: below the function passed to apply takes a Dataset as its argument and returns a Dataset or dictionary with one row for each row in each group. apply combines the result for each group together into a new Dataset:

```
>>> g.apply(lambda x: x.sum())
*A   B    C
--   -   --
a    3   10
b    3    5
```

```
>>> g.apply(lambda x: {'B':x.B.sum()})
*A   B
--   -
a    3
b    3
```

Example 2: The function passed to apply takes a Dataset as its argument and returns a Dataset with one row per group. apply combines the result for each group together into a new Dataset:

```
>>> g.apply(lambda x: x.max() - x.min())
*A   B   C
--   -   -
a    1   2
b    0   0
```

Example 3: The function passed to apply takes a Dataset as its argument and returns a Dataset with one row and one column per group (i.e., a scalar). apply combines the result for each group together into a Dataset:

```
>>> g.apply(lambda x: rt.Dataset({'val': [x.C.max() - x.B.min()]}))
*A   val
--   ---
a      5
b      2
```

Example 4: The function returns a Dataset with more than one row.

```
>>> g.apply(lambda x: x.cumsum())
*A   B    C
--   -   --
a    1    4
a    3   10
b    3    5
```

Example 5: A non-lambda, user-supplied function which creates a new column in the existing Dataset.

```
>>> def userfunc(x):
        x.Sub = x.C - x.B
        return x
>>> g.apply(userfunc)
*A   B   C   Sub
--   -   -   ---
a    1   4     3
a    2   6     4
b    3   5     2
```

**apply_helper**(*isreduce*, *origdict*, *userfunc*, **args*, *tups=0*, *filter=None*, *showfilter=False*, *label_keys=None*, *func_param=None*, *dtype=None*, *badrows=None*, *badcols=None*, *computable=True*, ***kwargs*)

Grouping apply_reduce/apply_nonreduce (for Categorical, groupby, accum2)

**For every column of data to be computed:**

**The userfunc will be called back per group as a single array. The order of the groups is either:**

1) Order of first apperance (when coming from a hash)

2) Lexigraphical order (when lex=True or a Categorical with ordered=True)

A reduce function must take an array as its first argument and return back a single scalar value. A non-reduce function must take an array as its first argument and return back another array. The first argument to apply MUST be the callable user function.

The second argument to apply contains one or more arrays to operate on.

- If passed as a list, the userfunc is called for each array in the list
- If passed as a tuple, the userfunc is called once with all the arrays as parameters

**Parameters**

- **isreduce** (*bool*) – Must be set. True for reduce, False for non-reduce.
- **origdict** (*dict of name:arrays*) – The column names and arrays to apply the functio on.
- **userfunc** (*callable*) – A callable that takes one or more arrays as its first argument, and returns an array or scalar. If `isreduce` is True, `userfunc` is a reduction and

should return a scalar; when `isreduce` is False, `userfunc` is a nonreduce/scan/prefix-sum and should return an array. In addition the callable may take positional arguments and keyword arguments.

- ***args** – Any additional user arguments to pass to `userfunc`.
- **tups** (*0*) – Set to 1 if `userfunc` wants multiple arrays passed fixed up by iGroup. Defaults to False. Set to 2 for passing in constants
- **showfilter** (*bool*) – Set to True to calculate filter. Defaults to False.
- **filter** (*ndarray of bools*) – optional boolean filter to apply
- **label_keys** (*rt.GroupByKeys, the labels on the left*) –
- **func_param** (*tuple, optional*) – Caller may pass `func_param=(arg1, arg2)` to pass arguments to `userfunc`.
- **dtype** (*str or np.dtype, or dict of np.dtypes, optional*) – Explicitly specify the dtype for the output array. Defaults to None, which means the function chooses a compatible dtype for the output. If a dict of np.dtypes is passed, multiple output arrays are allocated based on the specified dtypes.
- **badrows** – not used, may be passed from Acccum2
- **badcols** – not used

### Notes

All other arguments passed to this function (if any remaining) will be passed through to `userfunc`.

**See also:**

`GroupByOps.apply_reduce`, `GroupByOps.apply_nonreduce`

**as_filter**(*index*)

Returns an index filter for a given unique key

**copy**(*deep=True*)

Create a shallow or deep copy of the grouping object.

**Parameters**
**deep** (*bool, default True*) – If True, makes a deep copy of all array data.

**Returns**

- **newgrouping** (*Grouping*)
- **Note** (*a shallow copy will always make new dictionaries, but does not copy array data.*)

**copy_from**(*other=None*)

Initializes a new Grouping object if other is None. Otherwise shallow copy all necessary attributes from another grouping object to self.

**Parameters**
**other** (*Grouping*) –

**count**(*gbkeys=None*, *isortrows=None*, *keychain=None*, *filter=None*, *transform=False*, ***kwargs*)

Compute count of each unique key Returns a dataset containing a single column. The Grouping object has the ability to generate this column on its own, and therefore skips straight to _return_dataset versus other groupby calculations (which pass through _calculate_all) first.

**static extract_groups**(*condition*, *grouped_data*, *ncountgroup*, *ifirstgroup*)

Take groups of elements from an array, where the groups are selected by a boolean mask.

This function provides boolean-indexing over groups of data – so a boolean mask can be used to select _groups_ of data, rather than just individual elements, and the grouped elements will be copied to the output.

**Parameters**

- **condition** (*np.ndarray of bool*) – An array whose nonzero or True entries indicate the groups in *ncountgroup* whose elements will be extracted from grouped_data.
- **grouped_data** (*np.ndarray*) –
- **ncountgroup** (*np.ndarray of int*) –
- **ifirstgroup** (*np.ndarray of int*) –

**Return type**
np.ndarray

**Raises**
**ValueError** – When condition is not a boolean/logical array. When condition and *ncountgroup* have different shapes. When *ncountgroup* and *ifirstgroup* have different shapes.

**See also:**

numpy.extract

### Examples

Select data from an array, where the elements belong to even-numbered groups within the Grouping object.

```
>>> key_data = rt.FA([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6,␣
↪6, 6])
>>> data = rt.arange(len(key_data))
>>> g = rt.Grouping(key_data)
>>> group_mask = rt.arange(len(g.ncountgroup)) % 2 == 0
>>> Grouping.extract_groups(group_mask, data, g.ncountgroup, g.ifirstgroup)
FastArray([1, 2, 6, 7, 8, 9, 15, 16, 17, 18, 19, 20])
```

**get_name**()

List of grouping or grouping unique dict keys.

**isin**(*values*)

Used to match values

**Return type**
numpy array of bools where the values are found

**See also:**

rt.Grouping.isin

**ismember**(*values*, *reverse=False*)

Used to match against the unique categories NOTE: This does not match against the entire array, just the uniques

**Parameters**

**reverse** (*bool, defaults to False.*) – Set to True to reverse the `ismember(A, B)` to `ismember(B,A)`.

**Returns**

- **member_mask** (*np.ndarray of bool*) – boolean array of matches to unique categories
- **member_indices** (*np.ndarray of int*) – fancy index array of location in unique categories

### Examples

```
>>> a = rt.Cat(['b','c','d']).tile(5)
>>> b = rt.Cat(['a','b','d','e','f']).tile(5)
>>> tf1 = rt.ismember(a, b)[0]
>>> tf2 = b.grouping.ismember(a.categories())[1][b-1] != -128
>>> np.all(tf1 == tf2)
True
```

```
>>> a = rt.Cat(['BABL','COKE','DELT']).tile(50000)
>>> b = rt.Cat(['AAPL','BABL','DELT','ECHO','FB']).tile(33333333)
>>> %time tf1 = rt.ismember(a,b)[0]
 197 ms
>>> %time tf3 = rt.ismember(a.category_array, b.category_array)[1][a-1] != -128
 1 ms
>>> np.all(tf1 == tf3)
True
```

**See also:**

`rt.Grouping.isin`, `rt.ismember`

**classmethod newclassfrominstance**(*instance*, *origin*)

**newgroupfrominstance**(*newinstance*)

calculate_all may change the instance

**Parameters**

**newinstance** (*integer based array (codes or bins)*) –

**Return type**

a new grouping object

**onedict**(*unicode=False*, *invalid=True*, *sep='_'*)

Concatenates multikey groupings with underscore to make a single key. Adds 'Inv' to first element if kwarg Invalid=True.

**Parameters**

- **unicode** (*boolean, default False*) – whether to create a string or unicode based array
- **invalid** (*boolean, default True*) – whether or not to add 'Inv' as the first unique

**Returns**

- *a string of the new key name*

- *a new single array of the uniques concatenated*

**pack_by_group**(*filter=None*, *mustrepack=False*)

Used to prepare data for custom functions

Prepares 3 arrays:

- iGroup: array size is same as multikey, unique keys are grouped together
- iFirstGroup: array size is number of unique keys for that group, indexes into isort
- nCountGroup: array size is number of unique keys for the group

the user should use… igroup, ifirstgroup, ncountgroup

If a filter is passed, it is remembered

**classmethod possibly_recast**(*arr*, *unique_count*, *dtype=None*)

*unique_count* is checked and compared against preferred (minimal) dtype size is calculated.

If a dtype has been provided, it will be used (only if it is large enough to fit the maximum value for the calculated dtype).

**Parameters**

- **arr** (*ndarray of ints*) –
- **unique_count** (*int*) – The number of unique bins corresponding to `arr`.
- **dtype** (*str or np.dtype, optional*) – Optionally force a dtype for the returned integer array (see dtype keyword in the Categorical constructor), defaults to None.

**Returns**

**new_arr** – A recasted array with a smaller dtype, requested dtype, or possibly the same array as `arr` if no changes were needed.

**Return type**

ndarray of ints

**classmethod register_functions**(*functable*)

**regroup**(*filter=None*, *ikey=None*)

Regenerate the groupings iKey, possibly with a filter and/or eliminating unique values.

**Parameters**

- **filter** (*np.ndarray of bool, optional*) – Filtered bins will be marked as zero in the resulting iKey. If not provided, uniques will be reduced to the ones that occur in the iKey.
- **ikey** (*np.ndarray of int, optional*) – Only used when the grouping is in enum mode.

**Returns**

New Grouping object created by regenerating the *ikey*, *ifirstkey*, and *unique_count* using data from this instance.

**Return type**

*Grouping*

**set_dirty**()

If the shared information (like a Categorical's instance array) has been changed outside of the grouping object, the changing routine can call this on the grouping object.

**set_name**(*name*)

If the grouping dict contains a single item, rename it.

This will make categorical results consistent with groupby results if they've been constructed before being added to a dataset. Ensures that label names are consistent with categorical names.

**Parameters**

**name** (*str*) – The new name to use for the single column in the internal grouping dictionary.

### Examples

Single key Categorical added to a Dataset, grouping picks up name:

```
>>> c = rt.Categorical(['a','a','b','c','a'])
>>> print(c.get_name())
None
```

```
>>> ds = rt.Dataset({'catcol':c})
>>> ds.catcol.sum(rt.arange(5))
*catcol   col_0
-------   -----
a             5
b             2
c             3
```

Multikey Categorical, no names:

```
>>> c = rt.Categorical([rt.FA(['a','a','b','c','a']), rt.FA([1,1,2,3,1])])
>>> print(c.get_name())
None
```

```
>>> ds = rt.Dataset({'mkcol': c})
>>> ds.mkcol.sum(rt.arange(5))
*mkcol_0   *mkcol_1   col_0
--------   --------   -----
a                 1       5
b                 2       2
c                 3       3
```

Multikey Categorical, already has names for its columns (names are preserved):

```
>>> arr1 = rt.FA(['a','a','b','c','a'])
>>> arr1.set_name('mystrings')
>>> arr2 = rt.FA([1,1,2,3,1])
>>> arr2.set_name('myints')
>>> c = rt.Categorical([arr1, arr2])
>>> ds = rt.Dataset({'mkcol': c})
>>> ds.mkcol.sum(rt.arange(5))
*mystrings   *myints   col_0
----------   -------   -----
a                  1       5
b                  2       2
c                  3       3
```

**shrink**(*newcats*, *misc=None*, *inplace=False*, *name=None*)

**Parameters**

- **newcats** (*array_like*) – New categories to replace the old - typically a reduced set of strings
- **misc** (*scalar, optional*) – Value to use as category for items not found in new categories. This will be added to the new categories. If not provided, all items not found will be set to a filtered bin.
- **inplace** (*bool, not implemented*) – If True, re-index the categorical's underlying FastArray. Otherwise, return a new categorical with a new index and grouping object.
- **name** –

**Returns**

A new Grouping object based on this instance's data and the new set of labels provided in `newcats`.

**Return type**

*Grouping*

**sort**(*keylist*)

**static take_groups**(*grouped_data*, *indices*, *ncountgroup*, *ifirstgroup*)

Take groups of elements from an array.

This function provides fancy-indexing over groups of data – so a fancy index can be used to specify _groups_ of data, rather than just individual elements, and the grouped elements will be copied to the output.

**Parameters**

- **grouped_data** (*np.ndarray*) –
- **indices** (*np.ndarray of int*) –
- **ncountgroup** (*np.ndarray of int*) –
- **ifirstgroup** (*np.ndarray of int*) –

**Return type**

np.ndarray

**Raises**

**ValueError** – When *ncountgroup* and *ifirstgroup* have different shapes.

**See also:**

`numpy.take`

### Examples

Select data from an array, where the elements belong to the 2nd, 4th, and 6th groups within the Grouping object.

```
>>> key_data = rt.FA([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6,␣
↪6, 6])
>>> data = rt.arange(len(key_data))
>>> g = rt.Grouping(key_data)
>>> group_indices = rt.FA([2, 4, 6])
>>> Grouping.take_groups(data, group_indices, g.ncountgroup, g.ifirstgroup)
FastArray([1, 2, 6, 7, 8, 9, 15, 16, 17, 18, 19, 20])
```

riptable.rt_grouping.**combine2groups**(*group_row*, *group_col*, *filter=None*, *showfilter=False*)

The group_row unique keys are used in the grouping_dict returned. The group_cols unique keys are expected to become columns.

**Parameters**

- **group_row** (Grouping) – Grouping object for the rows
- **group_col** (Grouping) – Grouping object for the cols
- **filter** (*np.ndarray of bool, optional*) – A boolean filter of values to remove on the rows. Should be same length as group_row.ikey array (can pass in None).
- **showfilter** (*bool*) –

**Returns**

A new Grouping object The new ikey will always the number of `(group_row.unique_count+1)*(group_col.unique_count+1)`. The grouping_dict in the Grouping object will be for the rows only.

**Return type**

*Grouping*

riptable.rt_grouping.**hstack_groupings**(*ikey*, *uniques*, *i_cutoffs=None*, *u_cutoffs=None*, *from_mapping=False*, *base_index=1*, *ordered=False*, *verbose=False*)

For hstacking Categoricals or fixing indices in a categorical from a stacked .sds load Supports Categoricals from single array or dictionary mapping

**Parameters**

- **indices** (*single stacked array or list of indices*) – if single array, needs idx_cutoffs for slicing
- **uniques** (list of stacked unique category arrays (needs unique_cutoffs)) – or list of lists of uniques
- **i_cutoffs** –
- **u_cutoffs** –
- **from_mapping** (*bool*) –
- **base_index** (*int*) –
- **ordered** (*bool*) –
- **verbose** (*bool*) –

**Returns**

- *list or array_like* – list of fixed indices, or array of fixed contiguous indices.
- *list of ndarray* – stacked unique values

riptable.rt_grouping.**hstack_test**(*arr_list*)

riptable.rt_grouping.**merge_cats**(*indices*, *listcats*, *idx_cutoffs=None*, *unique_cutoffs=None*, *from_mapping=False*, *stack=True*, *base_index=1*, *ordered=False*, *verbose=False*)

For hstacking Categoricals possibly from a stacked .sds load.

Supports Categoricals from single array or dictionary mapping.

**Parameters**

- **indices** (*single stacked array or list of indices*) – if single array, needs idx_cutoffs for slicing
- **listcats** (*list of stacked unique category arrays (needs unique_cutoffs)*) – or list of lists of uniques if the uniques in file1 are 'A,'C' and the uniques in file2 are 'B','C,'D' then listcats is [FastArray('A','C','B','C','D')]
- **idx_cutoffs** (*ndarray of int64, optional*) – int64 array of the cutoffs to the `indices`. if the index length is 30 and 20 the idx_cutoffs is [30,50]
- **unique_cutoffs** (*list of one int64 array of the cutoffs to the listcats*) – if the index length is 2 and 3 the idx_cutoffs is [2,5]
- **from_mapping** (*bool*) –
- **stack** (*bool*) –
- **base_index** (*int*) –
- **ordered** (*bool*) –
- **verbose** (*bool*) –

**Returns**

- *Returns two items*
- *- list of fixed indices, or array of fixed contiguous indices.*
- *- stacked unique values*

### Notes

TODO: Needs to support multikey cats.

## 2.2.22 riptable.rt_hstack

### Functions

| | |
|---|---|
| *hstack_any*(itemlist[, cls, baseclass, destroy]) | stack_rows or hstack_any is the main routine to row stack riptable classes. |

### Attributes

| |
|---|
| *stack_rows* |

riptable.rt_hstack.**hstack_any**(*itemlist*, *cls=None*, *baseclass=None*, *destroy=False*, ***kwargs*)

stack_rows or hstack_any is the main routine to row stack riptable classes. It stacks categoricals, datasets, time objects, structs. It can now stack a dictionary of numpy arrays to return a single array and a categorical.

**Parameters**

- **itemlist** – a list of objects to stack for arrays, datasets, categoricals a dictionary of numpy arrays
- **cls** (*None. the type of class we are stacking*) –
- **baseclass** (*the baseclass we are stacking*) –
- **destroy** (*bool, False. Only valid for Datasets*) – !! This is dangerous so make sure you do not want the data anymore in the original datasets.

**Returns**

- **In the case of a list** (*returns a single new array, dataset, categorical or specified object.*)
- **In the case of a dict** (*returns a single new array and a new categorical (two objects returned).*)

### Examples

```
>>> stack_rows([arange(3), arange(2)])
FastArray([0, 1, 2, 0, 1])
```

```
>>> d={'test1':arange(3), 'test2':arange(1), 'test3':arange(2)}
>>> arr, cat = stack_rows(d)
>>> Dataset({'Data':arr, 'Cat': cat})
#   Data   Cat
-   ----   -----
0      0   test1
1      1   test1
2      2   test1
3      0   test2
4      0   test3
5      1   test3
```

**See also:**

np.hstack, rt.Categorical.align, rt.Dataset.concat_rows

riptable.rt_hstack.**stack_rows**

### 2.2.23 riptable.rt_imatrix

#### Classes

| | |
|---|---|
| *IMatrix* | Experimental class designed to take a Dataset and make a 2d matrix efficiently. |

**class** riptable.rt_imatrix.**IMatrix**(*ds*, *dtype=None*, *order='F'*, *colnames=None*)

Experimental class designed to take a Dataset and make a 2d matrix efficiently. It uses rt.vstack order='F' which uses rt.hstack plus np.reshape.

The matrix is shaped so that it can be inserted back into the Dataset.

**Parameters**

- **dtype** –
- **order** –
- **colnames** –

**See also:**

`rt.vstack`

**property dataset**

**property imatrix**

**__getitem__**(*fld*)

row slicing

**apply2d**(*func*, *name=None*, *showfilter=True*)

**Parameters**
**func** (*function or method name of function*) –

**Return type**
X and Y axis calculations

**rebuild**(*ds=None*, *dtype=None*, *order='F'*, *colnames=None*)

### 2.2.24 riptable.rt_io

#### Functions

| | |
|---|---|
| *printh*(data) | Allows jupyter lab/notebook to print multiple HTML renderings in the same output frame. |
| *read_dset_from_np*(outdir, fname[, mmap]) | Read columns stored as numpy follows to a Dataset. |
| *write_dset_to_np*(ds, outdir, fname) | Write the columns of a dataset to numpy binary files, one file per column in the specified directory. |

riptable.rt_io.**printh**(*data*)

Allows jupyter lab/notebook to print multiple HTML renderings in the same output frame.

Suppose you have three datasets: d1, d2, d3 In one jupyter cell you could write: printh(d1) printh(d2) printh(d3) And all three would be displayed, versus the default, where only the last is shown. Will also work for anything else with a _repr_html_ method.

You can also input a list of elements with _repr_html_ methods so that they display side by side. If the jupyter frame isn't wide enough, they'll just display below.

**Parameters**
**data** (*object or list of objects*) – The object(s) to be rendered for display.

riptable.rt_io.**read_dset_from_np**(*outdir*, *fname*, *mmap=False*)

Read columns stored as numpy follows to a Dataset.

**Parameters**

- **the** (*outdir is the path and fname is the name of*) –
- **dataset** (*subdirectory containing the columns of the*) –
- **will** (*set mmap = True for memmory mapping. Note this*) –
- **loading** (*allow quick*) –
- **elsewhere** (*but has some latency cost*) –

**Returns**
The dataset read in from the specified folder.

**Return type**
*Dataset*

**See also:**

*write_dset_to_np*

riptable.rt_io.**write_dset_to_np**(*ds*, *outdir*, *fname*)

Write the columns of a dataset to numpy binary files, one file per column in the specified directory.

**Parameters**

- **ds** (Dataset) – A Dataset to write out to disk.
- **outdir** (*str*) – The path to the folder where the output will be written.
- **fname** (*str*) – The name of the subdirectory to store the columns.

**See also:**

*read_dset_from_np*

### 2.2.25 riptable.rt_itemcontainer

#### Classes

| | |
|---|---|
| *ItemContainer* | Container for items in Struct -- all values are tuples with an attribute |

class riptable.rt_itemcontainer.**ItemContainer**(**args*, ***kwds*)

Container for items in Struct – all values are tuples with an attribute

**__contains__**(**args*)

**__delitem__**(*key*)

ic.__delitem__(y) <==> del ic[y]

**__eq__**(*other*)

Return self==value.

**__getitem__**(*key*)

**__iter__**()

**__len__**()

**__ne__**(*other*)

Return self!=value.

**__next__**()

**__repr__**()

ic.__repr__() <==> repr(ic)

**__setitem__**(*key*, *value*)

ic.__setitem__(i, y) <==> ic[i]=y

**_get_move_cols**(*cols*)

Possibly convert list/array/dictionary/string/index of items to move for item_move_to_front(), item_move_to_back()

**_set_attribute**(*item*, *name*, *value*)

**_tagged_as_dict**(*attrname*)

Returns dictionary of columns tagged with attrname.

**_tagged_get_dict_max**(*attrname*)

Returns unordered dictionary of columns tagged with attrname, max value for order.

**_tagged_get_names**(*attrname*)

Returns a list of item names tagged with attrname in order.

**_tagged_remove**(*attrname*)

Removes existing items tagged with attrname.

**_tagged_set_names**(*listnames*, *attrname*)

Removes existing items tagged with attrname. If items in listnames exist, they will be tagged with attrname.

**apply**(*func*, **args*, *cols=None*)

Performs a possibly inplace operation on items in the itemcontainer

**clear**()

**copy**(*cols=None*, *deep=False*)

Returns a shallow copy of the item container. cols list can be provided for specific selection.

**copy_apply**(*func*, **args*, *cols=None*)

Returns a copy of the itemcontainer, applying a function to items before swapping them out in the new ItemContainer object. Used in Dataset row masking.

**copy_inplace**(*selector*)

inplace row-selector (mask, fancy-index, or slice) applied

**footer_get_value**(*key*)

**footer_set_value**(*key*, *value*)

**get_dict_values**()

Returns a tuple of items in the item dict. Each item is a list.

**item_add_prefix**(*prefix*)

inplace operation. adds prefix in front of existing item name

faster than calling rename

**item_add_suffix**(*suffix*)

inplace operation. adds suffix in front of existing item name

faster than calling rename

**item_delete**(*key*)

**item_exists**(*item*)

**item_get_attribute**(*key*, *attrib_name*, *default=None*)

Retrieve the value of the attribute previously assigned with *item_set_attribute*.

**Parameters**

- **key** – name of the item
- **attrib_name** – name of the attribute

**item_get_dict**()

return the underlying dict

values are stored in the first tuple, attributes in the second tuple

**item_get_len**()

**item_get_value**(*key*)

return the value for the given key

NOTE: a good spot to put a counter for debugging

**item_get_values**(*keylist*)

return list of value for the given key used for fast dataset slicing/copy with column selection

**item_move_to_back**(*cols*)

Move single column or group of columns to front of list for iteration/indexing/display. Values of columns will remain unchanged.

**Parameters**
**cols** – list of column names to move.

**Returns**
None

item_move_to_front(*cols*)

Move single column or group of columns to front of list for iteration/indexing/display. Values of columns will remain unchanged.

**Parameters**
**cols** – list of column names to move.

**Returns**
None

item_rename(*old*, *new*)

Rename a single column.

**Parameters**

- **old** – Current column name.
- **new** – New column name.

**Returns**
value portion of item that was renamed

item_replace_all(*newdict*, *check_exists=True*)

Replace the data for each item in the item dict. Original attributes will be retained.

**Parameters**

- **newdict** (*dictionary of item names -> new item data (can also be a dataset)*) –
- **check_exists** (*if True, all newdict keys and old item keys will be compared to ensure a match*) –

item_set_attribute(*key*, *attrib_name*, *attrib_value*)

Attach an attribute (name,value) pair to the item.

Any valid dictionary name and any object can be assigned.

Note: see *item_get_attribute* to retrieve.

**Parameters**

- **key** – name of the item
- **attrib_name** – name of the attribute
- **attrib_value** – value of the attribute

item_set_value(*key*, *value*, *attr=None*)

item_set_value_internal(*key*, *value*)

item_str_match(*expression*, *flags=0*)

Create a boolean mask vector for items whose names match the regex. NB Uses re.match(), not re.search().

**Parameters**

- **expression** – regular expression
- **flags** – regex flags (from re module).

**Returns**
list array of bools (len ncols) which is true for columns which match the regex.

**item_str_replace**(*old*, *new*, *maxr=-1*)

**Parameters**

- **old** – string to look for within individual names of columns
- **new** – string to replace old string in column names

If an item name contains the old string, the old string will be replaced with the new one. If replacing the string conflicts with an existing item name, an error will be raised.

returns True if column names were replaced

**items**()

**items_as_dict**()

Return dictionary of items without attributes.

**items_tolist**()

**iter_values**()

This will yield the full values in _items dict (lists with item, attribute)

**keys**()

**label_as_dict**()

**label_get_names**()

**label_remove**()

**label_set_names**(*listnames*)

**pop**(**args*)

**setdefault**(**args*)

**summary_as_dict**()

**summary_get_names**()

**summary_remove**()

**summary_set_names**(*listnames*)

**update**(**args*)

**values**()

### 2.2.26 riptable.rt_ledger

#### Classes

| | |
|---|---|
| *MathLedger* | MathLedger class for tracking math calculations and times |

class riptable.rt_ledger.**MathLedger**

Bases: *riptable.rt_struct.Struct*

MathLedger class for tracking math calculations and times

**DOCRC: bool = False**

**DebugUFunc: bool = False**

**Verbose: int = 1**

**VerboseConversion: int = 1**

**classmethod _ARRAY_UFUNC**(*func*, *ufunc*, *method*, **args*, ***kwargs*)

**classmethod _ASTYPE**(*func*, *dtype*, **args*, ***kwargs*)

use numpy astype

**classmethod _AS_FA_TYPE**(*faself*, *dtypenum*, **args*, ***kwargs*)

use multithreaded conversion preserving sentinels

**classmethod _AS_FA_TYPE_UNSAFE**(*faself*, *dtypenum*, **args*, ***kwargs*)

use multithreaded conversion NOT preserving sentinels

**classmethod _BASICMATH_ONE_INPUT**(*tupleargs*, *fastfunction*, *final_num*)

**classmethod _BASICMATH_TWO_INPUTS**(*tupleargs*, *fastfunction*, *final_num*)

**classmethod _COPY**(*func*, **args*, ***kwargs*)

**classmethod _FUNNEL_ALL**(*func*, **args*, ***kwargs*)

**classmethod _GETITEM**(*func*, **args*)

**classmethod _INDEX_BOOL**(**args*)

**classmethod _LCLEAR**()

**classmethod _LDUMP**(*dataset=True*)

**classmethod _LOFF**()

**classmethod _LON**()

**classmethod _Ledger**()

**classmethod _LedgerClear**()

**classmethod _LedgerDump**(*dataset=True*)

**classmethod _LedgerDumpFile**(*filename*)

**classmethod _LedgerOff**()

**classmethod _LedgerOn**()

**classmethod _MBGET**(**args*)

**classmethod _REDUCE**(*arg1*, *reduceFunc*)

**classmethod _TRACEBACK**(*func*)

print the callback stack to help with debugging

### 2.2.27 riptable.rt_matplotlib

### 2.2.28 riptable.rt_merge

#### Classes

| | |
|---|---|
| *JoinIndices* | Holds fancy/logical indices into the left and right Datasets constructed by the join implementation, |

#### Functions

| | |
|---|---|
| *merge*(left, right[, on, left_on, right_on, how, ...]) | Merge Dataset by performing a database-style join operation by columns. |
| *merge2*(left, right[, on, left_on, right_on, how, ...]) | Merge Dataset by performing a database-style join operation by columns. |
| *merge_asof*(left, right[, on, left_on, right_on, by, ...]) | Combine two *Dataset* objects by performing a database-style left-join based |
| *merge_indices*(left, right, *[, on, how, validate, ...]) | Perform a join/merge of two `Dataset` objects, returning the left/right indices created by the join engine. |
| *merge_lookup*(left, right[, on, left_on, right_on, ...]) | Combine two *Dataset* objects by performing a database-style left-join |

**class** riptable.rt_merge.**JoinIndices**

Bases: `NamedTuple`

Holds fancy/logical indices into the left and right Datasets constructed by the join implementation, along with other relevant data needed to construct the resulting merged Dataset.

**left_index:** *riptable.rt_fastarray.FastArray* | None

Integer fancy index or boolean mask for selecting data from the columns of the left Dataset to create the columns of the merged Dataset. This index is optional; when None, indicates that the columns from the left Dataset can be used directly in the resulting merged Dataset without needing to be filtered or otherwise transformed.

**right_index:** *riptable.rt_fastarray.FastArray* | None

Integer fancy index or boolean mask for selecting data from the columns of the right Dataset to create the columns of the merged Dataset. This index is optional; when None, indicates that the columns from the right Dataset can be used directly in the resulting merged Dataset without needing to be filtered or otherwise transformed.

**right_only_rowcount:** int | None

Only populated for outer merges. Indices the number of rows in the right Dataset whose keys do not occur in the left Dataset. This value can be used to slice `right_fancyindex` to get just the part at the end which represents these "right-only rows".

**static result_rowcount**(*index_arr*, *dset_rowcount*)

Calculate the number of rows resulting from indexing into a `Dataset` with a fancy/logical index.

**Parameters**

- **index_arr** (*np.ndarray, optional*) – A fancy index or boolean mask array to be used to select rows from a `Dataset`.

- **dset_rowcount** (*int*) – The number of rows in the `Dataset` that `index_arr` will be applied to.

**Returns**

The number of rows resulting from indexing into a `Dataset` with a fancy/logical index. Guaranteed to be non-negative.

**Return type**

int

riptable.rt_merge.**merge**(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *how='left'*, *suffixes=('_x', '_y')*, *indicator=False*, *columns_left=None*, *columns_right=None*, *verbose=False*, *hint_size=0*)

Merge Dataset by performing a database-style join operation by columns.

**Parameters**

- **left** (Dataset) – Left Dataset
- **right** (Dataset) – Right Dataset
- **on** (*str or list of str, optional*) – Column names to join on. Must be found in both `left` and `right`.
- **left_on** (*str or list of str, optional*) – Column names from left Dataset to join on. When specified, overrides whatever is specified in `on`.
- **right_on** (*str or list of str, optional*) – Column names from right to join on. When specified, overrides whatever is specified in `on`.
- **how** (*{'left','right', 'inner', 'outer'}*) –
  - left: use only keys from the left. **The output rows will be in one-to-one correspondence with the left rows!** If multiple matches on the right occur, the last is taken.
  - right: use only keys from the right. **The output rows will be in one-to-one correspondence with the left rows!** If multiple matches on the left occur, the last is taken.
  - inner: use intersection of keys from both Datasets, similar to SQL inner join
  - outer: use union of keys from both Datasets, similar to SQL outer join
- **suffixes** (*tuple of (str, str), default ('_x', '_y')*) – Suffix to apply to overlapping column names in the left and right side, respectively. To raise an exception on overlapping columns use (False, False).
- **indicator** (*bool or str, default False*) – If True, adds a column to output Dataset called "merge_indicator" with information on the source of each row. If string, column with information on source of each row will be added to output Dataset, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in `left` Dataset, "right_only" for observations whose merge key only appears in `right` Dataset, and "both" if the observation's merge key is found in both.
- **columns_left** (*str or list of str, optional*) – Column names to include in the merge from `left`, defaults to None which causes all columns to be included.
- **columns_right** (*str or list of str, optional*) – Column names to include in the merge from `right`, defaults to None which causes all columns to be included.
- **verbose** (*boolean*) – For the stdout debris, defaults to False

- **hint_size** (*int*) – An estimate of the number of unique keys used for the join, to optimize performance by pre-allocating memory for the key hash table.

**Returns**
**merged**

**Return type**
*Dataset*

### Examples

```
>>> rt.merge(ds_simple_1, ds_simple_2, left_on = 'A', right_on = 'X', how = 'inner')
#   A      B   X       C
-   -   ----   -   -----
0   0   1.20   0    2.40
1   1   3.10   1    6.20
2   6   9.60   6   19.20

[3 rows x 4 columns] total bytes: 72.0 B
```

Demonstrating a 'left' merge.

```
>>> rt.merge(ds_complex_1, ds_complex_2, on = ['A','B'], how = 'left')
#   B    A       C       E
-   -   --   -----   -----
0   Q    0    2.40    1.50
1   R    6    6.20   11.20
2   S    9   19.20     nan
3   T   11   25.90     nan

[4 rows x 4 columns] total bytes: 84.0 B
```

**See also:**

*merge_asof*

riptable.rt_merge.**merge2**(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *how='left'*, *suffixes=None*, *copy=True*, *indicator=False*, *columns_left=None*, *columns_right=None*, *validate=None*, *keep=None*, *high_card=None*, *hint_size=None*, ***kwargs*)

Merge Dataset by performing a database-style join operation by columns.

**Parameters**

- **left** (Dataset) – Left Dataset
- **right** (Dataset) – Right Dataset
- **on** (*str or (str, str) or list of str or list of (str, str), optional*) – Column names to join on. Must be found in both `left` and `right`.
- **left_on** (*str or list of str, optional*) – Column names from left Dataset to join on. When specified, overrides whatever is specified in `on`.
- **right_on** (*str or list of str, optional*) – Column names from right to join on. When specified, overrides whatever is specified in `on`.
- **how** (*{'left', 'right', 'inner', 'outer'}*) – The type of merge to be performed.
  - left: use only keys from `left`, as in a SQL 'left join'. Preserves the ordering of keys.

  - right: use only keys from `right`, as in a SQL 'right join'. Preserves the ordering of keys.
  - inner: use intersection of keys from both Datasets, as in a SQL 'inner join'. Preserves the ordering of keys from `left`.
  - outer: use union of keys from both Datasets, as in a SQL 'full outer join'.
- **suffixes** (*tuple of (str, str), optional*) – Suffix to apply to overlapping column names in the left and right side, respectively. The default (`None`) causes an exception to be raised for any overlapping columns.
- **copy** (*bool, default True*) – If False, avoid copying data when possible; this can reduce memory usage but users must be aware that data can be shared between `left` and/or `right` and the Dataset returned by this function.
- **indicator** (*bool or str, default False*) – If True, adds a column to output Dataset called "merge_indicator" with information on the source of each row. If string, column with information on source of each row will be added to output Dataset, and column will be named value of string. Information column is Categorical-type and takes on a value of "left_only" for observations whose merge key only appears in `left` Dataset, "right_only" for observations whose merge key only appears in `right` Dataset, and "both" if the observation's merge key is found in both.
- **columns_left** (*str or list of str, optional*) – Column names to include in the merge from `left`, defaults to None which causes all columns to be included.
- **columns_right** (*str or list of str, optional*) – Column names to include in the merge from `right`, defaults to None which causes all columns to be included.
- **validate** (*{'one_to_one', '1:1', 'one_to_many', '1:m', 'many_to_one', 'm:1', 'many_to_many', 'm:m'}, optional*) – Validate the uniqueness of the values in the columns specified by the `on`, `left_on`, `right_on` parameters. In other words, allows the _multiplicity_ of the keys to be checked so the user can prevent the merge if they want to ensure the uniqueness of the keys in one or both of the Datasets being merged. Note: The `keep` parameter logically takes effect before `validate` when they're both specified.
- **keep** (*{'first', 'last'} or (str, str), optional*) – An optional string which specifies that only the first or last occurrence of each unique key within `left` and `right` should be kept. In other words, resolves multiple occurrences of keys (multiplicity > 1) to a single occurrence.
- **high_card** (*bool or (bool, bool), optional*) – Hint to low-level grouping implementation that the key(s) of `left` and/or `right` contain a high number of unique values (cardinality); the grouping logic *may* use this hint to select an algorithm that can provide better performance for such cases.
- **hint_size** (*int or (int, int), optional*) – An estimate of the number of unique keys used for the join. Used as a performance hint to the low-level grouping implementation. This hint is typically ignored when `high_card` is specified.

**Returns**
**merged**

**Return type**
*Dataset*

### Examples

```
>>> rt.merge2(ds_simple_1, ds_simple_2, left_on = 'A', right_on = 'X', how = 'inner
↪')
#   A      B   X       C
-   -   ----   -   -----
0   0   1.20   0    2.40
1   1   3.10   1    6.20
2   6   9.60   6   19.20

[3 rows x 4 columns] total bytes: 72.0 B
```

Demonstrating a 'left' merge.

```
>>> rt.merge2(ds_complex_1, ds_complex_2, on = ['A','B'], how = 'left')
#   B    A       C       E
-   -   --   -----   -----
0   Q    0    2.40    1.50
1   R    6    6.20   11.20
2   S    9   19.20     nan
3   T   11   25.90     nan

[4 rows x 4 columns] total bytes: 84.0 B
```

**See also:**

*merge_asof*

riptable.rt_merge.**merge_asof**(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *by=None*, *left_by=None*, *right_by=None*, *suffixes=None*, *copy=True*, *columns_left=None*, *columns_right=None*, *tolerance=None*, *allow_exact_matches=True*, *direction='backward'*, *verbose=False*, *action_on_unsorted='sort'*, *matched_on=False*, ***kwargs*)

Combine two *Dataset* objects by performing a database-style left-join based on the nearest numeric key.

An as-of merge is useful for keys that are times (or other numeric values) that aren't exact matches but are close enough to merge on.

Both *Dataset* objects must be sorted by the key column.

Use the `direction` argument to find the nearest key in the right *Dataset*:

- A `direction="backward"` search selects the closest key that's less than or equal to the key in the left *Dataset*.
- A `direction="forward"` search selects the closest key that's greater than or equal to the key in the left *Dataset*.
- A `direction="nearest"` search selects the key that's closest in absolute distance to the key in the left *Dataset*.

Optionally, you can match on equivalent keys with `by` before performing an as-of merge with `on`.

**Parameters**

- **left** (Dataset) – The first *Dataset* to merge. If a nearest match for a key in `left` isn't found in `right`, the returned `.Dataset` includes a row with the columns from `left`, but with NaN values in each column from `right`.

- **right** (Dataset) – The second *Dataset* to merge. If rows in right don't have nearest matches in left they will be discarded. If they match multiple rows in left they will be duplicated appropriately.
- **on** (*str or (str, str)*) – Name of the column to join on (the key column). If the column name is the same in both *Dataset* objects, use a single string. If the column names are different, use a tuple of strings. The values must be numeric (such as integers, floats, or datetimes). If on isn't specified, left_on and right_on must be specified.
- **left_on** (*str, optional*) – Use instead of on to specify the column in the left *Dataset* to join on.
- **right_on** (*str, optional*) – Use instead of on to specify the column in the right *Dataset* to join on.
- **by** (*str or (str, str) or list of str or list of (str, str), optional*) – Match equal keys in these columns before performing the as-of merge. Options for types:
  - Single string: Join on one column that has the same name in both *Dataset* objects.
  - List: A list of strings is treated as a multi-key in which all associated key column values in left must have matches in right. The column names must be the same in both *Dataset* objects, unless they're in a tuple; see below.
  - Tuple: Use a tuple to specify key columns that have different names. For example, ("col_a", "col_b") joins on col_a in left and col_b in right. Both columns are in the returned *Dataset* unless you specify otherwise using columns_left or columns_right.
- **left_by** (*str or list of str, optional*) – Use instead of by to specify names of columns to match equivalent values on in the left *Dataset*.
- **right_by** (*str or list of str, optional*) – Use instead of by to specify names of columns to match equivalent values on in the right *Dataset*.
- **suffixes** (*(str, str), optional*) – Suffixes to apply to returned overlapping non-key-column names in left and right, respectively. By default, an error is raised for any overlapping non-key columns that will be in the returned *Dataset*.
- **copy** (*bool, default True*) – Set to False to avoid copying data when possible. This can reduce memory usage, but be aware that data can be shared among left, right, and the *Dataset* returned by this function.
- **columns_left** (*str or list of str, optional*) – Names of columns from left to include in the merged Dataset. By default, all columns are included.
- **columns_right** (*str or list of str, optional*) – Names of columns from right to include in the merged *Dataset*. By default, all columns are included.
- **tolerance** (*int, float, or timedelta, optional*) – **Not implemented.** Tolerance allowed when performing the as-of part of the merge. When a row from left doesn't have a key in right within this distance, that row will have a NaN for any columns from right that appear in the merged result.
- **allow_exact_matches** (*boolean, default True*) – If True (the default), allow matching with an equivalent on value (i.e., allow less-than-or-equal-to or greater-than-or-equal-to matches). If False, don't match an equivalent on value (i.e., perform only strictly-less-than or strictly-greater-than matches).
- **direction** (*{"backward", "forward", "nearest"}, default "backward"*) – Whether to search for prior, subsequent, or closest matches in the right *Dataset*.

- **verbose** (*bool, default False*) – Show information used for debugging.
- **check_sorted** (*bool, default True*) – Deprecated since version 1.10.0: See `action_on_unsorted`.
- **action_on_unsorted** (*{"sort", "raise"}, default "sort"*) – New in version 1.10.0: The `on` columns are always checked to see if they

  are sorted. If they're unsorted, by default they are sorted before the merge; the original order is then restored in the returned merged *Dataset*. Set to "raise" to raise an error for any unsorted `on` column.
- **matched_on** (*bool or str, default False*) – Add a column to the merged *Dataset* that contains the `on` column value from `right` that was matched. When `True`, the column uses the default name "matched_on"; specify a string to name the column.
- **left_index** (*bool, default False*) – Deprecated since version 1.10.0.

  This parameter is only provided for compatibility with `pandas.merge_asof` and has no effect in a Riptable as-of merge.
- **right_index** (*bool, default False*) – Deprecated since version 1.10.0.

  This parameter is only provided for compatibility with `pandas.merge_asof` and has no effect in a Riptable as-of merge.

**Returns**

A new *Dataset* of the two merged objects.

**Return type**

*Dataset*

**See also:**

***merge_lookup***

Merge two `Dataset` objects based on equivalent keys.

***merge2***

Merge two *Dataset* objects using various database-style joins.

***merge_indices***

Return the left and right indices created by the join engine.

***Dataset.merge_asof***

Merge two *Dataset* objects using the nearest key.

***Dataset.merge_lookup***

Merge two *Dataset* objects with an in-place option.

***Dataset.merge2***

Merge two *Dataset* objects using various database-style joins.

### Examples

```
>>> left = rt.Dataset({"a": [1, 5, 10], "left_val": ["a", "b", "c"]})
>>> right = rt.Dataset({"a": [1, 2, 3, 6, 7], "right_val": [1, 2, 3, 6, 7]})
>>> left
#    a   left_val
-   --   --------
0    1   a
1    5   b
2   10   c

[3 rows x 2 columns] total bytes: 15.0 B
>>> right
#   a   right_val
-   -   ---------
0   1           1
1   2           2
2   3           3
3   6           6
4   7           7

[5 rows x 2 columns] total bytes: 40.0 B
```

Merge based on the integers in the "a" columns. The first match is exact; the second two are "backward" nearest matches (the default direction):

```
>>> rt.merge_asof(left, right, on="a")
#    a   left_val   right_val
-   --   --------   ---------
0    1   a                  1
1    5   b                  3
2   10   c                  7

[3 rows x 3 columns] total bytes: 27.0 B
```

When `allow_exact_matches=False`, a nearest match is used if there is one (as for row 0). Here, also, there's no "forward" nearest match in `right` for row 2:

```
>>> rt.merge_asof(left, right, on="a", direction="forward", allow_exact_
↪matches=False)
#    a   left_val   right_val
-   --   --------   ---------
0    1   a                  2
1    5   b                  6
2   10   c                Inv

[3 rows x 3 columns] total bytes: 27.0 B
```

If `allow_exact_matches=False` and there are no nearest matches, a NaN value is filled in:

```
>>> rt.merge_asof(left, right, on="a", allow_exact_matches=False)
#    a   left_val   right_val
-   --   --------   ---------
```

(continues on next page)

(continued from previous page)

```
0    1   a                Inv
1    5   b                  3
2   10   c                  7

[3 rows x 3 columns] total bytes: 27.0 B
```

As-of merges are good for time-series data. Here, the *Dataset* objects are merged with on="Time" and by="Symbol" to get the nearest time associated with the same symbol:

```
>>> # Left Dataset with trades and times.
>>> ds = rt.Dataset({"Symbol": ["AAPL", "AMZN", "AAPL"],
...                  "Venue": ["A", "I", "A"],
...                  "Time": rt.TimeSpan(["09:30", "10:00", "10:20"])})
>>> # Right Dataset with spot prices and nearby times.
>>> spot_ds = rt.Dataset({"Symbol": ["AMZN", "AMZN", "AMZN", "AAPL", "AAPL", "AAPL
↪"],
...                       "Spot Price": [2000.0, 2025.0, 2030.0, 500.0, 510.0, 520.
↪0],
...                       "Time": rt.TimeSpan(["09:30", "10:00", "10:25", "09:25",
↪"10:00", "10:25"])})
>>> ds
#   Symbol   Venue                Time
-   ------   -----   ------------------
0   AAPL     A       09:30:00.000000000
1   AMZN     I       10:00:00.000000000
2   AAPL     A       10:20:00.000000000

[3 rows x 3 columns] total bytes: 39.0 B
>>> spot_ds
#   Symbol   Spot Price                Time
-   ------   ----------   ------------------
0   AMZN       2,000.00   09:30:00.000000000
1   AMZN       2,025.00   10:00:00.000000000
2   AMZN       2,030.00   10:25:00.000000000
3   AAPL         500.00   09:25:00.000000000
4   AAPL         510.00   10:00:00.000000000
5   AAPL         520.00   10:25:00.000000000

[6 rows x 3 columns] total bytes: 120.0 B
```

Note that an as-of merge requires the on columns to be sorted. Before the merge, the on columns are always checked. If they're not sorted, by default they are sorted before the merge; the original order is then restored in the returned merged *Dataset*.

If you don't need to preserve the existing ordering, it's faster to sort the on columns in place first:

```
>>> spot_ds.sort_inplace("Time")
#   Symbol   Spot Price                Time
-   ------   ----------   ------------------
0   AAPL         500.00   09:25:00.000000000
1   AMZN       2,000.00   09:30:00.000000000
2   AMZN       2,025.00   10:00:00.000000000
3   AAPL         510.00   10:00:00.000000000
```

(continues on next page)

(continued from previous page)

```
4   AAPL         520.00   10:25:00.000000000
5   AMZN       2,030.00   10:25:00.000000000

[6 rows x 3 columns] total bytes: 120.0 B
```

Get the nearest earlier time:

```
>>> rt.merge_asof(ds, spot_ds, on="Time", by="Symbol", direction="backward",␣
↪matched_on=True)
#   Symbol                 Time   Venue   Spot Price           matched_on
-   ------   ------------------   -----   ----------   ------------------
0   AAPL     09:30:00.000000000   A           500.00   09:25:00.000000000
1   AMZN     10:00:00.000000000   I          2025.00   10:00:00.000000000
2   AAPL     10:20:00.000000000   A           510.00   10:00:00.000000000

[3 rows x 5 columns] total bytes: 87.0 B
```

Get the nearest later time:

```
>>> rt.merge_asof(ds, spot_ds, on="Time", by="Symbol", direction="forward", matched_
↪on=True)
#   Symbol                 Time   Venue   Spot Price           matched_on
-   ------   ------------------   -----   ----------   ------------------
0   AAPL     09:30:00.000000000   A           510.00   10:00:00.000000000
1   AMZN     10:00:00.000000000   I          2025.00   10:00:00.000000000
2   AAPL     10:20:00.000000000   A           520.00   10:25:00.000000000

[3 rows x 5 columns] total bytes: 87.0 B
```

Get the nearest time, whether earlier or later:

```
>>> rt.merge_asof(ds, spot_ds, on="Time", by="Symbol", direction="nearest", matched_
↪on=True)
#   Symbol                 Time   Venue   Spot Price           matched_on
-   ------   ------------------   -----   ----------   ------------------
0   AAPL     09:30:00.000000000   A           500.00   09:25:00.000000000
1   AMZN     10:00:00.000000000   I          2025.00   10:00:00.000000000
2   AAPL     10:20:00.000000000   A           520.00   10:25:00.000000000

[3 rows x 5 columns] total bytes: 87.0 B
```

riptable.rt_merge.**merge_indices**(*left*, *right*, *, *on=None*, *how='left'*, *validate=None*, *keep=None*, *high_card=None*, *hint_size=None*, ***kwargs*)

Perform a join/merge of two `Dataset` objects, returning the left/right indices created by the join engine.

The returned indices can be used to index into the left and right `Dataset` objects to construct a merged/joined `Dataset`.

**Parameters**

- **left** (`Dataset`) – Left Dataset
- **right** (`Dataset`) – Right Dataset
- **on** (*str or (str, str) or list of str or list of (str, str), optional*) – Column names to join on. Must be found in both `left` and `right`.

- **how** (*{'left', 'right', 'inner', 'outer'}*) – The type of merge to be performed.
  - left: use only keys from `left`, as in a SQL 'left join'. Preserves the ordering of keys.
  - right: use only keys from `right`, as in a SQL 'right join'. Preserves the ordering of keys.
  - inner: use intersection of keys from both Datasets, as in a SQL 'inner join'. Preserves the ordering of keys from `left`.
  - outer: use union of keys from both Datasets, as in a SQL 'full outer join'.
- **validate** (*{'one_to_one', '1:1', 'one_to_many', '1:m', 'many_to_one', 'm:1', 'many_to_many', 'm:m'}, optional*) – Validate the uniqueness of the values in the columns specified by the `on`, `left_on`, `right_on` parameters. In other words, allows the _multiplicity_ of the keys to be checked so the user can prevent the merge if they want to ensure the uniqueness of the keys in one or both of the Datasets being merged. Note: The `keep` parameter logically takes effect before `validate` when they're both specified.
- **keep** (*{'first', 'last'} or (str, str), optional*) – An optional string which specifies that only the first or last occurrence of each unique key within `left` and `right` should be kept. In other words, resolves multiple occurrences of keys (multiplicity > 1) to a single occurrence.
- **high_card** (*bool or (bool, bool), optional*) – Hint to low-level grouping implementation that the key(s) of `left` and/or `right` contain a high number of unique values (cardinality); the grouping logic *may* use this hint to select an algorithm that can provide better performance for such cases.
- **hint_size** (*int or (int, int), optional*) – An estimate of the number of unique keys used for the join. Used as a performance hint to the low-level grouping implementation. This hint is typically ignored when `high_card` is specified.

**Return type**
*JoinIndices*

### Examples

```
>>> rt.merge_indices(ds_simple_1, ds_simple_2, on=('A', 'X'), how = 'inner')
#   A      B   X       C
-   -   ----   -   -----
0   0   1.20   0    2.40
1   1   3.10   1    6.20
2   6   9.60   6   19.20

[3 rows x 4 columns] total bytes: 72.0 B
```

Demonstrating a 'left' merge.

```
>>> rt.merge_indices(ds_complex_1, ds_complex_2, on = ['A','B'], how = 'left')
#   B    A       C       E
-   -   --   -----   -----
0   Q    0    2.40    1.50
1   R    6    6.20   11.20
2   S    9   19.20     nan
3   T   11   25.90     nan
```

(continues on next page)

(continued from previous page)

```
[4 rows x 4 columns] total bytes: 84.0 B
```

**See also:**

*merge2*

riptable.rt_merge.**merge_lookup**(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *require_match=False*, *suffixes=None*, *copy=True*, *columns_left=None*, *columns_right=None*, *keep=None*, *high_card=None*, *hint_size=None*)

Combine two *Dataset* objects by performing a database-style left-join operation on columns.

This operation returns a new *Dataset* object. To do an in-place merge, use *Dataset.merge_lookup* with `inplace=True`.

**Parameters**

- **left** (*Dataset*) – The first *Dataset* to merge. If a matching value for a key in `left` isn't found in `right`, the returned *Dataset* includes a row with the columns from `left`, but with NaN values in each column from `right`.
- **right** (*Dataset*) – The second *Dataset* to merge. If rows in `right` don't have matches in `left` they will be discarded. If they match multiple rows in `left` they will be duplicated appropriately.
- **on** (*str or (str, str) or list of str or list of (str, str), optional*) – Names of columns (keys) to join on. If `on` isn't specified, `left_on` and `right_on` must be specified. Options for types:
  - Single string: Join on one column that has the same name in both *Dataset* objects.
  - List: A list of strings is treated as a multi-key in which all associated key column values in `left` must have matches in `right`. The column names must be the same in both *Dataset* objects, unless they're in a tuple; see below.
  - Tuple: Use a tuple to specify key columns that have different names. For example, `("col_a", "col_b")` joins on `col_a` in `left` and `col_b` in `right`. Both columns are in the returned *Dataset* unless you specify otherwise using `columns_left` or `columns_right`.
- **left_on** (*str or list of str, optional*) – Use instead of `on` to specify names of columns in the left *Dataset* to join on. A list of strings is treated as a multi-key in which all associated key column values in `left` must have matches in `right`. If both `on` and `left_on` are specified, an error is raised.
- **right_on** (*str or list of str, optional*) – Use instead of `on` to specify names of columns in the right *Dataset* to join on. A list of strings is treated as a multi-key in which all associated key column values in `right` must have matches in `left`. If both `on` and `right_on` are specified, an error is raised.
- **require_match** (bool, default `False`) – When `True`, all keys in `left` are required to have a matching key in `right`, and an error is raised when this requirement is not met.
- **suffixes** (*tuple of (str, str), optional*) – Suffixes to apply to returned overlapping non-key-column names in `left` and `right`, respectively. By default, an error is raised for any overlapping non-key columns that will be in the returned *Dataset*.
- **copy** (bool, default `True`) – Set to `False` to avoid copying data when possible. This can reduce memory usage, but be aware that data can be shared among `left`, `right`, and the *Dataset* returned by this function.

- **columns_left** (*str or list of str, optional*) – Names of columns from `left` to include in the merged `Dataset`. By default, all columns are included.
- **columns_right** (*str or list of str, optional*) – Names of columns from `right` to include in the merged *Dataset*. By default, all columns are included.
- **keep** (*{None, 'first', 'last'}, optional*) – When `right` contains multiple rows with a given unique key from `left`, keep only one such row; this parameter indicates whether it should be the first or last row with the given key. By default (`keep=None`), an error is raised if there are any non-unique keys in `right`.
- **high_card** (*bool or (bool, bool), optional*) – Hint to the low-level grouping implementation that the key(s) of `left` or `right` contain a high number of unique values (cardinality); the grouping logic *may* use this hint to select an algorithm that can provide better performance for such cases.
- **hint_size** (*int or (int, int), optional*) – An estimate of the number of unique keys used for the join. Used as a performance hint to the low-level grouping implementation. This hint is typically ignored when `high_card` is specified.

**Returns**
A new *Dataset* of the two merged objects.

**Return type**
*Dataset*

**See also:**

***merge_asof***
Merge two *Dataset* objects using the nearest key.

***merge2***
Merge two *Dataset* objects using various database-style joins.

***merge_indices***
Return the left and right indices created by the join engine.

***Dataset.merge_lookup***
Merge two *Dataset* objects with an in-place option.

***Dataset.merge2***
Merge two *Dataset* objects using various database-style joins.

***Dataset.merge_asof***
Merge two *Dataset* objects using the nearest key.

### Examples

A basic merge on a single column:

```
>>> ds_l = rt.Dataset({"Symbol": rt.FA(["GME", "AMZN", "TSLA", "SPY", "TSLA",
...                                     "AMZN", "GME", "SPY", "GME", "TSLA"])})
>>> ds_r = rt.Dataset({"Symbol": rt.FA(["TSLA", "GME", "AMZN", "SPY"]),
...                    "Trader": rt.FA(["Nate", "Elon", "Josh", "Dan"])})
>>> ds_l
#   Symbol
-   ------
0   GME
1   AMZN
```

(continues on next page)

(continued from previous page)

```
2   TSLA
3   SPY
4   TSLA
5   AMZN
6   GME
7   SPY
8   GME
9   TSLA

[10 rows x 1 columns] total bytes: 40.0 B
>>> ds_r
#   Symbol   Trader
-   ------   ------
0   TSLA     Nate
1   GME      Elon
2   AMZN     Josh
3   SPY      Dan

[4 rows x 2 columns] total bytes: 32.0 B
>>> rt.merge_lookup(ds_l, ds_r, on="Symbol")
#   Symbol   Trader
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY      Dan
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY      Dan
8   GME      Elon
9   TSLA     Nate

[10 rows x 2 columns] total bytes: 80.0 B
```

When key columns have different names, use `left_on` and `right_on` to specify them:

```
>>> ds_r.col_rename("Symbol", "Primary_Symbol")
>>> rt.merge_lookup(ds_l, ds_r, left_on="Symbol", right_on="Primary_Symbol",
...                 columns_right="Trader")
#   Symbol   Trader
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY      Dan
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY      Dan
8   GME      Elon
9   TSLA     Nate
```

(continues on next page)

(continued from previous page)

```
[10 rows x 2 columns] total bytes: 80.0 B
```

For non-key columns with the same name that will be returned, specify `suffixes`:

```
>>> # Add duplicate non-key columns.
>>> ds_l.Value = rt.FA([0.72, 0.85, 0.14, 0.55, 0.77, 0.65, 0.23, 0.15, 0.43, 0.25])
>>> ds_r.Value = rt.FA([0.28, 0.56, 0.89, 0.74])
>>> # You can also use a tuple to specify left and right key columns.
>>> rt.merge_lookup(ds_l, ds_r, on=("Symbol", "Primary_Symbol"),
...                 suffixes=["_1", "_2"], columns_right=["Value", "Trader"])
#   Symbol   Value_1   Value_2   Trader
-   ------   -------   -------   ------
0   GME         0.72      0.56   Elon
1   AMZN        0.85      0.89   Josh
2   TSLA        0.14      0.28   Nate
3   SPY         0.55      0.74   Dan
4   TSLA        0.77      0.28   Nate
5   AMZN        0.65      0.89   Josh
6   GME         0.23      0.56   Elon
7   SPY         0.15      0.74   Dan
8   GME         0.43      0.56   Elon
9   TSLA        0.25      0.28   Nate

[10 rows x 4 columns] total bytes: 240.0 B
```

When `on` is a list, a multi-key join is performed. All keys must match in the right *Dataset*.

If a matching value for a key in the left *Dataset* isn't found in the right *Dataset*, the returned *Dataset* includes a row with the columns from `left` but with NaN values in the columns from `right`.

```
>>> # Add associated Size values for multi-key join. Note that one
>>> # symbol-size pair in the left Dataset doesn't have a match in
>>> # the right Dataset.
>>> ds_l.Size = rt.FA([500, 150, 430, 225, 430, 320, 175, 620, 135, 260])
>>> ds_r.Size = rt.FA([430, 500, 150, 2250])
>>> # Pass a list of key columns that contains a tuple.
>>> rt.merge_lookup(ds_l, ds_r, on=[("Symbol", "Primary_Symbol"), "Size"],
...                 suffixes=["_1", "_2"])
#   Size   Symbol   Value_1   Trader   Value_2
-   ----   ------   -------   ------   -------
0    500   GME         0.72   Elon        0.56
1    150   AMZN        0.85   Josh        0.89
2    430   TSLA        0.14   Nate        0.28
3    225   SPY         0.55                nan
4    430   TSLA        0.77   Nate        0.28
5    320   AMZN        0.65                nan
6    175   GME         0.23                nan
7    620   SPY         0.15                nan
8    135   GME         0.43                nan
9    260   TSLA        0.25                nan

[10 rows x 5 columns] total bytes: 280.0 B
```

When the right *Dataset* has more than one matching key, use keep to specify which one to use:

```
>>> ds_l = rt.Dataset({"Symbol": rt.FA(["GME", "AMZN", "TSLA", "SPY", "TSLA",
...                                     "AMZN", "GME", "SPY", "GME", "TSLA"])})
>>> ds_r = rt.Dataset({"Symbol": rt.FA(["TSLA", "GME", "AMZN", "SPY", "SPY"]),
...                    "Trader": rt.FA(["Nate", "Elon", "Josh", "Dan", "Amy"])})
>>> rt.merge_lookup(ds_l, ds_r, on="Symbol", keep="last")
#   Symbol   Trader
-   ------   ------
0   GME      Elon
1   AMZN     Josh
2   TSLA     Nate
3   SPY      Amy
4   TSLA     Nate
5   AMZN     Josh
6   GME      Elon
7   SPY      Amy
8   GME      Elon
9   TSLA     Nate

[10 rows x 2 columns] total bytes: 80.0 B
```

Invalid values are not treated as equal keys:

```
>>> ds_l = rt.Dataset({"Key": [1.0, rt.nan, 2.0,], "Value1": [1.0, 2.0, 3.0]})
>>> ds_r = rt.Dataset({"Key": [1.0, 2.0, rt.nan], "Value2": [1.0, 2.0, 3.0]})
>>> rt.merge_lookup(ds_l, ds_r, on="Key", columns_right="Value2")
#    Key   Value1   Value2
-   ----   ------   ------
0   1.00     1.00     1.00
1    nan     2.00      nan
2   2.00     3.00     2.00

[3 rows x 3 columns] total bytes: 72.0 B
```

### 2.2.29 riptable.rt_merge_asof

Time/ordering-based merge implementations.

#### Functions

| | |
|---|---|
| *merge_asof2*(left, right[, on, left_on, right_on, by, ...]) | Perform an as-of merge. This is similar to a left-join except that we match on nearest key rather than equal keys. |

riptable.rt_merge_asof.**merge_asof2**(*left*, *right*, *on=None*, *left_on=None*, *right_on=None*, *by=None*, *left_by=None*, *right_by=None*, *suffixes=None*, *copy=True*, *columns_left=None*, *columns_right=None*, ***, *tolerance=None*, *allow_exact_matches=True*, *direction='backward'*, *matched_on=False*, ***kwargs*)

Perform an as-of merge. This is similar to a left-join except that we match on nearest key rather than equal keys.

Both Datasets must be sorted (ascending) by the 'on' column. When 'by' columns are specified, the 'on' column for each Dataset only needs to be sorted (ascending) within each unique 'key' of the 'by' columns. Sorting the entire `Dataset` by the 'on' column also meets this requirement, but some `Datasets` may have an 'on' column which is already pre-sorted within each 'by' key, in which case no additional sorting is required.

**For each row in the left Dataset:**

- A "backward" search selects the last row in the right Dataset whose 'on' key is less than or equal to the left's key.
- A "forward" search selects the first row in the right Dataset whose 'on' key is greater than or equal to the left's key.
- A "nearest" search selects the row in the right Dataset whose 'on' key is closest in absolute distance to the left's key.

Optionally match on equivalent keys with 'by' before searching with 'on'.

**Parameters**

- **left** (*Dataset*) – Left Dataset
- **right** (*Dataset*) – Right Dataset
- **on** (*str*) – Column name to join on. Must be found in both the `left` and `right` Datasets. This column in both left and right Datasets MUST be ordered. Furthermore this must be a numeric column, such as datetimelike, integer, or float. Either `on` or `left_on`/`right_on` must be specified.
- **left_on** (*str or list of str, optional*) – Column name to join on in `left` Dataset.
- **right_on** (*str or list of str, optional*) – Column name to join on in `right` Dataset.
- **by** (*str or (str, str) or list of str or list of (str, str), optional*) – Column name or list of column names. Match on these columns before performing merge operation.
- **left_by** (*str or list of str, optional*) – Column names to match on in the left Dataset.
- **right_by** (*str or list of str, optional*) – Column names to match on in the right Dataset.
- **suffixes** (*(str, str), optional, default None*) – Suffix to apply to overlapping column names in the left and right side, respectively.
- **copy** (*bool, default True*) – If False, avoid copying data when possible; this can reduce memory usage but users must be aware that data can be shared between `left` and/or `right` and the Dataset returned by this function.
- **columns_left** (*str or list of str, optional*) – Column names to include in the merge from `left`, defaults to None which causes all columns to be included.
- **columns_right** (*str or list of str, optional*) – Column names to include in the merge from `right`, defaults to None which causes all columns to be included.
- **tolerance** (*integer or float or Timedelta, optional, default None*) – Tolerance allowed when performing the 'asof' part of the merge; whenever a row from `left` doesn't have a key in `right` within this distance or less, that row will have a null/missing/NA value for any columns from the `right` Dataset which appear in the merged result.

- **allow_exact_matches** (*boolean, default True*) –
  - If True, allow matching with the same 'on' value (i.e. less-than-or-equal-to / greater-than-or-equal-to)
  - If False, don't match the same 'on' value (i.e., strictly less-than / strictly greater-than)
- **direction** (*{'backward', 'forward', or 'nearest'}, default 'backward'*) – Whether to search for prior, subsequent, or closest matches.
- **matched_on** (*bool or str, default False*) – If set to True or a string, an additional column is added to the result; for each row, it contains the value from the `on` column in `right` that was matched. When True, the column will use the default name 'matched_on'; specify a string to explicitly name the column.

**Returns**
**merged**

**Return type**
*Dataset*

**Raises**
`ValueError` – The `on`, `left_on`, or `right_on` columns are not sorted in ascending order within one or more keys of the `by` columns.

**See also:**

`riptable.merge_asof`

### Notes

**TODO: Consider allowing the `tolerance` parameter to also be a 1D array/sequence whose length is the same**
as the number of groups (or keys?) in the `left` dataset. That allows a per-group tolerance to be specified if needed.

### Examples

```
>>> left = rt.Dataset({'a': [1, 5, 10], 'left_val': ['a', 'b', 'c']})
>>> left
#    a   left_val
-   --   --------
0    1   a
1    5   b
2   10   c
>>> right = rt.Dataset({'a': [1, 2, 3, 6, 7],
...                     'right_val': [1, 2, 3, 6, 7]})
>>> right
#   a   right_val
-   -   ---------
0   1           1
1   2           2
2   3           3
3   6           6
4   7           7
>>> rt.merge_asof(left, right, on='a')
```

(continues on next page)

(continued from previous page)

```
#   a_x   left_val   a_y   right_val
-   ---   --------   ---   ---------
0     1   a            1           1
1     5   b            3           3
2    10   c            7           7
```

```
>>> rt.merge_asof(left, right, on='a', allow_exact_matches=False)
 #   a_x   left_val   a_y   right_val
-   ---   --------   ---   ---------
0     1   a          Inv         Inv
1     5   b            3           3
2    10   c            7           7
```

```
>>> rt.merge_asof(left, right, on='a', direction='forward')
#   a_x   left_val   a_y   right_val
-   ---   --------   ---   ---------
0     1   a            1           1
1     5   b            6           6
2    10   c          Inv         Inv
```

Here is a real-world time-series example

```
>>> quotes
#                   time   ticker   Bid   Ask
-   -------------------   ------   ----  ----
0   20191015 09:45:57.09    AAPL   3.40  3.50
1   20191015 11:35:09.76    AAPL   3.45  3.55
2   20191015 12:02:27.11    AAPL   3.50  3.60
3   20191015 12:43:13.73    MSFT   2.85  2.95
4   20191015 14:32:11.18    MSFT   2.90  3.00
```

```
>>> trades
#                   time   ticker   TradePrice   TradeSize
-   -------------------   ------   ----------   ---------
0   20191015 10:03:24.73    AAPL         3.45        1.00
1   20191015 10:41:22.79    MSFT         2.85        1.00
2   20191015 10:41:35.69    MSFT         2.86        1.00
3   20191015 11:04:32.55    AAPL         3.47        1.00
4   20191015 11:44:35.63    MSFT         2.91        1.00
5   20191015 12:26:17.68    AAPL         3.55        1.00
6   20191015 14:24:10.93    MSFT         2.98        1.00
7   20191015 15:45:13.41    AAPL         3.60        7.00
8   20191015 15:50:42.53    AAPL         3.58        1.00
9   20191015 15:53:59.60    AAPL         3.56        5.00
```

```
>>> rt.merge_asof(trades, quotes, on='time', by='ticker')
#                  time_x   ticker_x  TradePrice   TradeSize                  time_y ␣
↪ ticker_y    Bid    Ask
-   -------------------   ---------  ----------   ---------   -------------------- ␣
↪ --------   ----   ----
0   20191015 10:03:24.73       AAPL        3.45        1.00   20191015 09:45:57.09 ␣
```

(continues on next page)

(continued from previous page)

```
↪     AAPL   3.40   3.50
1   20191015 10:41:22.79       MSFT        2.85        1.00                    Inv ␣
↪ Filtered    nan    nan
2   20191015 10:41:35.69       MSFT        2.86        1.00                    Inv ␣
↪ Filtered    nan    nan
3   20191015 11:04:32.55       AAPL        3.47        1.00   20191015 09:45:57.09 ␣
↪     AAPL   3.40   3.50
4   20191015 11:44:35.63       MSFT        2.91        1.00                    Inv ␣
↪ Filtered    nan    nan
5   20191015 12:26:17.68       AAPL        3.55        1.00   20191015 12:02:27.11 ␣
↪     AAPL   3.50   3.60
6   20191015 14:24:10.93       MSFT        2.98        1.00   20191015 12:43:13.73 ␣
↪     MSFT   2.85   2.95
7   20191015 15:45:13.41       AAPL        3.60        7.00   20191015 12:02:27.11 ␣
↪     AAPL   3.50   3.60
8   20191015 15:50:42.53       AAPL        3.58        1.00   20191015 12:02:27.11 ␣
↪     AAPL   3.50   3.60
9   20191015 15:53:59.60       AAPL        3.56        5.00   20191015 12:02:27.11 ␣
↪     AAPL   3.50   3.60
```

only merge with the forward quotes

```
>>> rt.merge_asof(trades, quotes, on='time', by='ticker', direction='forward')
#                 time_x  ticker_x  TradePrice  TradeSize                time_y ␣
↪ticker_y   Bid   Ask
-   ------------------  --------  ----------  ---------  ------------------  ---
↪-----  ----  ----
0   20191015 10:03:24.73      AAPL        3.45       1.00  20191015 11:35:09.76    ␣
↪ AAPL  3.45  3.55
1   20191015 10:41:22.79      MSFT        2.85       1.00  20191015 12:43:13.73    ␣
↪ MSFT  2.85  2.95
2   20191015 10:41:35.69      MSFT        2.86       1.00  20191015 12:43:13.73    ␣
↪ MSFT  2.85  2.95
3   20191015 11:04:32.55      AAPL        3.47       1.00  20191015 11:35:09.76    ␣
↪ AAPL  3.45  3.55
4   20191015 11:44:35.63      MSFT        2.91       1.00  20191015 12:43:13.73    ␣
↪ MSFT  2.85  2.95
6   20191015 14:24:10.93      MSFT        2.98       1.00  20191015 14:32:11.18    ␣
↪ MSFT  2.90  3.00
7   20191015 15:45:13.41      AAPL        3.60       7.00                   Inv ␣
↪Filtered   nan   nan
8   20191015 15:50:42.53      AAPL        3.58       1.00                   Inv ␣
↪Filtered   nan   nan
9   20191015 15:53:59.60      AAPL        3.56       5.00                   Inv ␣
↪Filtered   nan   nan
5   20191015 12:26:17.68      AAPL        3.55       1.00                   Inv ␣
↪Filtered   nan   nan
```

### 2.2.30 riptable.rt_meta

#### Classes

| | |
|---|---|
| *Doc* | A document object containing metadata about a data object. |
| *Info* | A hierarchically structured container of descriptive information |
| *Item* | Descriptive information for a data object. |

#### Functions

| | |
|---|---|
| *apply_schema*(obj, schema[, doc]) | Apply a schema containing descriptive information recursively to the |
| *doc*(obj) | Return the *Doc* for the object, describing its contents. |
| *info*(obj[, title]) | Return the *Info* for the object, describing its contents. |

**class** riptable.rt_meta.**Doc**(*schema*)

Bases: *riptable.rt_struct.Struct*

A document object containing metadata about a data object.

**Parameters**
**schema** (*dict*) – See *apply_schema()* for more information on the format of the dictionary.

**_descrip**

**_detail**

**_steward**

**_type**

**__repr__**()
Return repr(self).

**__str__**()
Return str(self).

**_as_info**()

**_repr_html_**()

**class** riptable.rt_meta.**Info**

A hierarchically structured container of descriptive information for a data object.

**description: str | None**
The description of the data object.

**Type**
str

**detail**

Detail about the data object.

**Type**
str

**items: List[*Item*] | None**

For a *Struct* or *Dataset*, the items contained within it.

**Type**
list of *Item*

**steward: str | None**

The steward of the data object.

**Type**
str

**title = []**

The title of the data object

**Type**
list

**type: str | None**

The type of the data object.

**Type**
str

**__repr__()**

Return repr(self).

**__str__()**

Return str(self).

**_make_text()**

**_repr_html_()**

class riptable.rt_meta.**Item**(*name*, *type*, *description*, *steward*)

Descriptive information for a data object.

**Parameters**

- **name** (*str*) – The name of the data object.
- **type** (*str*) – The type of the data object.
- **description** (*str*) – A description of the data object.
- **steward** (*str*) – The steward of the data object.

**description: str**

A description of the data object.

**Type**
str

**name: str**

The name of the data object.

**Type**
str

**steward: str**

The steward of the data object.

**Type**
steward

**type: str**

The type of the data object.

**Type**
str

riptable.rt_meta.**apply_schema**(*obj*, *schema*, *doc=True*)

Apply a schema containing descriptive information recursively to the input data object.

The schema should be in the form of a hierarchical dictionary, where for the data object, and recursively for each element it may contain, there is a descriptive dictionary with the following keys and values:

- Type: 'Struct', 'Dataset', 'Multiset', 'FastArray', etc.
- Description: a brief description of the data object
- Steward: the name of the steward for that data object
- Detail: any additional descriptive information
- Contents: if the data object is a *Struct*, *Dataset*, or *Multiset*, a recursively formed dictionary where there is a descriptive dictionary of this form associated with the name of each element contained by the data object.

When the schema is applied to the data object, key/value pairs are set within the `_meta` dictionary attribute of the object and all of its elements, to enable subsequent retrieval of the descriptive information using the *rt_struct.Struct.info()* method or *rt_struct.Struct.doc()* property.

In addition, during the schema application process, the contents and type of each data object is compared to the expectation of the schema, with any differences returned in the form of a dictionary.

**Parameters**

- **obj** (Struct *or* FastArray) – The data object to apply the schema information to.
- **schema** (*dict*) – A descriptive dictionary defining the schema that should apply to the data object and any elements it may contain.
- **doc** (*bool*) – Indicates whether to create and attach a *Doc* to the object, so that the *doc()* method may be run on the object.

**Returns**
**res** – Dictionary of deviations from the schema

**Return type**
dict

**See also:**

*rt_struct.Struct.apply_schema()*

riptable.rt_meta.**doc**(*obj*)

Return the *Doc* for the object, describing its contents.

**Parameters**
**obj** (*Any*) – The object.

**Returns**
**doc** – Returns a *Doc* instance if the object contains documentation metadata, otherwise None.

**Return type**
*Doc*

riptable.rt_meta.**info**(*obj*, *title=None*)

Return the *Info* for the object, describing its contents.

**Parameters**

- **obj** (*Any*) – The object
- **title** (*str*) – The title to give the object, defaults to None.

**Returns**
**info** – Information about `obj`.

**Return type**
*Info*

### 2.2.31 riptable.rt_misc

#### Functions

| | |
|---|---|
| *autocomplete*([hook, jedi, greedy]) | Call rt.autocomplete() to specialize jupyter lab autcomplete output. |
| *build_header_tuples*(headers, span, group) | |
| *jedi_completions*(text, offset) | autocomplete() must be called first. |
| *output_cache_flush*() | used in ipython, jupyter, or spyder |
| *output_cache_none*() | used in ipython, jupyter, or spyder |
| *output_cache_setsize*([cache_size]) | used in ipython, jupyter, or spyder |
| *parse_header_tuples*(header_tups) | |
| *profile_func*(func[, sortby]) | Used to profile a function that has no arguments |
| *sub2ind*(aSize, aPosition) | **MATLAB** |

riptable.rt_misc.**autocomplete**(*hook=True*, *jedi=None*, *greedy=None*)

Call rt.autocomplete() to specialize jupyter lab autcomplete output. arrays, categoricals, datetime, struct, and datasets will be detected. array will be array followed by the dtype.

**Parameters**

- **hook** (*bool, default True*) – set to False to unhook riptable autocomplete
- **jedi** (*bool, default None*) – set to True to set use_jedi in IPython
- **greedy** (*bool, default None*) – set to True to set greedy in IPython for [' autocomplete

### Examples

```
>>> rt.autocomplete(); ds=Dataset({'test':arange(5), 'another':arange(5.0), 'mycat':
↪rt.Cat(arange(5)), 'mystr': arange(5).astype('S')})
Now in jupyter lab type 'ds.<tab>'
```

riptable.rt_misc.**build_header_tuples**(*headers*, *span*, *group*)

riptable.rt_misc.**jedi_completions**(*text*, *offset*)

autocomplete() must be called first. Not used yet. Returns the same completions jedi would.

### Examples

from riptable.rt_misc import jedi_completions st = Struct({'a': 5}) jedi_completions('st', 2)

riptable.rt_misc.**output_cache_flush**()

used in ipython, jupyter, or spyder calling output_cache_flush() will remove object reference in the output cache it is recommended this is called when there are memory concerns

riptable.rt_misc.**output_cache_none**()

used in ipython, jupyter, or spyder sets the terminalInteractiveShell output cache size to none Out[#] will no longer work the Out dictionary will be empty _# will no longer work

riptable.rt_misc.**output_cache_setsize**(*cache_size=100*)

used in ipython, jupyter, or spyder sets the terminalInteractiveShell output cache size to cache_size (100 is the default)

riptable.rt_misc.**parse_header_tuples**(*header_tups*)

riptable.rt_misc.**profile_func**(*func*, *sortby='time'*)

Used to profile a function that has no arguments

### Examples

This will time how long the __repr__ function to print out a dataset

```
>>> import riptable_docdata as rtd
>>> trips = rt.Dataset(rtd.get_bike_trips_data('trips'))
>>> profile_func(trips.__repr__)
```

riptable.rt_misc.**sub2ind**(*aSize*, *aPosition*)

**MATLAB**

sub2ind Linear index from multiple subscripts. sub2ind is used to determine the equivalent single index corresponding to a given set of subscript values.

IND = sub2ind(SIZ,I,J) returns the linear index equivalent to the row and column subscripts in the arrays I and J for a matrix of size SIZ.

IND = sub2ind(SIZ,I1,I2,...,IN) returns the linear index equivalent to the N subscripts in the arrays I1,I2,...,IN for an array of size SIZ.

I1,I2,...,IN must have the same size, and IND will have the same size as I1,I2,...,IN. For an array A, if IND = sub2ind(SIZE(A),I1,...,IN)), then A(IND(k))=A(I1(k),...,IN(k)) for all k.

**PYTHON**

ravel_multi_index(...) ravel_multi_index(multi_index, dims, mode='raise', order='C')

Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.

**Parameters**

- **multi_index** (*tuple of array_like*) – A tuple of integer arrays, one array for each dimension.
- **dims** (*tuple of ints*) – The shape of array into which the indices from `multi_index` apply.
- **mode** (*{'raise', 'wrap', 'clip'}, optional*) – Specifies how out-of-bounds indices are handled. Can specify either one mode or a tuple of modes, one mode per index.
  - 'raise' – raise an error (default)
  - 'wrap' – wrap around
  - 'clip' – clip to the range

  In 'clip' mode, a negative index which would normally wrap will clip to 0 instead.
- **order** (*{'C', 'F'}, optional*) – Determines whether the multi-index should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

**Returns**

**raveled_indices** – An array of indices into the flattened version of an array of dimensions `dims`.

**Return type**

ndarray

**See also:**

`unravel_index`

### Notes

New in version 1.6.0.

### Examples

```
>>> arr = np.array([[3,6,6],[4,5,1]])
>>> np.ravel_multi_index(arr, (7,6))
array([22, 41, 37])
>>> np.ravel_multi_index(arr, (7,6), order='F')
array([31, 41, 13])
>>> np.ravel_multi_index(arr, (4,6), mode='clip')
array([22, 23, 19])
>>> np.ravel_multi_index(arr, (4,4), mode=('clip','wrap'))
array([12, 13, 13])
```

```
>>> np.ravel_multi_index((3,1,4,1), (6,7,8,9))
1621
```

### 2.2.32 riptable.rt_mlutils

#### Functions

| | |
|---|---|
| *normalize_minmax*(arr) | |
| *normalize_zscore*(arr) | |

riptable.rt_mlutils.**normalize_minmax**(*arr*)

riptable.rt_mlutils.**normalize_zscore**(*arr*)

### 2.2.33 riptable.rt_multiset

#### Classes

| | |
|---|---|
| *Multiset* | Multisets contain datasets and/or multisets where all contained dataset have the |

**class** riptable.rt_multiset.**Multiset**(*input_value=None*)

Bases: *riptable.rt_struct.Struct*

Multisets contain datasets and/or multisets where all contained dataset have the same number of rows. Multisets can provide a convenient namespace for closely related datasets, such as those loaded from a single HDF5 file or generated by an aggregation applied to a GroupBy object.

The columns within contained datasets may be displayed in an interleaved way. Example: Assume Jan and Feb are two datasets with 3 columns each:

- Jan: Run1, Run2, Run3
- Feb: Run1, Run2, Run3

A Multiset containing these datasets would display with a multi-line header:

Run1 Run2 Run3

Jan Feb Jan Feb Jan Feb

One can access the Run1 column in the Jan dataset with the syntax: ms.Jan.Run1

#### Examples

```
>>> ds=rt.Dataset({'somenans': [0., 1., 2., nan, 4., 5.], 'morestuff': ['A','B','C',
↪'D','E','F']})
>>> ds2=rt.Dataset({'somenans': [0., 1., nan, 3., 4., 5.], 'morestuff':['H','I','J',
↪'K','L','M']})
>>> ms=rt.Multiset({'test':ds, 'test2':ds2})
>>> ms
      somenans      morestuff
#   test   test2   test   test2
```

(continues on next page)

(continued from previous page)

```
-   ----   -----   ----   -----
0   0.00    0.00   A      H
1   1.00    1.00   B      I
2   2.00     nan   C      J
3    nan    3.00   D      K
4   4.00    4.00   E      L
5   5.00    5.00   F      M
```

```
>>> ms['morestuff']
     morestuff
#   test   test2
-   ----   -----
0   A      H
1   B      I
2   C      J
3   D      K
4   E      L
5   F      M
```

```
>>> ms['test']
#   somenans   morestuff
-   --------   ---------
0       0.00   A
1       1.00   B
2       2.00   C
3        nan   D
4       4.00   E
5       5.00   F
```

```
>>> ms[[2,3],'somenans']
      somenans
#   test   test2
-   ----   -----
0   2.00     nan
1    nan    3.00
```

```
>>> ms[[2,3],'morestuff']
     morestuff
#   test   test2
-   ----   -----
0   C      J
1   D      K
```

```
>>> ms[[2,3],['morestuff','somenans']]
     morestuff       somenans
#   test   test2   test   test2
-   ----   -----   ----   -----
0   C      J       2.00     nan
1   D      K        nan    3.00
```

**property dtypes**

Returns dictionary of dtype for each column.

**Returns**
Dictionary containing the dtype for each column in the Multiset.

**Return type**
dict

`__getitem__`(*index*)

**Parameters**
**index** (*(rowspec, colspec) or colspec*) –

**Return type**
the indexed row(s), cols(s), sub-dataset or single value

### Examples

```
>>> ds=rt.Dataset({'somenans': [0., 1., 2., nan, 4., 5.]})
>>> ds2=rt.Dataset({'somenans': [0., 1., nan, 3., 4., 5.]})
>>> ms=rt.Multiset({'test':ds, 'test2':ds2})
>>> ms[2,:]
      somenans
#   test   test2
-   ----   -----
0   2.00     nan
```

**Raises**

- **IndexError** – When an invalid column name is supplied.
- **TypeError** –

`__len__`()

`__repr__`()
Return repr(self).

`__setitem__`(*index*, *value*)

**Parameters**

- **index** (*colspec*) –
- **value** (*A Dataset or Multiset*) –

**Return type**
None

**Raises**
**IndexError**, **TypeError**, **ValueError** –

`__str__`()
Return str(self).

`_autocomplete`()

`static _build_col_headers`(*rootobject*, *rootdict*)
return a list of lists of ColHeaders

**_build_footers**()

Still testing. TODO: speed up this python loop

**_check_addtype**(*name*, *value*)

called from subclassed Struct when a new item is added

**_copy**(*deep=False*, *rows=None*, *cols=None*, *base_index=0*, *cls=None*)

Bracket indexing that returns a multiset will funnel into this routine.

**Parameters**

- **deep** (*if True, perform a deep copy on column array*) –
- **rows** (*row mask*) –
- **cols** (*column mask*) –
- **base_index** (*used for head/tail slicing*) –
- **cls** (*class of return type, for subclass super() calls*) –
- **False.** (*First argument must be deep. Deep cannnot be set to None. It must be True or*) –

**static _depth_first**(*curobject*, *curdict*, *level*, *returnlist*)

returns the max depth, list of dictionaries

**_init_from_dict**(*dictionary*)

**_last_row_stats**()

**_repr_html_**()

**abs**(**args*, ***kwargs*)

**all**(**args*, ***kwargs*)

For use in boolean contexts: Is it true that for all elements (val) either:

1. val casts to True, or
2. returns True for val.all() or all(val)

**Return type**
bool

**any**(**args*, ***kwargs*)

For use in boolean contexts: Does there exist an element (val) which either:

1. val casts to True, or
2. returns True for val.any() or any(val)

**Return type**
bool

### Examples

```
>>> s=rt.Struct()
>>> s.a=rt.Dataset()
>>> s.any()
False
```

**apply**(**args*, ***kwargs*)

**apply_cols**(**args*, ***kwargs*)

**apply_rows**(**args*, ***kwargs*)

**astype**(**args*, ***kwargs*)

**cascade**(*funcname*, **args*, ***kwargs*)

Depth first calling of functions, often into a Dataset. For each Dataset in the Multiset, the function will be called with the `args` and `kwargs`. The return result is expected to be a Dataset which will then be added back into a new Multiset and returned to the called.

**Parameters**
**funcname** (*string or callable function*) –

**Return type**
*Multiset*

**copy**(*deep=True*)

Returns a shallow or deep copy of the multiset Defaults to a deepy copy.

**Parameters**
**deep** (*bool, default True*) – Set to False for a shallow copy.

**describe**(**args*, ***kwargs*)

**fillna**(**args*, ***kwargs*)

**flatten**(*horizontal=True*, *delimiter='_'*, *dset_col_name='Column'*)

Return a single dataset constructed by concatenating all of the datasets and flattened multisets contained within the multiset. Horizontal flattening will concatenate the datasets horizontally, prepending the dataset name to each dataset's column names. Vertical flattening requires the names and order of columns in each dataset to be identical, adding a single column to the returned dataset containing the name of the dataset from which each row derives.

**Parameters**

- **horizontal** (*bool*) – If True, concatenate the Datasets horizontally, otherwise vertically.
- **delimiter** (*char*) – The character used when joining dataset and column names to create unique names.
- **dset_col_name** (*string*) – For vertical flattening, the name for the column containing dataset names.

**Return type**
*Dataset*

**Raises**
**ValueError** –

**keep**(**args*, ***kwargs*)

**label_fixup**()

Auto scan for which column names can be used as labels in display

**label_set_names**(*listnames*)

Set which column names can be used as labels in display

**make_table**(*display_type*)

Pretty-print code used by infrastructure.

**Parameters**
**display_type** – See rt.rt_enum.DS_DISPLAY_TYPES.

**Returns**
Display object or string.

**max**(**args*, ***kwargs*)

**mean**(**args*, ***kwargs*)

**min**(**args*, ***kwargs*)

**multiget**(*index*, *deep=False*)

Returns a new Multiset representing a one-level sub-sampling of the original.

**Parameters**

- **index** (*An index specification.*) –
- **deep** (*bool, False*) – If set to True will make deep copies

**Return type**
A new Multiset.

**nanmax**(**args*, ***kwargs*)

**nanmean**(**args*, ***kwargs*)

**nanmin**(**args*, ***kwargs*)

**nanstd**(**args*, ***kwargs*)

**nansum**(**args*, ***kwargs*)

**nanvar**(**args*, ***kwargs*)

**pivot**(**args*, ***kwargs*)

**quantile**(**args*, ***kwargs*)

**sort_copy**(**args*, ***kwargs*)

**sort_inplace**(**args*, ***kwargs*)

**std**(**args*, ***kwargs*)

**sum**(**args*, ***kwargs*)

**trim**(**args*, ***kwargs*)

**var**(**args*, ***kwargs*)

### 2.2.34 riptable.rt_numpy

#### Classes

| | |
|---|---|
| *bool_* | The Riptable equivalent of numpy.bool_, with the concept of an invalid added. |
| *bytes_* | The Riptable equivalent of numpy.bytes_, with the concept of an invalid added. |
| *float32* | The Riptable equivalent of numpy.float32, with the concept of an invalid added. |
| *float64* | The Riptable equivalent of numpy.float64, with the concept of an invalid added. |
| *int0* | The Riptable equivalent of numpy.int64, with the concept of an invalid added. |
| *int16* | The Riptable equivalent of numpy.int16, with the concept of an invalid added. |
| *int32* | The Riptable equivalent of numpy.int32, with the concept of an invalid added. |
| *int64* | The Riptable equivalent of numpy.int64, with the concept of an invalid added. |
| *int8* | The Riptable equivalent of numpy.int8, with the concept of an invalid added. |
| *str_* | The Riptable equivalent of numpy.str_, with the concept of an invalid added. |
| *uint0* | The Riptable equivalent of numpy.uint64, with the concept of an invalid added. |
| *uint16* | The Riptable equivalent of numpy.uint16, with the concept of an invalid added. |
| *uint32* | The Riptable equivalent of numpy.uint32, with the concept of an invalid added. |
| *uint64* | The Riptable equivalent of numpy.uint64, with the concept of an invalid added. |
| *uint8* | The Riptable equivalent of numpy.uint8, with the concept of an invalid added. |

#### Functions

| | |
|---|---|
| *_searchsorted*(array, v[, side, sorter]) | |
| *abs*(*args, **kwargs) | This will check for numpy array first and call np.abs |
| *absolute*(*args, **kwargs) | |
| *all*(*args, **kwargs) | |
| *any*(*args, **kwargs) | |
| *arange*(*args, **kwargs) | Return an array of evenly spaced values within a specified interval. |
| *argsort*(*args, **kwargs) | |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| asanyarray(a[, dtype, order]) | |
| asarray(a[, dtype, order]) | |
| assoc_copy(key1, key2, arr) | **param key1**<br>Numpy arrays to match against; all arrays must be same length. |
| assoc_index(key1, key2) | **param key1**<br>Numpy arrays to match against; all arrays must be same length. |
| bincount(*args, **kwargs) | |
| bitcount(a) | Count the number of set (True) bits in an integer or in each integer within an array of |
| bool_to_fancy(arr[, both]) | **param arr**<br>A boolean array of True/False values |
| cat2keys(key1, key2[, filter, ordered, sort_gb, ...]) | Create a `Categorical` from two keys or two `Categorical` objects with all possible unique combinations. |
| ceil(*args, **kwargs) | |
| combine2keys(key1, key2, unique_count1, unique_count2) | **param key1**<br>First index array (int8, int16, int32 or int64). |
| combine_accum1_filter(key1, unique_count1[, filter]) | **param key1**<br>index array (int8, int16, int32 or int64) [must be base 1 -- if base 0, increment by 1] |
| combine_accum2_filter(key1, key2, unique_count1, ...) | **param key1**<br>First index array (int8, int16, int32 or int64). |
| combine_filter(key, filter) | **param key**<br>index array (int8, int16, int32 or int64) |
| concatenate(*args, **kwargs) | |
| crc32c(arr) | Calculate the 32-bit CRC of the data in an array using the Castagnoli polynomial (CRC32C). |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| crc64(arr) | |
| cumsum(*args, **kwargs) | |
| diff(*args, **kwargs) | |
| double(a) | |
| empty(shape[, dtype, order]) | Return a new array of specified shape and type, without initializing entries. |
| empty_like(array[, dtype, order, subok, shape]) | Return a new array with the same shape and type as the specified array, |
| floor(*args, **kwargs) | |
| full(shape, fill_value[, dtype, order]) | Return a new array of a specified shape and type, filled with a specified value. |
| full_like(a, fill_value[, dtype, order, subok, shape]) | Return a full array with the same shape and type as a given array. |
| get_common_dtype(x, y) | Return the dtype of two arrays, or two scalars, or a scalar and an array. |
| get_dtype(val) | Return the dtype of an array, list, or builtin int, float, bool, str, bytes. |
| groupby(list_arrays[, filter, cutoffs, base_index, ...]) | Main routine used to groupby one or more keys. |
| groupbyhash(list_arrays[, hint_size, filter, ...]) | Find unique values in an array using a linear hashing algorithm. |
| groupbylex(list_arrays[, filter, cutoffs, base_index, rec]) | **param list_arrays**<br>A list of numpy arrays to hash on (multikey). All arrays must be the same size. |
| groupbypack(ikey, ncountgroup[, unique_count, cutoffs]) | A routine often called after groupbyhash or groupbylex. |
| hstack(tup[, dtype]) | see numpy hstack |
| interp(x, xp, fp) | One-dimensional or two-dimensional linear interpolation with clipping. |
| interp_extrap(x, xp, fp) | One-dimensional or two-dimensional linear interpolation without clipping. |
| isfinite(*args, **kwargs) | Return True for each finite element, False otherwise. |
| isinf(*args, **kwargs) | Return True for each element that's positive or negative infinity, False otherwise. |
| ismember(a, b[, h, hint_size, base_index]) | The ismember function is meant to mimic the ismember function in MATLab. It takes two sets of data |
| isnan(*args, **kwargs) | Return True for each element that's a NaN (Not a Number), False otherwise. |
| isnanorzero(*args, **kwargs) | Return True for each element that's a NaN (Not a Number) or zero, False otherwise. |
| isnotfinite(*args, **kwargs) | Return True for each non-finite element, False otherwise. |
| isnotinf(*args, **kwargs) | Return True for each element that's not positive or negative infinity, |
| isnotnan(*args, **kwargs) | Return True for each element that's not a NaN (Not a Number), False otherwise. |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| issorted(*args) | Return True if the array is sorted, False otherwise. |
| lexsort(*args, **kwargs) | |
| log(*args, **kwargs) | |
| log10(*args, **kwargs) | |
| logical(a) | |
| makeifirst(key, unique_count[, filter]) | **param key**<br>Index array (int8, int16, int32 or int64). |
| makeilast(key, unique_count[, filter]) | **param key**<br>Index array (int8, int16, int32 or int64). |
| makeinext(key, unique_count) | **param key**<br>index array (int8, int16, int32 or int64) |
| makeiprev(key, unique_count) | **param key**<br>index array (int8, int16, int32 or int64) |
| mask_and(*args, **kwargs) | pass in a tuple or list of boolean arrays to AND together |
| mask_andi(*args, **kwargs) | inplace version: pass in a tuple or list of boolean arrays to AND together |
| mask_andnot(*args, **kwargs) | pass in a tuple or list of boolean arrays to ANDNOT together |
| mask_andnoti(*args, **kwargs) | inplace version: pass in a tuple or list of boolean arrays to ANDNOT together |
| mask_or(*args, **kwargs) | pass in a tuple or list of boolean arrays to OR together |
| mask_ori(*args, **kwargs) | inplace version: pass in a tuple or list of boolean arrays to OR together |
| mask_xor(*args, **kwargs) | pass in a tuple or list of boolean arrays to XOR together |
| mask_xori(*args, **kwargs) | inplace version: pass in a tuple or list of boolean arrays to XOR together |
| max(*args, **kwargs) | |
| maximum(x1, x2, *args, **kwargs) | |
| mean(*args[, filter, dtype]) | Compute the arithmetic mean of the values in the first argument. |
| median(*args, **kwargs) | |
| min(*args, **kwargs) | |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| minimum(x1, x2, *args, **kwargs) | |
| multikeyhash(*args) | Returns 7 arrays to help navigate data. |
| nan_to_num(*args, **kwargs) | arg1: ndarray |
| nan_to_zero(a) | Replace the NaN or invalid values in an array with zeroes. |
| nanargmax(*args, **kwargs) | |
| nanargmin(*args, **kwargs) | |
| nanmax(*args, **kwargs) | |
| nanmean(*args[, filter, dtype]) | Compute the arithmetic mean of the values in the first argument, ignoring NaNs. |
| nanmedian(*args, **kwargs) | |
| nanmin(*args, **kwargs) | |
| nanpercentile(*args, **kwargs) | |
| nanstd(*args[, filter, dtype]) | Compute the standard deviation of the values in the first argument, ignoring NaNs. |
| nansum(*args[, filter, dtype]) | Compute the sum of the values in the first argument, ignoring NaNs. |
| nanvar(*args[, filter, dtype]) | Compute the variance of the values in the first argument, ignoring NaNs. |
| ones(shape[, dtype, order, like]) | Return a new array of the specified shape and data type, filled with ones. |
| ones_like(a[, dtype, order, subok, shape]) | Return an array of ones with the same shape and data type as the specified array. |
| percentile(*args, **kwargs) | |
| putmask(a, mask, values) | This is roughly the equivalent of arr[mask] = arr2[mask]. |
| reindex_fast(index, array) | |
| reshape(*args, **kwargs) | |
| round(*args, **kwargs) | This will check for numpy array first and call np.round |
| searchsorted(a, v[, side, sorter]) | see np.searchsorted |
| single(a) | |
| sort(*args, **kwargs) | |
| sortinplaceindirect(*args, **kwargs) | |
| std(*args[, filter, dtype]) | Compute the standard deviation of the values in the first argument. |
| sum(*args[, filter, dtype]) | Compute the sum of the values in the first argument. |
| tile(arr, reps) | Construct an array by repeating a specified array a specified number of |
| transpose(*args, **kwargs) | |

continues on next page

Table 1 – continued from previous page

| | |
|---|---|
| *trunc*(*args, **kwargs) | |
| *unique32*(list_keys[, hintSize, filter]) | Returns the index location of the first occurence of each key. |
| *var*(*args[, filter, dtype]) | Compute the variance of the values in the first argument. |
| *vstack*(arrlist[, dtype, order]) | **param arrlist**<br>these arrays are considered the columns |
| *where*(condition[, x, y]) | Return a new `FastArray` or `Categorical` with elements from `x` or `y` |
| *zeros*(*args, **kwargs) | Return a new array of the specified shape and data type, filled with zeros. |
| *zeros_like*(a[, dtype, order, subok, shape]) | Return an array of zeros with the same shape and data type as the specified array. |

### Attributes

| | |
|---|---|
| *asanyarray*(a[, dtype, order]) | |
| *asarray*(a[, dtype, order]) | |

**class** riptable.rt_numpy.**bool_**(*value*)

Bases: numpy.bool_

The Riptable equivalent of numpy.bool_, with the concept of an invalid added.

**See also:**

numpy.bool_, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*

#### Examples

```
>>> rt.bool_.inv
False
```

**inv**

**class** riptable.rt_numpy.**bytes_**

Bases: numpy.bytes_

The Riptable equivalent of numpy.bytes_, with the concept of an invalid added.

**See also:**

np.bytes_, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *str_*, *bool_*

### Examples

```
>>> rt.bytes_.inv
b''
```

**inv**

**class** riptable.rt_numpy.**float32**(*value*)

Bases: numpy.float32

The Riptable equivalent of numpy.float32, with the concept of an invalid added.

**See also:**

numpy.float32, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.float32.inv
nan
```

**inv**

**class** riptable.rt_numpy.**float64**(*value*)

Bases: numpy.float64

The Riptable equivalent of numpy.float64, with the concept of an invalid added.

**See also:**

numpy.float64, *float32*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.float64.inv
nan
```

**inv**

**class** riptable.rt_numpy.**int0**(*value*)

Bases: *int64*

The Riptable equivalent of numpy.int64, with the concept of an invalid added.

**See also:**

numpy.int64, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.int64.inv
-9223372036854775808
```

class riptable.rt_numpy.**int16**(*value*)

Bases: numpy.int16

The Riptable equivalent of numpy.int16, with the concept of an invalid added.

**See also:**

numpy.int16, *float32*, *float64*, *int8*, *uint8*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.int16.inv
-32768
```

**inv**

class riptable.rt_numpy.**int32**(*value*)

Bases: numpy.int32

The Riptable equivalent of numpy.int32, with the concept of an invalid added.

**See also:**

numpy.int32, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.int32.inv
-2147483648
```

**inv**

class riptable.rt_numpy.**int64**(*value*)

Bases: numpy.int64

The Riptable equivalent of numpy.int64, with the concept of an invalid added.

**See also:**

numpy.int64, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.int64.inv
-9223372036854775808
```

**inv**

**class** riptable.rt_numpy.**int8**(*value*)

Bases: numpy.int8

The Riptable equivalent of numpy.int8, with the concept of an invalid added.

**See also:**

numpy.int8, *float32*, *float64*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.int8.inv
-128
```

**inv**

**class** riptable.rt_numpy.**str_**

Bases: numpy.str_

The Riptable equivalent of numpy.str_, with the concept of an invalid added.

**See also:**

numpy.str_, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *bool_*

### Examples

```
>>> rt.str_.inv
''
```

**inv**

**class** riptable.rt_numpy.**uint0**(*value*)

Bases: *uint64*

The Riptable equivalent of numpy.uint64, with the concept of an invalid added.

**See also:**

numpy.uint64, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.uint64.inv
18446744073709551615
```

class riptable.rt_numpy.**uint16**(*value*)

Bases: numpy.uint16

The Riptable equivalent of numpy.uint16, with the concept of an invalid added.

**See also:**

numpy.uint16, *float32*, *float64*, *int8*, *uint8*, *int16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.uint16.inv
65535
```

**inv**

class riptable.rt_numpy.**uint32**(*value*)

Bases: numpy.uint32

The Riptable equivalent of numpy.uint32, with the concept of an invalid added.

**See also:**

numpy.uint32, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.uint32.inv
4294967295
```

**inv**

class riptable.rt_numpy.**uint64**(*value*)

Bases: numpy.uint64

The Riptable equivalent of numpy.uint64, with the concept of an invalid added.

**See also:**

numpy.uint64, *float32*, *float64*, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.uint64.inv
18446744073709551615
```

**inv**

class riptable.rt_numpy.**uint8**(*value*)

Bases: numpy.uint8

The Riptable equivalent of numpy.uint8, with the concept of an invalid added.

**See also:**

numpy.uint8, *float32*, *float64*, *int8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *bytes_*, *str_*, *bool_*

### Examples

```
>>> rt.uint8.inv
255
```

**inv**

riptable.rt_numpy.**_searchsorted**(*array*, *v*, *side='left'*, *sorter=None*)

riptable.rt_numpy.**abs**(**args*, ***kwargs*)

This will check for numpy array first and call np.abs

riptable.rt_numpy.**absolute**(**args*, ***kwargs*)

riptable.rt_numpy.**all**(**args*, ***kwargs*)

riptable.rt_numpy.**any**(**args*, ***kwargs*)

riptable.rt_numpy.**arange**(**args*, ***kwargs*)

Return an array of evenly spaced values within a specified interval.

The half-open interval includes `start` but excludes `stop`: `[start, stop)`.

For integer arguments the function is roughly equivalent to the Python built-in range, but returns a `FastArray` rather than a range instance.

When using a non-integer step, such as 0.1, it's often better to use numpy.linspace().

For additional warnings, see numpy.arange().

**Parameters**

- **start** (*int or float, default 0*) – Start of interval. The interval includes this value.
- **stop** (*int or float*) – End of interval. The interval does not include this value, except in some cases where `step` is not an integer and floating point round-off affects the length of the output.
- **step** (*int or float, default 1*) – Spacing between values. For any output `out`, this is the distance between two adjacent values: `out[i+1] - out[i]`. If `step` is specified as a positional argument, `start` must also be given.

- **dtype** (*str or NumPy dtype or Riptable dtype, optional*) – The type of the output array. If `dtype` is not given, the data type is inferred from the other input arguments.
- **like** (*array_like, optional*) – Reference object to allow the creation of arrays that are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

**Returns**

A `FastArray` of evenly spaced numbers within the specified interval. For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of the output being greater than `stop`.

**Return type**

`FastArray`

**See also:**

`numpy.arange`, `riptable.ones`, `riptable.ones_like`, `riptable.zeros`, `riptable.zeros_like`, `riptable.empty`, `riptable.empty_like`, `riptable.full`, `riptable.arange`, `Categorical.full`

### Examples

```
>>> rt.arange(3)
FastArray([0, 1, 2])
```

```
>>> rt.arange(3.0)
FastArray([ 0.,  1.,  2.])
```

```
>>> rt.arange(3, 7)
FastArray([3, 4, 5, 6])
```

```
>>> rt.arange(3, 7, 2)
FastArray([3, 5])
```

riptable.rt_numpy.**argsort**(**args*, ***kwargs*)

riptable.rt_numpy.**asanyarray**(*a*, *dtype=None*, *order=None*)

riptable.rt_numpy.**asarray**(*a*, *dtype=None*, *order=None*)

riptable.rt_numpy.**assoc_copy**(*key1*, *key2*, *arr*)

**Parameters**

- **key1** (*ndarray / list thereof or a Dataset*) – Numpy arrays to match against; all arrays must be same length.
- **key2** (*ndarray / list thereof or a Dataset*) – Numpy arrays that will be matched with `key1`; all arrays must be same length.
- **arr** (*ndarray / Dataset*) – An array or Dataset the same length as key2 arrays which will be mapped to the size of `key1` In the case of an array, the output will be cast to FastArray to accomodate support of fancy-indexing with sentinel values

**Returns**
A new array the same length as `key1` arrays which has mapped the input `arr` from `key2` to `key1` the array's dtype will match the dtype of the input array (3rd parameter). However, outputs will be FastArrays when the input array is a numpy arrays such that fancy indexing with sentinels works correctly.

**Return type**
array_like

### Examples

```
>>> np.random.seed(12345)
>>> ds=Dataset({'time': rt.arange(200_000_000.0)})
>>> ds.data = np.random.randint(7, size=200_000_000)
>>> ds.symbol = rt.Cat(1 + rt.arange(200_000_000) % 7, ['AAPL','AMZN', 'FB', 'GOOG',
↪ 'IBM','MSFT','UBER'])
>>> dsa = rt.Dataset({'data': rt.repeat(rt.arange(7), 7), 'symbol': rt.tile(rt.
↪FastArray(['AAPL','AMZN', 'FB', 'GOOG', 'IBM','MSFT','UBER']), 7), 'time': 48 -␣
↪rt.arange(49.0)})
>>> rt.assoc_copy([ds.symbol, ds.data], [dsa.symbol, dsa.data], dsa.time)
FastArray([13.,  5., 46., ...,  5., 11., 24.])
```

riptable.rt_numpy.**assoc_index**(*key1*, *key2*)

**Parameters**

- **key1** (*ndarray / list thereof or a Dataset*) – Numpy arrays to match against; all arrays must be same length.
- **key2** (*ndarray / list thereof or a Dataset*) – Numpy arrays that will be matched with `key1`; all arrays must be same length.

**Returns**
**fancy_index** – Fancy index where the index of `key2` is matched against `key1`; if there was no match, the minimum integer (aka sentinel) is the index value.

**Return type**
ndarray of ints

### Examples

```
>>> np.random.seed(12345)
>>> ds = rt.Dataset({'time': rt.arange(200_000_000.0)})
>>> ds.data = np.random.randint(7, size=200_000_000)
>>> ds.symbol = rt.Cat(1 + rt.arange(200_000_000) % 7, ['AAPL','AMZN', 'FB', 'GOOG',
↪ 'IBM','MSFT','UBER'])
>>> dsa = rt.Dataset({'data': rt.repeat(rt.arange(7), 7), 'symbol': rt.tile(rt.
↪FastArray(['AAPL','AMZN', 'FB', 'GOOG', 'IBM','MSFT','UBER']), 7)})
>>> rt.assoc_index([ds.symbol, ds.data], [dsa.symbol, dsa.data])
FastArray([35, 43,  2, ..., 43, 37, 24])
```

riptable.rt_numpy.**bincount**(**args*, ***kwargs*)

riptable.rt_numpy.**bitcount**(*a*)

Count the number of set (True) bits in an integer or in each integer within an array of integers. This operation is also known as population count or Hamming weight.

**Parameters**

**a** (*int or sequence or numpy.array*) – A Python integer or a sequence of integers or a numpy integer array.

**Returns**

If the input is Python int the return is int. If the input is sequence or numpy array the return is a numpy array with dtype int8.

**Return type**

int or numpy.array

### Examples

```
>>> arr = rt.FastArray([741858, 77285, 916765, 395393, 347556, 896425, 921598,␣
↪86398])
>>> rt.bitcount(arr)
FastArray([10, 10, 14,  5,  9, 12, 14, 10], dtype=int8)
```

riptable.rt_numpy.**bool_to_fancy**(*arr*, *both=False*)

**Parameters**

- **arr** (*ndarray of bools*) – A boolean array of True/False values
- **both** (*bool*) – Controls whether to return a the True and False elements in `arr`. Defaults to False.

**Returns**

- **fancy_index** (*ndarray of bools*) – Fancy index array of where the True values are. If `both` is True, there are two fancy index array sections: The first array slice is where the True values are; The second array slice is where the False values are. The True count is returned.
- **true_count** (*int, optional*) – When `both` is True, this value is returned to indicate how many True values were in `arr`; this is then used to slice `fancy_index` into two slices indicating where the True and False values are, respectively, within `arr`.

### Notes

runs in parallel

### Examples

```
>>> np.random.seed(12345)
>>> bools = np.random.randint(2, size=20, dtype=np.int8).astype(bool)
>>> rt.bool_to_fancy(bools)
FastArray([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 12, 15, 17, 18, 19])
```

Setting the `both` parameter to True causes the function to return an array containing the indices of the True values in `arr` followed by the indices of the False values, along with the number (count) of True values. This count can be used to slice the returned array if you want just the True indices and False indices.

```
>>> fancy_index, true_count = rt.bool_to_fancy(bools, both=True)
>>> fancy_index[:true_count], fancy_index[true_count:]
(FastArray([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 12, 15, 17, 18, 19]),␣
↪FastArray([ 0, 11, 13, 14, 16]))
```

riptable.rt_numpy.**cat2keys**(*key1*, *key2*, *filter=None*, *ordered=True*, *sort_gb=False*, *invalid=False*, *fuse=False*)

Create a `Categorical` from two keys or two `Categorical` objects with all possible unique combinations.

### Notes

Code assumes Categoricals are base 1.

**Parameters**

- **key1** (*Categorical, ndarray, or list of ndarray*) – If a list of arrays is passed for this parameter, all arrays in the list must have the same length.
- **key2** (*Categorical, ndarray, or list of ndarray*) – If a list of arrays is passed for this parameter, all arrays in the list must have the same length.
- **filter** (*ndarray of bool, optional*) – only valid when invalid is set to True
- **ordered** (*bool, default True*) – only applies when key1 or key2 is not a categorical
- **sort_gb** (*bool, default False*) – only applies when key1 or key2 is not a categorical
- **invalid** (*bool, default False*) – Specifies whether or not to insert the invalid when creating the n x m unique matrix.
- **fuse** (*bool, default False*) – When True, forces the resulting categorical to have 2 keys, one for rows, and one for columns.

**Returns**

A multikey categorical that has at least 2 keys.

**Return type**

*Categorical*

### Examples

The following examples demonstrate using cat2keys on keys as lists and arrays, lists of arrays, and Categoricals. In each of the examples, you can determine the unique combinations by zipping the same position of each of the values of the category dictionary.

Creating a MultiKey Categorical from two lists of equal length.

```
>>> rt.cat2keys(list('abc'), list('xyz'))
Categorical([(a, x), (b, y), (c, z)]) Length: 3
  FastArray([1, 5, 9], dtype=int64) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'c', b'a', b'b', b'c', b'a', b'b', b'c'], dtype=
↪'|S1'), 'key_01': FastArray([b'x', b'x', b'x', b'y', b'y', b'y', b'z', b'z', b'z
↪'], dtype='|S1')} Unique count: 9
```

```
>>> rt.cat2keys(np.array(list('abc')), np.array(list('xyz')))
Categorical([(a, x), (b, y), (c, z)]) Length: 3
  FastArray([1, 5, 9], dtype=int64) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'c', b'a', b'b', b'c', b'a', b'b', b'c'], dtype=
↪'|S1'), 'key_01': FastArray([b'x', b'x', b'x', b'y', b'y', b'y', b'z', b'z', b'z
↪'], dtype='|S1')} Unique count: 9
```

```
>>> key1, key2 = [rt.FA(list('abc')), rt.FA(list('def'))], [rt.FA(list('uvw')), rt.
↪FA(list('xyz'))]
>>> rt.cat2keys(key1, key2)
Categorical([(a, d, u, x), (b, e, v, y), (c, f, w, z)]) Length: 3
  FastArray([1, 5, 9], dtype=int64) Base Index: 1
  {'key_0': FastArray([b'a', b'b', b'c', b'a', b'b', b'c', b'a', b'b', b'c'], dtype=
↪'|S1'), 'key_1': FastArray([b'd', b'e', b'f', b'd', b'e', b'f', b'd', b'e', b'f'],
↪ dtype='|S1'), 'key_01': FastArray([b'u', b'u', b'u', b'v', b'v', b'v', b'w', b'w
↪', b'w'], dtype='|S1'), 'key_11': FastArray([b'x', b'x', b'x', b'y', b'y', b'y', b
↪'z', b'z', b'z'], dtype='|S1')} Unique count: 9
```

```
>>> cat.category_dict
{'key_0': FastArray([b'a', b'b', b'c', b'a', b'b', b'c', b'a', b'b', b'c'],
           dtype='|S1'),
 'key_1': FastArray([b'd', b'e', b'f', b'd', b'e', b'f', b'd', b'e', b'f'],
           dtype='|S1'),
 'key_01': FastArray([b'u', b'u', b'u', b'v', b'v', b'v', b'w', b'w', b'w'],
           dtype='|S1'),
 'key_11': FastArray([b'x', b'x', b'x', b'y', b'y', b'y', b'z', b'z', b'z'],
           dtype='|S1')}
```

riptable.rt_numpy.**ceil**(**args*, ***kwargs*)

riptable.rt_numpy.**combine2keys**(*key1*, *key2*, *unique_count1*, *unique_count2*, *filter=None*)

**Parameters**

- **key1** (*ndarray of ints*) – First index array (int8, int16, int32 or int64).
- **key2** (*ndarray of ints*) – Second index array (int8, int16, int32 or int64).
- **unique_count1** (*int*) – Number of unique values in key1 (often returned by groupbyhash/groupbylex).

- **unique_count2** (*int*) – Number of unique values in key2.
- **filter** (*ndarray of bools, optional*) – Boolean array with same length as key1 array, defaults to None.

**Returns**

- *TWO ARRAYs (iKey (for 2 dims), nCountGroup)*
- *bin is a 1 based index array with each False value setting the index to 0*
- *nCountGroup is INT32 array with size = to (unique_count1 + 1)*(unique_count2 + 1)*

riptable.rt_numpy.**combine_accum1_filter**(*key1*, *unique_count1*, *filter=None*)

**Parameters**

- **key1** (*ndarray of ints*) – index array (int8, int16, int32 or int64) [must be base 1 – if base 0, increment by 1] often referred to as iKey or the bin array for categoricals
- **unique_count1** (*int*) – Maximum number of uniques in key1 array.
- **filter** (*ndarray of bool, optional*) – Boolean array same length as key1 array, defaults to None.

**Returns**

- **iKey** (*a new 1 based index array with each False value setting the index to 0*) – iKey dtype will match the dtype in Arg1
- **iFirstKey** (*an INT32 array, the fixup for first since some bins may have been removed*)
- **unique_count** (INT32 and is the new unique_count1. It is the length of iFirstKey)

### Example

```
>>> a = rt.arange(20) % 10
>>> b = a.astype('S')
>>> c = rt.Cat(b)
>>> rt.combine_accum1_filter(c, c.unique_count, rt.logical(rt.arange(20) % 2))
{'iKey': FastArray([0, 1, 0, 2, 0, 3, 0, 4, 0, 5, 0, 1, 0, 2, 0, 3, 0, 4, 0, 5],
           dtype=int8),
 'iFirstKey': FastArray([1, 3, 5, 7, 9]),
 'unique_count': 5}
```

riptable.rt_numpy.**combine_accum2_filter**(*key1*, *key2*, *unique_count1*, *unique_count2*, *filter=None*)

**Parameters**

- **key1** (*ndarray of ints*) – First index array (int8, int16, int32 or int64).
- **key2** (*ndarray of ints*) – Second index array (int8, int16, int32 or int64).
- **unique_count1** (*int*) – Maximum number of unique values in key1.
- **unique_count2** (*int*) – Maximum number of unique values in key2.
- **filter** (*ndarray of bools, optional*) – Boolean array with same length as key1 array, defaults to None.

**Returns**

- *TWO ARRAYs (iKey (for 2 dims), nCountGroup)*

- *bin is a 1 based index array with each False value setting the index to 0*
- *nCountGroup is INT32 array with size = to (unique_count1 + 1)*(unique_count2 + 1)*

riptable.rt_numpy.**combine_filter**(*key*, *filter*)

**Parameters**

- **key** (*ndarray of ints*) – index array (int8, int16, int32 or int64)
- **filter** (*ndarray of bools*) – Boolean array same length as key.

**Returns**

1 based index array with each False value setting the index to 0. The equivalent function is `return index*filter` or `np.where(filter, index, 0)`.

**Return type**

ndarray of ints

### Notes

This routine can run in parallel.

riptable.rt_numpy.**concatenate**(**args*, ***kwargs*)

riptable.rt_numpy.**crc32c**(*arr*)

Calculate the 32-bit CRC of the data in an array using the Castagnoli polynomial (CRC32C).

This function does not consider the array's shape or strides when calculating the CRC, it simply calculates the CRC value over the entire buffer described by the array.

**Parameters**

**arr** –

**Returns**

The 32-bit CRC value calculated from the array data.

**Return type**

int

### Notes

**TODO: Warn when the array has non-default striding, as that is not currently respected by**
the implementation of this function.

riptable.rt_numpy.**crc64**(*arr*)

riptable.rt_numpy.**cumsum**(**args*, ***kwargs*)

riptable.rt_numpy.**diff**(**args*, ***kwargs*)

riptable.rt_numpy.**double**(*a*)

riptable.rt_numpy.**empty**(*shape*, *dtype=float*, *order='C'*)

Return a new array of specified shape and type, without initializing entries.

**Parameters**

- **shape** (*int or tuple of int*) – Shape of the empty array, e.g., `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (str or NumPy dtype or Riptable dtype, default `numpy.float64`) – The desired data type for the array.
- **order** (*{'C', 'F'}, default 'C'*) – Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**
A new `FastArray` of uninitialized (arbitrary) data of the specified shape and type.

**Return type**
`FastArray`

**See also:**

`riptable.empty_like`, `riptable.ones`, `riptable.ones_like`, `riptable.zeros`, `riptable.zeros_like`, `riptable.empty`, `riptable.full`, `Categorical.full`

### Notes

Unlike *zeros*, *empty* doesn't set the array values to zero, so it may be marginally faster. On the other hand, it requires the user to manually set all the values in the array, so it should be used with caution.

### Examples

```
>>> rt.empty(5)
FastArray([0.  , 0.25, 0.5 , 0.75, 1.  ])  # uninitialized
```

```
>>> rt.empty(5, dtype = int)
FastArray([80288976,        0,        0,        0,        1])  # uninitialized
```

riptable.rt_numpy.**empty_like**(*array*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return a new array with the same shape and type as the specified array, without initializing entries.

**Parameters**

- **array** (*array*) – The shape and data type of `array` define the same attributes of the returned array. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str or NumPy dtype or Riptable dtype, optional*) – Overrides the data type of the result.
- **order** (*{'K', C', 'F', or 'A'}, default 'K'*) – Overrides the memory layout of the result. 'K' (the default) means match the layout of `array` as closely as possible. 'C' means row-major (C-style); 'F' means column-major (Fortran-style); 'A' means 'F' if `array` is Fortran-contiguous, 'C' otherwise.
- **subok** (*bool, default True*) – If True (the default), then the newly created array will use the sub-class type of `array`, otherwise it will be a base-class array.
- **shape** (*int or sequence of ints, optional*) – Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, it will try to keep the same order; otherwise, order='C' is implied. Note that although multi-dimensional arrays are

technically supported by Riptable, you may get unexpected results when working with them.

**Returns**
A new `FastArray` of uninitialized (arbitrary) data with the same shape and type as `array`.

**Return type**
`FastArray`

**See also:**

`riptable.empty`, `riptable.ones`, `riptable.ones_like`, `riptable.zeros`, `riptable.zeros_like`, `riptable.full`, `Categorical.full`

### Examples

```
>>> a = rt.FastArray([1, 2, 3, 4])
>>> rt.empty_like(a)
FastArray([ 1814376192,  1668069856, -1994737310,   746250422])  # uninitialized
```

```
>>> rt.empty_like(a, dtype = float)
FastArray([0.25, 0.5 , 0.75, 1.  ])  # uninitialized
```

`riptable.rt_numpy.floor(*args, **kwargs)`

`riptable.rt_numpy.full(shape, fill_value, dtype=None, order='C')`

Return a new array of a specified shape and type, filled with a specified value.

**Parameters**

- **shape** (*int or sequence of int*) – Shape of the new array, e.g., `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **fill_value** (*scalar or array*) – Fill value. For 1-dimensional arrays, only scalar values are accepted.
- **dtype** (*str or NumPy dtype or Riptable dtype, optional*) – The desired data type for the array. The default is the data type that would result from creating a `FastArray` with the specified `fill_value`: `rt.FastArray(fill_value).dtype`.
- **order** (*{'C', 'F'}, default 'C'*) – Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**
A new `FastArray` of the specified shape and type, filled with the specified value.

**Return type**
`FastArray`

**See also:**

`Categorical.full`, `riptable.ones`, `riptable.ones_like`, `riptable.zeros`, `riptable.zeros_like`, `riptable.empty`, `riptable.empty_like`

### Examples

```
>>> rt.full(5, 2)
FastArray([2, 2, 2, 2, 2])
```

```
>>> rt.full(5, 2.0)
FastArray([2., 2., 2., 2., 2.])
```

Specify a data type:

```
>>> rt.full(5, 2, dtype = float)
FastArray([2., 2., 2., 2., 2.])
```

riptable.rt_numpy.**full_like**(*a*, *fill_value*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return a full array with the same shape and type as a given array.

**Parameters**

- **a** (*array*) – The shape and data type of `a` define the same attributes of the returned array. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **fill_value** (*scalar or array_like*) – Fill value.
- **dtype** (*str or NumPy dtype or Riptable dtype, optional*) – Overrides the data type of the result.
- **order** (*{'C', 'F', 'A', or 'K'}, default 'K'*) – Overrides the memory layout of the result. 'C' means row-major (C-style), 'F' means column-major (Fortran-style), 'A' means 'F' if `a` is Fortran-contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*bool, default True*) – If True (the default), then the newly created array will use the sub-class type of `a`, otherwise it will be a base-class array.
- **shape** (*int or sequence of int, optional*) – Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, it will try to keep the same order; otherwise, order='C' is implied. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.

**Returns**

A `FastArray` with the same shape and data type as the specified array, filled with `fill_value`.

**Return type**

`FastArray`

**See also:**

`riptable.ones`, `riptable.zeros`, `riptable.zeros_like`, `riptable.empty`, `riptable.empty_like`, `riptable.full`

### Examples

```
>>> a = rt.FastArray([1, 2, 3, 4])
>>> rt.full_like(a, 9)
FastArray([9, 9, 9, 9])
```

```
>>> rt.ones_like(a, dtype = float)
FastArray([1., 1., 1., 1.])
```

riptable.rt_numpy.**get_common_dtype**(*x*, *y*)

Return the dtype of two arrays, or two scalars, or a scalar and an array.

Will dtype normal python ints to int32 or int64 (not int8 or int16). Used in where, put, take, putmask.

**Parameters**

- **x** (*scalar or array_like*) – A scalar and/or array to find the common dtype of.
- **y** (*scalar or array_like*) – A scalar and/or array to find the common dtype of.

**Returns**

The data type (dtype) common to both `x` and `y`. If the objects don't have exactly the same dtype, returns the dtype which both types could be implicitly coerced to.

**Return type**

data-type

### Examples

```
>>> get_common_type('test','hello')
dtype('<U5')
```

```
>>> get_common_type(14,'hello')
dtype('<U16')
```

```
>>> get_common_type(14,b'hello')
dtype('<S16')
```

```
>>> get_common_type(14, 17)
dtype('int32')
```

```
>>> get_common_type(arange(10), arange(10.0))
dtype('float64')
```

```
>>> get_common_type(arange(10).astype(bool), True)
dtype('bool')
```

riptable.rt_numpy.**get_dtype**(*val*)

Return the dtype of an array, list, or builtin int, float, bool, str, bytes.

**Parameters**

**val** – An object to get the dtype of.

**Returns**

The data-type (dtype) for `val` (if it has a dtype), or a dtype compatible with `val`.

**Return type**
data-type

### Notes

if a python integer, will use int32 or int64 (never uint) for a python float, always returns float64 for a string, will return U or S with size

TODO: consider pushing down into C++

### Examples

```
>>> get_dtype(10)
dtype('int32')
```

```
>>> get_dtype(123.45)
dtype('float64')
```

```
>>> get_dtype('hello')
dtype('<U5')
```

```
>>> get_dtype(b'hello')
dtype('S5')
```

riptable.rt_numpy.**groupby**(*list_arrays*, *filter=None*, *cutoffs=None*, *base_index=1*, *lex=False*, *rec=False*, *pack=False*, *hint_size=0*)

Main routine used to groupby one or more keys.

**Parameters**

- **list_arrays** (*list of ndarray*) – A list of numpy arrays to hash on (multikey). All arrays must be the same size.
- **filter** (*ndarray of bool, optional*) – A boolean array the same length as the arrays in `list_arrays` used to pre-filter the input data before passing it to the grouping algorithm, defaults to None.
- **cutoffs** (*ndarray, optional*) – INT64 array of cutoffs
- **base_index** (*int*) –
- **lex** (*defaults to False. if False will call groupbyhash*) – If set to true will call groupbylex
- **rec** (*bool*) – When set to true, a record array is created, and then the data is sorted. A record array is faster, but may not produce a true lexicographical sort. Defaults to False. Only applicable when `lex` is True.
- **pack** (*bool*) – Set to True to return iGroup, iFirstGroup, nCountGroup also; defaults to False. This is only meaningful when using hash-based grouping – when `lex` is True, the sorting-based grouping always computes and returns this information.
- **hint_size** (*int*) – An integer hint if the number of unique keys is known in advance, defaults to zero. Only applicable when using hash-based grouping (i.e. `lex` is False).

### Notes

Ends up calling groupbyhash or groupbylex.

**See also:**

*groupbyhash*, *groupbylex*

riptable.rt_numpy.**groupbyhash**(*list_arrays*, *hint_size=0*, *filter=None*, *hash_mode=2*, *cutoffs=None*, *pack=False*)

Find unique values in an array using a linear hashing algorithm.

Find unique values in an array using a linear hashing algorithm; it will then bin each group according to first appearance. The zero bin is reserved for anything filtered out.

**Parameters**

- **list_arrays** (*ndarray or list of ndarray*) – a single numpy array or a list of numpy arrays to hash on (multikey) - all arrays must be the same size
- **hint_size** (*int, optional*) – An integer hint if the number of unique keys is known in advance, defaults to zero.
- **filter** (*ndarray of bool, optional*) – A boolean filter to pre-filter the values on, defaults to None.
- **hash_mode** (*int*) – Setting for controlling the hashing mode; defaults to 2. Users generally should not override the default value of this parameter.
- **cutoffs** (*ndarray, optional*) – An int64 array of cutoffs, defaults to None.
- **pack** (*bool*) – Set to True to return iGroup, iFirstGroup, nCountGroup also; defaults to False.

**Returns**

- *A dictionary of 3 arrays*
- **'iKey'** (*array size is same as multikey, the unique key for which this row in multikey belongs*)
- **'iFirstKey'** (*array size is same as unique keys, index into the first row for that unique key*)
- **'unique_count'** (*number of uniques (not including the zero bin)*)

### Examples

```
>>> np.random.seed(12345)
>>> c = np.random.randint(0, 8000, 2_000_000)
>>> rt.groupbyhash(c)
{'iKey': FastArray([   1,    2,    3, ..., 6061, 7889, 3002]),
 'iFirstKey': FastArray([    0,     1,     2, ..., 67072, 67697, 68250]),
 'unique_count': 8000,
 'iGroup': None,
 'iFirstGroup': None,
 'nCountGroup': None}
```

The 'pack' parameter can be overridden to True to calculate additional information about the relationship between elements in the input array and their group. Note this information is the same type of information `groupbylex` returns by default.

```
>>> rt.groupbyhash(c, pack=True)
{'iKey': FastArray([1, 2, 2, ..., 4, 6, 1]),
 'iFirstKey': FastArray([ 0,  1,  3,  4,  6, 14, 18, 20]),
 'unique_count': 8,
 'iGroup': FastArray([   0,    9,   21, ..., 9988, 9991, 9992]),
 'iFirstGroup': FastArray([   0,    0, 1213, 2465, 3761, 4987, 6239, 7522, 8797]),
 'nCountGroup': FastArray([   0, 1213, 1252, 1296, 1226, 1252, 1283, 1275, 1203])}
```

The output from `groupbyhash` is useful as an input to `rc.BinCount`:

```
>>> x = rt.groupbyhash(c)
>>> rc.BinCount(x['iKey'], x['unique_count'] + 1)
FastArray([  0, 251, 262, ..., 239, 217, 246])
```

A filter (boolean array) can be passed to `groupbyhash`; this causes `groupbyhash` to only operate on the elements of the input array where the filter has a corresponding True value.

```
>>> f = (c % 3).astype(bool)
>>> rt.groupbyhash(c, filter=f)
{'iKey': FastArray([   0,    1,    2, ...,    0, 5250, 1973]),
 'iFirstKey': FastArray([    1,     2,     3, ..., 54422, 58655, 68250]),
 'unique_count': 5333,
 'iGroup': None,
 'iFirstGroup': None,
 'nCountGroup': None}
```

The `groupbyhash` function can also operate on multikeys (tuple keys).

```
>>> d = np.random.randint(0, 8000, 2_000_000)
>>> rt.groupbyhash([c, d])
{'iKey': FastArray([      1,       2,       3, ..., 1968854, 1968855, 1968856]),
 'iFirstKey': FastArray([      0,       1,       2, ..., 1999997, 1999998,␣
↪1999999]),
 'unique_count': 1968856,
 'iGroup': None,
 'iFirstGroup': None,
 'nCountGroup': None}
```

riptable.rt_numpy.**groupbylex**(*list_arrays*, *filter=None*, *cutoffs=None*, *base_index=1*, *rec=False*)

**Parameters**

- **list_arrays** (*ndarray or list of ndarray*) – A list of numpy arrays to hash on (multikey). All arrays must be the same size.
- **filter** (*ndarray of bool, optional*) – A boolean array of true/false filters, defaults to None.
- **cutoffs** (*ndarray, optional*) – INT64 array of cutoffs
- **base_index** (*int*) –
- **rec** (*bool*) – When set to true, a record array is created, and then the data is sorted. A record array is faster, but may not produce a true lexicographical sort. Defaults to False.

**Returns**

- *A dict of 6 numpy arrays*

- **iKey** (*array size is same as multikey, the unique key for which this row in multikey belongs*)
- **iFirstKey** (*array size is same as unique keys, index into the first row for that unique key*)
- **unique_count** (*number of uniques*)
- **iGroup** (*result from lexsort (fancy index sort of list_arrays)*)
- **iFirstGroup** (*array size is same as unique keys + 1: offset into iGroup*)
- **nCountGroup** (*array size is same as unique keys + 1: length of slice in iGroup*)

### Examples

```
>>> a = rt.arange(100).astype('S')
>>> f = rt.logical(rt.arange(100) % 3)
>>> rt.groupbylex([a], filter=f)
{'iKey': FastArray([ 0,  1,  9,  0, 23, 31,  0, 45, 53,  0,  2,  3,  0,  4,  5,  0,
        6,  7,  0,  8, 10,  0, 11, 12,  0, 13, 14,  0, 15, 16,  0, 17,
       18,  0, 19, 20,  0, 21, 22,  0, 24, 25,  0, 26, 27,  0, 28, 29,
        0, 30, 32,  0, 33, 34,  0, 35, 36,  0, 37, 38,  0, 39, 40,  0,
       41, 42,  0, 43, 44,  0, 46, 47,  0, 48, 49,  0, 50, 51,  0, 52,
       54,  0, 55, 56,  0, 57, 58,  0, 59, 60,  0, 61, 62,  0, 63, 64,
        0, 65, 66,  0]),
 'iFirstKey': FastArray([ 1, 10, 11, 13, 14, 16, 17, 19,  2, 20, 22, 23, 25, 26, 28,
↪ 29,
       31, 32, 34, 35, 37, 38,  4, 40, 41, 43, 44, 46, 47, 49,  5, 50,
       52, 53, 55, 56, 58, 59, 61, 62, 64, 65, 67, 68,  7, 70, 71, 73,
       74, 76, 77, 79,  8, 80, 82, 83, 85, 86, 88, 89, 91, 92, 94, 95,
       97, 98]),
 'unique_count': 66,
 'iGroup': FastArray([ 1, 10, 11, 13, 14, 16, 17, 19,  2, 20, 22, 23, 25, 26, 28,␣
↪29,
       31, 32, 34, 35, 37, 38,  4, 40, 41, 43, 44, 46, 47, 49,  5, 50,
       52, 53, 55, 56, 58, 59, 61, 62, 64, 65, 67, 68,  7, 70, 71, 73,
       74, 76, 77, 79,  8, 80, 82, 83, 85, 86, 88, 89, 91, 92, 94, 95,
       97, 98]),
 'iFirstGroup': FastArray([66,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,␣
↪13, 14,
       15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
       31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
       47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
       63, 64, 65]),
 'nCountGroup': FastArray([34,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,␣
↪1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1,  1])}
```

riptable.rt_numpy.**groupbypack**(*ikey*, *ncountgroup*, *unique_count=None*, *cutoffs=None*)

A routine often called after groupbyhash or groupbylex. Operates on binned integer arrays only (int8, int16, int32, or int64).

**Parameters**

- **ikey** (*ndarray of ints*) – iKey from groupbyhash or groupbylex
- **ncountgroup** (*ndarray of ints, optional*) – From rc.BinCount or hash, if passed in it will be returned unchanged as part of this function's output.
- **unique_count** (*int, optional*) – required if ncountgroup is None, otherwise not unique_count (scalar int) (must include the 0 bin so +1 often added)
- **cutoffs** (*array_like, optional*) – cutoff array for parallel processing

**Returns**

- *3 arrays in a dict*
- **['iGroup']** (*array size is same as ikey, unique keys are grouped together*)
- **['iFirstGroup']** (*array size is number of unique keys, indexes into iGroup*)
- **['nCountGroup']** (*array size is number of unique keys, how many in each group*)

### Examples

```
>>> np.random.seed(12345)
>>> c = np.random.randint(0, 8, 10_000)
>>> x = rt.groupbyhash(c)
>>> ncountgroup = rc.BinCount(x['iKey'], x['unique_count'] + 1)
>>> rt.groupbypack(x['iKey'], ncountgroup)
{'iGroup': FastArray([   0,    9,   21, ..., 9988, 9991, 9992]),
 'iFirstGroup': FastArray([   0,    0, 1213, 2465, 3761, 4987, 6239, 7522, 8797]),
 'nCountGroup': FastArray([   0, 1213, 1252, 1296, 1226, 1252, 1283, 1275, 1203])}
```

The sum of the entries in the nCountGroup array returned by groupbypack matches the length of the original array.

```
>>> rt.groupbypack(x['iKey'], ncountgroup)['nCountGroup'].sum()
10000
```

riptable.rt_numpy.**hstack**(*tup*, *dtype=None*, ***kwargs*)

see numpy hstack riptable can also take a dtype (it will convert all arrays to that dtype while stacking) riptable version will preserve sentinels riptable version is multithreaded for special classes like categorical and dataset, it will check to see if the class has it's own hstack and it will call that

riptable.rt_numpy.**interp**(*x*, *xp*, *fp*)

One-dimensional or two-dimensional linear interpolation with clipping.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (xp, fp), evaluated at x.

**Parameters**

- **x** (*array of float32 or float64*) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (*1-D or 2-D sequence of float32 or float64*) – The x-coordinates of the data points, must be increasing if argument period is not specified. Otherwise, xp is internally sorted after normalizing the periodic boundaries with xp = xp % period.
- **fp** (*1-D or 2-D sequence of float32 or float64*) – The y-coordinates of the data points, same length as xp.

**Returns**
**y** – The interpolated values, same shape as `x`.

**Return type**
*float32* or *float64* (corresponding to fp) or ndarray

**See also:**

`np.interp`, `rt.interp_extrap`

### Notes

riptable version does not handle kwargs left/right whereas np does riptable version handles floats or doubles, whereas np is always a double riptable will warn if first parameter is a float32, but xp or yp is a double

`riptable.rt_numpy.`**`interp_extrap`**(*x*, *xp*, *fp*)

One-dimensional or two-dimensional linear interpolation without clipping.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (`xp`, `fp`), evaluated at `x`.

**See also:**

`np.interp`, `rt.interp`

### Notes

- riptable version handles floats or doubles, wheras np is always a double
- 2d mode is auto-detected based on `xp`/`fp`

`riptable.rt_numpy.`**`isfinite`**(**args*, ***kwargs*)

Return True for each finite element, False otherwise.

A value is considered to be finite if it's not positive or negative infinity or a NaN (Not a Number).

**Parameters**

- **`*args`** – See `numpy.isfinite`.
- **`**kwargs`** – See `numpy.isfinite`.

**Returns**
For array input, a `FastArray` of booleans is returned that's True for each element that's finite, False otherwise. For scalar input, a boolean is returned.

**Return type**
`FastArray` or bool

**See also:**

`riptable.isnotfinite`, `riptable.isinf`, `riptable.isnotinf`, `FastArray.isfinite`, `FastArray.isnotfinite`, `FastArray.isinf`, `FastArray.isnotinf`

**`Dataset.mask_or_isfinite`**
Return a boolean array that's True for each `Dataset` row that has at least one finite value.

**`Dataset.mask_and_isfinite`**
Return a boolean array that's True for each `Dataset` row that contains all finite values.

**Dataset.mask_or_isinf**
Return a boolean array that's True for each Dataset row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**
Return a boolean array that's True for each Dataset row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> rt.isfinite(a)
FastArray([False, False, False,  True])
```

```
>>> rt.isfinite(1)
True
```

riptable.rt_numpy.**isinf**(**args*, ***kwargs*)

Return True for each element that's positive or negative infinity, False otherwise.

**Parameters**

- ***args** – See numpy.isinf.
- ****kwargs** – See numpy.isinf.

**Returns**
For array input, a FastArray of booleans is returned that's True for each element that's positive or negative infinity, False otherwise. For scalar input, a boolean is returned.

**Return type**
FastArray or bool

**See also:**

riptable.isnotinf, riptable.isfinite, riptable.isnotfinite, FastArray.isinf, FastArray.isnotinf, FastArray.isfinite, FastArray.isnotfinite

**Dataset.mask_or_isfinite**
Return a boolean array that's True for each Dataset row that has at least one finite value.

**Dataset.mask_and_isfinite**
Return a boolean array that's True for each Dataset row that contains all finite values.

**Dataset.mask_or_isinf**
Return a boolean array that's True for each Dataset row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**
Return a boolean array that's True for each Dataset row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> rt.isinf(a)
FastArray([ True,  True, False, False])
```

```
>>> rt.isinf(1)
False
```

riptable.rt_numpy.**ismember**(*a*, *b*, *h=2*, *hint_size=0*, *base_index=0*)

The ismember function is meant to mimic the ismember function in MATLab. It takes two sets of data and returns two - a boolean array and array of indices of the first occurrence of an element in a in b - otherwise NaN.

**Parameters**

- **a** (*A python list (strings), python tuple (strings), chararray, ndarray of unicode strings,*) – ndarray of int32, int64, float32, or float64.
- **b** (A list with the same constraints as a. Note: if a contains string data, b must also contain) – string data. If it contains different numerical data, casting will occur in either a or b.
- **h** (*There are currently two different hashing functions that can be used to execute ismember.*) – Depending on the size, type, and number of matches in the data, the hashes perform differently. Currently accepts 1 or 2. 1=PRIME number (might be faster for floats - uses less memory) 2=MASK using power of 2 (usually faster but uses more memory)
- **hint_size** (*int, default 0*) – For large arrays with a low unique count, setting this value to 4*expected unique count may speed up hashing.
- **base_index** (*int, default 0*) – When set to 1 the first return argument is no longer a boolean array but an integer that is 1 or 0. A return value of 1 indicates there exists values in b that do not exist in a.

**Returns**

- **c** (*int or np.ndarray of bool*) – A boolean array the same size as a indicating whether or not the element at the corresponding index in a was found in b.
- **d** (*np.ndarray of int*) – An array of indices the same size as a which each indicate where an element in a first occured in b or NaN otherwise.

**Raises**

- **TypeError** – input must be ndarray, python list, or python tuple
- **ValueError** – data must be int32, int64, float32, float64, chararray, or unicode strings. If a contains string data, b must also contain string data and vice versa.

### Examples

```
>>> a = [1.0, 2.0, 3.0, 4.0]
>>> b = [1.0, 3.0, 4.0, 4.0]
>>> c,d = ismember(a,b)
>>> c
FastArray([ True, False,  True,  True])
>>> d
FastArray([   0, -128,    1,    2], dtype=int8)
```

NaN values do not behave the same way as other elements. A NaN in the first will not register as existing in the second array. This is the expected behavior (to match MatLab nan MATLab nan handling):

```
>>> a = FastArray([1.,2.,3.,np.nan])
>>> b = FastArray([2.,3.,np.nan])
>>> c,d = ismember(a,b)
>>> c
FastArray([False,  True,  True, False])
>>> d
FastArray([-128,    0,    1, -128], dtype=int8)
```

riptable.rt_numpy.**isnan**(**args*, ***kwargs*)

Return True for each element that's a NaN (Not a Number), False otherwise.

**Parameters**

- ***args** – See numpy.isnan.
- ****kwargs** – See numpy.isnan.

**Returns**

For array input, a FastArray of booleans is returned that's True for each element that's a NaN, False otherwise. For scalar input, a boolean is returned.

**Return type**

FastArray or bool

**See also:**

riptable.isnotnan, riptable.isnanorzero, FastArray.isnan, FastArray.isnotnan, FastArray.notna, FastArray.isnanorzero, Categorical.isnan, Categorical.isnotnan, Categorical.notna, Date.isnan, Date.isnotnan, DateTimeNano.isnan, DateTimeNano.isnotnan

**Dataset.mask_or_isnan**

Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**

Return a boolean array that's True for each all-NaN Dataset row.

### Examples

```
>>> a = rt.FastArray([rt.nan, rt.inf, 2])
>>> rt.isnan(a)
FastArray([ True, False, False])
```

```
>>> rt.isnan(0)
False
```

riptable.rt_numpy.**isnanorzero**(**args*, ***kwargs*)

Return True for each element that's a NaN (Not a Number) or zero, False otherwise.

**Parameters**

- ***args** – See numpy.isnan.
- ****kwargs** – See numpy.isnan.

**Returns**

For array input, a FastArray of booleans is returned that's True for each element that's a NaN or zero, False otherwise. For scalar input, a boolean is returned.

**Return type**

FastArray or bool

**See also:**

FastArray.isnanorzero, riptable.isnan, riptable.isnotnan, FastArray.isnan, FastArray.isnotnan, Categorical.isnan, Categorical.isnotnan, Date.isnan, Date.isnotnan, DateTimeNano.isnan, DateTimeNano.isnotnan

**Dataset.mask_or_isnan**

Return a boolean array that's True for each Dataset row that contains at least one NaN.

**Dataset.mask_and_isnan**

Return a boolean array that's True for each all-NaN Dataset row.

### Examples

```
>>> a = rt.FastArray([0, rt.nan, rt.inf, 3])
>>> rt.isnanorzero(a)
FastArray([ True,  True, False, False])
```

```
>>> rt.isnanorzero(0)
True
```

riptable.rt_numpy.**isnotfinite**(**args*, ***kwargs*)

Return True for each non-finite element, False otherwise.

A value is considered to be finite if it's not positive or negative infinity or a NaN (Not a Number).

**Parameters**

- ***args** – See numpy.isfinite.
- ****kwargs** – See numpy.isfinite.

**Returns**
For array input, a `FastArray` of booleans is returned that's True for each non-finite element, False otherwise. For scalar input, a boolean is returned.

**Return type**
`FastArray` or bool

**See also:**

`riptable.isfinite`, `riptable.isinf`, `riptable.isnotinf`, `FastArray.isfinite`, `FastArray.isnotfinite`, `FastArray.isinf`, `FastArray.isnotinf`

**`Dataset.mask_or_isfinite`**
Return a boolean array that's True for each `Dataset` row that has at least one finite value.

**`Dataset.mask_and_isfinite`**
Return a boolean array that's True for each `Dataset` row that contains all finite values.

**`Dataset.mask_or_isinf`**
Return a boolean array that's True for each `Dataset` row that has at least one value that's positive or negative infinity.

**`Dataset.mask_and_isinf`**
Return a boolean array that's True for each `Dataset` row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> rt.isnotfinite(a)
FastArray([ True,  True,  True, False])
```

```
>>> rt.isnotfinite(1)
False
```

`riptable.rt_numpy.`**`isnotinf`**(**args*, ***kwargs*)

Return True for each element that's not positive or negative infinity, False otherwise.

**Parameters**

- **`*args`** – See `numpy.isinf`.
- **`**kwargs`** – See `numpy.isinf`.

**Returns**
For array input, a `FastArray` of booleans is returned that's True for each element that's not positive or negative infinity, False otherwise. For scalar input, a boolean is returned.

**Return type**
`FastArray` or bool

**See also:**

`riptable.isinf`, `FastArray.isnotinf`, `FastArray.isinf`, `riptable.isfinite`, `riptable.isnotfinite`, `FastArray.isfinite`, `FastArray.isnotfinite`

**`Dataset.mask_or_isfinite`**
Return a boolean array that's True for each `Dataset` row that has at least one finite value.

**`Dataset.mask_and_isfinite`**
Return a boolean array that's True for each `Dataset` row that contains all finite values.

**Dataset.mask_or_isinf**
Return a boolean array that's True for each `Dataset` row that has at least one value that's positive or negative infinity.

**Dataset.mask_and_isinf**
Return a boolean array that's True for each `Dataset` row that contains all infinite values.

### Examples

```
>>> a = rt.FastArray([rt.inf, -rt.inf, rt.nan, 0])
>>> rt.isnotinf(a)
FastArray([False, False,  True,  True])
```

```
>>> rt.isnotinf(1)
True
```

riptable.rt_numpy.**isnotnan**(**args*, ***kwargs*)

Return True for each element that's not a NaN (Not a Number), False otherwise.

**Parameters**

- ***args** – See `numpy.isnan`.
- ****kwargs** – See `numpy.isnan`.

**Returns**
For array input, a `FastArray` of booleans is returned that's True for each element that's not a NaN, False otherwise. For scalar input, a boolean is returned.

**Return type**
`FastArray` or bool

**See also:**

`riptable.isnan`, `riptable.isnanorzero`, `FastArray.isnan`, `FastArray.isnotnan`, `FastArray.notna`, `FastArray.isnanorzero`, `Categorical.isnan`, `Categorical.isnotnan`, `Categorical.notna`, `Date.isnan`, `Date.isnotnan`, `DateTimeNano.isnan`, `DateTimeNano.isnotnan`

**Dataset.mask_or_isnan**
Return a boolean array that's True for each `Dataset` row that contains at least one NaN.

**Dataset.mask_and_isnan**
Return a boolean array that's True for each all-NaN `Dataset` row.

### Examples

```
>>> a = rt.FastArray([rt.nan, rt.inf, 2])
>>> rt.isnotnan(a)
FastArray([False,  True,  True])
```

```
>>> rt.isnotnan(0)
True
```

riptable.rt_numpy.**issorted**(**args*)

Return True if the array is sorted, False otherwise.

NaNs at the end of an array are considered sorted.

**Parameters**
***args** (*ndarray*) – The array to check. It must be one-dimensional and contiguous.

**Returns**
True if the array is sorted, False otherwise.

**Return type**
bool

**See also:**

`FastArray.issorted`

### Examples

```
>>> a = rt.FastArray(['a', 'c', 'b'])
>>> rt.issorted(a)
False
```

```
>>> a = rt.FastArray([1.0, 2.0, 3.0, rt.nan])
>>> rt.issorted(a)
True
```

```
>>> cat = rt.Categorical(['a', 'a', 'a', 'b', 'b'])
>>> rt.issorted(cat)
True
```

```
>>> dt = rt.Date.range('20190201', '20190208')
>>> rt.issorted(dt)
True
```

```
>>> dtn = rt.DateTimeNano(['6/30/19', '1/30/19'], format='%m/%d/%y', from_tz='NYC')
>>> rt.issorted(dtn)
False
```

riptable.rt_numpy.**lexsort**(**args*, ***kwargs*)

riptable.rt_numpy.**log**(**args*, ***kwargs*)

riptable.rt_numpy.**log10**(**args*, ***kwargs*)

riptable.rt_numpy.**logical**(*a*)

riptable.rt_numpy.**makeifirst**(*key*, *unique_count*, *filter=None*)

**Parameters**

- **key** (*ndarray of ints*) – Index array (int8, int16, int32 or int64).
- **unique_count** (*int*) – Maximum number of unique values in `key` array.
- **filter** (*ndarray of bools, optional*) – Boolean array same length as `key` array, defaults to None.

**Returns**
**index** – An index array of the same dtype and length of the `key` passed in. The index array will have the invalid value for the array's dtype set at any locations it could not find a first occurrence.

**Return type**
ndarray of ints

### Notes

makeifirst will NOT reduce the index/ikey unique size even when a filter is passed. Based on the integer dtype int8/16/32/64, all locations that have no first will be set to invalid. If an invalid is used as a riptable fancy index, it will pull in another invalid, for example '' empty string

riptable.rt_numpy.**makeilast**(*key*, *unique_count*, *filter=None*)

**Parameters**

- **key** (*ndarray of ints*) – Index array (int8, int16, int32 or int64).
- **unique_count** (*int*) – Maximum number of unique values in key array.
- **filter** (*ndarray of bools, optional*) – Boolean array same length as key array, defaults to None.

**Returns**
**index** – An index array of the same dtype and length of the key passed in. The index array will have the invalid value for the array's dtype set at any locations it could not find a last occurrence.

**Return type**
ndarray of ints

### Notes

makeilast will NOT reduce the index/ikey unique size even when a filter is passed. Based on the integer dtype int8/16/32/64, all locations that have no last will be set to invalid. If an invalid is used as a riptable fancy index, it will pull in another invalid, for example '' empty string

riptable.rt_numpy.**makeinext**(*key*, *unique_count*)

**Parameters**

- **key** (*ndarray of integers*) – index array (int8, int16, int32 or int64)
- **unique_count** (*int*) – max uniques in 'key' array

**Returns**

- *An index array of the same dtype and length of the next row*
- *The index array will have -MAX_INT set to any locations it could not find a next*

riptable.rt_numpy.**makeiprev**(*key*, *unique_count*)

**Parameters**

- **key** (*ndarray of integers*) – index array (int8, int16, int32 or int64)
- **unique_count** (*int*) – max uniques in 'key' array

**Return type**
The index array will have -MAX_INT set to any locations it could not find a previous

riptable.rt_numpy.**mask_and**(**args*, ***kwargs*)

pass in a tuple or list of boolean arrays to AND together

riptable.rt_numpy.**mask_andi**(**args*, ***kwargs*)

inplace version: pass in a tuple or list of boolean arrays to AND together

riptable.rt_numpy.**mask_andnot**(**args*, ***kwargs*)

pass in a tuple or list of boolean arrays to ANDNOT together

riptable.rt_numpy.**mask_andnoti**(**args*, ***kwargs*)

inplace version: pass in a tuple or list of boolean arrays to ANDNOT together

riptable.rt_numpy.**mask_or**(**args*, ***kwargs*)

pass in a tuple or list of boolean arrays to OR together

riptable.rt_numpy.**mask_ori**(**args*, ***kwargs*)

inplace version: pass in a tuple or list of boolean arrays to OR together

riptable.rt_numpy.**mask_xor**(**args*, ***kwargs*)

pass in a tuple or list of boolean arrays to XOR together

riptable.rt_numpy.**mask_xori**(**args*, ***kwargs*)

inplace version: pass in a tuple or list of boolean arrays to XOR together

riptable.rt_numpy.**max**(**args*, ***kwargs*)

riptable.rt_numpy.**maximum**(*x1*, *x2*, **args*, ***kwargs*)

riptable.rt_numpy.**mean**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the arithmetic mean of the values in the first argument.

When possible, `rt.mean(x, *args)` calls `x.mean(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.mean` accepts the `filter` and `dtype` keyword arguments, but `Dataset.mean` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the mean calculation. If the filter is uniformly `False`, `rt.mean` returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.mean(dtype = my_type)` is equivalent to `my_type(x.mean())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's mean.

**Return type**

scalar or `Dataset`

**See also:**

*nanmean*

Computes the mean, ignoring NaNs.

**Dataset.mean**

Computes the mean of numerical `Dataset` columns.

**FastArray.mean**

Computes the mean of `FastArray` values.

**GroupByOps.mean**
Computes the mean of each group. Used by `Categorical` objects.

### Notes

The `dtype` keyword for `rt.mean` specifies the data type of the result. This differs from `numpy.mean`, where it specifies the data type used to compute the mean.

### Examples

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> rt.mean(a)
4.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> rt.mean(a, dtype = rt.int32)
4
```

With a filter:

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> b = rt.FastArray([False, True, False, True])
>>> rt.mean(a, filter = b)
5.0
```

riptable.rt_numpy.**median**(**args*, ***kwargs*)

riptable.rt_numpy.**min**(**args*, ***kwargs*)

riptable.rt_numpy.**minimum**(*x1*, *x2*, **args*, ***kwargs*)

riptable.rt_numpy.**multikeyhash**(**args*)
Returns 7 arrays to help navigate data.

**Parameters**

- **key** – the unique occurence
- **nth** – the nth unique occurence
- **bktsize** – how many unique occurences occur
- **next** – index to the next unique occurence and previous
- **prev** – index to the next unique occurence and previous
- **first** – index to the first unique occurence and last
- **last** – index to the first unique occurence and last

### Examples

```
>>> myarr = rt.arange(10) % 3
>>> myarr
FastArray([0, 1, 2, 0, 1, 2, 0, 1, 2, 0])
```

```
>>> mkgrp = rt.Dataset(rt.multikeyhash([myarr]).asdict())
>>> mkgrp.a = myarr
>>> mkgrp
#   key   nth   bktsize   next   prev   first   last   a
-   ---   ---   -------   ----   ----   -----   ----   -
0     1     1         4      3     -1       0      9   0
1     2     1         3      4     -1       1      7   1
2     3     1         3      5     -1       2      8   2
3     1     2         4      6      0       0      9   0
4     2     2         3      7      1       1      7   1
5     3     2         3      8      2       2      8   2
6     1     3         4      9      3       0      9   0
7     2     3         3     -1      4       1      7   1
8     3     3         3     -1      5       2      8   2
9     1     4         4     -1      6       0      9   0
```

riptable.rt_numpy.**nan_to_num**(**args*, ***kwargs*)

arg1: ndarray returns: ndarray with nan_to_num notes: if you want to do this inplace contact TJD

riptable.rt_numpy.**nan_to_zero**(*a*)

Replace the NaN or invalid values in an array with zeroes.

This is an in-place operation – the input array is returned after being modified.

**Parameters**
**a** (*ndarray*) – The input array.

**Returns**
The input array `a` (after it's been modified).

**Return type**
ndarray

riptable.rt_numpy.**nanargmax**(**args*, ***kwargs*)

riptable.rt_numpy.**nanargmin**(**args*, ***kwargs*)

riptable.rt_numpy.**nanmax**(**args*, ***kwargs*)

riptable.rt_numpy.**nanmean**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the arithmetic mean of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `0.0` is returned.

When possible, `rt.nanmean(x, *args)` calls `x.nanmean(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.nanmean` accepts the `filter` and `dtype` keyword arguments, but `Dataset.nanmean` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the mean calculation. If the filter is uniformly `False`, `rt.nanmean` returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.nanmean(dtype = my_type)` is equivalent to `my_type(x.nanmean())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's mean.

**Return type**

scalar or `Dataset`

**See also:**

*mean*
Computes the mean.

**Dataset.nanmean**
Computes the mean of numerical `Dataset` columns, ignoring NaNs.

**FastArray.nanmean**
Computes the mean of `FastArray` values, ignoring NaNs.

**GroupByOps.nanmean**
Computes the mean of each group, ignoring NaNs. Used by `Categorical` objects.

### Notes

The `dtype` keyword for `rt.nanmean` specifies the data type of the result. This differs from `numpy.nanmean`, where it specifies the data type used to compute the mean.

### Examples

```
>>> a = rt.FastArray([1, 3, 5, rt.nan])
>>> rt.nanmean(a)
3.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 3, 5, rt.nan])
>>> rt.nanmean(a, dtype = rt.int32)
3
```

With a filter:

```
>>> a = rt.FastArray([1, 3, 5, rt.nan])
>>> b = rt.FastArray([False, True, True, True])
>>> rt.nanmean(a, filter = b)
4.0
```

`riptable.rt_numpy.`**nanmedian**(**args*, ***kwargs*)

riptable.rt_numpy.**nanmin**(**args*, ***kwargs*)

riptable.rt_numpy.**nanpercentile**(**args*, ***kwargs*)

riptable.rt_numpy.**nanstd**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the standard deviation of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `NaN` is returned.

Riptable uses the convention that `ddof = 1`, meaning the standard deviation of `[x_1, ..., x_n]` is defined by `std = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

When possible, `rt.nanstd(x, *args)` calls `x.nanstd(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.nanstd` accepts the `filter` and `dtype` keyword arguments, but `Dataset.nanstd` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the standard deviation calculation. If the filter is uniformly `False`, `rt.nanstd` returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.nanstd(dtype = my_type)` is equivalent to `my_type(x.nanstd())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's standard deviation.

**Return type**

scalar or `Dataset`

**See also:**

*std*

Computes the standard deviation.

`FastArray.nanstd`

Computes the standard deviation of `FastArray` values, ignoring NaNs.

`Dataset.nanstd`

Computes the standard deviation of numerical `Dataset` columns, ignoring NaNs.

`GroupByOps.nanstd`

Computes the standard deviation of each group, ignoring NaNs. Used by `Categorical` objects.

## Notes

The `dtype` keyword for `rt.nanstd` specifies the data type of the result. This differs from `numpy.nanstd`, where it specifies the data type used to compute the standard deviation.

### Examples

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> rt.nanstd(a)
1.0
```

With a dtype specified:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> rt.nanstd(a, dtype = rt.int32)
1
```

With filter:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> b = rt.FastArray([False, True, True, True])
>>> rt.nanstd(a, filter = b)
0.7071067811865476
```

riptable.rt_numpy.**nansum**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the sum of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `0.0` is returned.

When possible, `rt.nansum(x, *args)` calls `x.nansum(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.nansum` accepts the `filter` and `dtype` keyword arguments, but `Dataset.nansum` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the sum calculation. If the filter is uniformly `False`, `rt.nansum` returns `0.0`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.nansum(dtype = my_type)` is equivalent to `my_type(x.nansum())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's sum.

**Return type**

scalar or `Dataset`

**See also:**

***sum***

Sums the values of the input.

**FastArray.nansum**

Sums the values of a `FastArray`, ignoring NaNs.

**Dataset.nansum**

Sums the values of numerical `Dataset` columns, ignoring NaNs.

**GroupByOps.nansum**

Sums the values of each group, ignoring NaNs. Used by `Categorical` objects.

### Notes

The `dtype` keyword for `rt.nansum` specifies the data type of the result. This differs from `numpy.nansum`, where it specifies the data type used to compute the sum.

### Examples

```
>>> a = rt.FastArray( [1, 3, 5, 7, rt.nan])
>>> rt.nansum(a)
16.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1.0, 3.0, 5.0, 7.0, rt.nan])
>>> rt.nansum(a, dtype = rt.int32)
16
```

With a filter:

```
>>> a = rt.FastArray([1, 3, 5, 7, rt.nan])
>>> b = rt.FastArray([False, True, False, True, True])
>>> rt.nansum(a, filter = b)
10.0
```

riptable.rt_numpy.**nanvar**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the variance of the values in the first argument, ignoring NaNs.

If all values in the first argument are NaNs, `NaN` is returned.

Riptable uses the convention that `ddof = 1`, meaning the variance of `[x_1, ..., x_n]` is defined by `var = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

When possible, `rt.nanvar(x, *args)` calls `x.nanvar(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.nanvar` accepts the `filter` and `dtype` keyword arguments, but `Dataset.nanvar` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the variance calculation. If the filter is uniformly `False`, `rt.nanvar` returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.nanvar(dtype = my_type)` is equivalent to `my_type(x.nanvar())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's variance.

**Return type**

scalar or `Dataset`

**See also:**

*var*
Computes the variance.

**FastArray.nanvar**
Computes the variance of `FastArray` values, ignoring NaNs.

**Dataset.nanvar**
Computes the variance of numerical `Dataset` columns, ignoring NaNs.

**GroupByOps.nanvar**
Computes the variance of each group, ignoring NaNs. Used by `Categorical` objects.

### Notes

The `dtype` keyword for `rt.nanvar` specifies the data type of the result. This differs from `numpy.nanvar`, where it specifies the data type used to compute the variance.

### Examples

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> rt.nanvar(a)
1.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> rt.nanvar(a, dtype = rt.int32)
1
```

With a filter:

```
>>> a = rt.FastArray([1, 2, 3, rt.nan])
>>> b = rt.FastArray([False, True, True, True])
>>> rt.nanvar(a, filter = b)
0.5
```

`riptable.rt_numpy.`**ones**(*shape*, *dtype=None*, *order='C'*, ***, *like=None*)

Return a new array of the specified shape and data type, filled with ones.

**Parameters**

- **shape** (*int or sequence of int*) – Shape of the new array, e.g., `(2, 3)` or `2`. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (str or NumPy dtype or Riptable dtype, default `numpy.float64`) – The desired data type for the array.
- **order** (*{'C', 'F'}, default 'C'*) – Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.
- **like** (*array_like, default None*) – Reference object to allow the creation of arrays that are not NumPy arrays. If an array-like passed in as `like` supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

**Returns**
A new `FastArray` of the specified shape and type, filled with ones.

**Return type**
`FastArray`

**See also:**

`riptable.ones_like`, `riptable.zeros`, `riptable.zeros_like`, `riptable.empty`, `riptable.empty_like`, `riptable.full`

### Examples

```
>>> rt.ones(5)
FastArray([1., 1., 1., 1., 1.])
```

```
>>> rt.ones(5, dtype='int8')
FastArray([1, 1, 1, 1, 1], dtype=int8)
```

`riptable.rt_numpy.`**`ones_like`**(*a*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return an array of ones with the same shape and data type as the specified array.

**Parameters**

- **a** (*array*) – The shape and data type of `a` define the same attributes of the returned array. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str or NumPy dtype or Riptable dtype, optional*) – Overrides the data type of the result.
- **order** (*{'C', 'F', 'A', or 'K'}, default 'K'*) – Overrides the memory layout of the result. 'C' means row-major (C-style), 'F' means column-major (Fortran-style), 'A' means 'F' if `a` is Fortran-contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*bool, default True*) – If True (the default), then the newly created array will use the sub-class type of `a`, otherwise it will be a base-class array.
- **shape** (*int or sequence of int, optional*) – Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, it will try to keep the same order; otherwise, order='C' is implied. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.

**Returns**
A `FastArray` with the same shape and data type as the specified array, filled with ones.

**Return type**
`FastArray`

**See also:**

`riptable.ones`, `riptable.zeros`, `riptable.zeros_like`, `riptable.empty`, `riptable.empty_like`, `riptable.full`

### Examples

```
>>> a = rt.FastArray([1, 2, 3, 4])
>>> rt.ones_like(a)
FastArray([1, 1, 1, 1])
```

```
>>> rt.ones_like(a, dtype = float)
FastArray([1., 1., 1., 1.])
```

riptable.rt_numpy.**percentile**(**args*, ***kwargs*)

riptable.rt_numpy.**putmask**(*a*, *mask*, *values*)

This is roughly the equivalent of arr[mask] = arr2[mask].

### Examples

```
>>> arr = rt.FastArray([10, 10, 10, 10])
>>> arr2 = rt.FastArray([1, 2, 3, 4])
>>> mask = rt.FastArray([False, True, True, False])
>>> rt.putmask(arr, mask, arr2)
>>> arr
FastArray([10,  2,  3, 10])
```

It's important to note that the length of `arr` and `arr2` are presumed to be the same, otherwise the values in `arr2` are repeated until they have the same dimension.

It should NOT be used to replace this operation:

```
>>> arr = rt.FastArray([10, 10, 10, 10])
>>> arr2 = rt.FastArray([1, 2])
>>> mask = rt.FastArray([False, True, True, False])
>>> arr[mask] = arr2
>>> arr
FastArray([10,  1,  2, 10])
```

`arr2` is repeated to create `rt.FastArray([1, 2, 1, 2])` before performing the operation, hence the different result.

```
>>> arr = rt.FastArray([10, 10, 10, 10])
>>> arr2 = rt.FastArray([1, 2])
>>> mask = rt.FastArray([False, True, True, False])
>>> rt.putmask(arr, mask, arr2)
>>> arr
FastArray([10,  2,  1, 10])
```

riptable.rt_numpy.**reindex_fast**(*index*, *array*)

riptable.rt_numpy.**reshape**(**args*, ***kwargs*)

riptable.rt_numpy.**round**(**args*, ***kwargs*)

This will check for numpy array first and call np.round

riptable.rt_numpy.**searchsorted**(*a*, *v*, *side='left'*, *sorter=None*)

see np.searchsorted side ='leftplus' is a new option in riptable where values > get a 0

riptable.rt_numpy.**single**(*a*)

riptable.rt_numpy.**sort**(**args*, ***kwargs*)

riptable.rt_numpy.**sortinplaceindirect**(**args*, ***kwargs*)

riptable.rt_numpy.**std**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the standard deviation of the values in the first argument.

Riptable uses the convention that `ddof = 1`, meaning the standard deviation of `[x_1, ..., x_n]` is defined by `std = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

When possible, `rt.std(x, *args)` calls `x.std(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.std` accepts the `filter` and `dtype` keyword arguments, but `Dataset.std` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the standard deviation calculation. If the filter is uniformly `False`, `rt.std` returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.std(dtype = my_type)` is equivalent to `my_type(x.std())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's standard deviation.

**Return type**

scalar or `Dataset`

**See also:**

*nanstd*
Computes the standard deviation, ignoring NaNs.

**FastArray.std**
Computes the standard deviation of `FastArray` values.

**Dataset.std**
Computes the standard deviation of numerical `Dataset` columns.

**GroupByOps.std**
Computes the standard deviation of each group. Used by `Categorical` objects.

## Notes

The `dtype` keyword for `rt.std` specifies the data type of the result. This differs from `numpy.std`, where it specifies the data type used to compute the standard deviation.

### Examples

```
>>> a = rt.FastArray([1, 2, 3])
>>> rt.std(a)
1.0
```

With a dtype specified:

```
>>> a = rt.FastArray([1, 2, 3])
>>> rt.std(a, dtype = rt.int32)
1
```

With a filter:

```
>>> a = rt.FastArray([1, 2, 3])
>>> b = rt.FA([False, True, True])
>>> rt.std(a, filter = b)
0.7071067811865476
```

riptable.rt_numpy.**sum**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the sum of the values in the first argument.

When possible, `rt.sum(x, *args)` calls `x.sum(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below. For example, `Dataset.sum()` does not accept the `filter` or `dtype` keyword arguments.

For `FastArray.sum`, see `numpy.sum` for documentation but note the following:

- Until a reported bug is fixed, the `dtype` keyword argument may not work as expected:
  - Riptable data types (for example, `rt.float64`) are ignored.
  - NumPy integer data types (for example, `numpy.int32`) are also ignored.
  - NumPy floating point data types are applied as `numpy.float64`.
- If you include another NumPy parameter (for example, `axis=0`), the NumPy implementation of `sum` will be used and the `dtype` will be used to compute the sum.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the sum calculation.
- **dtype** (*rt.dtype or numpy.dtype, optional*) – The data type of the result. By default, for integer input the result `dtype` is `int64` and for floating point input the result `dtype` is `float64`. See the notes above about using this keyword argument with `FastArray` objects as input.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's sum.

**Return type**

scalar or `Dataset`

**See also:**

`numpy.sum`

*nansum*
Sums the values, ignoring NaNs.

**FastArray.sum**
Sums the values of a FastArray.

**Dataset.sum**
Sums the values of numerical Dataset columns.

**GroupByOps.sum**
Sums the values of each group. Used by Categorical objects.

### Examples

```
>>> a = rt.FastArray([1, 3, 5, 7])
>>> rt.sum(a)
16
```

```
>>> a = rt.FastArray([1.0, 3.0, 5.0, 7.0])
>>> rt.sum(a)
16.0
```

riptable.rt_numpy.**tile**(*arr*, *reps*)

Construct an array by repeating a specified array a specified number of times.

**Parameters**

- **a** (*array or scalar*) – The input array or scalar.
- **reps** (*int or array of int*) – The number of repetitions of a along each axis. For examples of *tile* used with multi-dimensional arrays, see numpy.tile(). Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.

**Returns**
A new FastArray of the repeated input arrays.

**Return type**
FastArray

**See also:**

**riptable.repeat**
Construct an array by repeating each element of a specified array.

### Examples

Tile a scalar:

```
>>> rt.tile(2, 5)
FastArray([2, 2, 2, 2, 2])
```

Tile an array:

```
>>> x = rt.FA([1, 2, 3, 4])
>>> rt.tile(x, 2)
FastArray([1, 2, 3, 4, 1, 2, 3, 4])
```

riptable.rt_numpy.**transpose**(**args*, ***kwargs*)

riptable.rt_numpy.**trunc**(**args*, ***kwargs*)

riptable.rt_numpy.**unique32**(*list_keys*, *hintSize=0*, *filter=None*)

Returns the index location of the first occurence of each key.

**Parameters**

- **list_keys** (*list of ndarray*) – A list of numpy arrays to hash on (multikey); if there is just one item it still needs to be in a list such as `[array1]`.
- **hintSize** (*int*) – Integer hint if the number of unique keys (in `list_keys`) is known in advance, defaults to 0.
- **filter** (*ndarray of bools, optional*) – Boolean array used to pre-filter the array(s) in `list_keys` prior to processing them, defaults to None.

**Returns**

An array the size of the total unique values; the array contains the INDEX to the first occurence of the unique value. the second array contains the INDEX to the last occurence of the unique value.

**Return type**

ndarray of ints

riptable.rt_numpy.**var**(**args*, *filter=None*, *dtype=None*, ***kwargs*)

Compute the variance of the values in the first argument.

Riptable uses the convention that `ddof = 1`, meaning the variance of `[x_1, ..., x_n]` is defined by `var = 1/(n - 1) * sum(x_i - mean )**2` (note the `n - 1` instead of `n`). This differs from NumPy, which uses `ddof = 0` by default.

When possible, `rt.var(x, *args)` calls `x.var(*args)`; look there for documentation. In particular, note whether the called function accepts the keyword arguments listed below.

For example, `FastArray.var` accepts the `filter` and `dtype` keyword arguments, but `Dataset.var` does not.

**Parameters**

- **filter** (*array of bool, default None*) – Specifies which elements to include in the variance calculation. If the filter is uniformly `False`, `rt.var` returns a `ZeroDivisionError`.
- **dtype** (*rt.dtype or numpy.dtype, default float64*) – The data type of the result. For a `FastArray` `x`, `x.var(dtype = my_type)` is equivalent to `my_type(x.var())`.

**Returns**

Scalar for `FastArray` input. For `Dataset` input, returns a `Dataset` consisting of a row with each numerical column's variance.

**Return type**

scalar or `Dataset`

**See also:**

*nanvar*
Computes the variance, ignoring NaNs.

**FastArray.var**
Computes the variance of `FastArray` values.

**Dataset.var**
Computes the variance of numerical `Dataset` columns.

**GroupByOps.var**
Computes the variance of each group. Used by `Categorical` objects.

### Notes

The `dtype` keyword for `rt.var` specifies the data type of the result. This differs from `numpy.var`, where it specifies the data type used to compute the variance.

### Examples

```
>>> a = rt.FastArray([1, 2, 3])
>>> rt.var(a)
1.0
```

With a `dtype` specified:

```
>>> a = rt.FastArray([1, 2, 3])
>>> rt.var(a, dtype = rt.int32)
1
```

With a filter:

```
>>> a = rt.FastArray([1, 2, 3])
>>> b = rt.FastArray([False, True, True])
>>> rt.var(a, filter = b)
0.5
```

riptable.rt_numpy.**vstack**(*arrlist*, *dtype=None*, *order='C'*)

**Parameters**

- **arrlist** (*list of 1d numpy arrays of the same length*) – these arrays are considered the columns
- **order** (*defaults to 'C' for row major. 'F' will be column major.*) –
- **dtype** (*defaults to None. Can specifiy the final dtype for order='F' only.*) –
- **WARNING** (*when order='F' riptable vstack will return a diffrent shape*) –
- **length** (*from np.vstack since it will try to keep the first dim the same*) –
- **contiguous.** (*while keeping data*) –
- **passed** (*If order='F' is not*) –

- **assumed.** (*order='C' is*) –
- **fails** (*If riptable*) –
- **called.** (*then normal np.vstack will be*) –
- **arrays** (*For large*) –
- **fly.** (*riptable can run in parallel while converting to the dtype on the*) –

**Returns**

- *a 2d array that is column major and can be insert into a dataset*
- **Use v[** (*,0] then v[:,1] to access the columns instead of*)
- *v[0] and v[1] which would be the method with np.vstack*

**See also:**

`np.vstack`, `np.column_stack`

### Examples

```
>>> a = rt.arange(100)
>>> b = rt.arange(100.0)
>>> v = rt.vstack([a,b], order='F')
>>> v.strides
(8, 800)
```

```
>>> v.flags
C_CONTIGUOUS : False
F_CONTIGUOUS : True
```

```
>>> v.shape
(100,2)
```

`riptable.rt_numpy.`**`where`**(*condition*, *x=None*, *y=None*)

Return a new `FastArray` or `Categorical` with elements from `x` or `y` depending on whether `condition` is True.

For 1-dimensional arrays, this function is equivalent to:

```
[xv if c else yv
 for c, xv, yv in zip(condition, x, y)]
```

If only `condition` is provided, this function returns a tuple containing an integer `FastArray` with the indices where the condition is True. Note that this usage of *where* is not supported for `FastArray` objects of more than one dimension.

Note also that this case of *where* uses `riptable.bool_to_fancy()`. Using *bool_to_fancy* directly is preferred, as it behaves correctly for subclasses.

**Parameters**

- **condition** (*bool or array of bool*) – Where True, yield `x`, otherwise yield `y`.

- **x** (*scalar, array, or callable, optional*) – The value to use where condition is True. If x is provided, y must also be provided, and x and y should be the same type. If x is an array, a callable that returns an array, or a Categorical, it must be the same length as condition. The value of x that corresponds to the True value is used.
- **y** (*scalar, array, or callable, optional*) – The value to use where condition is False. If y is provided, x must also be provided, and x and y should be the same type. If y is an array, a callable that returns an array, or a Categorical, it must be the same length as condition. The value of y that corresponds to the False value is used.

**Returns**

If x and y are Categorical objects, a Categorical is returned. Otherwise, if x and y are provided a FastArray is returned. When only condition is provided, a tuple is returned containing an integer FastArray with the indices where the condition is True.

**Return type**

*FastArray* or *Categorical* or tuple

**See also:**

***FastArray.where***

Replace values where a given condition is False.

**riptable.bool_to_fancy**

The function called when x and y are omitted.

## Examples

condition is a comparison that creates an array of booleans, and x and y are scalars:

```
>>> a = rt.FastArray(rt.arange(5))
>>> a
FastArray([0, 1, 2, 3, 4])
>>> rt.where(a < 2, 100, 200)
FastArray([100, 100, 200, 200, 200])
```

condition and x are same-length arrays, and y is a scalar:

```
>>> condition = rt.FastArray([False, False, True, True, True])
>>> x = rt.FastArray([100, 101, 102, 103, 104])
>>> y = 200
>>> rt.where(condition, x, y)
FastArray([200, 200, 102, 103, 104])
```

When x and y are Categorical objects, a Categorical is returned:

```
>>> primary_traders = rt.Cat(['John', 'Mary', 'John', 'Mary', 'John', 'Mary'])
>>> secondary_traders = rt.Cat(['Chris', 'Duncan', 'Chris', 'Duncan', 'Duncan',
↪'Chris'])
>>> is_primary = rt.FA([True, True, False, True, False, True])
>>> rt.where(is_primary, primary_traders, secondary_traders)
Categorical([John, Mary, Chris, Mary, Duncan, Mary]) Length: 6
  FastArray([3, 4, 1, 4, 2, 4], dtype=int8) Base Index: 1
  FastArray([b'Chris', b'Duncan', b'John', b'Mary'], dtype='|S6') Unique count: 4
```

When x and y are Date objects, a FastArray of integers is returned that can be converted to a Date (other datetime objects are similar):

```
>>> x = rt.Date(['20230101', '20220101', '20210101'])
>>> y = rt.Date(['20190101', '20180101', '20170101'])
>>> condition = x > rt.Date(['20211231'])
>>> rt.where(condition, x, y)
>>> FastArray([19358, 18993, 17167])
>>> rt.Date(_)
Date(['2023-01-01', '2022-01-01', '2017-01-01'])
```

When only a condition is provided, a tuple is returned containing a FastArray with the indices where the condition is True:

```
>>> a = rt.FastArray([10, 20, 30, 40, 50])
>>> rt.where(a < 40)
(FastArray([0, 1, 2]),)
```

riptable.rt_numpy.**zeros**(**args*, ***kwargs*)

Return a new array of the specified shape and data type, filled with zeros.

**Parameters**

- **shape** (*int or sequence of int*) – Shape of the new array, e.g., (2, 3) or 2. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (str or NumPy dtype or Riptable dtype, default numpy.float64) – The desired data type for the array.
- **order** (*{'C', 'F'}, default 'C'*) – Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.
- **like** (*array_like, default None*) – Reference object to allow the creation of arrays that are not NumPy arrays. If an array-like passed in as like supports the __array_function__ protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

**Returns**

A new FastArray of the specified shape and type, filled with zeros.

**Return type**

FastArray

**See also:**

riptable.zeros_like, riptable.ones, riptable.ones_like, riptable.empty, riptable.empty_like, riptable.full

### Examples

```
>>> rt.zeros(5)
FastArray([0., 0., 0., 0., 0.])
```

```
>>> rt.zeros(5, dtype = 'int8')
FastArray([0, 0, 0, 0, 0], dtype=int8)
```

riptable.rt_numpy.**zeros_like**(*a*, *dtype=None*, *order='k'*, *subok=True*, *shape=None*)

Return an array of zeros with the same shape and data type as the specified array.

**Parameters**

- **a** (*array*) – The shape and data type of `a` define the same attributes of the returned array. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.
- **dtype** (*str or NumPy dtype or Riptable dtype, optional*) – Overrides the data type of the result.
- **order** (*{'C', 'F', 'A', or 'K'}, default 'K'*) – Overrides the memory layout of the result. 'C' means row-major (C-style), 'F' means column-major (Fortran-style), 'A' means 'F' if `a` is Fortran-contiguous, 'C' otherwise. 'K' means match the layout of `a` as closely as possible.
- **subok** (*bool, default True*) – If True (the default), then the newly created array will use the sub-class type of `a`, otherwise it will be a base-class array.
- **shape** (*int or sequence of int, optional*) – Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, it will try to keep the same order; otherwise, order='C' is implied. Note that although multi-dimensional arrays are technically supported by Riptable, you may get unexpected results when working with them.

**Returns**

A `FastArray` with the same shape and data type as the specified array, filled with zeros.

**Return type**

`FastArray`

**See also:**

`riptable.zeros`, `riptable.ones`, `riptable.ones_like`, `riptable.empty`, `riptable.empty_like`, `riptable.full`

### Examples

```
>>> a = rt.FastArray([1, 2, 3, 4])
>>> rt.zeros_like(a)
FastArray([0, 0, 0, 0])
```

```
>>> rt.zeros_like(a, dtype = float)
FastArray([1., 1., 1., 1.])
```

riptable.rt_numpy.**asanyarray**

riptable.rt_numpy.**asarray**

### 2.2.35 riptable.rt_pdataset

#### Classes

| | |
|---|---|
| *PDataset* | The PDataset class inherits from Dataset. It holds multiple datasets (preivously stacked together) in contiguous slices. |

**class** riptable.rt_pdataset.**PDataset**(*inputval=None*, *cutoffs=None*, *filenames=None*, *pnames=None*, *showpartitions=True*, ***kwargs*)

Bases: *riptable.rt_dataset.Dataset*

The PDataset class inherits from Dataset. It holds multiple datasets (preivously stacked together) in contiguous slices. Each partition has a name and a contiguous slice that can be used to extract it from the larger Dataset. Extracting a partition is zero-copy. Partitions can be extracted using partition(), or bracket [] indexing.

**A PDataset is often returned when:**

Multiple Datasets are hstacked, i.e. hstack([ds1, ds2, ds3]) Calling load_sds with stack=True, i.e. load_sds([file1, file2, file3], stack=True)

Properties: prows, pdict, pnames, pcount, pgb, pgbu, pgroupby, pslices, piter, pcutoffs Methods: partition(), pslice(), showpartitions()

pds['20190204'] or pds[20190204] will return a dataset for the given partition name

**Construction**

**inputval**

[-list of files to load and stack] -list of datasets to stack -regular dataset inputval (will only have one partition)

PDataset([path1, path2, path3], (pnames)) -call load_sds(stack=True) -paths become filenames -if pnames specified, use those, otherwise look for dates -if no dates, auto generate pnames

PDataset([ds1, ds2, ds3], (filenames, pnames)) PDataset(ds, (filenames, pnames)) -call Dataset.hstack() -if pnames specified, use those -if filenames, look for dates -if no dates, auto generate pnames

PDataset(arraydict, cutoffs, (filenames, pnames)) -constructor from load_sds() -if pnames specified, use those -if filenames, look for dates -if no dates, auto generate pnames

**property _row_numbers**

Subclasses can define their own callback function to customize the left side of the table. If not defined, normal row numbers will be displayed

**Parameters**

- **arr** (*array*) – Fancy index array of row numbers
- **style** (ColumnStyle) – Default style object for final row numbers column.

**Returns**

- **header** (*string*)
- **label_array** (*ndarray*)
- **style** (ColumnStyle)

**property pcat**

Lazy generates a categorical for row labels callback or pgroupby

**property pcount**

rtype: Number of partitions

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1, file2, file3], stack=True)
>>> pds.pcount
3
```

**property pcutoffs**

rtype: Cutoffs for partition. For slicing, maintain contiguous arrays.

### Examples

```
>>> pds.pcutoffs
FastArray([1447138, 3046565, 5344567], dtype=int64)
```

**property pdict**

rtype: A dictionary with the partition names and the partition slices.

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1, file2, file3], stack=True)
>>> pds.pdict
{'20190204': slice(0, 1447138, None),
 '20190205': slice(1447138, 3046565, None),
 '20190206': slice(3046565, 4509322, None)}
```

**property piter**

Iterate over dictionary of arrays for each partition. Yields key (load source) -> value (dataset as dictionary)

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1,file2, file2], stack=True)
>>> for name, ds in pds.iter: print(name)
20190204
20190205
20190206
```

**property pnames**

rtype: A list with the names of the partitions

### Example

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1, file2, file3], stack=True)
>>> pds.pnames
['20190205', '20190206', '20190207']
```

**property prows**

rtype: An array with the number of rows in each partition.

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1, file2, file3], stack=True)
>>> pds.prows
FastArray([1447138, 2599427, 1909895], dtype=int64)
```

**property pslices**

Return the slice (start,end) associated with the partition number

**See also:**

*pslices*, *pdict*

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1,file2, file2], stack=True)
>>> pds.pslices
[slice(0, 1447138, None),
 slice(1447138, 3046565, None),
 slice(3046565, 4509322, None)]
```

**__getitem__**(*index*)

**Parameters**
**index** – (rowspec, colspec) or colspec

**Returns**
the indexed row(s), cols(s), sub-dataset or single value

**Raises**

- **IndexError** –
- **TypeError** –
- **KeyError** –

**classmethod _auto_pnames**(*pcount*)

Auto generate partition names if none provided and no date found in filenames.

**_autocomplete**()

**_copy**(*deep=False*, *rows=None*, *cols=None*, *base_index=0*, *cls=None*)

returns a PDataset if no row selection, otherwise Dataset

**classmethod _filenames_to_pnames**(*filenames*)

At least two filenames must be present to compare Algo will reverse the string on the assumption that pathnames can vary in the front of the string It also assumes that the filenames end similarly, such as ".SDS" It will search for the difference and look for digits, then try to extract the digits

**classmethod _init_from_list**(*dlist*, *filenames*, *pnames*)

Construct a PDataset from multiple datasets, or by loading multiple files.

**classmethod _init_pnames_filenames**(*pcount*, *pnames*, *filenames*)

Initialize filenames, pnames based on what was provided to the constructor.

If no pnames provided, try to derive a date from filenames If no date found, or no filenames provided, use default names [p0, p1, p2 . . . ]

**Parameters**

- **pcount** (*int*) – number of partitions, in case names need to be auto generated
- **pnames** (*list of str, optional*) – list of partition names or None
- **filenames** (*sequence of str, optional*) – list of file paths (possibly empty)

**_ipython_key_completions_**()

**_post_init**(*cutoffs*, *filenames*, *pnames*, *showpartitions*)

Final initializer for variables specific to PDataset. Also initializes variables from parent class.

**_pre_init**()

Keep this in for chaining pre-inits in parent classes.

**abstract classmethod hstack**(*pds_list*)

Stacks columns from multiple datasets. see: Dataset.concat_rows

**igroupby**()

Lazily generate a categorical binned by each partition. Data will be attached to categorical, so operations can be called without specifying data. This allows reduce functions to be applied per partion.

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1,file2, file2], stack=True)
>>> pds.pgroupby['AskSize'].sum()
*Partition   TradeSize
----------   ---------
20190204     1.561e+07
20190205     1.950e+07
20190206     1.532e+07
```

See Also: Dataset.groupby, Dataset.gb, Dataset.gbu

**partition**(*index*)

Return the Dataset associated with the partition number

### Examples

Example below assumes 3 filenames with datasets

```
>>> pds = load_sds([file1, file2, file2], stack=True)
>>> pds.partition(0)
```

**pgb**(*by*, ***kwargs*)

Equivalent to `pgroupby()`

**pgroupby**(*by*, ***kwargs*)

**classmethod pload**(*path*, *start*, *end*, *include=None*, *threads=None*, *folders=None*)

Returns a PDataset of stacked files from multiple days. Will load all files found within the date range provided.

**Parameters**

- **path** (*format string for filepath, {} in place of YYYYMMDD. {} may appear multiple times.*) –
- **start** (*integer or string start date in format YYYYMMDD*) –
- **end** (*integer or string end date in format YYYYMMDD*) –

**prow_labeler**(*rownumbers*, *style*)

Display calls this routine back to replace row numbers. rownumbers : fancy index of row numbers being displayed style : ColumnStyle object - default from DisplayTable, can be changed

Returns: label header, label array, style

**abstract psave**()

Does not work yet. Would save backout all the partitions.

**pslice**(*index*)

Return the slice (start,end) associated with the partition number

**See also:**

*pslices*, *pdict*

### Examples

```
>>> pds.pslice(0)
slice(0, 1447138, None)
```

**save**(*path=''*, *share=None*, *compress=True*, *overwrite=True*, *name=None*, *onefile=False*, *bandsize=None*, *append=None*, *complevel=None*)

Save a dataset to a single .sds file or shared memory.

**Parameters**

- **path** (*str or os.PathLike*) – full path to save location + file name (if no .sds extension is included, it will be added)
- **share** (*str, optional*) – Shared memory name. If set, dataset will be saved to shared memory and NOT to disk when shared memory is specified, a filename must be included in path. only this will be used, the rest of the path will be discarded.

- **compress** (*bool*) – Use compression when saving the file. Shared memory is always saved uncompressed.
- **overwrite** (*bool*) – Defaults to True. If False, prompt the user when overwriting an existing .sds file; mainly useful for Struct.save(), which may call Dataset.save() multiple times.
- **name** (*str, optional*) –
- **bandsize** (*int, optional*) – If set to an integer > 10000 it will compress column data every bandsize rows
- **append** (*str, optional*) – If set to a string it will append to the file with the section name.
- **complevel** (*int, optional*) – Compression level from 0 to 9. 2 (default) is average. 1 is faster, less compressed, 3 is slower, more compressed.

### Examples

```
>>> ds = rt.Dataset({'col_'+str(i):a rt.range(5) for i in range(3)})
>>> ds.save('my_data')
>>> os.path.exists('my_data.sds')
True
```

```
>>> ds.save('my_data', overwrite=False)
my_data.sds already exists and is a file. Overwrite? (y/n) n
No file was saved.
```

```
>>> ds.save('my_data', overwrite=True)
Overwriting file with my_data.sds
```

```
>>> ds.save('shareds1', share='sharename')
>>> os.path.exists('shareds1.sds')
False
```

**See also:**

`Dataset.load`, `Struct.save`, `Struct.load`, `load_sds`, `load_h5`

**set_pnames**(*pnames*)

**Parameters**

**pnames** (*list of str*) –

### Examples

Example below assumes 3 filenames date encoded with datasets

```
>>> pds = load_sds([file1, file2, file3], stack=True)
>>> pds.pnames
['20190205', '20190206', '20190207']
>>> pds.set_pnames(['Jane', 'John', 'Jill'])
['Jane', 'John', 'Jill']
```

**showpartitions**(*show=True*)

toggle whether partitions are shown on the left

### 2.2.36 riptable.rt_pgroupby

#### Classes

| | |
|---|---|
| *PGroupBy* | **param dataset**<br>The dataset object |

**class** riptable.rt_pgroupby.**PGroupBy**(**args*, ***kwargs*)

Bases: *riptable.rt_groupby.GroupBy*

**Parameters**

- **dataset** (Dataset) – The dataset object
- **keys** (*list*) – List of column names to groupby
- **filter** (*array of bools*) – Boolean mask array applied as filter before grouping
- **return_all** (*bool*) – Default to False. When set to True will return all the dataset columns for every operation.
- **hintSize** (*int*) – Hint size for the hash
- **sort** (*bool*) – Default to True. Indicates

**gbkeys**

**Type**
dictionary of numpy arrays binned from

**isortrows**

**Type**
sorted index or None

**DebugMode = False**

**ShowEmpty = True**

**TestCatGb = True**

**copy**(*deep=True*)

Called from getitem when user follows gb with []

### 2.2.37 riptable.rt_sds

#### Functions

| | |
|---|---|
| *SDSMakeDirsOff*() | Disables SDSMakeDirs. |
| *SDSMakeDirsOn*() | Enables SDSMakeDirs. |
| *SDSRebuildRootOff*() | Disables SDSRebuildRoot. |
| *SDSRebuildRootOn*() | Enables SDSRebuildRoot. |
| *SDSVerboseOff*() | Disables SDSVerbose. |
| *SDSVerboseOn*() | Enables SDSVerbose. |
| *compress_dataset_internal*(filename, metadata, listarrays) | All SDS saves will hit this routine before the final call to riptable_cpp.CompressFile() |
| *container_from_filetype*(filetype) | Returns the appropriate container class based on the SDSFileType enum saved in the SDS file header. |
| *decompress_dataset_internal*(filename[, mode, ...]) | **param filename**<br>A string (or list of strings) of fully qualified path name, or shared memory location (e.g., Global\...) |
| *load_sds*(filepath[, share, info, include_all_sds, ...]) | Load a dataset from single .sds file or struct from directory of .sds files. |
| *load_sds_mem*(filepath, share[, include, threads, filter]) | Explicitly load data from shared memory. |
| *save_sds*(filepath, item[, share, compress, overwrite, ...]) | Datasets and arrays will be saved into a single .sds file. |
| *save_struct*([data, path, sharename, name, overwrite, ...]) | |
| *sds_concat*(filenames[, output, include]) | **param filenames**<br>List of fully qualified pathnames |
| *sds_flatten*(rootpath) | *sds_flatten* brings all structs and nested structures in sub-directories into the main directory. |
| *sds_info*(filepath[, share, sections, threads]) | |
| *sds_tree*(filepath[, threads]) | Explicitly display a tree of data for .sds file or directory. |

riptable.rt_sds.**SDSMakeDirsOff**()

Disables SDSMakeDirs.

riptable.rt_sds.**SDSMakeDirsOn**()

Enables SDSMakeDirs.

riptable.rt_sds.**SDSRebuildRootOff**()

Disables SDSRebuildRoot.

riptable.rt_sds.**SDSRebuildRootOn**()

Enables SDSRebuildRoot.

riptable.rt_sds.**SDSVerboseOff**()

Disables SDSVerbose.

riptable.rt_sds.**SDSVerboseOn**()

Enables SDSVerbose.

riptable.rt_sds.**compress_dataset_internal**(*filename*, *metadata*, *listarrays*, *meta_tups=None*, *comptype=CompressionType.ZStd*, *complevel=2*, *fileType=0*, *sharename=None*, *bandsize=None*, *append=None*)

All SDS saves will hit this routine before the final call to `riptable_cpp.CompressFile()`

**Parameters**

- **filename** (*str or bytes or os.PathLike*) – Fully qualified filename (path has already been checked by save_sds wrapper)
- **metadata** (*bytes*) – JSON metadata as a bytestring
- **listarrays** (*list of numpy arrays*) –
- **meta_tups** (*Tuples of (itemname, SDSFlag) - see SDSFlag enum in rt_enum.py*) –
- **comptype** (*CompressionType*) – Specify the type of compression to use when saving the Dataset.
- **complevel** (*int*) – Compression level. 2 (default) is average. 1 is faster, less compressed, 3 is slower, more compressed.
- **fileType** (*SDSFileType*) – See SDSFileType in rt_enum.py - distinguishes between Struct, Dataset, Single item, or Matlab Table
- **sharename** (*str or bytes, optional*) – If provided, data will be saved (uncompressed) into shared memory. No file will be saved to disk.

**Return type**
None

riptable.rt_sds.**container_from_filetype**(*filetype*)

Returns the appropriate container class based on the `SDSFileType` enum saved in the SDS file header. Older files where the file type is not set will default to 0, and container will default to `Struct`.

**Parameters**
**filetype** (*SDSFileType*) –

**Return type**
type

riptable.rt_sds.**decompress_dataset_internal**(*filename*, *mode=CompressionMode.DecompressFile*, *sharename=None*, *info=False*, *include=None*, *stack=None*, *threads=None*, *folders=None*, *sections=None*, *filter=None*, *mustexist=False*, *goodfiles=None*)

**Parameters**

- **filename** (*str or bytes or os.PathLike or sequence of str*) – A string (or list of strings) of fully qualified path name, or shared memory location (e.g., `Global\...`)
- **mode** (*CompressionMode*) – When set to `CompressionMode.Info`, tup2 is replaced with a tuple of numpy attributes (shape, dtype, flags, itemsize) (default CompressionMode).
- **sharename** (*str, or bytes, optional*) – Unique bytestring for shared memory location. Prevents mistakenly overwriting data in shared memory (defaults to None).
- **include** (*str, bytes, or list of str*) – Which items to include in the load. If items were omitted, tuples will still appear, but None will be loaded as their corresponding data (defaults to None).

- **stack** (*bool, optional*) – Set to `True` to stack array data before loading into python (see docstring for `stack_sds`). Set to `False` when appending many files into one and want columns flattening. Defaults to None.
- **threads** (*int, optional*) – How many threads to read, stack, and decompress with (defaults to None).
- **info** (*boolean*) – Instead of decompressing numpy arrays, return a summary of each one's contents (shape/dtype/itemsize/etc.)
- **folders** (*str, bytes, or list of strings, optional*) – When saving with `onefile=True` (will filter out only those subfolders) list of strings (defaults to None)
- **filter** (*ndarray, optional*) – A boolean or fancy index filter (only rows in the filter will be added) (defaults to None).
- **mustexist** (*bool*) – When true will raise exception if any file is missing.
- **sections** (*list of str, optional*) – List of strings with sections to load (file must have been saved with `append=`) (defaults to None).
- **goodfiles** (*list of str, optional*) – Tuples of two objects (list of filenames, path the files came from) – often from `os.walk` (defaults to None).

**Returns**

tup1: json metadata in a bytestring tup2: list of numpy arrays or tuple of (shape, dtype, flags, itemsize) if info mode tup3: list of tuples containing (itemname, SDSFlags bitmask) for all items in container (might not correspond with 2nd item's arrays) tup4: dictionary of file header meta data

**Return type**

list of tuples, optional

**Raises**

**ValueError** – If `include` is not a list of column names. If the result doesn't contain any data.

riptable.rt_sds.**load_sds**(*filepath*, *share=None*, *info=False*, *include_all_sds=False*, *include=None*, *name=None*, *threads=None*, *stack=None*, *folders=None*, *sections=None*, *filter=None*, *mustexist=False*, *verbose=False*, *reserve=0.0*)

Load a dataset from single `.sds` file or struct from directory of `.sds` files.

When `stack=True`, generic loader for a single `.sds` file or directory of multiple `.sds` files.

**Parameters**

- **filepath** (*str or bytes or os.PathLike or sequence of str*) – Full path to file or directory. When `stack` is `True` can be list of `.sds` files to stack When `stack` is `True` list of directories containing `.sds` files to stack (must also use kwarg `include`)
- **share** (*str, optional*) – The shared memory name. loader will check for dataset in shared memory first and if it's not there, the data (if the filepath is found on disk) will be loaded into the user's workspace AND shared memory. A sharename must be accompanied by a file name. The rest of a full path will be trimmed off internally. Defaults to None. For Windows make sure SE_CREATE_GLOBAL_NAME flag is set.
- **info** (*bool*) – No item data will be loaded, the hierarchy will be displayed in a tree (defaults to False).
- **include_all_sds** (*bool*) – If `True`, any extra files in saved struct's directory will be loaded into final struct (skips user prompt) (defaults to False).

- **include** (*list of str, optional*) – A list of strings of which columns to load, e.g. `['Ask','Bid']`. When `stack` is `True` and directories passed, list of filenames to stack across each directory (defaults to None).
- **name** (*str, optional*) – Optionally specify the name of the struct being loaded. This might be different than directory (defaults to None).
- **threads** (*int, optional*) – How many threads to read, stack, and decompress with (defaults to None).
- **stack** (*bool, optional*) – Set to `True` to stack array data before loading into python (see docstring for `stack_sds`). Set to `False` when appending many files into one and want columns flattening. This parameter is not compatible with the `share` or `info` parameters (defaults to None).
- **folders** (*list of str, optional*) – A list of strings on which folders to include e.g., `['zz/','xtra/']` (must be saved with `onefile=True`) (defaults to None).
- **sections** (*list of str, optional*) – A list of strings on which sections to include (must be saved with `append="name"`) (defaults to None).
- **filter** (*ndarray, optional*) – Optional fancy index or boolean array. Does not work with `stack=True`. Designed to read in contiguous sections; for example, `filter=arange(10)` to read first 10 elements (defaults to None).
- **mustexist** (*bool*) – Set to True to ensure that all files exist or raise an exception (defaults to False).
- **verbose** (*bool*) – Prints time related data to stdout (defaults to False).
- **reserve** (*float*) – When set greater than 0.0 and less than 1.0, this is how much extra room is reserved when stacking. If set to 0.10, it will allocate 10% more memory for future partitions. Defaults to 0.0.

**Return type**
*Struct*

### Notes

When `stack` is `True`: - columns with the same name must have matching types or upcastable types - bytestring widths will be fixed internally - numeric types will be upcast appropriately - missing columns will be filled with the invalid value for the column type

### Examples

Stacking multiple files together while loading:

```
>>> files = [ r'D:\dir1\ds1.sds' r'D:\dir2\ds1.sds' ]
>>> load_sds(files, stack=True)
#   col_0   col_1   col_2   col_3   col_4
-   -----   -----   -----   -----   -----
0    0.71    0.86    0.44    0.97    0.47
1    0.89    0.40    0.10    0.94    0.66
2    0.03    0.56    0.80    0.85    0.30
```

Stacking multiple files together while loading, explicitly specifying the list of columns to be loaded.

```
>>> files = [ r'D:\dir1\ds1.sds' r'D:\dir2\ds1.sds' ]
>>> include = ['col_0', 'col_1', 'col_4']
>>> load_sds(files, include=include, stack=True)
#   col_0   col_1   col_4
-   -----   -----   -----
0    0.71    0.86    0.47
1    0.89    0.40    0.66
2    0.03    0.56    0.30
```

Stacking multiple directories together while loading, explicitly specifying the list of `Dataset` objects to load (from each directory, then stack together).

```
>>> files = [ r'D:\dir1', r'D:\dir2' ]
>>> include = [ 'ds1', 'ds2', 'ds3' ]
>>> load_sds(files, include=include, stack=True)
#   Name   Type      Size                0   1   2
-   ----   -------   -----------------   -   -   -
0   ds1    Dataset   20 rows x 10 cols
1   ds2    Dataset   20 rows x 10 cols
2   ds3    Dataset   20 rows x 10 cols
```

**See also:**

*sds_tree*, *sds_info*

riptable.rt_sds.**load_sds_mem**(*filepath*, *share*, *include=None*, *threads=None*, *filter=None*)

Explicitly load data from shared memory.

**Parameters**

- **filepath** (*str or bytes or os.PathLike*) – name of sds file or directory. if no .sds extension, _load_sds will look for _root.sds if no _root.sds is found, extension will be added and shared memory will be checked again.
- **share** (*str*) – shared memory name. For Windows make sure SE_CREATE_GLOBAL_NAME flag is set.
- **include** (*list of str, optional*) –
- **threads** (*int, optional, defaults to None*) – how many threads to used
- **filter** (*int array or bool array, optional, defaults to None*) –

**Return type**

*Struct*, *Dataset* or array loaded from shared memory.

### Notes

To load a single dataset that belongs to a struct, the extension must be included. Otherwise, the path is assumed to be a directory, and the entire Struct is loaded.

riptable.rt_sds.**save_sds**(*filepath*, *item*, *share=None*, *compress=True*, *overwrite=True*, *name=None*, *onefile=False*, *bandsize=None*, *append=None*, *complevel=None*)

Datasets and arrays will be saved into a single .sds file. Structs will create a directory of `.sds` files for potential nested structures.

**Parameters**

- **filepath** (*str or bytes or os.PathLike*) – Path to directory for Struct, path to `.sds` file for Dataset/array (extension will be added if necessary).
- **item** (*Struct, dataset, array, or array subclass*) –
- **share** – If the shared memory name is set, `item` will be saved to shared memory and NOT to disk. When shared memory is specified, a filename must be included in path. Only this will be used, the rest of the path will be discarded. For Windows make sure SE_CREATE_GLOBAL_NAME flag is set.
- **compress** (*bool, default True*) – Use compression when saving the file (shared memory is always saved uncompressed)
- **overwrite** (*bool, default False*) – If `True`, do not prompt the user when overwriting an existing `.sds` file (mainly useful for `Struct.save()`, which may call `Dataset.save()` multiple times)
- **name** (*str, optional*) – Name of the sds file.
- **onefile** (*bool, default False*) – If True will flatten() a nested struct before saving to make it one file.
- **bandsize** (*int, optional*) – If set to an integer greater than 10000 it will compress column datas every `bandsize` rows.
- **append** (*str, optional*) – If set to a string it will append to the file with the section name
- **complevel** (*int, optional*) – Compression level from 0 to 9. 2 (default) is average. 1 is faster, less compressed, 3 is slower, more compressed.

**Raises**

**TypeError** – If `item` type cannot be saved

### Notes

`save()` can also be called from a `Struct` or `Dataset` object.

### Examples

Saving a Struct:

```
>>> st = Struct({ \
    'a': Struct({ \
        'arr' : arange(10), \
        'a2'  : Dataset({ 'col1': arange(5) }) \
    }), \
    'b': Struct({ \
        'ds1' : Dataset({ 'ds1col': arange(6) }), \
        'ds2' : Dataset({ 'ds2col' : arange(7) }) \
    }), \
})
```

```
>>> st.tree()
Struct
    ├──── a (Struct)
    │     ├──── arr int32 (10,) 4
```

(continues on next page)

(continued from previous page)

```
    │      └──── a2 (Dataset)
    │             └──── col1 int32 (5,) 4
    └──── b (Struct)
        ├──── ds1 (Dataset)
        │     └──── ds1col int32 (6,) 4
        └──── ds2 (Dataset)
              └──── ds2col int32 (7,) 4
```

```
>>> save_sds(r'D:\\junk\\nested', st)
>>> os.listdir(r'D:\\junk\\nested')
_root.sds
a!a2.sds
a.sds
b!ds1.sds
b!ds2.sds
```

Saving a Dataset:

```
>>> ds = Dataset({'col_'+str(i):arange(5) for i in range(5)})
>>> save_sds(r'D:\\junk\\test', ds)
>>> os.listdir(r'D:\\junk')
test.sds
```

Saving an Array:

```
>>> a = arange(100)
>>> save_sds('D:\\junk\\test_arr', a)
>>> os.listdir('D:\\junk')
test_arr.sds
```

Saving an Array Subclass:

```
>>> c = Categorical(np.random.choice(['a','b','c'],500))
>>> save_sds(r'D:\\junk\\cat', c)
>>> os.listdir(r'D:\\junk')
cat.sds
```

riptable.rt_sds.**save_struct**(*data=None*, *path=None*, *sharename=None*, *name=None*, *overwrite=True*, *compress=True*, *onefile=False*, *bandsize=None*, *complevel=None*)

riptable.rt_sds.**sds_concat**(*filenames*, *output=None*, *include=None*)

**Parameters**

- **filenames** (*sequence of str or os.PathLike.*) – List of fully qualified path-names
- **output** (*str or os.PathLike, optional*) – Single string of the filename to create (defaults to None).
- **include** (*list of str, optional*) – A list of strings indicating which columns to include in the load (currently not supported). Defaults to None.

**Return type**

A new file created with the name in `output`. This output file has all the filenames appended.

**Raises**

**ValueError** – If output filename is not specified.

### Notes

The `include` parameter is not currently implemented.

### Examples

```
>>> flist=['/nfs/file1.sds', '/nfs/file2.sds', '/nfs/file3.sds']
>>> sds_concat(flist, output='/nfs/mydata/concattest.sds')
>>> sds_load('/nfs/mydata/concattest.sds', stack=True)
```

riptable.rt_sds.**sds_flatten**(*rootpath*)

*sds_flatten* brings all structs and nested structures in sub-directories into the main directory.

**Parameters**

**rootpath** (*str or bytes or os.PathLike*) – The pathname to the SDS root directory.

### Examples

```
>>> sds_flatten(r'D:\junk\PYTHON_SDS')
```

### Notes

- The current implementation of *sds_flatten* crawls one subdirectory.
- If a nested directory contains items that are not sds files, the flatten will be skipped for the nested directory.
- If a there is a name conflict with items already in the base directory, the flatten will be skipped for the nested directory.
- No files will be moved or renamed until all conflicts are checked.
- If there were directories that couldn't be flattened, lists them at the end.

riptable.rt_sds.**sds_info**(*filepath*, *share=None*, *sections=None*, *threads=None*)

riptable.rt_sds.**sds_tree**(*filepath*, *threads=None*)

Explicitly display a tree of data for .sds file or directory. Only loads info, not data.

**Parameters**

- **filepath** (*str or bytes or os.PathLike*) –
- **threads** (*int, optional*) –

### Examples

```
>>> ds = Dataset({'col_'+str(i):arange(5) for i in range(5)})
>>> ds.save(r'D:\junk\treeds')
>>> sds_tree(r'D:\junk\treeds')
treeds
  ├──── col_0 FA  (5,)  int32  i4
  ├──── col_1 FA  (5,)  int32  i4
  ├──── col_2 FA  (5,)  int32  i4
  ├──── col_3 FA  (5,)  int32  i4
  └──── col_4 FA  (5,)  int32  i4
```

## 2.2.38 riptable.rt_sharedmemory

### Classes

| | |
|---|---|
| *SharedMemory* | |

**class** riptable.rt_sharedmemory.**SharedMemory**

## 2.2.39 riptable.rt_sort_cache

### Classes

| | |
|---|---|
| *SortCache* | Global sort cache for uid - unique ids which are often generated from GetTSC (CPU time stamp counter) |

**class** riptable.rt_sort_cache.**SortCache**

Bases: object

Global sort cache for uid - unique ids which are often generated from GetTSC (CPU time stamp counter)

to ensure that the values have not changed underneath, it performs a crc check and compares via the crc of a known sorted index array

**_cache**

**_logging = False**

**classmethod get_sorted_row_index**(*uid*, *nrows*, *sortdict*)

**classmethod invalidate**(*uid*)

**classmethod invalidate_all**()

**classmethod logging_off**()

**classmethod logging_on**()

**classmethod store_sort**(*uid*, *sortlist*, *sortidx*)

Restore a sort index from file.

### 2.2.40 riptable.rt_stats

#### Functions

| | |
|---|---|
| *class_error*(X, Y) | |
| *groupScatter*(*arg, **kwarg) | This function has been moved to playa.stats. |
| *linear_spline*(X0, Y0, knots[, display]) | |
| *lm*(X, Y[, intercept, removeNaN, displayStats]) | |
| *mae*(X, Y) | |
| *plotPrediction*(X, Yhat, Y, N[, lb, ub]) | |
| *plot_hist*(Y, bins) | |
| *r2*(X, Y) | |
| *statx*(X) | |
| *winsorize*(Y, lb, ub) | |

riptable.rt_stats.**class_error**(*X*, *Y*)

riptable.rt_stats.**groupScatter**(**arg*, ***kwarg*)

This function has been moved to playa.stats.

riptable.rt_stats.**linear_spline**(*X0*, *Y0*, *knots*, *display=True*)

riptable.rt_stats.**lm**(*X*, *Y*, *intercept=True*, *removeNaN=True*, *displayStats=True*)

riptable.rt_stats.**mae**(*X*, *Y*)

riptable.rt_stats.**plotPrediction**(*X*, *Yhat*, *Y*, *N*, *lb=None*, *ub=None*)

riptable.rt_stats.**plot_hist**(*Y*, *bins*)

riptable.rt_stats.**r2**(*X*, *Y*)

riptable.rt_stats.**statx**(*X*)

riptable.rt_stats.**winsorize**(*Y*, *lb*, *ub*)

### 2.2.41 riptable.rt_str

#### Classes

| | |
|---|---|
| *CatString* | Provides access to FAString methods for Categoricals. |
| *FAString* | String accessor class for FastArray. |

**class** riptable.rt_str.**CatString**(*cat*)

Provides access to FAString methods for Categoricals. All string methods are wrappers of the FAString equivalent with categorical re-expansion and option for how to fill filtered elements.

**property _isfiltered**

**property substr**

**classmethod _build_method**(*method*)

General purpose factory for FAString function wrappers.

**classmethod _build_property**(*name*)

General purpose factory for FAString property wrappers.

**_convert_fastring_output**(*out*)

**extract**(*regex*, *expand=None*, *fillna=''*, *names=None*)

**class** riptable.rt_str.**FAString**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: *riptable.rt_fastarray.FastArray*

String accessor class for FastArray.

**property backtostring**

convert back to FastArray or np.ndarray 'S' or 'U' string 'S12' or 'U40'

**property lower**

upper case a string (bytes or unicode) makes a copy

##### Examples

```
>>> FAString(['THIS','THAT','TEST']).lower
FastArray(['this','that','test'], dtype='<U4')
```

**property n_elements**

The number of elements in the original string array

**property reverse**

upper case a string (bytes or unicode) does not make a copy

### Examples

FAString(['this','that','test']).reverse

**property reverse_inplace**

upper case a string (bytes or unicode) does not make a copy

### Examples

FAString(['this','that','test']).reverse_inplace

**property str**

Casts an array of byte strings or unicode as `FAString`.

Enables a variety of useful string manipulation methods.

**Return type**
*FAString*

**Raises**
`TypeError` – If the FastArray is of dtype other than byte string or unicode

**See also:**

`np.chararray`, `np.char`, `rt.FAString.apply`

### Examples

```
>>> s=FA(['this','that','test ']*100_000)
>>> s.str.upper
FastArray([b'THIS', b'THAT', b'TEST ', ..., b'THIS', b'THAT', b'TEST '],
          dtype='|S5')
```

```
>>> s.str.lower
FastArray([b'this', b'that', b'test ', ..., b'this', b'that', b'test '],
          dtype='|S5')
```

```
>>> s.str.removetrailing()
FastArray([b'this', b'that', b'test', ..., b'this', b'that', b'test'],
          dtype='|S5')
```

**property strlen**

return the string length of every string (bytes or unicode)

### Examples

```
>>> FAString(['this  ','that ','test']).strlen
FastArray([6, 5, 4])
```

**property substr**

**property upper**

upper case a string (bytes or unicode) makes a copy

#### Examples

```
>>> FAString(['this','that','test']).upper
FastArray(['THIS','THAT','TEST'], dtype='<U4')
```

**property upper_inplace**

upper case a string (bytes or unicode) does not make a copy

#### Examples

FAString([‘this’,’that’,’test’]).upper_inplace

**_APPLY_PARALLEL_THRESHOLD = 10000**

**nb_char**

**nb_char_par**

**nb_contains**

**nb_contains_par**

**nb_endswith**

**nb_endswith_par**

**nb_find**

**nb_index**

**nb_index_any_of**

**nb_index_any_of_par**

**nb_index_par**

**nb_lower**

**nb_lower_par**

**nb_removetrailing**

**nb_removetrailing_par**

**nb_replace**

**nb_replace_par**

**nb_reverse**

**nb_reverse_inplace**

**nb_reverse_inplace_par**

**nb_reverse_par**

**nb_startswith**

**nb_startswith_par**

**nb_strlen**

**nb_strlen_par**

**nb_substr**

**nb_substr_par**

**nb_upper**

**nb_upper_inplace**

**nb_upper_inplace_par**

**nb_upper_par**

**_apply_func**(*func*, *funcp*, **args*, *dtype=None*, *input=None*)

**_find**(*str2*)

Searches src for occurences of str2 and build a Boolean mask the same size as src indicating the starting point of all such occurences.

**Parameters**

**for** (*str2 - a string with one or more characters to search*) –

### Examples

```
>>> FAString(['this','that','test']).find('t')
FastArray([
    [True, False, False, False],
    [True, False, False, True],
    [True, False, False, True]
])
```

**_nb_char**(*position*, *itemsize*, *strlen*, *out*)

**_nb_contains**(*itemsize*, *dest*, *str2*)

**_nb_endswith**(*itemsize*, *dest*, *str2*)

**_nb_find**(*itemsize*, *dest*, *str2*)

Searches src for occurrences of str2 and build a Boolean array with a row per string indicating indicating the starting points of all such occurrences.

**_nb_index**(*itemsize*, *dest*, *str2*)

**_nb_index_any_of**(*itemsize*, *dest*, *str2*)

**_nb_lower**(*itemsize*, *dest*)

**_nb_removetrailing**(*itemsize*, *dest*, *removechar*)

**_nb_replace**(*itemsize*, *dest*, *dest_itemsize*, *old*, *new*, *locations*)

**_nb_reverse**(*itemsize*, *dest*)

**_nb_reverse_inplace**(*itemsize*)

**_nb_startswith**(*itemsize*, *dest*, *str2*)

**_nb_strlen**(*itemsize*, *dest*)

**_nb_substr**(*out*, *itemsize*, *start*, *stop*, *strlen*)

**_nb_upper**(*itemsize*, *dest*)

**_nb_upper_inplace**(*itemsize*)

**_substr**(*start*, *stop=None*)

Take a substring of each element using slice args. Behaves like slice, such that a single argument is treated as the stop. start, stop may be integers or arrays of integers aligned with self.

### Examples

```
>>> a = rt.FA(['abc', 'xyzQ'])
>>> a.str.substr(2)
FastArray([b'ab', b'xy'], dtype='|S2')
>>> a.str.substr(0, 2)
FastArray([b'ab', b'xy'], dtype='|S2')
>>> a.str.substr(1, 2)
FastArray([b'b', b'y'], dtype='|S2')
>>> a.str.substr([1, 2])    # element-wise bounds
FastArray([b'a', b'xy'], dtype='|S2')
```

**_validate_input**(*str2*)

**apply**(*func*, **args*, *dtype=None*)

Write your own string apply function NOTE: byte strings are passed as uint8 NOTE: unicode strings are passed as uint32

default signature must match

@nb.njit(cache=get_global_settings().enable_numba_cache, nogil=True) def nb_upper(src, itemsize, dest):

src: is uint array itemsize: is how wide the string is per row dest: is return uint array

**Parameters**

- ***args** (*pass in zero or more arguments (the arguments are always at the end)*) –
- **dtype** (*specify a different dtype*) –

### Example

```
>>> import numba as nb
... @nb.njit(cache=get_global_settings().enable_numba_cache, nogil=True)
... def nb_upper(src, itemsize, dest):
...     for i in nb.prange(len(src) / itemsize):
...         rowpos = i * itemsize
...        for j in range(itemsize):
...             c=src[rowpos+j]
...             if c >= 97 and c <= 122:
...                 # convert to ASCII upper
...                 dest[rowpos+j] = c-32
...             else:
...                dest[rowpos+j] = c
```

```
>>> FAString(['this  ','that ','test']).apply(nb_upper)
```

**char**(*position*)

Take a single character from each element.

**Parameters**

**position** (*int or list of int or np.ndarray*) – The position of the character to be extracted. Negative values respect the length of the individual strings. If an array, the length must be equal to the number of strings. An error is raised if any positions are out of bounds (>= self._itemsize).

**contains**(*str2*)

Return a boolean array that's True for each string element that contains the given substring, otherwise False.

The entire substring must match.

**Parameters**

**str2** (*str*) – A string with one or more characters to search for. To search using regular expressions, use *FAString.regex_match()*.

**Returns**

A boolean array where the value is True if the string contains the entire substring specified in `str2`, otherwise False.

**Return type**

`FastArray`

**See also:**

*FAString.startswith*, *FAString.endswith*, *FAString.regex_match*

### Examples

```
>>> FAString(['this  ','that ','test']).contains('at')
FastArray([False, True, False])
```

This can be called on a FastArray using .str.contains().

```
>>> a = rt.FastArray(['this  ','that ','test'])
>>> a.str.contains('at')
FastArray([False,  True, False])
```

**endswith**(*str2*)

Return a boolean array that's True where the given substring matches the end of each string element, otherwise False.

The entire substring must match.

**Parameters**

**str2** (*str*) – A string with one or more characters to search for. To search using regular expressions, use *FAString.regex_match()*.

**Returns**

A boolean array where the value is True if the string ends with the entire substring specified in str2, otherwise False.

**Return type**

FastArray

**See also:**

*FAString.startswith*, *FAString.contains*, *FAString.regex_match*

### Examples

```
>>> FAString(['abab','ababa','abababb']).endswith('ab')
FastArray([True, False, False])
```

This can be called on a FastArray using .str.endswith().

```
>>> a = rt.FastArray(['abab','ababa','abababb'])
>>> a.str.endswith('ab')
FastArray([True, False, False])
```

**extract**(*regex*, *expand=None*, *fillna=''*, *names=None*, *apply_unique=True*)

Extract one or more pattern groups from each element of an array into a FastArray or Dataset.

This is useful when you have pieces of data in a string that you want to split into separate elements.

For one capture group, the default is to return a FastArray, but this can be overridden by setting expand to True or by providing a name of a Dataset column to populate. For more than one capture group, a Dataset is returned.

Column names for the resulting Dataset can be specified within the regex using (?P<name>) in the capture group(s) or by passing the names argument, which may be more convenient.

**Parameters**

- **regex** (*str*) – The pattern(s) to search for. Define multiple capture groups using parentheses.
- **expand** (*bool, default False*) – Set to True to return a `Dataset` for a single capture group. If False, a `FastArray` is returned.
- **fillna** (*str, default '' (empty string)*) – For elements where there's no match, this is the fill value for the resulting `FastArray` or `Dataset` column.
- **names** (*list of str, default None*) – For more than one capture group, a `Dataset` is returned. Optionally, you can provide column names (keys) for the extracted data.
- **apply_unique** (*bool*) – When True, the regex is applied to the unique values and then expanded using the reverse index (see `riptable.unique()`). This is optimal for repetitive data and benign for unique or highly non-repetitive data.

**Returns**

For one capture group, a `FastArray` (or optionally a `Dataset`) is returned. For more than one capture group, a `Dataset` is returned.

**Return type**

`FastArray` or `Dataset`

**See also:**

*FAString.regex_match*
Return a boolean array that indicates whether given string or regular expression pattern is contained in each string element.

*FAString.regex_replace*
Replace each instance of a specified string or pattern.

### Examples

These examples use a `FastArray` containing OSI symbols.

```
>>> osi = rt.FastArray(['SPX UO 12/15/23 C5700', 'SPXW UO 09/17/21 C3650'])
```

Extract one substring:

```
>>> osi.str.extract('\w+')
FastArray([b'SPX', b'SPXW'], dtype='|S4')
```

Provide a name for the resulting `Dataset` column:

```
>>> osi.str.extract('(?P<root>\w+)')
#   root
-   ----
0   SPX
1   SPXW
```

Define two capture groups and provide names for the resulting `Dataset` columns:

```
>>> osi.str.extract('(\w+).* (\d{2}/\d{2}/\d{2})', names = ['root', 'expiration
↪'])
#   root   expiration
```

(continues on next page)

(continued from previous page)

```
-   ----   ----------
0   SPX    12/15/23
1   SPXW   09/17/21
```

Extract one substring into a `Dataset` column using `expand = True`. (Note that for the element with an unmatched pattern, an empty string is returned).

```
>>> osi.str.extract('\w+W', expand = True)
#   group_0
-   -------
0
1   SPXW
```

**index**(*str2*)

return the first index location of the entire substring specified in str2, or -1 if the substring does not exist

**Parameters**
**for** (*str2 - a string with one or more characters to search*) –

### Examples

```
>>> FAString(['this  ','that ','test']).index('at')
FastArray([-1, 2, -1])
```

**index_any_of**(*str2*)

return the first index location any of the characters that are part of str2, or -1 if none of the characters match

**Parameters**
**for** (*str2 - a string with one or more characters to search*) –

### Examples

```
>>> FAString(['this  ','that ','test']).index_any_of('ia')
FastArray([2, 2, -1])
```

**possibly_convert_tostr**(*arr*)

converts list like or an array to the same string type

**regex_match**(*regex*, *apply_unique=True*)

Return a boolean array that's True where the given substring or regular expression pattern is contained in each string element, otherwise False.

The entire substring or pattern must match.

Applies `re.search()` on each element with `regex` as the pattern.

**Parameters**

- **regex** (*str*) – String or regular expression pattern to search for.
- **apply_unique** (*bool, default True*) – When True, the regex is applied to the unique values and then expanded using the reverse index (see `riptable.unique()`). This is optimal for repetitive data and benign for unique or highly non-repetitive data.

**Returns**
A boolean array where the value is True if the string element contains the entire substring or regex pattern specified in `regex`, otherwise False.

**Return type**
`FastArray`

**See also:**

*FAString.regex_replace*
Replace each instance of a specified substring or pattern.

*FAString.extract*
Extract one or more pattern groups into a `Dataset` or `FastArray`.

*FAString.contains*, *FAString.startswith*, *FAString.endswith*

### Examples

Find any instance of 'ab' that appears at the end of a string:

```
>>> FAString(['abab','ababa','abababb']).regex_match('ab$')
FastArray([True, False, False])
```

This can be called on a `FastArray` using `.str.regex_match()`.

```
>>> a = rt.FastArray(['abab','ababa','abababb'])
>>> a.str.regex_match('ab$')
FastArray([True, False, False])
```

**regex_replace**(*regex*, *repl*, *apply_unique=True*)

Replace each instance of a specified substring or pattern.

The entire substring or pattern must match. If the substring or pattern isn't found, the original string is returned unchanged.

The behavior is identical to that of `re.sub()`. In particular, the returned string is obtained by replacing the leftmost non-overlapping occurrences of the substring or pattern with the replacement string.

**Parameters**

- **regex** (*str*) – String or regular expression pattern to search for.
- **repl** (*str*) – The replacement string.
- **apply_unique** (*bool, default True*) – When True, the regex is applied to the unique values and then expanded using the reverse index (see `riptable.unique()`). This is optimal for repetitive data and benign for unique or highly non-repetitive data.

**Returns**
An array with all occurrences of the substring or pattern replaced.

**Return type**
`FastArray`

**See also:**

*FAString.regex_match*
Return a boolean array that indicates whether given substring or regular expression pattern is contained in each string element.

*FAString.extract*
Extract one or more pattern groups into a `Dataset` or `FastArray`.

*FAString.contains*, *FAString.startswith*, *FAString.endswith*

### Examples

Replace instances of 'aa' with 'b'. All non-overlapping occurrences are replaced, starting from the left:

```
>>> FAString(['aaa', 'aaaa', 'aaaaa']).regex_replace('aa', 'b')
FastArray(['ba', 'bb', 'bba'], dtype='<U3')
```

Replace any instance of 'ab' that appears at the end of a string with 'b'.

```
>>> FAString(['abab','ababa','abababb']).regex_replace('ab$', 'b')
FastArray(['abb', 'ababa', 'abababb'], dtype='<U7')
```

This can be called on a FastArray using `.str.regex_replace()`. The returned `FastArray` elements are byte strings.

```
>>> a = rt.FastArray(['abab','ababa','abababb'])
>>> a.str.regex_replace('ab$', 'b')
FastArray([b'abb', b'ababa', b'abababb'], dtype='|S7')
```

**removetrailing**(*remove=32*)

removes spaces at end of string (often to fixup matlab string) makes a copy

**Parameters**
**character)** (*remove=32. defaults to removing ascii 32 (space)* –

### Examples

```
>>> FAString(['this  ','that ','test']).removetrailing()
FastArray(['this','that','test'], dtype='<U6')
```

**replace**(*old*, *new*)

Replace all occurrences of `old` with `new`

**startswith**(*str2*)

Return a boolean array that's True where the given substring matches the start of each string element, otherwise False.

The entire substring must match.

**Parameters**
**str2** (*str*) – A string with one or more characters to search for. To search using regular expressions, use *FAString.regex_match()*.

**Returns**
A boolean array where the value is True if the string starts with the entire substring specified in `str2`, otherwise False.

**Return type**
`FastArray`

**See also:**

*FAString.endswith*, *FAString.contains*, *FAString.regex_match*

### Examples

```
>>> FAString(['this  ','that ','test']).startswith('thi')
FastArray([True, False, False])
```

This can be called on a FastArray using .str.startswith().

```
>>> a = rt.FastArray(['this  ','that ','test'])
>>> a.str.startswith('thi')
FastArray([True, False, False])
```

**strpbrk**(*str2*)

**strstr**(*str2*)

**strstrb**(*str2*)

**substr_char_stop**(*stop*, *inclusive=False*)

Take a substring of each element using characters as bounds.

**Parameters**

- **stop** – A string used to determine the start of the sub-string. Excluded from the result by default. We go to the end of the string where stop is not in found in the corresponding element
- **inclusive** (*bool*) – If True, include the stopping string in the result

### Examples

```
>>> s = FastArray(['ABC', 'A_B', 'AB_C', 'AB_C_DD'])
>>> s.str.substr_char_stop('_')
FastArray([b'ABC', b'A', b'AB', b'AB'], dtype='|S2')
>>> s.str.substr_char_stop('_', inclusive=True)
FastArray([b'ABC', b'A_', b'AB_', b'AB_'], dtype='|S2')
```

## 2.2.42 riptable.rt_struct

### Classes

| | |
|---|---|
| *Struct* | The Struct class is at the root of much of the riptable class design; both Dataset and Multiset |

**class** riptable.rt_struct.**Struct**(*dictionary={}*)

The Struct class is at the root of much of the riptable class design; both Dataset and Multiset inherit from Struct.

Struct represents a collection of (mixed-type) data members, with standard attribute get/set behavior, as well as dictionary-style retrieval.

The Struct constructor takes a dictionary (dict, OrderedDict, etc...) as its required argument. When *Struct.UseFastArray* is True (the default), any numpy arrays among the dictionary values will be cast into FastArray. Struct() := Struct({}).

The constructor dictionary keys (or element/column names added later) must not conflict with any Struct member names. Additionally, if *Struct.AllowAnyName* is False (it is True by default), a column name must be a legal Python variable name, not starting with '_'.

**Parameters**

**dictionary** (*dict*) – A dictionary of named objects.

### Examples

```
>>> st = rt.Struct({'a': 1, 'b': 'fish', 'c': [5.6, 7.8], 'd': {'A': 'david', 'B':
↪'matthew'},
... 'e': np.ones(7, dtype=np.int64)})
>>> print(st)
#   Name   Type    Size   0      1     2
-   ----   -----   ----   ----   ---   -
0   a      int     0      1
1   b      str     0      fish
2   c      list    2      5.6    7.8
3   d      dict    2      A      B
4   e      int64   7      1      1     1
>>> st.a
1
>>> st['a']
1
>>> print(st[3:])
#   Name   Type    Rows   0   1   2
-   ----   -----   ----   -   -   -
0   d      dict    2      A   B
1   e      int64   7      1   1   1
>>> st.newcol = 5  # okay, a new entry
>>> st.newcol = [5, 7]  # okay, replace the entry
>>> st['another'] = 6  # also works
>>> st['newcol'] = 6  # and this works as well
```

**Indexing behavior**

```
>>> st['b'] # get a 'column' (equiv. st.b)
>>> st[['a', 'e']] # get some columns
>>> st[[0, 4]] # get some columns (order is that of iterating st (== list(st))
>>> st[1:5:2] # standard slice notation, indexing corresponding to previous
>>> st[bool_vector] # get 'True' columns
```

**Equivalents**

```
>>> assert len(st) == st.get_ncols()
>>> for _k in st: print(_k, st[_k])
>>> for _k, _v in st.items(): print(_k, _v)
>>> for _k, _v in zip(st.keys(), st.values()): print(_k, _v)
>>> for _k, _v in zip(st, st.values()): print(_k, _v)
```

(continues on next page)

(continued from previous page)

```
>>> if key in st:
...     assert getattr(st, key) is st[key]
```

**Context manager**

```
>>> with Struct({'a': 1, 'b': 'fish'}) as st:
...     st.a)
>>> assert not hasattr(st, 'a')
```

**property _A**

Display all columns, all rows (up to 10,000), and long strings of a *Dataset* or *Struct*.

Without this property, columns are elided when the maximum display width is reached, rows are elided when there are more then 30 to display, and strings are truncated after 15 characters.

**Returns**

A wrapper for display operations that don't return a *Dataset* or *Struct*.

**Return type**

*DisplayString*

**See also:**

***Struct._V***

Display all rows of a *Dataset* or *Struct*.

***Struct._H***

Display all columns and long strings of a *Dataset* or *Struct*.

***Struct._G***

Display all columns of a *Dataset* or *Struct*, wrapping the table as needed.

***Struct._T***

Display a transposed view of a *Dataset* or *Struct*.

### Examples

```
>>> ds = rt.Dataset({'col_'+str(i):rt.arange(31) for i in range(12)})
>>> ds[0] = 'long_string_long_string'
```

By default, columns are elided when the maximum display width is reached, rows are elided when there are more then 30 to display, and strings are truncated after 15 characters:

```
>>> ds
  #   col_0             col_1   col_2   col_3   col_4   col_5   ...   col_7  ␣
↪col_8   col_9   col_10   col_11
---   ---------------   -----   -----   -----   -----   -----   ---   -----  -
↪----   -----   ------   ------
  0   long_string_lon       0       0       0       0       0   ...       0  ␣
↪   0       0        0        0
  1   long_string_lon       1       1       1       1       1   ...       1  ␣
↪   1       1        1        1
  2   long_string_lon       2       2       2       2       2   ...       2  ␣
↪   2       2        2        2
  3   long_string_lon       3       3       3       3       3   ...       3  ␣
```

(continues on next page)

(continued from previous page)

```
↪   3       3        3        3
 4   long_string_lon       4       4       4       4       4   ...       4  ␣
↪   4       4        4        4
 5   long_string_lon       5       5       5       5       5   ...       5  ␣
↪   5       5        5        5
 6   long_string_lon       6       6       6       6       6   ...       6  ␣
↪   6       6        6        6
 7   long_string_lon       7       7       7       7       7   ...       7  ␣
↪   7       7        7        7
 8   long_string_lon       8       8       8       8       8   ...       8  ␣
↪   8       8        8        8
 9   long_string_lon       9       9       9       9       9   ...       9  ␣
↪   9       9        9        9
10   long_string_lon      10      10      10      10      10   ...      10  ␣
↪  10      10       10       10
11   long_string_lon      11      11      11      11      11   ...      11  ␣
↪  11      11       11       11
12   long_string_lon      12      12      12      12      12   ...      12  ␣
↪  12      12       12       12
13   long_string_lon      13      13      13      13      13   ...      13  ␣
↪  13      13       13       13
14   long_string_lon      14      14      14      14      14   ...      14  ␣
↪  14      14       14       14
...   ...                ...     ...     ...     ...     ...   ...     ...  ␣
↪ ...     ...      ...      ...
16   long_string_lon      16      16      16      16      16   ...      16  ␣
↪  16      16       16       16
17   long_string_lon      17      17      17      17      17   ...      17  ␣
↪  17      17       17       17
18   long_string_lon      18      18      18      18      18   ...      18  ␣
↪  18      18       18       18
19   long_string_lon      19      19      19      19      19   ...      19  ␣
↪  19      19       19       19
20   long_string_lon      20      20      20      20      20   ...      20  ␣
↪  20      20       20       20
21   long_string_lon      21      21      21      21      21   ...      21  ␣
↪  21      21       21       21
22   long_string_lon      22      22      22      22      22   ...      22  ␣
↪  22      22       22       22
23   long_string_lon      23      23      23      23      23   ...      23  ␣
↪  23      23       23       23
24   long_string_lon      24      24      24      24      24   ...      24  ␣
↪  24      24       24       24
25   long_string_lon      25      25      25      25      25   ...      25  ␣
↪  25      25       25       25
26   long_string_lon      26      26      26      26      26   ...      26  ␣
↪  26      26       26       26
27   long_string_lon      27      27      27      27      27   ...      27  ␣
↪  27      27       27       27
28   long_string_lon      28      28      28      28      28   ...      28  ␣
↪  28      28       28       28
29   long_string_lon      29      29      29      29      29   ...      29  ␣
```

(continues on next page)

(continued from previous page)

```
↪  29      29       29       29
 30   long_string_lon      30      30      30      30      30   ...      30   ␣
↪  30      30       30       30
```

Display all columns, rows, and long strings:

```
>>> ds._A
  #   col_0                    col_1   col_2   col_3   col_4   col_5   col_6 ␣
↪ col_7   col_8   col_9   col_10   col_11
---   ----------------------   -----   -----   -----   -----   -----   ----- ␣
↪ -----   -----   -----   ------   ------
  0   long_string_long_string       0       0       0       0       0       0 ␣
↪     0       0       0        0        0
  1   long_string_long_string       1       1       1       1       1       1 ␣
↪     1       1       1        1        1
  2   long_string_long_string       2       2       2       2       2       2 ␣
↪     2       2       2        2        2
  3   long_string_long_string       3       3       3       3       3       3 ␣
↪     3       3       3        3        3
  4   long_string_long_string       4       4       4       4       4       4 ␣
↪     4       4       4        4        4
  5   long_string_long_string       5       5       5       5       5       5 ␣
↪     5       5       5        5        5
  6   long_string_long_string       6       6       6       6       6       6 ␣
↪     6       6       6        6        6
  7   long_string_long_string       7       7       7       7       7       7 ␣
↪     7       7       7        7        7
  8   long_string_long_string       8       8       8       8       8       8 ␣
↪     8       8       8        8        8
  9   long_string_long_string       9       9       9       9       9       9 ␣
↪     9       9       9        9        9
 10   long_string_long_string      10      10      10      10      10      10 ␣
↪    10      10      10       10       10
 11   long_string_long_string      11      11      11      11      11      11 ␣
↪    11      11      11       11       11
 12   long_string_long_string      12      12      12      12      12      12 ␣
↪    12      12      12       12       12
 13   long_string_long_string      13      13      13      13      13      13 ␣
↪    13      13      13       13       13
 14   long_string_long_string      14      14      14      14      14      14 ␣
↪    14      14      14       14       14
 15   long_string_long_string      15      15      15      15      15      15 ␣
↪    15      15      15       15       15
 16   long_string_long_string      16      16      16      16      16      16 ␣
↪    16      16      16       16       16
 17   long_string_long_string      17      17      17      17      17      17 ␣
↪    17      17      17       17       17
 18   long_string_long_string      18      18      18      18      18      18 ␣
↪    18      18      18       18       18
 19   long_string_long_string      19      19      19      19      19      19 ␣
↪    19      19      19       19       19
 20   long_string_long_string      20      20      20      20      20      20 ␣
```

(continues on next page)

(continued from previous page)

```
↪    20      20      20       20       20
 21   long_string_long_string      21       21      21      21      21      21 ␣
↪    21      21      21       21       21
 22   long_string_long_string      22       22      22      22      22      22 ␣
↪    22      22      22       22       22
 23   long_string_long_string      23       23      23      23      23      23 ␣
↪    23      23      23       23       23
 24   long_string_long_string      24       24      24      24      24      24 ␣
↪    24      24      24       24       24
 25   long_string_long_string      25       25      25      25      25      25 ␣
↪    25      25      25       25       25
 26   long_string_long_string      26       26      26      26      26      26 ␣
↪    26      26      26       26       26
 27   long_string_long_string      27       27      27      27      27      27 ␣
↪    27      27      27       27       27
 28   long_string_long_string      28       28      28      28      28      28 ␣
↪    28      28      28       28       28
 29   long_string_long_string      29       29      29      29      29      29 ␣
↪    29      29      29       29       29
 30   long_string_long_string      30       30      30      30      30      30 ␣
↪    30      30      30       30       30
```

**property _G**

Display all columns of a *Dataset* or *Struct*, wrapping the table after the maximum display width is reached.

Note: The table is displayed as text, not HTML.

**Return type**
None

**See also:**

***Struct._V***
Display all rows of a *Dataset* or *Struct*.

***Struct._H***
Display all columns and long strings of a *Dataset* or *Struct*.

***Struct._A***
Display all columns, rows, and long strings of a *Dataset* or *Struct*.

***Struct._T***
Display a transposed view of a *Dataset* or *Struct*.

### Examples

```
>>> ds = rt.Dataset(
...     {key: rt.FA([i, 2 * i, 3 * i, 4 * i]) % 3 == 0 for i, key in enumerate(
↪'abcdefghijklmno')}
... )
```

Default behavior:

```
>>> ds
#      a       b       c      d       e       f      g    ...      j       k   ␣
↪   l      m       n       o
-   ----   -----   -----   ----   -----   -----   ----   ---   ----   -----   -
↪----   ----   -----   -----
0   True   False   False   True   False   False   True   ...   True   False  ␣
↪False   True   False   False
1   True   False   False   True   False   False   True   ...   True   False  ␣
↪False   True   False   False
2   True    True    True   True    True    True   True   ...   True    True  ␣
↪True   True    True    True
3   True   False   False   True   False   False   True   ...   True   False  ␣
↪False   True   False   False
```

Show all rows, wrapping the table as needed:

```
>>> ds._G
#      a       b       c      d       e       f      g       h       i      j ␣
↪     k       l      m
-   ----   -----   -----   ----   -----   -----   ----   -----   -----   ---- ␣
↪ -----   -----   ----
0   True   False   False   True   False   False   True   False   False   True ␣
↪ False   False   True
1   True   False   False   True   False   False   True   False   False   True ␣
↪ False   False   True
2   True    True    True   True    True    True   True    True    True   True ␣
↪  True    True   True
3   True   False   False   True   False   False   True   False   False   True ␣
↪ False   False   True

#       n       o
-   -----   -----
0   False   False
1   False   False
2    True    True
3   False   False
```

**property _H**

Display all columns and long strings of a *Dataset* or *Struct*.

Without this property, columns are elided when the maximum display width is reached, and strings are truncated after 15 characters.

**Returns**

A wrapper for display operations that don't return a *Dataset* or *Struct*.

**Return type**
*DisplayString*

**See also:**

***Struct._V***
Display all rows of a *Dataset* or *Struct*.

***Struct._A***
Display all columns, rows, and long strings of a *Dataset* or *Struct*.

***Struct._G***
Display all columns of a *Dataset* or *Struct*, wrapping the table as needed.

***Struct._T***
Display a transposed view of a *Dataset* or *Struct*.

### Examples

By default, columns are elided when the maximum display width is reached, and strings are truncated after 15 characters.

```
>>> ds = rt.Dataset({key : rt.FA([i, 2*i, 3*i, 4*i])%3 == 0 for i, key in␣
↪enumerate('abcdefghijklm')})
>>> ds[0] = rt.FA('long_string_long_string')
>>> ds
#   a                    b       c      d       e       f   ...       h      ␣
↪i      j       k       l      m
-   ---------------   -----   -----   ----   -----   -----   ---   -----   ----
↪-   ----   -----   -----   ----
0   long_string_lon   False   False   True   False   False   ...   False  ␣
↪False   True   False   False   True
1   long_string_lon   False   False   True   False   False   ...   False  ␣
↪False   True   False   False   True
2   long_string_lon    True    True   True    True    True   ...    True  ␣
↪True   True    True    True   True
3   long_string_lon   False   False   True   False   False   ...   False  ␣
↪False   True   False   False   True
```

Display all columns and long strings:

```
>>> ds._H
#   a                           b       c      d       e       f      g     ␣
↪ h       i      j       k       l      m
-   ----------------------   -----   -----   ----   -----   -----   ----   ---
↪--   -----   ----   -----   -----   ----
0   long_string_long_string   False   False   True   False   False   True  ␣
↪False   False   True   False   False   True
1   long_string_long_string   False   False   True   False   False   True  ␣
↪False   False   True   False   False   True
2   long_string_long_string    True    True   True    True    True   True   ␣
↪True    True   True    True    True   True
3   long_string_long_string   False   False   True   False   False   True  ␣
↪False   False   True   False   False   True
```

**property _T**

Display a transposed view of the *Dataset* or *Struct*.

All columns are shown as rows and vice-versa. Strings up to 32 characters are fully displayed.

**Returns**
A wrapper for display operations that don't return a *Dataset* or *Struct*.

**Return type**
*DisplayString*

**See also:**

***Struct.dtranspose***
Called by this method.

***Struct._V***
Show all rows of a *Dataset* or *Struct*.

***Struct._H***
Show all columns and long strings of a *Dataset* or *Struct*.

***Struct._A***
Show all columns, rows, and long strings of a *Dataset* or *Struct*.

***Struct._G***
Show all columns of a *Dataset* or *Struct*, wrapping the table as needed.

### Examples

```
>>> ds = rt.Dataset({'a': [1, 2, 3], 'b' : ['longstring_longstring_longstring_
↪longstring',
...                  'fish', 'david']})
>>> ds
#   a   b
-   -   ---------------
0   1   longstring_long
1   2   fish
2   3   david
>>> ds._T
Fields:                                0     1      2
      a                                1     2      3
      b  longstring_longstring_lonstring  fish  david
```

**property _V**

Display all rows (up to 10,000) of a *Dataset* or *Struct*.

Without this property, rows are elided when there are more than 30 to display.

**Returns**
A wrapper for display operations that don't return a *Dataset* or *Struct*.

**Return type**
*DisplayString*

**See also:**

***Struct._H***
Display all columns and long strings of a *Dataset* or *Struct*.

*Struct._A*
Display all columns, rows, and long strings of a *Dataset* or *Struct*.

*Struct._G*
Display all columns of a *Dataset* or *Struct*, wrapping the table as needed.

*Struct._T*
Display a transposed view of a *Dataset* or *Struct*.

### Examples

By default, rows are elided when there are more than 30 to display.

```
>>> ds = rt.Dataset({'a' : rt.arange(31)})
>>> ds
  #     a
---   ---
  0     0
  1     1
  2     2
  3     3
  4     4
  5     5
  6     6
  7     7
  8     8
  9     9
 10    10
 11    11
 12    12
 13    13
 14    14
...   ...
 16    16
 17    17
 18    18
 19    19
 20    20
 21    21
 22    22
 23    23
 24    24
 25    25
 26    26
 27    27
 28    28
 29    29
 30    30
```

Display all rows:

```
>>> ds._V
  #     a
---   ---
```

(continues on next page)

(continued from previous page)

```
 0     0
 1     1
 2     2
 3     3
 4     4
 5     5
 6     6
 7     7
 8     8
 9     9
10    10
11    11
12    12
13    13
14    14
15    15
16    16
17    17
18    18
19    19
20    20
21    21
22    22
23    23
24    24
25    25
26    26
27    27
28    28
29    29
30    30
```

**property _row_numbers**

Subclasses can define their own callback function to customize the left side of the table. If not defined, normal row numbers will be displayed

**Parameters**

- **arr** (*array*) – Fancy index array of row numbers
- **style** (ColumnStyle) – Default style object for final row numbers column.

**Returns**

- **header** (*string*)
- **label_array** (*ndarray*)
- **style** (ColumnStyle)

**property _sort_columns**

Subclasses can define their own callback function to return columns they were sorted by, and styles. Callback function will receive trimmed fancy index (based on sort index) and return a dictionary of column headers -> (masked_array, ColumnStyle objects) These columns will be moved to the left side of the table (but to the right of row labels, groupbykeys, row numbers, etc.)

**property _styles**

Subclasses can return a callback function which takes no arguments Returns dictionary of column names -> ColumnStyle objects

**property doc**

*rt_meta.Doc* The descriptive documentation object for the structure.

**property footers**

Returns the footer attributes.

For example, Accum2 and AccumTable objects can have footers.

**property has_nested_containers: bool**

True if the Struct contains other Struct-like objects.

**Type**
bool

**property shape**

Return the number of rows and columns.

**Returns**
Number of rows, columns.

**Return type**
tuple of int

**See also:**

**riptable.reshape**
Return an array containing the same data with a new shape.

**FastArray.reshape**
Return an array containing the same data with a new shape.

### Examples

```
>>> ds = rt.Dataset({"one": rt.arange(3), "two": rt.arange(3) % 2})
>>> ds
#   one   two
-   ---   ---
0     0     0
1     1     1
2     2     0
>>> ds.shape
(3, 2)
```

**property total_sizes: Tuple[int, int]**

The total physical and logical size of all (columnar) data in bytes within this Struct.

**Returns**

- **total_physical_size** (*int*) – The total size, in bytes, of all columnar data in this instance, not counting any duplicate/alias object instances.
- **total_logical_size** (*int*) – The total size, in bytes, of all columnar data in this instance, including duplicate/alias object instances. This value is always at least as large as `total_physical_size`.

**AllNames = False**

True if any name for a column name is permitted without renaming.

**Type**
bool

**AllowAnyName = True**

True if any name for a column name is permitted, but will be renamed.

**Type**
bool

**UseFastArray = True**

True if np.ndarray is flipped to FastArray on init.

**Type**
bool

**WarnOnInvalidNames = False**

True if a warning is generated on invalid names.

**Type**
bool

**_lastrepr = 0**

**_lastreprhtml = 0**

**_restricted_names**

**_summary_len = 3**

**col_delete**

**__bool__**()

**__contains__**(*item*)

**__delattr__**(*name*)

Implement delattr(self, name).

**__delitem__**(*name*)

**__dir__**()

Default dir() implementation.

**__enter__**()

**__eq__**(*lhs*)

Return self==value.

**__exit__**(*exc_type*, *exc_val*, *exc_tb*)

**__ge__**(*lhs*)

Return self>=value.

**__getattr__**(*name*)

__getitem__(*index*)

**Parameters**
**index** (*colspec*) –

**Returns**
The indexed item(s), that is, 'column(s)'. If index resolves to multiple 'cols' then another 'Struct' will be returned with those items as a shallow copy.

**Return type**
result

**Raises**
- **IndexError** –
- **TypeError** –

__gt__(*lhs*)
Return self>value.

__iter__()

__le__(*lhs*)
Return self<=value.

__len__()

__lt__(*lhs*)
Return self<value.

__ne__(*lhs*)
Return self!=value.

__repr__()
Return repr(self).

__reversed__()

__setattr__(*name*, *value*)
Implement setattr(self, name, value).

__setitem__(*index*, *value*)

**Parameters**
- **index** – colspec
- **value** – May be any type

**Returns**
None

**Raises**
- **IndexError** –
- **TypeError** –

__str__()
Return str(self).

_addnewitem(*name*, *value*)

**_addnewitem_allnames**(*name*, *value*)

**_aggregate_column_matches**(*items=None*, *like=None*, *regex=None*, *on_missing='raise'*, *func=None*)

Aggregate the matches returned by the methods defined for `items`, `like`, and `regex`, and return them in order.

At least one of `items`, `like`, or `regex` must be specified.

**Parameters**

- **items** (*str, int, or iterable of str or int, optional*) – Names or indices of columns to be removed. An iterable can contain both string and int values.
- **like** (*str, optional*) – Substring to match in column names.
- **regex** (*str, optional*) – Regular expression string to match in column names.
- **on_missing** (*{"raise", "warn", "ignore"}, default "raise"*) – Governs how to handle a column in `items` that doesn't exist:
  - "raise" (default): Raises an IndexError. No columns are returned.
  - "warn": Issues a warning. All columns in `items` that do exist are included in match list.
  - "ignore": No error or warning. All columns in `items` that do exist are included in match list.
- **func** (*callable, optional*) – Method calling `_aggregate_column_matches`. Used to enrich exception / log messages.

**Returns**
A list of strings corresponding to column name matches.

**Return type**
list of str

**classmethod _align_array_info**(*allinfo*, *maxwidths*)

**classmethod _array_info_list**(*arrinfo*)

Build list of info for single array. Used for all arrays in a container or a single array stored in single SDS file.

returns ['FA', 'shape', 'dtype name', 'i+itemsize']

**classmethod _array_summary**(*data*, *name=None*)

**Parameters**

- **data** – Tuple of array info from CompressionType.Info tup1: (tuple) shape tup2: (int) dtype.num tup3: (int) bitmask for numpy flags tup4: (int) itemsize
- **name** – Optional name for top-level Struct.

Intenal routine for tree from meta summary (info only, no arrays)

**Returns**
String of array info for a single struct.

**classmethod _array_summary_single**(*arrinfo*)

**_as_dictionary**(*copy=False*, *rows=None*, *cols=None*)

Return a dictionary of numpy arrays.

_as_meta_data(*name=None*, *nested=True*)

_autocomplete()

_build_sds_meta_data(*name=None*, *nesting=True*, ***kwargs*)

Final SDS file will be laid out as follows:

- header
- meta data string (json, includes scalars)
- arrays
- special arrays
- meta tuples [tuple(item name, SDSFlags) for all items]

Nested data structures will generate their own SDS files.

_check_addtype(*name*, *value*)

override to check types

_copy(*deep=False*, *cls=None*)

**Parameters**

- **deep** (*bool, default True*) – if True, perform a deep copy calling each object depth first with `.copy(True)` if False, a shallow `.copy(False)` is called, often just copying the containers dict.
- **cls** (*type, optional*) – Class of return type, for subclass super() calls
- **False.** (*First argument must be deep. Deep cannnot be set to None. It must be True or*) –

_copy_base(*from_Struct*)

This copies the underlying special variables but does not copy _all_items or _uniqueid or any of the 'columns'.

**Parameters**
**from_Struct** – the Struct being copied

**Returns**
is_locked() (must unlock/relock around rest of copy)

_copy_from_dict(*source_dict*, *copy=False*, *rows=None*, *cols=None*)

_deleteitem(*name*)

_ensure_atomic(*colnames*, *func*)

Only proceed with certain operations if all columns exist in table. Pass in the function for a more informative error.

_escape_invalid_file_chars(*name*)

Certain characters will cause problems in item names if a Struct needs to name an SDS file. ('', ':', '<', '>', '!', '|', '*', '?')

_extract_indexing(*index*)

Internal method common to get/set item.

**Parameters**
**index** – (rowspec, colspec) or colspec (=> rowspec of :)

**Returns**

- *col_idx*
- *row_idx*
- *ncols*
- *nrows*
- *row_arg* – NB Any column names will be converted to str (from bytes or numpy.str_).

**classmethod _flatten_undo**(*sep*, *startpos*, *startname*, *obj_array*, *meta=None*, *cutoffs=None*)

internal routine

**classmethod _from_meta_data**(*itemdict*, *itemflags*, *meta*)

**classmethod _from_sds_onefile**(*arrs*, *meta_tups*, *meta=None*, *folders=None*)

Special routine called after loading an SDS onefile to re-expand

**_get_count_for_slice**(*idx*, *for_rows*)

**_get_final_display_mode**(*plain=False*)

**static _get_seq**(*map*, *protected*, *start*)

**_index_from_row_labels**(*fld*)

Use this if row index was a string or tuple. Will only be applied to the Dataset's label columns (if it has any).

**classmethod _info_tree**(*path*, *data*)

Converts nested structure to tree view of file info for Struct and Dataset. Top level will be named based on single file or directory.

**_init_from_dict**(*dictionary*)

**_ipython_key_completions_**()

**_last_row_stats**()

**classmethod _load_from_sds_meta_data**(*meta*, *arrays*, *meta_tups=[]*, *file_header={}*, *include=None*)

Iterates over sections of the meta data object to rebuild a data structure.

Arrays will be in the following order: - Main arrays (or underlying FastArrays for subclasses) - Secondary arrays for FastArray subclasses that require additional contiguous data (e.g. Categorical) - Array of fancy indices to sort by

A dictionary will be constructed. All arrays will be inserted by name from 'item_names' in meta object. All (if any) meta data will be read from 'item_meta' in meta object, and FastArray subclasses will be constructed. The container object will be constructed from the dictionary. Any labels (gbkeys) will be set. If sorted column names exist, they will be set, and the sorted index will be added to the SortCache.

**Parameters**

- **meta** (*riptable MetaData object (see Utils/rt_metadata.py)*) –
- **arrays** (*list of numpy arrays from an expanded SDS file*) –

**Returns**

For now, Struct, Dataset, and Multiset all use this parent method.

**Return type**
*Struct*, *Dataset*, or *Multiset*

**classmethod _load_from_sds_meta_data_nested**(*name*, *meta*, *arrdict*)

**classmethod _load_without_meta_data**(*meta*, *arrays*, *meta_tups*, *file_header=None*)

Loads from meta tuples only (e.g. when no metadata is generated by Matlab)

**_lock**()

**_mask_get_item**(*idx*, *by_col_arg=True*)

_mask_get_item applies a mask to a row or a column

**Parameters**

- **idx** – the argument from the get/set-item [] brackets
- **by_col_arg** – is this a column mask (instead of row mask)

**Returns**
list of actual indexes or None

**_meta_dict**(*name=None*)

**_post_init**()

Call self._run_once() to cleanup or init anything else, override _run_once() in subclasses if needed. :return: None

**_pre_init**()

**_prepare_display_data**()

Returns a list of lists (all column data) and a list of header tuples for display.

**Returns**
list(list), list(tuple)

**_replaceitem**(*name*, *value*)

**_replaceitem_allnames**(*name*, *value*)

**_repr_html_**()

**_run_once**()

Other classes may override _run_once to initialize data, see _post_init() :return: None

**_safe_reordering_of_renames**(*orig_dict*)

**classmethod _scalar_summary**(*scalar_tup*)

Scalars are stored as arrays in SDS, but a flag is set in the meta tuple. They will be labeled as scalar and their dtype will be displayed.

**classmethod _serialize_item**(*item*, *itemname*)

return a dict of {name: array} a matching list of ints which are the arrayflags a metastring if it exists

**static _sizeof_fmt**(*num*, *suffix='B'*)

**_sort_column_styles**(*style*)

Callback to return sort-by columns.

style : default sort style from display

Returns dictionary of column name -> tuple( array, ColumnStyle ) These columns will be moved to the left of the table.

**_struct_compare_check**(*func_name*, *lhs*)

Returns a Struct consisting of union of key names with value self.X == self.Y. If a key is missing from one or the other it will have value False. If any comparison fails (exception) the value will be False. If any comparisons value Y cannot be cast to bool, Y.all() and all(Y) will be attempted.

**Parameters**

- **func_name** – comparison function name (e.g., '__eq__')
- **lhs** –

**Returns**

Struct of bools

**_superadditem**(*name*, *value*)

**_temp_display**(*option*, *value*)

Temporarily modify a display option when generating dataset display. User configured option (or default) will be restored after display string is generated.

**classmethod _tree_from_sds_meta_data**(*meta*, *arrays*, *meta_tups*, *file_header*)

SDS loads in info mode (no data loaded, just metadata + file header information)

**Returns**

Tree display of nested structures in SDS directory.

**Return type**

str

**_unlock**()

**_update_sort**(*name*)

Discard sort index if sortby item was removed or replaced.

**_validate_names**(*names*)

**all**()

For use in boolean contexts: Is it true that for all elements (val) either:

1. val casts to True, or
2. returns True for val.all() or all(val)

**Return type**

bool

**any**()

For use in boolean contexts: Does there exist an element (val) which either:

1. val casts to True, or
2. returns True for val.any() or any(val)

**Return type**

bool

### Examples

```
>>> s=rt.Struct()
>>> s.a=rt.Dataset()
>>> s.any()
False
```

**apply_schema**(*schema*)

Apply a schema containing descriptive information recursively to the Struct.

**Parameters**
**schema** (*dict*) – A dictionary of schema information. See *rt_meta.apply_schema()* for more information on the format of the dictionary.

**Returns**
**res** – Dictionary of deviations from the schema

**Return type**
dict

**See also:**

*info*, *doc*, *rt_meta.apply_schema()*

**as_ordered_dictionary**(*sublist=None*)

Returns contents of Struct as a collections.OrderedDict instance.

**Parameters**
**sublist** (*list of str*) – Optional list restricting columns to return.

**Return type**
OrderedDict

**asdict**(*sublist=None*, *copy=False*)

Return contents of Struct as a dictionary.

**Parameters**

- **sublist** (*list of str*) – Optional list restricting columns to return.
- **copy** (*bool*) – If set to True then copy() will be called on columns where appropriate.

**Return type**
dict

### Examples

This is useful if, for whatever reason, a riptable Dataset needs to go into a pandas DataFrame:

```
>>> ds = rt.Dataset({'col_'+str(i): rt.arange(5) for i in range(5)})
>>> df = pd.DataFrame(ds.asdict())
>>> df
   col_0  col_1  col_2  col_3  col_4
0      0      0      0      0      0
1      1      1      1      1      1
2      2      2      2      2      2
3      3      3      3      3      3
4      4      4      4      4      4
```

Certain items can be requested with the `sublist` keyword:

```
>>> ds.asdict(sublist=['col_1','col_3'])
{'col_1': FastArray([0, 1, 2, 3, 4]), 'col_3': FastArray([0, 1, 2, 3, 4])}
```

**col_add_prefix**(*prefix*)

Add the same prefix to all items in the Struct/Dataset.

Rather than renaming the columns in a col_rename loop - which would have to rebuild the underlying dictionary N times, this clears the original dictionary, and rebuilds a new one once. Label columns and sortby columns will also be fixed to match the new names.

**Parameters**
**prefix** (*str*) – String to add before every each item name

**Return type**
None

### Examples

```
>>> #TODO Need to call np.random.seed(12345) first to ensure example runs␣
↪deterministically
>>> ds = rt.Dataset({'col_'+str(i):np.random.rand(5) for i in range(5)})
>>> ds.col_add_prefix('NEW_')
#   NEW_col_0   NEW_col_1   NEW_col_2   NEW_col_3   NEW_col_4
-   ---------   ---------   ---------   ---------   ---------
0        0.70        0.52        0.07        0.81        0.26
1        0.13        0.43        0.01        0.46        0.45
2        0.34        0.24        0.87        0.81        0.80
3        0.63        0.22        0.85        0.60        0.91
4        0.46        0.70        0.02        0.49        0.34
```

**col_add_suffix**(*suffix*)

Add the same suffix to all items in the Struct/Dataset.

Rather than renaming the columns in a col_rename loop - which would have to rebuild the underlying dictionary N times, this clears the original dictionary, and rebuilds a new one once. Label columns and sortby columns will also be fixed to match the new names.

**Parameters**
**suffix** (*str*) – String to add before every each item name

**Return type**
None

### Examples

```
>>> #TODO Need to call np.random.seed(12345) first to ensure example runs␣
↪deterministically
>>> ds = rt.Dataset({'col_'+str(i):np.random.rand(5) for i in range(5)})
>>> ds.col_add_suffix('_NEW')
#   col_0_NEW   col_1_NEW   col_2_NEW   col_3_NEW   col_4_NEW
-   ---------   ---------   ---------   ---------   ---------
0        0.70        0.52        0.07        0.81        0.26
```

(continues on next page)

(continued from previous page)

```
1        0.13        0.43        0.01        0.46        0.45
2        0.34        0.24        0.87        0.81        0.80
3        0.63        0.22        0.85        0.60        0.91
4        0.46        0.70        0.02        0.49        0.34
```

**col_exists**(*name*)

Return True if the column name already exists

**col_filter**(*items=None*, *like=None*, *regex=None*, *on_missing='raise'*)

Return the columns specified by indices or matches on column names.

Note that this method doesn't filter a *Dataset* or *Struct* on its contents, only on the column index or name.

At least one of `items`, `like`, or `regex` must be specified.

**Parameters**

- **items** (*str, int, or iterable of str or int, optional*) – Names or indices of columns to be removed. An iterable can contain both string and int values.
- **like** (*str, optional*) – Substring to match in column names.
- **regex** (*str, optional*) – Regular expression string to match in column names.
- **on_missing** (*{"raise", "warn", "ignore"}, default "raise"*) – Governs how to handle a column in `items` that doesn't exist:
  - "raise" (default): Raises an IndexError. Nothing is returned.
  - "warn": Issues a warning. Any columns in `items` that do exist are returned.
  - "ignore": No error or warning. Any columns in `items` that do exist are returned.

**Returns**

Same type as the input object.

**Return type**

*Dataset* or *Struct*

**See also:**

*Struct.__getitem__*, *Dataset.__getitem__*

### Examples

Select columns by name:

```
>>> ds = rt.Dataset({"one": rt.arange(3), "two": rt.arange(3) % 2, "three": rt.
↪arange(3) % 3})
>>> ds
#   one   two   three
-   ---   ---   -----
0     0     0       0
1     1     1       1
2     2     0       2
```

```
>>> ds.col_filter(items=["one", "three"])
#   one   three
-   ---   -----
0     0       0
1     1       1
2     2       2
```

Select columns by index:

```
>>> ds.col_filter(items=[0, 1])
#   one   two
-   ---   ---
0     0     0
1     1     1
2     2     0
```

Select columns by substring:

```
>>> ds.col_filter(like="thr")
#   three
-   -----
0       0
1       1
2       2
```

Select columns by regular expression:

```
>>> ds.col_filter(regex="e$")
#   one   three
-   ---   -----
0     0       0
1     1       1
2     2       2
```

Select *Dataset* and *FastArray* objects from a *Struct*:

```
>>> ds2 = rt.Dataset({"four": rt.arange(3), "five": rt.arange(3) % 2})
>>> fa = rt.FastArray([1, 2, 3])
>>> s = rt.Struct()
>>> s.ds = ds
>>> s.ds2 = ds2
>>> s.fa = fa
>>> s.col_filter([0])
#   Name   Type      Size              0   1   2
-   ----   -------   ---------------   -   -   -
0   ds     Dataset   3 rows x 3 cols
>>> s.col_filter("fa")
#   Name   Type    Size   0   1   2
-   ----   -----   ----   -   -   -
0   fa     int32   3      1   2   3
```

**col_get_attribute**(*name*, *attrib_name*, *default=None*)

Gets the attribute of the specified column, the `attrib_name` must be used to indicate which attribute.

**Parameters**

- **name** (*str*) – The name of the column
- **attrib_name** (*str*) – The name of the attribute
- **default** – Default value returned when attribute not found.

### Examples

```
>>> ds.col_set_attribute('col1', 'TEST', 417)
>>> ds.col_get_attribute('col1', 'TEST')
417
>>> ds.col_get_attribute('col1', 'TEST', nan)
417
>>> ds.col_get_attribute('col1', 'DOESNOTEXIST', nan)
nan
```

**col_get_len**()

Gets the number of columns (or items) in the Struct

**col_get_value**(*name*)

Return a single item.

**Parameters**
**name** (*string*) – Item name.

**Returns**
Item from item container (no attribute).

**Return type**
obj

**Raises**
**KeyError** – Item not found with given name.

**col_map**(*rename_dict*)

Rename columns and re-arrange names of columns based on the rules set forth in the supplied dictionary.

**Parameters**
**rename_dict** (*dict*) – Dictionary defining a remapping of (some/all) column names.

**Return type**
None

### Examples

```
>>> #TODO Call np.random.seed(12345) here to make the example output␣
↪deterministic
>>> ds = rt.Dataset({'col_'+str(i): np.random.rand(5) for i in range(5)})
>>> ds.col_map({'col_1':'AAA', 'col_2':'BBB'})
>>> ds
#   col_0    AAA    BBB   col_3   col_4
-   -----   ----   ----   -----   -----
0    0.55   0.21   0.27    0.85    0.03
1    0.77   0.75   0.65    0.97    0.24
2    0.09   0.07   0.40    0.81    0.62
```

(continues on next page)

(continued from previous page)

```
3    0.50   0.93   0.98    0.99    0.99
4    0.40   0.45   0.53    0.49    0.76
```

**col_move**(*flist*, *blist*)

Move single column or group of columns to back of list for iteration/indexing/display. Values of columns will remain unchanged.

**Parameters**

- **flist** (list of str) – Item names to move to front.
- **blist** (list of str) – Item names to move to back.

**See also:**

*Struct.col_move_to_front*, *Struct.col_move_to_back*

**col_move_to_back**(*cols*)

Move single column or group of columns to front of list for iteration/indexing/display.

Values of columns will remain unchanged.

**Parameters**

**flist** (list of str) – Item names to move to back.

### Examples

```
>>> #TODO Call np.random.seed(12345) here to make the example output␣
↪deterministic
>>> ds = rt.Dataset({'col_'+str(i): np.random.rand(5) for i in range(5)})
>>> ds
#   col_0   col_1   col_2   col_3   col_4
-   -----   -----   -----   -----   -----
0    0.28    0.84    0.24    0.72    0.81
1    0.72    0.44    0.41    0.53    0.17
2    0.37    0.66    0.61    0.52    0.50
3    0.08    0.31    0.15    0.65    0.98
4    0.63    0.89    0.25    0.13    0.16
```

```
>>> ds.col_move_to_back(['col_2','col_0'])
#   col_1   col_3   col_4   col_2   col_0
-   -----   -----   -----   -----   -----
0    0.84    0.72    0.81    0.24    0.28
1    0.44    0.53    0.17    0.41    0.72
2    0.66    0.52    0.50    0.61    0.37
3    0.31    0.65    0.98    0.15    0.08
4    0.89    0.13    0.16    0.25    0.63
```

**See also:**

*Struct.col_move_to_back*, *Struct.col_move*

**col_move_to_front**(*cols*)

Move single column or group of columns to front of list for iteration/indexing/display. Values of columns will remain unchanged.

**Parameters**
**flist** (list of str) – Item names to move to front.

### Examples

```
>>> #TODO Call np.random.seed(12345) here to make the example output␣
↪deterministic
>>> ds = rt.Dataset({'col_'+str(i): np.random.rand(5) for i in range(5)})
>>> ds
#   col_0   col_1   col_2   col_3   col_4
-   -----   -----   -----   -----   -----
0    0.60    0.50    0.77    0.72    0.73
1    0.48    0.65    0.96    0.17    0.99
2    0.06    0.54    0.81    0.20    0.30
3    0.18    0.85    0.24    0.44    0.38
4    0.04    0.84    0.64    0.66    0.97
```

```
>>> ds.col_move_to_front(['col_4', 'col_2'])
>>> ds
#   col_4   col_2   col_0   col_1   col_3
-   -----   -----   -----   -----   -----
0    0.73    0.77    0.60    0.50    0.72
1    0.99    0.96    0.48    0.65    0.17
2    0.30    0.81    0.06    0.54    0.20
3    0.38    0.24    0.18    0.85    0.44
4    0.97    0.64    0.04    0.84    0.66
```

**See also:**

*Struct.col_move_to_front*, *Struct.col_move*

**col_pop**(*colspec*)

colspec is as for [] (getitem). List input will return a sub-Struct, removing it from current object. Single-column ("string", single integer) input will return a single "column".

**Parameters**
**colspec** (*list, string, or integer*) –

**Returns**
Single value or new (same-type) object containing the removed data.

**Return type**
obj

### Examples

```
>>> ds = rt.Dataset({'col_'+str(i): rt.arange(5) for i in range(3)})
>>> ds
#   col_0   col_1   col_2
-   -----   -----   -----
0       0       0       0
1       1       1       1
2       2       2       2
```

(continues on next page)

(continued from previous page)

```
3       3       3       3
4       4       4       4
>>> col = ds.col_pop('col_1')
>>> ds
#   col_0   col_2
-   -----   -----
0       0       0
1       1       1
2       2       2
3       3       3
4       4       4
>>> col
FastArray([0, 1, 2, 3, 4])
```

**col_remove**(*items=None*, *like=None*, *regex=None*, *on_missing='warn'*)

Remove the columns specified by indices or matches on column names.

This can be done only if the *Dataset* or *Struct* is unlocked.

At least one of `items`, `like`, or `regex` must be specified.

**Parameters**

- **items** (*str, int, or iterable of str or int, optional*) – Names or indices of columns to be removed. An iterable can contain both string and int values.
- **like** (*str, optional*) – Substring to match in column names.
- **regex** (*str, optional*) – Regular expression string to match in column names.
- **on_missing** (*{"warn", "raise", "ignore"}, default "warn"*) – Governs how to handle a column in `items` that doesn't exist:
  - "warn" (default): Issues a warning. All columns in `items` that do exist are removed.

    Changed in version 1.13.0: Previously, the default value was `"raise"`.
  - "raise" : Raises an IndexError. No columns are removed.
  - "ignore": No error or warning. All columns in `items` that do exist are removed.

**Return type**

None

### Examples

```
>>> ds = rt.Dataset({"col_" + str(i): rt.arange(5) for i in range(5)})
>>> ds
#   col_0   col_1   col_2   col_3   col_4
-   -----   -----   -----   -----   -----
0       0       0       0       0       0
1       1       1       1       1       1
2       2       2       2       2       2
3       3       3       3       3       3
4       4       4       4       4       4
```

```
>>> ds.col_remove(["col_2", "col_0"])
>>> ds
#   col_1   col_3   col_4
-   -----   -----   -----
0       0       0       0
1       1       1       1
2       2       2       2
3       3       3       3
4       4       4       4
```

Try to remove a column that doesn't exist with the default `on_missing="warn"`. A warning is raised, and any columns that do exist are removed:

```
>>> ds.col_remove(["col_1", "col_2"])
UserWarning: Column col_2 doesn't exist and couldn't be removed.
>>> ds
#   col_3   col_4
-   -----   -----
0       0       0
1       1       1
2       2       2
3       3       3
4       4       4
```

Remove a column by its index:

```
>>> ds.col_remove([0])
>>> ds
#   col_4
-   -----
0       0
1       1
2       2
3       3
4       4
```

```
>>> ds2 = rt.Dataset({"aabb": rt.arange(3), "abab": rt.arange(3), "ccdd": rt.
↪arange(3),
...                   "cdcd": rt.arange(3)})
>>> ds2
#   aabb   abab   ccdd   cdcd
-   ----   ----   ----   ----
0      0      0      0      0
1      1      1      1      1
2      2      2      2      2
```

Remove columns by substring:

```
>>> ds2.col_remove(like="cd")
>>> ds2
#   aabb   abab
-   ----   ----
0      0      0
```

(continues on next page)

(continued from previous page)

```
1      1      1
2      2      2
```

Remove columns by regular expression:

```
>>> ds2.col_remove(regex="^ab")
>>> ds2
#   aabb
-   ----
0      0
1      1
2      2
```

**col_rename**(*old*, *new*)

Rename a single column.

The new name must be a valid column name; that is, it must not be a Python keyword or a *Struct* or *Dataset* class method name.

To check whether a name is valid, use *is_valid_colname*. To see a list of invalid column names, use *get_restricted_names*.

Note that column names that don't meet Python's rules for well-formed variable names can't be accessed using attribute access. For example, a column named 'my-column' can't be accessed with `ds.my-column`, but can be accessed with `ds['my-column']`.

**Parameters**

- **old** (*str*) – Current column name.
- **new** (*str*) – New column name.

**Return type**
None

**See also:**

*is_valid_colname*
Check whether a string is a valid column name.

*get_restricted_names*
Get a list of invalid column names.

### Examples

```
>>> ds = rt.Dataset({'a': [1, 2, 3]), 'b': [4.0, 5.0, 6.0]})
>>> ds
#   a      b
-   -   ----
0   1   4.00
1   2   5.00
2   3   6.00
>>> ds.col_rename('a', 'new_a')
>>> ds
#   new_a      b
```

(continues on next page)

(continued from previous page)

```
-   -----   ----
0       1   4.00
1       2   5.00
2       3   6.00
```

**col_set_attribute**(*name*, *attrib_name*, *attrib_value*)

Sets the attribute of the specified column, the attrib_name must be used to indicate which attribute.

**Parameters**

- **name** (*str*) – The name of the column
- **attrib_name** (*str*) – The name of the attribute
- **attrib_value** – The value of the attribute

### Examples

```
>>> ds.col_set_attribute('col1', 'TEST', 417)
>>> ds.col_get_attribute('col1', 'TEST')
417
```

**col_set_value**(*name*, *value*)

Check if item name is allowed, possibly escape. Set the value portion of the item to value.

**Parameters**

- **name** (*str*) – Item name.
- **value** (*object*) – For structs, nearly anything. For datasets, array.

**col_str_match**(*expression*, *flags=0*)

Create a boolean mask vector for columns whose names match the regex.

Uses `re.match()`, not `re.search()`.

**Parameters**

- **expression** (*str*) – regular expression
- **flags** – regex flags (from `re` module).

**Returns**

Array of bools (len ncols) which is true for columns which match the regex.

**Return type**

*FastArray*

### Examples

```
>>> st = rt.Struct({
... 'price' : arange(5),
... 'trade_time' : rt.arange(5) * 1000,        # expected to regex match `.
↪*time.*`
... 'name' : rt.FA(['a','b','c','d','e']),
... 'other_trade_time' : rt.arange(5) * 1000,  # expected to regex match `.
↪*time.*`
... })
>>> st.col_str_match(r'.*time.*')
FastArray([False,  True, False, True])
```

**col_str_replace**(*old*, *new*, *max=-1*)

If a column name contains the old string, the old string will be replaced with the new one. If replacing the string will conflict with an existing column name, an error will be raised. Labels / sortby columns will be fixed if their names are modified.

**Parameters**

- **old** (*str*) – String to look for within individual names of columns.
- **new** (*str*) – String to replace old string in column names.
- **max** (*int*) – Optionally limit the number of occurrences per column name to replace; defaults to -1 which will replace all.

### Examples

Replace all occurrences in each names:

```
>>> ds = rt.Dataset({
... 'aaa': rt.arange(5),
... 'a' : rt.arange(5),
... 'aab': rt.arange(5)
... })
>>> ds.col_str_replace('a', 'A')
>>> ds
#   AAA   A   AAb
-   ---   -   ---
0     0   0     0
1     1   1     1
2     2   2     2
3     3   3     3
4     4   4     4
```

Limit number of replacements per name:

```
>>> ds = rt.Dataset({
... 'aaa': rt.arange(5),
... 'a' : rt.arange(5),
... 'aab': rt.arange(5)
... })
>>> ds.col_str_replace('a','A',max=1)
```

(continues on next page)

(continued from previous page)

```
>>> ds
#   Aaa   A   Aab
-   ---   -   ---
0     0   0     0
1     1   1     1
2     2   2     2
3     3   3     3
4     4   4     4
```

Replacing will create a conflict:

```
>>> ds = rt.Dataset({'a': rt.arange(5), 'A': rt.arange(5)})
ValueError: Item A already existed, cannot make replacement in item.
```

**col_swap**(*from_cols*, *to_cols*)

Swaps column values, names retain current order.

**Parameters**

- **from_cols** (*list*) – a list of unique extant column names
- **to_cols** (*list*) – a list of unique extant column names

### Examples

```
>>> st = Struct({'a': 1, 'b': 'fish', 'c': [5.6, 7.8], 'd': {'A': 'david', 'B':
↪ 'matthew'},
... 'e': np.ones(7, dtype=np.int64)})
>>> st
#   Name   Type    Rows   0      1     2
-   ----   -----   ----   ----   ---   -
0   a      int     0      1
1   b      str     0      fish
2   c      list    2      5.6    7.8
3   d      dict    2      A      B
4   e      int64   7      1      1     1
```

```
>>> st.col_swap(list('abc'), list('cba'))
>>> st
#   Name   Type    Rows   0      1     2
-   ----   -----   ----   ----   ---   -
0   a      list    2      5.6    7.8
1   b      str     0      fish
2   c      int     0      1
3   d      dict    2      A      B
4   e      int64   7      1      1     1
```

**classmethod concat_structs**(*struct_list*)

Merges data from multiple structs.

Structs must have the same keys, and contain only Structs, Datasets, arrays, and riptable arrays.

A struct utility for merging data from multiple structs (useful for multiday loading). Structs must have the same keys, and contain only Structs, Datasets, Categoricals, and Numpy Arrays.

**Parameters**
**struct_list** (list of *Struct*) –

**Returns**
**obj**

**Return type**
*Struct*

**See also:**

*hstack()*

**copy**(*deep=True*)

Returns a shallow or deep copy of the *Struct*. Defaults to a deep copy.

**Parameters**
**deep** (*bool, default True*) – if True, perform a deep copy calling each object depth first with `.copy(True)` if False, a shallow `.copy(False)` is called, often just copying the containers dict.

### Examples

```
>>> ds=rt.Dataset({'somenans': [0., 1., 2., nan, 4., 5.], 'morestuff': ['A','B
↪','C','D','E','F']})
>>> ds2=rt.Dataset({'somenans': [0., 1., nan, 3., 4., 5.], 'morestuff':['H','I
↪','J','K','L','M']})
>>> st=Struct({'test':ds, 'test2': Struct({'ds2':ds2}), 'arr': arange(10)})
>>> st.copy()
#   Name    Type      Size              0     1   2
-   -----   -------   ---------------   ---   -   -
0   test    Dataset   6 rows x 2 cols
1   test2   Struct    1                 ds2
2   arr     int32     10                0     1   2
```

**display_attributes**()

Returns a dict of display attributes, currently consisting of NumberOfFooterRows and a list of MarginColumns.

**Returns**
**d** – A dictionary of display attributes

**Return type**
dict

**dtranspose**(*plain=False*)

For display only. Return a transposed version of the container's string representation.

**Parameters**
**plain** (*bool, False*) – If true then should not be colored.

**Returns**
Formatted, transposed version of this instance; intended for display.

**Return type**
string

### Examples

```
>>> st = rt.Struct({'a': 1, 'b': 'fish', 'c': [5.6, 7.8], 'd': {'A': 'david',
↪'B': 'matthew'},
... 'e': np.ones(7, dtype=np.int64)})
>>> st
#   Name    Type   Size      0     1   2
-   ----   -----   ----   ----   ---   -
0      a     int      0      1
1      b     str      0   fish
2      c    list      2    5.6   7.8
3      d    dict      2      A     B
4      e   int64      7      1     1   1
[5 columns]
>>> st.dtranspose()
Fields:     0      1      2      3       4
-------   ---   ----   ----   ----   -----
   Name     a      b      c      d       e
   Type   int    str   list   dict   int64
   Size     0      0      2      2       7
      0     1   fish    5.6      A       1
      1                 7.8      B       1
      2                                   1
[5 columns]
```

**equals**(*other*)

Test whether two Structs contain the same elements in each column. NaNs in the same location are considered equal.

**Parameters**

**other** (*another Struct or dict to compare to*) –

**Return type**

bool

**See also:**

Dataset.crc, ==, >=, <=, >, <

### Examples

```
>>> s1 = rt.Struct({'t': 54, 'test': np.int64(34), 'test2': rt.arange(200)})
>>> s2 = rt.Struct({'t': 54, 'test': np.int64(34), 'test2': rt.arange(200)})
>>> s1.equals(s2)
True
```

**flatten**(*sep='/'*, *level=0*)

Flattens or collapses a Struct, recursively called

**Parameters**

**use** (*sep='/' the separating string to*) – Please note that some chars are not allowed and will be replaced with _.

**Return type**

New Struct with collapsed names (separated by specified char) which can then be saved

**Note:** _sep is stored in the __dict__ to help with undo or saving to file arrayflags, metastring are now exposed

**See also:**

*flatten_undo*

**flatten_undo**(*sep=None*, *startname=''*, *obj_array=None*)

Restores a Struct to original form before Struct.flatten()

**Parameters**

- **sep=None** –
- **'/'** (*user may pass in the separating string to use such as*) –

**Return type**
New Struct that is back to original form before Struct.flatten()

**See also:**

*flatten*

**get_attribute**(*attrib_name*, *default=None*)

Get an attribute that applies to all items/columns.

**Parameters**

- **attrib_name** – name of the attribute
- **default** – return value if attrib_name is not a valid attribute

**Returns**
**val**

**Return type**
attribute value or None

**See also:**

*col_get_attribute*, *set_attribute*

**get_ncols**()

Return the number of items in the Struct.

**Returns**
**ncols** – The number of items in the Struct

**Return type**
int

**get_nrows**()

Retunrs 0, as a Struct has no rows.

**Returns**
0

**Return type**
int

**Note:** Subclasses need to define this explicitly.

**get_restricted_names**()

Return a list of invalid column names.

Invalid column names are Python keywords and *Struct* or *Dataset* class method names.

This method generates the result only once. Afterward, it is stored as a class variable.

**Returns**
A set of strings that are invalid column names.

**Return type**
set

**See also:**

*is_valid_colname*
Check whether a string is a valid column name.

### Examples

```
>>> # Limit and format the output.
>>> print("Some of the restricted names include: ")
>>> print(", ".join(list(ds.get_restricted_names())[::10]))
Some of the restricted names include: mask_or_isinf, __reduce_ex__,
imatrix_xy, __weakref__, dtypes, _get_columns, from_arrow, elif,
__imul__, _deleteitem, __rsub__, _index_from_row_labels, as_matrix,
putmask, _as_meta_data, shape, cat, __invert__, try, _init_columns_as_dict,
label_as_dict, col_str_replace, _replaceitem, label_set_names, __contains__,
__floordiv__, _row_numbers, filter, __init__, sorts_on, flatten_undo,
col_str_match, __dict__, size, __rand__, info, col_remove, as, or
```

**classmethod hstack**(*struct_list*)

Merges data from multiple structs. Structs must have the same keys, and contain only Structs, Datasets, arrays, and riptable arrays.

**Parameters**

- **struct_list** (list of *Struct*) –
- **loading).** (*A struct utility for merging data from multiple structs (useful for multiday*) –
- **keys** (*Structs must have the same*) –
- **Structs** (*and contain only*) –
- **Datasets** –
- **Categoricals** –
- **Arrays.** (*and Numpy*) –

**Returns**
**obj**

**Return type**
*Struct*

**See also:**

riptable.hstack

**info**(***kwargs*)

Return an object containing a description of the structure's contents.

**Parameters**
**kwargs** (*dict*) – Optional keyword arguments passed to *rt_meta.info()*

**Returns**
**info** – A description of the structure's contents.

**Return type**
*rt_meta.Info*

**is_locked**()

Returns True if object is locked (unable to add/remove/rename elements).

NB: Currently behaves as does tuple: the contained data will still be mutable when possible.

**Returns**
True if object is locked

**Return type**
bool

**is_valid_colname**(*name*)

Check whether a string is a valid column name.

Python keywords and *Struct* or *Dataset* class method names are not valid column names.

To see a list of invalid column names, use *get_restricted_names*.

**Parameters**
**name** (*str*) – The string to be checked.

**Returns**
True if `name` is valid, otherwise False.

**Return type**
bool

**See also:**

***get_restricted_names***
Get a list of invalid column names.

### Examples

```
>>> ds.is_valid_colname('yield')  # Python keyword
False
>>> ds.is_valid_colname('sample')  # Dataset method
False
>>> ds.is_valid_colname('Yield')  # OK because keywords are case-sensitive
True
>>> ds.is_valid_colname('Sample')  # Method names are also case-sensitive
True
```

**items**()

Dictionary-iterator access to Struct items.

**Returns**

- *dict_items* – Name, Item pairs.
- *return: iterator to column keys and values*

**key_search**(*regex*, *case_sensitive=False*, *recursive=True*, *path=''*)

**keys**()

**Returns**
Item names.

**Return type**
list

**label_as_dict**()

Gets the column names used as labels in display

**label_filter**(*items=None*, *like=None*, *regex=None*, *axis=None*)

Subset rows of dataset according to value in its label column.

TODO: how should multikey be handled?

**Parameters**

- **items** (*list-like*) – List of specific values to match in label column.
- **like** (*string*) – Keep items where 'like' occurs in label column.
- **regex** (*string (regular expression)*) – Keep axis with re.search(regex, col) == True.

### Examples

```
>>> ds
 #   col_7   col_8   col_9   keycol
--   -----   -----   -----   --------------
 0    0.53    0.52    0.47   paul
 1    0.10    0.78    0.09   ray
 2    0.50    0.79    0.50   paul
 3    0.81    0.68    0.72   ray
 4    0.08    0.71    0.02   john
 5    0.38    0.19    0.90   ray
 6    0.53    0.33    0.46   mary katherine
 7    0.75    0.48    0.94   john
 8    1.00    0.70    0.79   mary ann
 9    0.47    0.64    0.16   ray
10    0.80    0.43    0.08   mary ann
11    0.54    0.19    0.43   joe
12    0.89    0.08    0.81   mary katherine
13    0.96    0.91    0.33   paul
14    0.18    0.55    0.44   ray
15    0.42    0.49    0.66   mary ann
16    0.05    0.53    0.66   paul
17    0.60    0.56    0.03   joe
18    0.62    0.42    0.56   mary ann
19    0.63    0.33    0.95   paul
```

```
>>> gb = ds.gb('keycol').sum()
>>> gb.label_filter(items='john')
*keycol   col_7   col_8   col_9
-------   -----   -----   -----
john       0.82    1.19    0.96
```

```
>>> gb.label_filter(like=['ar', 'p'])
*keycol          col_7   col_8   col_9
--------------   -----   -----   -----
mary ann          2.85    2.05    2.08
mary katherine    1.43    0.41    1.27
paul              2.66    3.08    2.92
```

```
>>> gb.label_filter(regex='n$')
*keycol    col_7   col_8   col_9
--------   -----   -----   -----
john        0.82    1.19    0.96
mary ann    2.85    2.05    2.08
```

**label_get_names**()

Gets the column names used as labels in display

**label_remove**()

Reomves any labels used in display

**label_set_names**(*listnames*)

Set which column names can be used as labels in display

**classmethod load**(*path=''*, *name=None*, *share=None*, *info=False*, *columns=None*, *include_all_sds=False*, *include=None*, *threads=None*, *folders=None*)

Load a Struct from a directory or single SDS file.

**Parameters**

- **path** (*str or os.PathLike*) – Full path to directory or single SDS file with Struct data.
- **name** (str, optional, default None) – Name of a nested container to search for in the root directory. Multiple tiers can be separated by '//'
- **info** (*bool, optional, default False*) – If True, no array data will be loaded, instead a display tree of information about nested structures and their contents will be returned.
- **columns** (list, optional, default None) – Not implemented
- **include_all_sds** (*bool, optional, default False*) – If False, when additional files were found in a directory, and they were not in the root structs meta data, the user will be prompted to load them. If True, all files will be automatically loaded.
- **include** (*list of str, optional, default None*) – A list of specific items to load. This list will only be applied to the root Struct - not to nested containers.
- **threads** (*int, optional, default None*) – Number of threads to use during the SDS load. Number of threads before the load will be restored after the load or if the load fails. See also riptide_cpp.SetThreadWakeUp.

**Returns**
Loaded data with possibly nested containers and riptable classes restored.

**Return type**
*Struct*

**See also:**

`riptable.load_sds`

**make_categoricals**(*columnlist=None*, *dtype=None*)

Converts specified string/bytes columns or all string/bytes columns to Categorical. Will also crawl through nested structs/datasets and convert their strings to categoricals.

**Parameters**

- **columnlist** (str or list, optional) – Single name, or list of names of items to convert to categoricals.
- **dtype** (numpy.dtype, optional) – Integer dtype for the categoricals' underlying arrays.

**Raises**

- **TypeError** – If the dtype was set to a non-dtype object.
- **ValueError** – If a requested item could not be found in the container.

### Notes

Error checking will complete in the root structure before any conversion begins.

**make_matlab_categoricals**(*xtra*, *remove_trailing=True*, *dtype=None*, *prefix='p'*, *keep_prefix=True*)

Turn matlab categorical indices and corresponding unique arrays into riptable categoricals.

**Parameters**

- **xtra** (*Struct*) – Container holding unique arrays.
- **remove_trailing** (*bool, optional, default True*) – If True, remove trailing spaces from Matlab strings.
- **dtype** (numpy.dtype, optional, default None) – Integer dtype for underlying array of constructed categoricals.
- **prefix** (str, optional, default 'p') – Prefix for integer arrays in calling dataset - columns that will be looked for in the struct.
- **keep_prefix** (*bool, default True*) – If True, Drop the prefix after flipping the column to categorical in the dataset. If the a column exists with that name, the user will be warned.

**make_matlab_datetimes**(*dtcols=None*, *gmt=False*, *auto=True*)

Convert datetime columns from Matlab to DateTimeNano and TimeSpan arrays.

**Parameters**

- **dtcols** (str or list) – Name or list of names of columns to convert to DateTimeNano arrays.
- **gmt** (*bool, optional, default False*) – Not implemented.
- **auto** (*bool, optional, default True*) – If True, look for 'MS' in the names of all columns, and flip them to TimeSpan objects.

**make_struct_from_categories**(*prefix=None*, *keep_prefix=False*)

Build a struct of unique arrays from all categoricals in the container, or those with a specified prefix.

**Parameters**

- **prefix** (str, optional) – Only include columns with names that begin with this string.
- **keep_prefix** (*bool, default False*) – If True, keep the prefix when naming the item in the new structure.

### Examples

TODO - sanitize - add example that makes a struct from categoricals and prints its representation See the version history for structure of older examples.

**Returns**
**cats**

**Return type**
*Struct*

### Notes

This is a partial inverse operation of Struct.make_matlab_categoricals

**make_table**(*display_type*)

Pretty-print code used by infrastructure.

**Parameters**
**display_type** (*rt.rt_enum.DS_DISPLAY_TYPES*) –

**Returns**
Display object or string.

**Return type**
obj or str

**save**(*path=''*, *name=None*, *share=None*, *overwrite=True*, *compress=True*, *onefile=False*, *bandsize=None*)

Save a struct to a directory. If the struct contains only arrays, will be saved as a single .SDS file.

**Parameters**

- **path** (*str or os.PathLike*) – Full path to save. Directory will be created automatically if it doesn't exist. .SDS extension will be appended if a single file is being saved and is necessary.
- **name** (*str, optional*) – Name for the root structure if it's being appended to an existing struct's directory. The existing _root.sds does not get overwritten, and structs can be combined without a full load.
- **share** (*str, optional*) –
- **overwrite** (*bool, optional, default True*) – If True, user will not be prompted on whether or not to overwrite existing .SDS files. Otherwise, prompt will appear if directory exists.
- **compress** (*bool, optional, default True*) – If True, ZStandard compression will be used when writing to SDS, otherwise, no compression will be used.

- **onefile** (*bool, optional, default False*) – If True will collapse all nesting Structs
- **bandsize** (*int, optional, default None*) –

**set_attribute**(*attrib_name*, *attrib_value*)

Set an attribute that applies to all items/columns.

**Parameters**

- **attrib_name** – name of the attribute
- **attrib_value** – value of the attribute

**See also:**

*col_set_attribute*, *get_attribute*

**set_display_callback**(*userfunc*, *scope=None*)

Set the user display callback for styling text.

**Parameters**

- **userfunc** (*function*) – This function must take two arguments, `userfunc(cols, **kwargs)`.
- **scope** (*default None*) – The callback for just this Dataset, or all Datasets. Can be None, `Dataset`, or *Struct*.

### Examples

```
>>> from riptable.rt_display import DisplayColumnColors
>>> def make_red(cols, **kwargs):
...     location = kwargs['location']  # could left, right, or main
...     if location == 'main':
...         for col in cols:
...             for cell in col:
...                 if cell.string.startswith('-'): cell.string = '(' + cell.
↪string[1:] + ')'; cell.color=DisplayColumnColors.Red
>>> ds=rt.Dataset({'test':rt.arange(5)-3, 'another':rt.arange(5.0)-2})
>>> ds.set_display_callback(make_red)
>>> ds
```

**static set_fast_array**(*val*)

Set to true to force the casting of numpy arrays to FastArray when constructing a Struct or adding a new column.

**Parameters**

**val** (*bool*) – True or False

**summary_as_dict**()

Gets the column names used as rights in display

**summary_get_names**()

Gets the column names used as rights in display

**summary_remove**()

Reomves any rights used in display

**summary_set_names**(*listnames*)

Set which column names can be used as rights in display

**tolist**()

Returns data values in a list. Output equivalent to list(st.values()).

**Returns**
list

**tree**(*name=None*, *showpaths=False*, *info=False*)

Returns a hierarchical view of the Struct.

**Parameters**

- **name** (*str*) – Optional name for the top of the tree
- **showpaths** – TODO purpose unknown, may raise error if true
- **info** – TODO purpose unknown

**Returns**
**tree** – A hierarchical view of the Struct

**Return type**
str

### Examples

```
>>> st1 = rt.Struct({'A': rt.FA([1, 2, 3]), 'B': rt.FA([4, 5])})
>>> st2 = rt.Struct({'C': st1, 'D': rt.FA([6, 7, 8])})
>>> st2.tree()
Struct
  ├──── C (Struct)
  │     ├──── A int32 (3,) 4
  │     └──── B int32 (2,) 4
  └──── D int32 (3,) 4
>>> st2.tree(name='foo')
foo
  ├──── C (Struct)
  │     ├──── A int32 (3,) 4
  │     └──── B int32 (2,) 4
  └──── D int32 (3,) 4
```

**values**()

Values are individual items from the struct (no attribute from item container).

**Returns**
Items.

**Return type**
dict_values

### 2.2.43 riptable.rt_timers

#### Functions

| | |
|---|---|
| *GetNanoTime*() | Returns: a long integer in unix epoch nanoseconds |
| *GetTSC*() | Returns: a long integer from the CPUs's current time stamp counter |
| *tic*() | Call tic() followed by code followed by toc() to time a routine in nanoseconds. |
| *ticf*() | Call ticf() followed by code followed by tocf() to time fastarrays |
| *ticp*() | Call ticp() followed by code followed by tocp() to profile function calls |
| *ticx*() | Call ticx() followed by code followed by tocx() to time a routine in TSC |
| *toc*([logger]) | Call tic() followed by code followed by toc() to time a routine in nanoseconds. |
| *tocf*([dataset]) | Call ticf() followed by code followed by tocf() to time fastarrays |
| *tocp*([dataset, logfile, sort, strip, stats, calls, find]) | Call ticp() followed by code followed by tocp() to profile anything between the ticp/tocp |
| *tocx*([logger]) | Call ticx() followed by code followed by tocx() to time a routine in TSC |
| *tt*(expression[, loops, return_time]) | tictoc time an expression in nanoseconds. use ; to separate lines |
| *ttx*(expression[, loops]) | tictoc time an expression in TSC (time stamp counters). use ; to separate lines |
| *utcnow*([count]) | Call *GetNanoTime* one or more times and return the timestamps in a `DateTimeNano` array. |

riptable.rt_timers.**GetNanoTime**()

Returns: a long integer in unix epoch nanoseconds Note: this function is written as fast as possible for both Windows and Linux

riptable.rt_timers.**GetTSC**()

Returns: a long integer from the CPUs's current time stamp counter

time stamp counter (TSC) are based on the CPUs's clock cycle, which is often above 1GHz thus GetTSC return values are guaranteed to be both unique and subsample below 1 nanosecond

Note: this function is written as fast as possible for both Windows and Linux

riptable.rt_timers.**tic**()

Call tic() followed by code followed by toc() to time a routine in nanoseconds.

**See also:**

*toc*, *ticx*, *ticp*, *ticf*

riptable.rt_timers.**ticf**()

Call ticf() followed by code followed by tocf() to time fastarrays

See also: toc, ticx, ticp, ticf

riptable.rt_timers.**ticp**()

Call ticp() followed by code followed by tocp() to profile function calls

See also: toc, ticx, ticp, ticf

riptable.rt_timers.**ticx**()

Call ticx() followed by code followed by tocx() to time a routine in TSC

See also: toc, ticx, ticp, ticf

riptable.rt_timers.**toc**(*logger=None*)

Call tic() followed by code followed by toc() to time a routine in nanoseconds.

**Parameters**

**logger** (*logging.Logger, optional*) – An optionally-specified logger where the collected timing information is recorded. If not specified (the default), the timing information is written to stdout.

**See also:**

*toc*, *ticx*, *ticp*, *ticf*

riptable.rt_timers.**tocf**(*dataset=True*)

Call ticf() followed by code followed by tocf() to time fastarrays

**Parameters**

**dataset** (*bool, defaults to True.*) – If specified, returns a Dataset. Set to False to print out instead.

riptable.rt_timers.**tocp**(*dataset=True*, *logfile=None*, *sort='time'*, *strip=True*, *stats=False*, *calls=False*, *find=None*)

Call ticp() followed by code followed by tocp() to profile anything between the ticp/tocp tocp() may be called again to retrieve data in a different manner

### Examples

ticp(); ds.sort_copy(by='Symbol'); tocp()._H ticp(); ds.sort_copy(by='Symbol'); tocp().sort_view('cumtime')._A ticp(); ds.sort_copy(by='Symbol'); tocp(find='rt_fastarray.py:332')._H ticp(); ds.sort_copy(by='Symbol'); tocp(find='rt_fastarray.py')._H ticp(); ds.sort_copy(by='Symbol'); ds=tocp(calls=True); ds.gb('filepath').sum()._H tocp(calls=True).gb(['function','filepath'])['tottime'].sum().sort_view('tottime')._A ticp(); ds.sort_copy(by='Symbol'); stats=tocp(stats=True); ticp(); ds.sort_copy(by='Symbol'); tocp(False); ticp(); ds.sort_copy(by='Symbol'); tocp(False, strip=False); ticp(); ds.sort_copy(by='Symbol'); tocp(False, sort='cumtime');

**Parameters**

- **output** (*dataset=False. set to True to return a Dataset otherwise use pstats*) –
- **format** (*logfile=None. set to filename to save the Dataset in SDS*) – NOTE: consider pickling the result when stats=True to save for later analysis
- **dataset=False** (*sort='time' by default when*) –
- **info** (*calls=False. set to True to include 'callee' and 'filepath' to determine caller*) –
- **'filename** (*find=None. set to a string with*) –

- **dataset=False** – “calls” –> “call count” “ncalls” –> “call count” “cumtime” –> “cumulative time” “cumulative”–> “cumulative time” “file” –> “file name” “filename” –> “file name” “line” –> “line number” “module” –> “file name” “name” –> “function name” “nfl” –> “name/file/line” “pcalls” –> “primitive call count” “stdname” –> “standard name” “time” –> “internal time” “tottime” –> “internal time”
- **include** (*other options*) – “calls” –> “call count” “ncalls” –> “call count” “cumtime” –> “cumulative time” “cumulative”–> “cumulative time” “file” –> “file name” “filename” –> “file name” “line” –> “line number” “module” –> “file name” “name” –> “function name” “nfl” –> “name/file/line” “pcalls” –> “primitive call count” “stdname” –> “standard name” “time” –> “internal time” “tottime” –> “internal time”
- **_lsprof.c** (*stats=False. set to True to return all stats collected by*) – return all information collected by the profiler. Each profiler_entry is a tuple-like object with the following attributes:

  code code object callcount how many times this was called reccallcount how many times called recursively totaltime total time in this entry inlinetime inline time in this entry (not in subcalls) calls details of the calls

  The calls attribute is either None or a list of profiler_subentry objects:

  code called code object callcount how many times this is called reccallcount how many times this is called recursively totaltime total time spent in this call inlinetime inline time (not in further subcalls)

riptable.rt_timers.**tocx**(*logger=None*)

Call ticx() followed by code followed by tocx() to time a routine in TSC

**Parameters**

- **logger** (*logging.Logger, optional*) – An optionally-specified logger where the collected timing information is recorded. If not specified (the default), the timing information is written to stdout.
- **also** (*See*) –

riptable.rt_timers.**tt**(*expression*, *loops=1*, *return_time=False*)

tictoc time an expression in nanoseconds. use ; to separate lines

**Parameters**

- **execute** (*arg2 is optional and is how many loops to*) –
- **execute** –

riptable.rt_timers.**ttx**(*expression*, *loops=1*)

tictoc time an expression in TSC (time stamp counters). use ; to separate lines

**Parameters**

- **execute** (*arg2 is optional and is how many loops to*) –
- **execute** –

riptable.rt_timers.**utcnow**(*count=1*)

Call *GetNanoTime* one or more times and return the timestamps in a `DateTimeNano` array.

**Parameters**

**count** (*int, default to 1*) – The number of timestamp samples to collect.

**Returns**
A DateTimeNano array containing the sampled timestamps (representing the current time in UTC nanoseconds).

**Return type**
*DateTimeNano*

### Examples

```
>>> import riptable as rt
>>> rt.utcnow()
DateTimeNano([20190215 11:29:44.022382600])
>>> rt.utcnow()._fa
FastArray([1550248297734812800], dtype=int64)
```

To make an array containing multiple timestamps:

```
>>> len(rt.utcnow(1_000_000))
1000000
```

**See also:**

*GetNanoTime*, datetime.datetime.utcnow

## 2.2.44 riptable.rt_timezone

### Classes

| | |
|---|---|
| *TimeZone* | Stores daylight savings cutoff information so UTC times can be translated to zone-specific times. |

**class** riptable.rt_timezone.**TimeZone**(*from_tz=None*, *to_tz='NYC'*)

Stores daylight savings cutoff information so UTC times can be translated to zone-specific times. Every DateTimeNano object holds a *TimeZone* object. All timezone-related conversions / fixups will be handled by the *TimeZone* class.

**Parameters**

- **from_tz** (*str, defaults to None*) –
- **to_tz** (*str*) –

**_from_tz**

tz databse timezone name - the timezone that the time originates from

**Type**
str

**_dst_cutoffs**

lookup array for converting times from constructor to UTC nano in GMT time

**Type**
numpy.ndarray

**_to_tz**
tz database timezone name - the timezone that the time will be displayed in

**Type**
str

**_timezone_str**
same as _to_str. NOTE: This is actually a property, not a regular attribute.

**_dst_reverse**
lookup array for DateTimeNano to display time in the correct timezone, accounting for daylight savings.

**Type**
numpy.ndarray

**_offset**
offset from GMT for display (non daylight savings)

**_fix_offset**
the offset from the timezone of origin

### Notes

'UTC' is not a timezone, but accepted as an alias for GMT

**property _timezone_str**
Get _to_tz, which is the name for this timezone within the 'tz' database.

**_ALIAS_TIMEZONE_NAMES**

**_TZDB_TIMEZONE_NAMES = ['America/New_York', 'Europe/Dublin', 'UTC', 'GMT', 'Australia/Sydney', 'Asia/Hong_Kong',...**

**tz_error_msg**

**valid_timezones**

**__eq__**(*other*)
Return self==value.

**__repr__**()
Return repr(self).

**classmethod _init_from_tz**(*from_tz*)

**classmethod _init_to_tz**(*to_tz*)
Return daylight savings information, timezone string for correctly displaying the datetime based on the to_tz keyword in the constructor.

**_is_dst**(*arr*)

**_mask_dst**(*arr*, *cutoffs=None*)

**Parameters**

- **arr** – int64 UTC nanoseconds
- **cutoffs** – an array containing daylight savings time starts/ends at midnight possibly a reverse array for GMT that compensates for New York timezone (see DST_REVERSE_NYC)

**_set_timezone**(*tz*)

See DateTimeNano.set_timezone()

**_tz_offset**(*arr*)

**copy**()

A shallow copy of the TimeZone - all attributes are scalars or references to constants.

**fix_dst**(*arr*, *cutoffs=None*)

Called by DateTimeNano routines that need to adjust time for timezone. Also called by DateTimeNanoScalar

**Parameters**

- **arr** (*underlying array of int64, UTC nanoseconds OR a scalar np.int64*) –
- **cutoffs** (*lookup array for daylight savings time cutoffs for the active timezone*) –

### Notes

There is a difference in daylight savings fixup for Dublin timezone. The python datetime.astimezone() routine works differently than fromutctimestamp(). Python datetime may set a 'fold' attribute, indicating that the time is invalid, within an ambiguous daylight savings hour.

```
>>> import datetime
>>> from dateutil import tz
```

```
>>> zone = tz.gettz('Europe/Dublin')
>>> pdt0 = datetime.datetime(2018, 10, 28, 1, 59, 0, tzinfo=zone)
>>> pdt1 = datetime.datetime(2018, 10, 28, 2, 59, 0, tzinfo=zone)
>>> dtn = DateTimeNano(['2018-10-28 01:59', '2018-10-28 02:59'], from_tz=
↪'DUBLIN', to_tz='DUBLIN')
>>> utc = datetime.timezone.utc
```

```
>>> pdt0.astimezone(utc)
datetime.datetime(2018, 10, 28, 0, 59, tzinfo=datetime.timezone.utc)
```

```
>>> pdt1.astimezone(utc)
datetime.datetime(2018, 10, 28, 1, 59, tzinfo=datetime.timezone.utc)
```

```
>>> dtn.astimezone('GMT')
DateTimeNano([20181028 00:59:00.000000000, 20181028 02:59:00.000000000])
```

**static normalize_tz_to_tzdb_name**(*tz_name*)

**to_utc**(*dtn*, *inv_mask=None*)

Called in the DateTimeNano constructor. If necessary, integer arrays of nanoseconds are converted from their timezone of origin to UTC nanoseconds in GMT. Restores any invalids (0) from the original array. This differs from fix_dst() because it adds the offset to the array.

### 2.2.45 riptable.rt_utils

#### Functions

| | |
|---|---|
| *alignmk*(key1, key2, time1, time2[, direction, ...]) | Core routine for merge_asof. |
| *bytes_to_str*(b) | |
| *crc_match*(arrlist) | Perform a CRC check on every array in list, returns True if they were all a match. |
| *describe*(arr[, q, fill_value]) | Similar to pandas describe; columns remain stable, with extra column (Stats) added for names. |
| *findTrueWidth*(string) | Find the length of a byte string without trailing zeros. Useful for optimizing string matching functions. |
| *get_default_value*(arr) | |
| *ischararray*(a) | |
| *islogical*(a) | |
| *load_h5*(filepath[, name, columns, format, fixblocks, ...]) | Load from h5 file and flip hdf5.io objects to riptable structures. |
| *mbget*(aValues, aIndex[, d]) | Provides fancy-indexing functionality similar to `np.take`, but where out-of-bounds indices 'retrieve' a |
| *merge_prebinned*(key1, key2, val1, val2, totalUniqueSize) | merge_prebinned |
| *normalize_keys*(key1, key2[, verbose]) | Helper function to make two different lists of keys the same itemsize. Handles categoricals. |
| *str_to_bytes*(s) | |
| *to_str*(s) | |

riptable.rt_utils.**alignmk**(*key1*, *key2*, *time1*, *time2*, *direction='backward'*, *allow_exact_matches=True*, *verbose=False*)

Core routine for merge_asof.

```
Takes a key1 on the left and a key2 on the right (multikey is allowed).
When going forward, it will check if time1 <= time2
    if so
        it will hash on key1 and return the last row number for key2 or INVALID
        it will increment the index into time1
    else
        it will return the last row number from key2
        it will increment the index into time2

When going backward, it will start on the last time, it will check if time1 >= time2
    if so
        it will hash on key1 and return the last row number for key2 or INVALID
        it will decrement the index into time1
    else
        it will return the last row number from key2
        it will decrement the index into time2
```

**Parameters**

- **key1** (*a numpy array or a list/tuple of numpy arrays*) –
- **key2** (*a numpy array or a list/tuple of numpy arrays*) –
- **time1** (*a monotonic integer array often indicating time, must be same length as key1*) –
- **time2** (*a monotonic integer array often indicating time, must be same length as key2*) –
- **direction** (*{'backward', 'forward', 'nearest'}*) – The alignment direction.
- **allow_exact_matches** (*bool*) –
- **verbose** (*bool*) – When True, enables more-verbose logging output. Defaults to False.

**Returns**

- *Fancy index the same length as key1/time1 (may have invalids)*
- *use the return index to pull from right hand side, for example key2[return]*
- *to populate a dataset with length key1*

### Examples

```
>>> time1=rt.FA([0, 1, 4, 6, 8, 9, 11, 16, 19, 20, 22, 27])
>>> time2=rt.FA([4, 5, 7, 8, 10, 12, 15, 16, 24])
>>> alignmk(rt.ones(time1.shape), rt.ones(time2.shape), time1, time2, direction=
↪'backward')
FastArray([-2147483648, -2147483648, 0, 1, 3, 3, 4, 7, 7, 7, 7, 8])
>>> alignmk(rt.ones(time1.shape), rt.ones(time2.shape), time1, time2, direction=
↪'forward')
FastArray([0, 0, 0, 2, 3, 4, 5, 7, 8, 8, 8, -2147483648])
```

riptable.rt_utils.**bytes_to_str**(*b*)

riptable.rt_utils.**crc_match**(*arrlist*)

Perform a CRC check on every array in list, returns True if they were all a match.

**Parameters**
**arrlist** (*list of numpy arrays*) –

**Returns**
True if all arrays in `arrlist` are structurally equal; otherwise, False.

**Return type**
bool

**See also:**

`numpy.array_equal`

riptable.rt_utils.**describe**(*arr*, *q=None*, *fill_value=None*)

Similar to pandas describe; columns remain stable, with extra column (Stats) added for names.

**Parameters**

- **arr** (*array, list-like, or* `Dataset`) – The data to be described.

- **q** (*list of float, optional*) – List of quantiles, defaults to `[0.10, 0.25, 0.50, 0.75, 0.90]`.
- **fill_value** (*optional*) – Place-holder value for non-computable columns.

**Return type**
*Dataset*

### Examples

```
>>> describe(arange(100) %3)
*Stats     Col0
------   ------
Count    100.00
Valid    100.00
Nans       0.00
Mean       0.99
Std        0.82
Min        0.00
P10        0.00
P25        0.00
P50        1.00
P75        2.00
P90        2.00
Max        2.00
MeanM      0.99

[13 rows x 2 columns] total bytes: 169.0 B
```

riptable.rt_utils.**findTrueWidth**(*string*)

Find the length of a byte string without trailing zeros. Useful for optimizing string matching functions.

**Parameters**
**string** (*a byte string as an array of* int8) – A byte string as an array of int8

**Returns**
Number of bytes in string.

**Return type**
int

### Examples

```
>>> a = np.chararray(1, itemsize=5)
>>> a[0] = b'abc'
>>> findTrueWidth(np.frombuffer(a,dtype=np.int8))
3
```

riptable.rt_utils.**get_default_value**(*arr*)

riptable.rt_utils.**ischararray**(*a*)

riptable.rt_utils.**islogical**(*a*)

riptable.rt_utils.**load_h5**(*filepath*, *name='/'*, *columns=''*, *format=None*, *fixblocks=False*, *drop_short=False*, *verbose=0*, ***kwargs*)

Load from h5 file and flip hdf5.io objects to riptable structures.

In some h5 files, the arrays are saved as rows in "blocks". If `fixblocks` is `True`, this routine will transpose the rows in the blocks.

**Parameters**

- **filepath** (*str or os.PathLike*) – The path to the HDF5 file to load.
- **name** (*str*) – Set to table name, defaults to '/'.
- **columns** (*sequence of str or re.Pattern or callable, defaults to ''*) – Return the given subset of columns, or those matching regex. If a function is passed, it will be called with column names, dtypes and shapes, and should return a subset of column names. Passing an empty string (the default) loads all columns.
- **format** (*hdf5.Format*) – TODO, defaults to hdf5.Format.NDARRAY
- **fixblocks** (*bool*) – True will transpose the rows when the H5 file are as ???, defaults to False.
- **drop_short** (*bool*) – Set to True to drop short rows and never return a Struct, defaults to False.
- **verbose** – TODO

**Returns**

A `Dataset` or `Struct` with all workspace contents.

**Return type**

*Dataset* or *Struct*

### Notes

block<#>_items is a list of column names (bytes) block<#>_values is a numpy array of numpy array (rows) columns (for riptable) can be generated by zipping names from the list with transposed columns

axis0 appears to be all column names - not sure what to do with this also what is axis1? should it get added like the other columns?

riptable.rt_utils.**mbget**(*aValues*, *aIndex*, *d=None*)

Provides fancy-indexing functionality similar to `np.take`, but where out-of-bounds indices 'retrieve' a default value instead of e.g. raising an exception.

It returns an array the same size as the `aIndex` array, with `aValues` in place of the indices and delimiter values (use `d` to customize) for invalid indices.

**Parameters**

- **aValues** (*np.ndarray*) – A single dimension of array values (strings only accepted as chararray).
- **aIndex** (*np.ndarray*) – A single dimension array of int64 indices.
- **d** – An optional argument for a custom default for string operations to use when the index is out of range. (currently always uses the default) `d` is character byte `b''` when `aValues` is a chararray `np.nan` when aValues are floats, `INVALID_POINTER_32` or `INVALID_POINTER_64` when aValues are ints.

**Returns**

**vout** – An array of values in `aValues` that have been looked up according to the indices in `aIndex`. The array will have the same shape as `aIndex`, and the same dtype and class as `aValues`.

**Return type**

np.ndarray

**Raises**

`KeyError` – When the dtype for `aValues` is not int32,int64,float32,float64 and `aValues` is not a chararray.

### Notes

**Tests Performed:**

Large aValues size (28 million) Large aValues typesize (50 for chararray) Large aIndex size (28 million) All indices valid for aIndex in aValues. No indices valid for aIndex in aValues. Empty input arrays. Invalid types for aValues array. Invalid types for aIndex array (not int64 or int32)

The return array vout is the same size as the p array. Suppose we have a position i. If the index stored at position i of p is a valid index for array v, vout at position i will contain the value of v at that index. If the index stored at position i of p is an invalid index, vout at position i will contain the default or custom delimiter value (d).

Match: 4 is at position 2 of the p array. 4 is a valid index in array v (within range). 50 is at position 4 of the v array. Therefore, position 2 of the result vout will contain 50.

Miss: -7 is at position 1 of the p array. -7 is an invalid index in array v (out of range). Therefore, position 1 of the result vout will contain the delimiter.

**Edge Case Tests:**

(TODO)

### Examples

Start with two arrays:

```
>>> v = np.array([10, 20, 30, 40, 50, 60, 70])          #MATLab: v = [10 20 30 40␣
↪50 60 70];
>>> p = np.array([0, -7, 4, 3, 7, 1, 2])               #MATLab: p = [1 -6 5 4 8 2␣
↪3];
>>> vout = mbget(v,p)                                  #MATLab: vout = mbget(v,p);
>>> print(vout)                                        #MATLab: vout
[10  -2147483648  50  40 -2147483648  20  30]    #MATLab: [10.00  NaN  50.00  40.00␣
↪ NaN  20.00  30.00]
```

`riptable.rt_utils.`**`merge_prebinned`**(*key1*, *key2*, *val1*, *val2*, *totalUniqueSize*)

merge_prebinned TODO: Improve docs when working properly

**Parameters**

- **key1** (*a numpy array already binned (like a categorical)*) –
- **key2** (*a numpy array already binned*) –
- **val1** (*int32/64 or float32/64*) –
- **val2** (*int32/64 or float32/64*) –

### Notes

key1 and key2 must be same dtype val1 and val2 must be same dtype

riptable.rt_utils.**normalize_keys**(*key1*, *key2*, *verbose=False*)

Helper function to make two different lists of keys the same itemsize. Handles categoricals.

**Parameters**

- **key1** (*a numpy array or a list/tuple of numpy arrays*) –
- **key2** (*a numpy array or a list/tuple of numpy arrays*) –

**Returns**

If the keys were passed in as single arrays they will be returned as a list of 1 array Integers, Float, String may be upcast if necessary. Categoricals may be aligned if necessary.

**Return type**

Two lists of arrays that are aligned (same itemsize)

### Examples

```
>>> c1 = rt.Cat(['A','B','C'])
>>> c2 = rt.Cat(rt.arange(3) + 1, ['A','B','C'])
>>> [d1], [d2] = rt.normalize_keys(c1, c2)
```

### Notes

TODO: integer, float and string upcasting can be done while rotating.

riptable.rt_utils.**str_to_bytes**(*s*)

riptable.rt_utils.**to_str**(*s*)

Riptable is an open source library built for high-performance data analysis. It's similar to Pandas by design, but it's been optimized to meet the needs of Riptable's core users: quantitative analysts interacting live with large volumes of trading data.

Riptable is based on NumPy, so it shares many core NumPy methods for array-based operations. For users who work with large datasets, Riptable improves on NumPy and Pandas by using multi-threading and efficient memory management, much of it implemented at the C++ level.

Getting Started

New to Riptable? Check out the Intro to Riptable, which takes you through Riptable's main concepts.

*To the Intro*

API Reference

The reference guide has more detailed descriptions of the functions, modules, and objects included in Riptable.

*To the API Reference*

# PYTHON MODULE INDEX

## r


riptable, 167
riptable.config, 195
riptable.conftest, 196
riptable.numba, 192
riptable.numba.indexing, 192
riptable.numba.invalid_values, 193
riptable.rt_accum2, 196
riptable.rt_accumtable, 202
riptable.rt_algos, 206
riptable.rt_bin, 206
riptable.rt_categorical, 211
riptable.rt_compressedarray, 262
riptable.rt_csv, 266
riptable.rt_dataset, 266
riptable.rt_datetime, 340
riptable.rt_display, 394
riptable.rt_ema, 400
riptable.rt_enum, 400
riptable.rt_fastarray, 411
riptable.rt_fastarraynumba, 473
riptable.rt_groupby, 475
riptable.rt_groupbykeys, 482
riptable.rt_groupbynumba, 485
riptable.rt_groupbyops, 488
riptable.rt_grouping, 518
riptable.rt_hstack, 537
riptable.rt_imatrix, 539
riptable.rt_io, 539
riptable.rt_itemcontainer, 540
riptable.rt_ledger, 544
riptable.rt_matplotlib, 546
riptable.rt_merge, 546
riptable.rt_merge_asof, 561
riptable.rt_meta, 566
riptable.rt_misc, 569
riptable.rt_mlutils, 572
riptable.rt_multiset, 572
riptable.rt_numpy, 578
riptable.rt_pdataset, 633
riptable.rt_pgroupby, 639
riptable.rt_sds, 640
riptable.rt_sharedmemory, 648
riptable.rt_sort_cache, 648
riptable.rt_stats, 649
riptable.rt_str, 650
riptable.rt_struct, 661
riptable.rt_timers, 704
riptable.rt_timezone, 707
riptable.rt_utils, 710
riptable.Utils, 167
riptable.Utils.appdirs, 167
riptable.Utils.common, 173
riptable.Utils.conversion_utils, 176
riptable.Utils.display_options, 177
riptable.Utils.ipython_utils, 184
riptable.Utils.pandas_utils, 185
riptable.Utils.rt_display_nested, 186
riptable.Utils.rt_display_properties, 189
riptable.Utils.rt_metadata, 190
riptable.Utils.teamcity_helper, 192
riptable.Utils.terminalsize, 192

# INDEX

## Symbols


_A (*riptable.rt_struct.Struct property*), 663
_ALIAS_TIMEZONE_NAMES (*riptable.rt_timezone.TimeZone attribute*), 708
_APPLY_PARALLEL_THRESHOLD (*riptable.rt_str.FAString attribute*), 652
_ARRAY_UFUNC() (*riptable.rt_ledger.MathLedger class method*), 545
_ASTYPE() (*riptable.rt_ledger.MathLedger class method*), 545
_AS_FA_TYPE() (*riptable.rt_ledger.MathLedger class method*), 545
_AS_FA_TYPE_UNSAFE() (*riptable.rt_ledger.MathLedger class method*), 545
_AUTO_SAVE (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_BAR_GRAPH (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_BASICMATH_ONE_INPUT() (*riptable.rt_ledger.MathLedger class method*), 545
_BASICMATH_TWO_INPUTS() (*riptable.rt_ledger.MathLedger class method*), 545
_BOUNDS (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_CONFIG_LOADED (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_COPY() (*riptable.rt_ledger.MathLedger class method*), 545
_DOUBLING_TRIAL_16 (*in module riptable.Utils.common*), 175
_DOUBLING_TRIAL_32 (*in module riptable.Utils.common*), 175
_DOUBLING_TRIAL_MAX (*in module riptable.Utils.common*), 175
_FUNNEL_ALL() (*riptable.rt_ledger.MathLedger class method*), 545
_FastFunctionsOff() (*riptable.rt_fastarray.FastArray static method*), 420
_FastFunctionsOn() (*riptable.rt_fastarray.FastArray static method*), 420
_G (*riptable.rt_struct.Struct property*), 666
_GCNOW() (*riptable.rt_fastarray.FastArray static method*), 420
_GCSET() (*riptable.rt_fastarray.FastArray static method*), 420
_GETITEM() (*riptable.rt_ledger.MathLedger class method*), 545
_H (*riptable.rt_struct.Struct property*), 667
_HEAT_MAP (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_INDEX_BOOL() (*riptable.rt_ledger.MathLedger class method*), 545
_INT_16_MAX (*in module riptable.Utils.common*), 175
_INT_32_MAX (*in module riptable.Utils.common*), 175
_INT_MAX (*in module riptable.Utils.common*), 175
_INVALID_FREQ_ERROR (*riptable.rt_datetime.DateTimeNano attribute*), 368
_LCLEAR() (*riptable.rt_fastarray.FastArray static method*), 420
_LCLEAR() (*riptable.rt_ledger.MathLedger class method*), 545
_LDUMP() (*riptable.rt_fastarray.FastArray static method*), 420
_LDUMP() (*riptable.rt_ledger.MathLedger class method*), 545
_LDUMPF() (*riptable.rt_fastarray.FastArray static method*), 420
_LOFF() (*riptable.rt_fastarray.FastArray static method*), 420
_LOFF() (*riptable.rt_ledger.MathLedger class method*), 545
_LON() (*riptable.rt_fastarray.FastArray static method*), 420
_LON() (*riptable.rt_ledger.MathLedger class method*), 545
_Ledger() (*riptable.rt_ledger.MathLedger class method*), 545
_LedgerClear() (*riptable.rt_ledger.MathLedger class method*), 545

_LedgerDump() (*riptable.rt_ledger.MathLedger class method*), 545
_LedgerDumpFile() (*riptable.rt_ledger.MathLedger class method*), 545
_LedgerOff() (*riptable.rt_ledger.MathLedger class method*), 545
_LedgerOn() (*riptable.rt_ledger.MathLedger class method*), 545
_MBGET() (*riptable.rt_ledger.MathLedger class method*), 545
_NO_OBJECT (*riptable.Utils.common.cached_weakref_property attribute*), 174
_OFF() (*riptable.rt_fastarray.FastArray static method*), 420
_ON() (*riptable.rt_fastarray.FastArray static method*), 420
_PAINT_MAX (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_PAINT_MIN (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_PAINT_SIGNS (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_PAINT_ZEROS (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_RDUMP() (*riptable.rt_fastarray.FastArray static method*), 420
_REDUCE() (*riptable.rt_ledger.MathLedger class method*), 545
_RESET_OPTIONS (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_ROFF() (*riptable.rt_fastarray.FastArray static method*), 420
_RON() (*riptable.rt_fastarray.FastArray static method*), 420
_SEED (*in module riptable.Utils.common*), 175
_T (*riptable.rt_struct.Struct property*), 668
_TEST_FOOTERS (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_TEST_ONE_PASS (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_TOFF() (*riptable.rt_fastarray.FastArray static method*), 421
_TON() (*riptable.rt_fastarray.FastArray static method*), 421
_TRACEBACK() (*riptable.rt_ledger.MathLedger class method*), 545
_TRANSPOSE (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_TZDB_TIMEZONE_NAMES (*riptable.rt_timezone.TimeZone attribute*), 708
_USERNAME (*riptable.Utils.display_options.DisplayOptions attribute*), 182
_USE_FAST_COUNT_UNIQUES (*riptable.rt_groupbyops.GroupByOps attribute*), 490
_V (*riptable.rt_struct.Struct property*), 669
_V0() (*riptable.rt_fastarray.FastArray static method*), 421
_V1() (*riptable.rt_fastarray.FastArray static method*), 421
_V2() (*riptable.rt_fastarray.FastArray static method*), 421
_WEAKREF_CACHE_NAME (*riptable.Utils.common.cached_weakref_property attribute*), 174
__abs__() (*riptable.rt_dataset.Dataset method*), 273
__abs__() (*riptable.rt_datetime.Date method*), 347
__abs__() (*riptable.rt_datetime.DateTimeNano method*), 368
__abs__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__add__() (*riptable.rt_dataset.Dataset method*), 273
__add__() (*riptable.rt_datetime.Date method*), 347
__add__() (*riptable.rt_datetime.DateSpan method*), 358
__add__() (*riptable.rt_datetime.DateTimeNano method*), 368
__add__() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
__add__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__and__() (*riptable.rt_dataset.Dataset method*), 273
__and__() (*riptable.rt_datetime.Date method*), 347
__and__() (*riptable.rt_datetime.DateTimeNano method*), 368
__array_finalize__() (*riptable.rt_datetime.DateTimeBase method*), 362
__array_finalize__() (*riptable.rt_fastarray.FastArray method*), 421
__array_function__() (*riptable.rt_fastarray.FastArray method*), 421
__array_ufunc__() (*riptable.rt_fastarray.FastArray method*), 421
__arrow_array__() (*riptable.rt_categorical.Categorical method*), 227
__arrow_array__() (*riptable.rt_datetime.Date method*), 347
__arrow_array__() (*riptable.rt_datetime.DateTimeNano method*), 368
__arrow_array__() (*riptable.rt_datetime.TimeSpan method*), 386
__arrow_array__() (*riptable.rt_fastarray.FastArray method*), 422
__bool__() (*riptable.rt_struct.Struct method*), 673

__call__() (*riptable.Utils.rt_display_nested.LeftAligned method*), 188
__ceil__() (*riptable.rt_datetime.Date method*), 348
__ceil__() (*riptable.rt_datetime.DateTimeNano method*), 368
__complex__() (*riptable.rt_datetime.Date method*), 348
__complex__() (*riptable.rt_datetime.DateTimeNano method*), 368
__contains__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__contains__() (*riptable.rt_struct.Struct method*), 673
__del__() (*riptable.rt_accum2.Accum2 method*), 198
__del__() (*riptable.rt_categorical.Categorical method*), 227
__del__() (*riptable.rt_dataset.Dataset method*), 273
__delattr__() (*riptable.rt_struct.Struct method*), 673
__delitem__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__delitem__() (*riptable.rt_struct.Struct method*), 673
__dir__() (*riptable.rt_struct.Struct method*), 673
__enter__() (*riptable.rt_struct.Struct method*), 673
__eq__() (*riptable.rt_categorical.Categorical method*), 227
__eq__() (*riptable.rt_dataset.Dataset method*), 273
__eq__() (*riptable.rt_datetime.Date method*), 348
__eq__() (*riptable.rt_datetime.DateSpan method*), 358
__eq__() (*riptable.rt_datetime.DateTimeNano method*), 368
__eq__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__eq__() (*riptable.rt_fastarray.FastArray method*), 422
__eq__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__eq__() (*riptable.rt_struct.Struct method*), 673
__eq__() (*riptable.rt_timezone.TimeZone method*), 708
__exit__() (*riptable.rt_struct.Struct method*), 673
__float__() (*riptable.rt_datetime.Date method*), 348
__float__() (*riptable.rt_datetime.DateTimeNano method*), 368
__floor__() (*riptable.rt_datetime.Date method*), 348
__floor__() (*riptable.rt_datetime.DateTimeNano method*), 368
__floordiv__() (*riptable.rt_dataset.Dataset method*), 273
__floordiv__() (*riptable.rt_datetime.DateTimeNano method*), 369
__floordiv__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__ge__() (*riptable.rt_categorical.Categorical method*), 227
__ge__() (*riptable.rt_dataset.Dataset method*), 273
__ge__() (*riptable.rt_datetime.Date method*), 348
__ge__() (*riptable.rt_datetime.DateSpan method*), 358
__ge__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ge__() (*riptable.rt_fastarray.FastArray method*), 422
__ge__() (*riptable.rt_struct.Struct method*), 673
__get__() (*riptable.Utils.common.cached_weakref_property method*), 174
__getattr__() (*riptable.rt_groupby.GroupBy method*), 476
__getattr__() (*riptable.rt_struct.Struct method*), 673
__getattribute__() (*riptable.rt_compressedarray.CompressedArray method*), 265
__getitem__() (*riptable.Utils.rt_metadata.MetaData method*), 191
__getitem__() (*riptable.rt_accum2.Accum2 method*), 198
__getitem__() (*riptable.rt_accumtable.AccumTable method*), 202
__getitem__() (*riptable.rt_categorical.Categorical method*), 227
__getitem__() (*riptable.rt_categorical.Categories method*), 259
__getitem__() (*riptable.rt_dataset.Dataset method*), 273
__getitem__() (*riptable.rt_datetime.DateTimeBase method*), 363
__getitem__() (*riptable.rt_display.DisplayColumn method*), 395
__getitem__() (*riptable.rt_fastarray.FastArray method*), 422
__getitem__() (*riptable.rt_groupby.GroupBy method*), 476
__getitem__() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
__getitem__() (*riptable.rt_grouping.Grouping method*), 525
__getitem__() (*riptable.rt_imatrix.IMatrix method*), 539
__getitem__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__getitem__() (*riptable.rt_multiset.Multiset method*), 574
__getitem__() (*riptable.rt_pdataset.PDataset method*), 635
__getitem__() (*riptable.rt_struct.Struct method*), 673
__gt__() (*riptable.rt_categorical.Categorical method*), 228
__gt__() (*riptable.rt_dataset.Dataset method*), 273
__gt__() (*riptable.rt_datetime.Date method*), 348
__gt__() (*riptable.rt_datetime.DateSpan method*), 358
__gt__() (*riptable.rt_datetime.DateTimeNano method*), 369
__gt__() (*riptable.rt_fastarray.FastArray method*), 422
__gt__() (*riptable.rt_struct.Struct method*), 674

__iadd__() (*riptable.rt_dataset.Dataset method*), 273
__iadd__() (*riptable.rt_datetime.Date method*), 348
__iadd__() (*riptable.rt_datetime.DateSpan method*), 358
__iadd__() (*riptable.rt_datetime.DateTimeNano method*), 369
__iand__() (*riptable.rt_dataset.Dataset method*), 273
__iand__() (*riptable.rt_datetime.Date method*), 348
__iand__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ifloordiv__() (*riptable.rt_dataset.Dataset method*), 273
__ifloordiv__() (*riptable.rt_datetime.Date method*), 348
__ifloordiv__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ilshift__() (*riptable.rt_dataset.Dataset method*), 273
__ilshift__() (*riptable.rt_datetime.Date method*), 348
__ilshift__() (*riptable.rt_datetime.DateTimeNano method*), 369
__imatmul__() (*riptable.rt_datetime.Date method*), 348
__imatmul__() (*riptable.rt_datetime.DateTimeNano method*), 369
__imod__() (*riptable.rt_dataset.Dataset method*), 273
__imod__() (*riptable.rt_datetime.Date method*), 348
__imod__() (*riptable.rt_datetime.DateTimeNano method*), 369
__imul__() (*riptable.rt_dataset.Dataset method*), 273
__imul__() (*riptable.rt_datetime.Date method*), 348
__imul__() (*riptable.rt_datetime.DateTimeNano method*), 369
__int__() (*riptable.rt_datetime.Date method*), 348
__int__() (*riptable.rt_datetime.DateTimeNano method*), 369
__invert__() (*riptable.rt_dataset.Dataset method*), 273
__invert__() (*riptable.rt_datetime.Date method*), 348
__invert__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ior__() (*riptable.rt_dataset.Dataset method*), 274
__ior__() (*riptable.rt_datetime.Date method*), 348
__ior__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ipow__() (*riptable.rt_dataset.Dataset method*), 274
__ipow__() (*riptable.rt_datetime.Date method*), 348
__ipow__() (*riptable.rt_datetime.DateTimeNano method*), 369
__irshift__() (*riptable.rt_dataset.Dataset method*), 274
__irshift__() (*riptable.rt_datetime.Date method*), 348
__irshift__() (*riptable.rt_datetime.DateTimeNano method*), 369
__isub__() (*riptable.rt_dataset.Dataset method*), 274
__isub__() (*riptable.rt_datetime.Date method*), 348
__isub__() (*riptable.rt_datetime.DateSpan method*), 358
__isub__() (*riptable.rt_datetime.DateTimeNano method*), 369
__iter__() (*riptable.rt_groupby.GroupBy method*), 476
__iter__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__iter__() (*riptable.rt_struct.Struct method*), 674
__itruediv__() (*riptable.rt_dataset.Dataset method*), 274
__itruediv__() (*riptable.rt_datetime.Date method*), 348
__itruediv__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ixor__() (*riptable.rt_dataset.Dataset method*), 274
__ixor__() (*riptable.rt_datetime.Date method*), 348
__ixor__() (*riptable.rt_datetime.DateTimeNano method*), 369
__le__() (*riptable.rt_categorical.Categorical method*), 228
__le__() (*riptable.rt_dataset.Dataset method*), 274
__le__() (*riptable.rt_datetime.Date method*), 348
__le__() (*riptable.rt_datetime.DateSpan method*), 358
__le__() (*riptable.rt_datetime.DateTimeNano method*), 369
__le__() (*riptable.rt_fastarray.FastArray method*), 422
__le__() (*riptable.rt_struct.Struct method*), 674
__len__() (*riptable.rt_accum2.Accum2 method*), 198
__len__() (*riptable.rt_categorical.Categories method*), 259
__len__() (*riptable.rt_dataset.Dataset method*), 274
__len__() (*riptable.rt_display.DisplayCell method*), 394
__len__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__len__() (*riptable.rt_multiset.Multiset method*), 574
__len__() (*riptable.rt_struct.Struct method*), 674
__lshift__() (*riptable.rt_dataset.Dataset method*), 274
__lshift__() (*riptable.rt_datetime.Date method*), 348
__lshift__() (*riptable.rt_datetime.DateTimeNano method*), 369
__lt__() (*riptable.rt_categorical.Categorical method*), 228
__lt__() (*riptable.rt_dataset.Dataset method*), 274
__lt__() (*riptable.rt_datetime.Date method*), 348
__lt__() (*riptable.rt_datetime.DateSpan method*), 359
__lt__() (*riptable.rt_datetime.DateTimeNano method*), 369
__lt__() (*riptable.rt_fastarray.FastArray method*), 422
__lt__() (*riptable.rt_struct.Struct method*), 674
__matmul__() (*riptable.rt_datetime.Date method*), 348
__matmul__() (*riptable.rt_datetime.DateTimeNano*

*method*), 369
__mod__() (*riptable.rt_dataset.Dataset method*), 274
__mul__() (*riptable.rt_dataset.Dataset method*), 274
__mul__() (*riptable.rt_datetime.Date method*), 348
__mul__() (*riptable.rt_datetime.DateTimeNano method*), 369
__mul__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__ne__() (*riptable.rt_categorical.Categorical method*), 228
__ne__() (*riptable.rt_dataset.Dataset method*), 274
__ne__() (*riptable.rt_datetime.Date method*), 349
__ne__() (*riptable.rt_datetime.DateSpan method*), 359
__ne__() (*riptable.rt_datetime.DateTimeNano method*), 369
__ne__() (*riptable.rt_fastarray.FastArray method*), 422
__ne__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__ne__() (*riptable.rt_struct.Struct method*), 674
__neg__() (*riptable.rt_dataset.Dataset method*), 274
__neg__() (*riptable.rt_datetime.Date method*), 349
__neg__() (*riptable.rt_datetime.DateTimeNano method*), 369
__neg__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__next__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__or__() (*riptable.rt_dataset.Dataset method*), 274
__or__() (*riptable.rt_datetime.Date method*), 349
__or__() (*riptable.rt_datetime.DateTimeNano method*), 369
__pos__() (*riptable.rt_dataset.Dataset method*), 274
__pos__() (*riptable.rt_datetime.Date method*), 349
__pos__() (*riptable.rt_datetime.DateTimeNano method*), 369
__pos__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__pow__() (*riptable.rt_dataset.Dataset method*), 274
__pow__() (*riptable.rt_datetime.Date method*), 349
__pow__() (*riptable.rt_datetime.DateTimeNano method*), 369
__radd__() (*riptable.rt_dataset.Dataset method*), 274
__radd__() (*riptable.rt_datetime.Date method*), 349
__radd__() (*riptable.rt_datetime.DateTimeNano method*), 370
__radd__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__rand__() (*riptable.rt_dataset.Dataset method*), 274
__rand__() (*riptable.rt_datetime.Date method*), 349
__rand__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rdivmod__() (*riptable.rt_datetime.Date method*), 349
__rdivmod__() (*riptable.rt_datetime.DateTimeNano method*), 370
__reduce__() (*riptable.rt_fastarray.FastArray method*), 422
__repr__() (*riptable.Utils.rt_display_properties.ItemFormat method*), 190
__repr__() (*riptable.Utils.rt_metadata.MetaData method*), 191
__repr__() (*riptable.rt_accum2.Accum2 method*), 198
__repr__() (*riptable.rt_accumtable.AccumTable method*), 203
__repr__() (*riptable.rt_categorical.Categorical method*), 228
__repr__() (*riptable.rt_categorical.Categories method*), 259
__repr__() (*riptable.rt_compressedarray.CompressedArray method*), 265
__repr__() (*riptable.rt_dataset.Dataset method*), 274
__repr__() (*riptable.rt_datetime.DateScalar method*), 357
__repr__() (*riptable.rt_datetime.DateSpanScalar method*), 362
__repr__() (*riptable.rt_datetime.DateTimeBase method*), 363
__repr__() (*riptable.rt_datetime.DateTimeNano method*), 370
__repr__() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
__repr__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__repr__() (*riptable.rt_display.DisplayCell method*), 395
__repr__() (*riptable.rt_display.DisplayColumn method*), 395
__repr__() (*riptable.rt_display.DisplayString method*), 396
__repr__() (*riptable.rt_display.DisplayText method*), 400
__repr__() (*riptable.rt_groupby.GroupBy method*), 476
__repr__() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
__repr__() (*riptable.rt_grouping.Grouping method*), 525
__repr__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__repr__() (*riptable.rt_meta.Doc method*), 566
__repr__() (*riptable.rt_meta.Info method*), 567
__repr__() (*riptable.rt_multiset.Multiset method*), 574
__repr__() (*riptable.rt_struct.Struct method*), 674
__repr__() (*riptable.rt_timezone.TimeZone method*), 708
__reversed__() (*riptable.rt_struct.Struct method*), 674
__rfloordiv__() (*riptable.rt_dataset.Dataset method*), 274
__rfloordiv__() (*riptable.rt_datetime.Date method*), 349

__rfloordiv__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rlshift__() (*riptable.rt_datetime.Date method*), 349
__rlshift__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rmatmul__() (*riptable.rt_datetime.Date method*), 349
__rmatmul__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rmod__() (*riptable.rt_dataset.Dataset method*), 274
__rmod__() (*riptable.rt_datetime.Date method*), 349
__rmod__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rmul__() (*riptable.rt_dataset.Dataset method*), 274
__rmul__() (*riptable.rt_datetime.Date method*), 349
__rmul__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rmul__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__ror__() (*riptable.rt_dataset.Dataset method*), 274
__ror__() (*riptable.rt_datetime.Date method*), 349
__ror__() (*riptable.rt_datetime.DateTimeNano method*), 370
__round__() (*riptable.rt_datetime.Date method*), 349
__round__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rpow__() (*riptable.rt_dataset.Dataset method*), 274
__rpow__() (*riptable.rt_datetime.Date method*), 349
__rpow__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rrshift__() (*riptable.rt_datetime.Date method*), 349
__rrshift__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rshift__() (*riptable.rt_dataset.Dataset method*), 274
__rshift__() (*riptable.rt_datetime.Date method*), 349
__rshift__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rsub__() (*riptable.rt_dataset.Dataset method*), 274
__rsub__() (*riptable.rt_datetime.Date method*), 349
__rsub__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rsub__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__rtruediv__() (*riptable.rt_dataset.Dataset method*), 274
__rtruediv__() (*riptable.rt_datetime.Date method*), 349
__rtruediv__() (*riptable.rt_datetime.DateTimeNano method*), 370
__rxor__() (*riptable.rt_dataset.Dataset method*), 275
__rxor__() (*riptable.rt_datetime.Date method*), 349
__rxor__() (*riptable.rt_datetime.DateTimeNano method*), 370
__set_name__() (*riptable.Utils.common.cached_weakref_property method*), 174
__setattr__() (*riptable.Utils.display_options.DisplayOptions method*), 183
__setattr__() (*riptable.rt_struct.Struct method*), 674
__setitem2__() (*riptable.rt_categorical.Categorical method*), 228
__setitem__() (*riptable.Utils.rt_metadata.MetaData method*), 191
__setitem__() (*riptable.rt_accumtable.AccumTable method*), 203
__setitem__() (*riptable.rt_categorical.Categorical method*), 228
__setitem__() (*riptable.rt_dataset.Dataset method*), 275
__setitem__() (*riptable.rt_display.DisplayColumn method*), 395
__setitem__() (*riptable.rt_fastarray.FastArray method*), 423
__setitem__() (*riptable.rt_itemcontainer.ItemContainer method*), 541
__setitem__() (*riptable.rt_multiset.Multiset method*), 574
__setitem__() (*riptable.rt_struct.Struct method*), 674
__slots__ (*riptable.rt_datetime.DateScalar attribute*), 357
__slots__ (*riptable.rt_datetime.DateSpanScalar attribute*), 362
__slots__ (*riptable.rt_datetime.DateTimeNanoScalar attribute*), 384
__slots__ (*riptable.rt_datetime.TimeSpanScalar attribute*), 388
__str__() (*riptable.Utils.rt_display_properties.ItemFormat method*), 190
__str__() (*riptable.Utils.rt_metadata.MetaData method*), 191
__str__() (*riptable.rt_accum2.Accum2 method*), 199
__str__() (*riptable.rt_categorical.Categorical method*), 228
__str__() (*riptable.rt_categorical.Categories method*), 259
__str__() (*riptable.rt_compressedarray.CompressedArray method*), 265
__str__() (*riptable.rt_dataset.Dataset method*), 275
__str__() (*riptable.rt_datetime.DateScalar method*), 357
__str__() (*riptable.rt_datetime.DateSpanScalar method*), 362
__str__() (*riptable.rt_datetime.DateTimeBase method*), 363
__str__() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
__str__() (*riptable.rt_datetime.TimeSpanScalar method*), 389

__str__() (*riptable.rt_display.DisplayCell method*), 395
__str__() (*riptable.rt_display.DisplayColumn method*), 395
__str__() (*riptable.rt_display.DisplayString method*), 396
__str__() (*riptable.rt_display.DisplayText method*), 400
__str__() (*riptable.rt_groupby.GroupBy method*), 477
__str__() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
__str__() (*riptable.rt_meta.Doc method*), 566
__str__() (*riptable.rt_meta.Info method*), 567
__str__() (*riptable.rt_multiset.Multiset method*), 574
__str__() (*riptable.rt_struct.Struct method*), 674
__sub__() (*riptable.rt_dataset.Dataset method*), 275
__sub__() (*riptable.rt_datetime.Date method*), 349
__sub__() (*riptable.rt_datetime.DateSpan method*), 359
__sub__() (*riptable.rt_datetime.DateTimeNano method*), 370
__sub__() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
__sub__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__truediv__() (*riptable.rt_dataset.Dataset method*), 275
__truediv__() (*riptable.rt_datetime.DateTimeNano method*), 370
__truediv__() (*riptable.rt_datetime.TimeSpanScalar method*), 389
__trunc__() (*riptable.rt_datetime.Date method*), 349
__trunc__() (*riptable.rt_datetime.DateTimeNano method*), 370
__version__ (*in module riptable.Utils.appdirs*), 173
__version_info__ (*in module riptable.Utils.appdirs*), 173
__xor__() (*riptable.rt_dataset.Dataset method*), 275
__xor__() (*riptable.rt_datetime.Date method*), 349
__xor__() (*riptable.rt_datetime.DateTimeNano method*), 370
_accum1_pass() (*riptable.rt_accum2.Accum2 class method*), 199
_add_allnames() (*riptable.rt_dataset.Dataset method*), 275
_add_labels_footers_summaries() (*riptable.rt_dataset.Dataset method*), 275
_add_nano_ext() (*riptable.rt_datetime.DateTimeBase static method*), 363
_add_totals() (*riptable.rt_accum2.Accum2 class method*), 199
_addnewitem() (*riptable.rt_struct.Struct method*), 674
_addnewitem_allnames() (*riptable.rt_struct.Struct method*), 674
_aggregate_column_matches() (*riptable.rt_struct.Struct method*), 675
_align_array_info() (*riptable.rt_struct.Struct class method*), 675
_anydict (*riptable.rt_grouping.Grouping property*), 522
_apply_2d_operation() (*riptable.rt_accum2.Accum2 class method*), 199
_apply_func() (*riptable.rt_str.FAString method*), 653
_apply_outlier() (*riptable.rt_dataset.Dataset method*), 275
_argmax() (*riptable.rt_fastarray.FastArray static method*), 423
_argmin() (*riptable.rt_fastarray.FastArray static method*), 423
_arr_info() (*in module riptable.Utils.rt_display_nested*), 188
_array_compiled_numba_apply() (*riptable.rt_categorical.Categorical static method*), 228
_array_edit() (*riptable.rt_categorical.Categories method*), 259
_array_info_list() (*riptable.rt_struct.Struct class method*), 675
_array_summary() (*riptable.rt_struct.Struct class method*), 675
_array_summary_single() (*riptable.rt_struct.Struct class method*), 675
_as_dictionary() (*riptable.rt_struct.Struct method*), 675
_as_if_dark() (*riptable.rt_display.DisplayText static method*), 400
_as_info() (*riptable.rt_meta.Doc method*), 566
_as_itemcontainer() (*riptable.rt_dataset.Dataset method*), 275
_as_meta_data() (*riptable.rt_categorical.Categorical method*), 228
_as_meta_data() (*riptable.rt_datetime.DateTimeBase method*), 363
_as_meta_data() (*riptable.rt_struct.Struct method*), 675
_attach_self_as_key_column() (*riptable.rt_categorical.Categorical method*), 229
_auto_pnames() (*riptable.rt_pdataset.PDataset class method*), 635
_autocomplete() (*riptable.rt_categorical.Categorical method*), 229
_autocomplete() (*riptable.rt_dataset.Dataset method*), 275
_autocomplete() (*riptable.rt_multiset.Multiset method*), 574
_autocomplete() (*riptable.rt_pdataset.PDataset method*), 635
_autocomplete() (*riptable.rt_struct.Struct method*), 676

_axis_key() (*riptable.rt_dataset.Dataset static method*), 275
_build_col_headers() (*riptable.rt_multiset.Multiset static method*), 574
_build_end() (*riptable.rt_display.DisplayColumn method*), 395
_build_footers() (*riptable.rt_multiset.Multiset method*), 574
_build_method() (*riptable.rt_str.CatString class method*), 650
_build_nested_ascii() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
_build_nested_html() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
_build_property() (*riptable.rt_str.CatString class method*), 650
_build_sds_meta_data() (*riptable.rt_accum2.Accum2 method*), 199
_build_sds_meta_data() (*riptable.rt_categorical.Categorical method*), 229
_build_sds_meta_data() (*riptable.rt_datetime.DateTimeBase method*), 363
_build_sds_meta_data() (*riptable.rt_struct.Struct method*), 676
_build_string() (*riptable.rt_accum2.Accum2 method*), 199
_build_string() (*riptable.rt_categorical.Categorical method*), 229
_build_string() (*riptable.rt_categorical.Categories method*), 259
_build_string() (*riptable.rt_compressedarray.CompressedArray method*), 265
_build_string() (*riptable.rt_datetime.DateTimeBase method*), 363
_build_string() (*riptable.rt_groupby.GroupBy method*), 477
_build_string() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
_build_unique_dict() (*riptable.rt_grouping.Grouping method*), 525
_cache (*riptable.rt_sort_cache.SortCache attribute*), 648
_calc_badslots() (*riptable.rt_accum2.Accum2 class method*), 199
_calc_multipass() (*riptable.rt_accum2.Accum2 class method*), 199
_calc_onepass() (*riptable.rt_accum2.Accum2 class method*), 199
_calculate_all() (*riptable.rt_accum2.Accum2 method*), 200
_calculate_all() (*riptable.rt_categorical.Categorical method*), 229
_calculate_all() (*riptable.rt_groupby.GroupBy method*), 477
_calculate_all() (*riptable.rt_groupbyops.GroupByOps method*), 490
_calculate_all() (*riptable.rt_grouping.Grouping method*), 525
_cat_info() (*in module riptable.Utils.rt_display_nested*), 188
_categorical_compare_check() (*riptable.rt_categorical.Categorical method*), 229
_categories (*riptable.rt_categorical.Categorical property*), 217
_category_make_unique_multi_key() (*riptable.rt_categorical.Categorical method*), 229
_check_add_dimensions() (*riptable.rt_dataset.Dataset method*), 275
_check_addtype() (*riptable.rt_dataset.Dataset method*), 275
_check_addtype() (*riptable.rt_multiset.Multiset method*), 575
_check_addtype() (*riptable.rt_struct.Struct method*), 676
_check_mathops() (*riptable.rt_datetime.Date method*), 349
_check_mathops() (*riptable.rt_datetime.DateSpan method*), 359
_check_mathops_nano() (*riptable.rt_datetime.Date method*), 350
_check_mathops_nano() (*riptable.rt_datetime.DateSpan method*), 359
_check_ndim() (*riptable.rt_fastarray.FastArray class method*), 423
_compare_check() (*riptable.rt_fastarray.FastArray method*), 423
_construct_new_footers() (*riptable.rt_dataset.Dataset method*), 275
_convert_datestring() (*riptable.rt_datetime.Date class method*), 350
_convert_fastring_output() (*riptable.rt_str.CatString method*), 650
_convert_matlab_days() (*riptable.rt_datetime.Date class method*), 350
_convert_matlab_days() (*riptable.rt_datetime.DateTimeNano class method*), 370
_copy() (*riptable.rt_categorical.Categories method*), 259

_copy() (*riptable.rt_dataset.Dataset method*), 275
_copy() (*riptable.rt_multiset.Multiset method*), 575
_copy() (*riptable.rt_pdataset.PDataset method*), 635
_copy() (*riptable.rt_struct.Struct method*), 676
_copy_attributes() (*riptable.rt_dataset.Dataset method*), 276
_copy_base() (*riptable.rt_struct.Struct method*), 676
_copy_extra() (*riptable.rt_categorical.Categorical method*), 229
_copy_from_dict() (*riptable.rt_struct.Struct method*), 676
_dataset (*riptable.rt_groupbyops.GroupByOps attribute*), 490
_dataset_compare_check() (*riptable.rt_dataset.Dataset method*), 276
_date_compare_check() (*riptable.rt_datetime.Date method*), 350
_datespan_compare_check() (*riptable.rt_datetime.DateSpan method*), 359
_datetimenano_compare_check() (*riptable.rt_datetime.DateTimeNano method*), 370
_default_info() (*in module riptable.Utils.rt_display_nested*), 188
_deleteitem() (*riptable.rt_struct.Struct method*), 676
_depth_first() (*riptable.rt_multiset.Multiset static method*), 575
_descrip (*riptable.rt_meta.Doc attribute*), 566
_detail (*riptable.rt_meta.Doc attribute*), 566
_dict_val_at_index() (*riptable.rt_groupbyops.GroupByOps method*), 490
_dst_cutoffs (*riptable.rt_timezone.TimeZone attribute*), 707
_dst_reverse (*riptable.rt_timezone.TimeZone attribute*), 708
_dtypes_by_group (*in module riptable.Utils.common*), 175
_ema_op() (*riptable.rt_groupbyops.GroupByOps method*), 490
_empty_allowed() (*riptable.rt_grouping.Grouping method*), 526
_empty_like() (*riptable.rt_fastarray.FastArray static method*), 423
_ensure_atomic() (*riptable.rt_struct.Struct method*), 676
_ensure_vector() (*riptable.rt_dataset.Dataset method*), 276
_escape_invalid_file_chars() (*riptable.rt_struct.Struct method*), 676
_expand_array() (*riptable.rt_categorical.Categorical method*), 229
_extract_indexing() (*riptable.rt_struct.Struct method*), 676
_fa (*riptable.rt_categorical.Categorical property*), 217
_fa (*riptable.rt_datetime.DateScalar property*), 357
_fa (*riptable.rt_datetime.DateSpanScalar property*), 362
_fa (*riptable.rt_datetime.DateTimeBase property*), 362
_fa (*riptable.rt_datetime.DateTimeNanoScalar property*), 384
_fa (*riptable.rt_datetime.TimeSpanScalar property*), 388
_fa_filter_wrapper() (*riptable.rt_fastarray.FastArray method*), 423
_fa_keyword_wrapper() (*riptable.rt_fastarray.FastArray method*), 423
_filenames_to_pnames() (*riptable.rt_pdataset.PDataset class method*), 636
_fill_invalid_internal() (*riptable.rt_fastarray.FastArray method*), 423
_finalize_dataset() (*riptable.rt_grouping.Grouping method*), 526
_find() (*riptable.rt_str.FAString method*), 653
_finish_calculate_all() (*riptable.rt_accum2.Accum2 method*), 200
_first_list (*riptable.rt_categorical.Categories property*), 258
_fix_offset (*riptable.rt_timezone.TimeZone attribute*), 708
_flatten_undo() (*riptable.rt_struct.Struct class method*), 677
_footers_exist() (*riptable.rt_dataset.Dataset method*), 276
_format() (*riptable.rt_display.DisplayText static method*), 400
_from_arrow() (*riptable.rt_categorical.Categorical static method*), 229
_from_arrow() (*riptable.rt_datetime.Date static method*), 350
_from_arrow() (*riptable.rt_datetime.DateTimeNano static method*), 370
_from_arrow() (*riptable.rt_datetime.TimeSpan static method*), 386
_from_arrow() (*riptable.rt_fastarray.FastArray static method*), 423
_from_categories() (*riptable.rt_grouping.Grouping method*), 526
_from_maybe_non_unique_labels() (*riptable.rt_categorical.Categorical class method*), 230
_from_meta_data() (*riptable.rt_categorical.Categorical class method*), 230
_from_meta_data() (*riptable.rt_datetime.DateTimeNano class method*), 371
_from_meta_data() (*riptable.rt_datetime.TimeSpan class method*), 386

_from_meta_data() (*riptable.rt_struct.Struct class method*), 677
_from_sds_onefile() (*riptable.rt_struct.Struct class method*), 677
_from_tz (*riptable.rt_timezone.TimeZone attribute*), 707
_funnel_mathops() (*riptable.rt_datetime.DateTimeBase method*), 363
_gb_keyword_wrapper() (*riptable.rt_groupbyops.GroupByOps method*), 490
_gb_quantile_name() (*riptable.rt_groupbyops.GroupByOps static method*), 491
_get_agg_func() (*riptable.rt_groupbyops.GroupByOps method*), 491
_get_array() (*riptable.rt_categorical.Categories method*), 259
_get_calculate_dict() (*riptable.rt_grouping.Grouping method*), 526
_get_codes() (*riptable.rt_categorical.Categories method*), 259
_get_columns() (*riptable.rt_dataset.Dataset method*), 276
_get_count_for_slice() (*riptable.rt_struct.Struct method*), 677
_get_default_path() (*riptable.Utils.display_options.DisplayOptions static method*), 183
_get_dict() (*riptable.rt_categorical.Categories method*), 260
_get_filter_bin_name() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
_get_final_display_mode() (*riptable.rt_struct.Struct method*), 677
_get_gbkeyname() (*riptable.rt_accum2.Accum2 method*), 200
_get_gbkeys() (*riptable.rt_accum2.Accum2 method*), 200
_get_index_from_tuple() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
_get_mapping() (*riptable.rt_categorical.Categories method*), 260
_get_move_cols() (*riptable.rt_itemcontainer.ItemContainer method*), 541
_get_seq() (*riptable.rt_struct.Struct static method*), 677
_get_username() (*riptable.Utils.display_options.DisplayOptions static method*), 183
_get_win_folder (*in module riptable.Utils.appdirs*), 173
_get_win_folder_from_registry() (*in module riptable.Utils.appdirs*), 169
_get_win_folder_with_ctypes() (*in module riptable.Utils.appdirs*), 169
_get_win_folder_with_jna() (*in module riptable.Utils.appdirs*), 169
_get_win_folder_with_pywin32() (*in module riptable.Utils.appdirs*), 169
_getitem() (*riptable.rt_groupby.GroupBy method*), 477
_getitem_enum() (*riptable.rt_categorical.Categories method*), 260
_getitem_multikey() (*riptable.rt_categorical.Categories method*), 260
_getitem_singlekey() (*riptable.rt_categorical.Categories method*), 260
_getsingleitem() (*riptable.rt_categorical.Categorical method*), 230
_grouping (*riptable.rt_categorical.Categories attribute*), 259
_grouping_data_as_dict() (*riptable.rt_groupby.GroupBy method*), 477
_guard_math_op() (*riptable.rt_datetime.DateTimeNano method*), 371
_header_color() (*riptable.rt_display.DisplayText static method*), 400
_hstack() (*riptable.rt_grouping.Grouping static method*), 526
_imatrix_y_internal() (*riptable.rt_dataset.Dataset method*), 276
_index_from_row_labels() (*riptable.rt_struct.Struct method*), 677
_info_tree() (*riptable.rt_struct.Struct class method*), 677
_init_columns_as_dict() (*riptable.rt_dataset.Dataset method*), 276
_init_from_dict() (*riptable.rt_dataset.Dataset method*), 276
_init_from_dict() (*riptable.rt_multiset.Multiset method*), 575
_init_from_dict() (*riptable.rt_struct.Struct method*), 677
_init_from_itemcontainer() (*riptable.rt_dataset.Dataset method*), 276
_init_from_list() (*riptable.rt_pdataset.PDataset class method*), 636
_init_from_pandas_df() (*riptable.rt_dataset.Dataset method*), 276
_init_from_tz() (*riptable.rt_timezone.TimeZone class method*), 708

_init_pnames_filenames() (*riptable.rt_pdataset.PDataset class method*), 636
_init_to_tz() (*riptable.rt_timezone.TimeZone class method*), 708
_insert_filter_bin() (*riptable.rt_groupbykeys.GroupByKeys method*), 483
_insert_filter_label() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
_internal_getitem() (*riptable.rt_accum2.Accum2 method*), 200
_internal_self_compare() (*riptable.rt_fastarray.FastArray method*), 424
_ipython_key_completions_() (*riptable.rt_categorical.Categorical method*), 230
_ipython_key_completions_() (*riptable.rt_dataset.Dataset method*), 276
_ipython_key_completions_() (*riptable.rt_pdataset.PDataset method*), 636
_ipython_key_completions_() (*riptable.rt_struct.Struct method*), 677
_is_dst() (*riptable.rt_timezone.TimeZone method*), 708
_is_float_encodable() (*riptable.rt_dataset.Dataset method*), 276
_is_not_supported() (*riptable.rt_fastarray.FastArray method*), 424
_is_valid_mapping_code() (*riptable.rt_categorical.Categories method*), 260
_isfiltered (*riptable.rt_str.CatString property*), 650
_iter_internal() (*riptable.rt_groupbyops.GroupByOps method*), 491
_iter_internal_contiguous() (*riptable.rt_groupbyops.GroupByOps method*), 491
_keys_as_list() (*riptable.rt_groupbyops.GroupByOps method*), 491
_kwarg_check() (*riptable.rt_fastarray.FastArray method*), 424
_labels_footers_summaries_conform() (*riptable.rt_dataset.Dataset method*), 276
_last_row_stats() (*riptable.rt_dataset.Dataset method*), 276
_last_row_stats() (*riptable.rt_multiset.Multiset method*), 575
_last_row_stats() (*riptable.rt_struct.Struct method*), 677
_lastrepr (*riptable.rt_struct.Struct attribute*), 673
_lastreprhtml (*riptable.rt_struct.Struct attribute*), 673
_legacy_array_function() (*riptable.rt_fastarray.FastArray method*), 424
_load_from_sds_meta_data() (*riptable.rt_accum2.Accum2 class method*), 200
_load_from_sds_meta_data() (*riptable.rt_categorical.Categorical class method*), 230
_load_from_sds_meta_data() (*riptable.rt_datetime.Date class method*), 350
_load_from_sds_meta_data() (*riptable.rt_datetime.DateSpan class method*), 359
_load_from_sds_meta_data() (*riptable.rt_datetime.DateTimeNano class method*), 371
_load_from_sds_meta_data() (*riptable.rt_datetime.TimeSpan class method*), 386
_load_from_sds_meta_data() (*riptable.rt_struct.Struct class method*), 677
_load_from_sds_meta_data_nested() (*riptable.rt_struct.Struct class method*), 678
_load_without_meta_data() (*riptable.rt_struct.Struct class method*), 678
_lock() (*riptable.rt_struct.Struct method*), 678
_logging (*riptable.rt_sort_cache.SortCache attribute*), 648
_make_accum_dataset() (*riptable.rt_grouping.Grouping method*), 527
_make_enumikey() (*riptable.rt_grouping.Grouping method*), 527
_make_imatrix() (*riptable.rt_accum2.Accum2 method*), 200
_make_isortrows() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
_make_isortrows() (*riptable.rt_grouping.Grouping method*), 527
_make_text() (*riptable.rt_meta.Info method*), 567
_makecat() (*riptable.rt_dataset.Dataset method*), 276
_map_asciitree() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
_map_full_paths() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
_map_htmltree() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
_mapping_edit() (*riptable.rt_categorical.Categories method*), 260
_mapping_new() (*riptable.rt_categorical.Categories*

*method*), 260
_mask_dst() (*riptable.rt_timezone.TimeZone method*), 708
_mask_flags() (*in module riptable.Utils.rt_display_nested*), 188
_mask_get_item() (*riptable.rt_struct.Struct method*), 678
_mask_reduce() (*riptable.rt_dataset.Dataset method*), 276
_math_error_string() (*riptable.rt_datetime.DateTimeBase method*), 363
_max() (*riptable.rt_fastarray.FastArray static method*), 424
_mean() (*riptable.rt_fastarray.FastArray static method*), 424
_meta_dict() (*riptable.rt_categorical.Categorical method*), 231
_meta_dict() (*riptable.rt_datetime.DateTimeBase method*), 363
_meta_dict() (*riptable.rt_datetime.DateTimeNano method*), 371
_meta_dict() (*riptable.rt_datetime.TimeSpan method*), 386
_meta_dict() (*riptable.rt_struct.Struct method*), 678
_min() (*riptable.rt_fastarray.FastArray static method*), 424
_nan_idx() (*riptable.rt_categorical.Categorical method*), 231
_nanargmax() (*riptable.rt_fastarray.FastArray static method*), 424
_nanargmin() (*riptable.rt_fastarray.FastArray static method*), 424
_nanfunc() (*riptable.rt_categorical.Categorical method*), 231
_nanmax() (*riptable.rt_fastarray.FastArray static method*), 424
_nanmean() (*riptable.rt_fastarray.FastArray static method*), 424
_nanmin() (*riptable.rt_fastarray.FastArray static method*), 424
_nanstd() (*riptable.rt_fastarray.FastArray static method*), 424
_nansum() (*riptable.rt_fastarray.FastArray static method*), 424
_nanvar() (*riptable.rt_fastarray.FastArray static method*), 424
_nb_char() (*riptable.rt_str.FAString method*), 653
_nb_contains() (*riptable.rt_str.FAString method*), 653
_nb_endswith() (*riptable.rt_str.FAString method*), 653
_nb_fill_backend() (*riptable.rt_groupbynumba.GroupbyNumba static method*), 485
_nb_find() (*riptable.rt_str.FAString method*), 653
_nb_groupbycalculateall() (*riptable.rt_groupbynumba.GroupbyNumba method*), 485
_nb_groupbycalculateallpack() (*riptable.rt_groupbynumba.GroupbyNumba method*), 485
_nb_index() (*riptable.rt_str.FAString method*), 653
_nb_index_any_of() (*riptable.rt_str.FAString method*), 653
_nb_lower() (*riptable.rt_str.FAString method*), 653
_nb_removetrailing() (*riptable.rt_str.FAString method*), 653
_nb_replace() (*riptable.rt_str.FAString method*), 653
_nb_reverse() (*riptable.rt_str.FAString method*), 653
_nb_reverse_inplace() (*riptable.rt_str.FAString method*), 653
_nb_startswith() (*riptable.rt_str.FAString method*), 654
_nb_strlen() (*riptable.rt_str.FAString method*), 654
_nb_substr() (*riptable.rt_str.FAString method*), 654
_nb_upper() (*riptable.rt_str.FAString method*), 654
_nb_upper_inplace() (*riptable.rt_str.FAString method*), 654
_new_array_function() (*riptable.rt_fastarray.FastArray method*), 424
_normalize_column() (*riptable.rt_dataset.Dataset method*), 277
_np (*riptable.rt_datetime.DateScalar property*), 357
_np (*riptable.rt_datetime.DateSpanScalar property*), 362
_np (*riptable.rt_datetime.DateTimeNanoScalar property*), 384
_np (*riptable.rt_datetime.TimeSpanScalar property*), 388
_np (*riptable.rt_fastarray.FastArray property*), 416
_nth() (*riptable.rt_groupbyops.GroupByOps method*), 491
_numbaEMA() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486
_numbaEMA2() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486
_numbaFillBackward() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486
_numbaFillForward() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486
_numbaTrim() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486
_numba_fill_direction() (*riptable.rt_groupbynumba.GroupbyNumba static method*), 486
_numbamin() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486
_numbasum() (*riptable.rt_groupbynumba.GroupbyNumba method*), 486

_object_as_string() (*riptable.rt_dataset.Dataset method*), 277
_offset (*riptable.rt_timezone.TimeZone attribute*), 708
_operate_iter_input_cols() (*riptable.rt_dataset.Dataset method*), 277
_parse_item_format() (*riptable.rt_datetime.Date static method*), 350
_parse_item_format() (*riptable.rt_datetime.DateTimeNano class method*), 371
_pop_gb_data() (*riptable.rt_groupby.GroupBy method*), 477
_pop_gb_data() (*riptable.rt_groupbyops.GroupByOps method*), 491
_possibly_add_categories() (*riptable.rt_categorical.Categories method*), 260
_possibly_convert() (*riptable.rt_dataset.Dataset method*), 277
_possibly_convert_array() (*riptable.rt_dataset.Dataset method*), 277
_possibly_transform() (*riptable.rt_groupbyops.GroupByOps method*), 491
_possibly_warn() (*riptable.rt_fastarray.FastArray class method*), 425
_post_init() (*riptable.rt_dataset.Dataset method*), 277
_post_init() (*riptable.rt_pdataset.PDataset method*), 636
_post_init() (*riptable.rt_struct.Struct method*), 678
_pre_init() (*riptable.rt_dataset.Dataset method*), 278
_pre_init() (*riptable.rt_pdataset.PDataset method*), 636
_pre_init() (*riptable.rt_struct.Struct method*), 678
_prepare_display_data() (*riptable.rt_dataset.Dataset method*), 278
_prepare_display_data() (*riptable.rt_struct.Struct method*), 678
_prepare_gb_data() (*riptable.rt_groupbyops.GroupByOps method*), 491
_prepend_invalid() (*riptable.rt_categorical.Categorical method*), 231
_pull_from_ifirstkey() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
_py_number_to_np_dtype() (*riptable.rt_fastarray.FastArray class method*), 425
_quantile() (*riptable.rt_groupbyops.GroupByOps method*), 492
_quantile_funcParam_from_q() (*riptable.rt_groupbyops.GroupByOps static method*), 493
_quantile_q_from_funcParam() (*riptable.rt_groupbyops.GroupByOps static method*), 493
_random() (*riptable.rt_datetime.DateTimeNano class method*), 371
_reduce_check() (*riptable.rt_fastarray.FastArray method*), 425
_reduce_op_identity_value (*riptable.rt_fastarray.FastArray attribute*), 419
_rename_summary_row_and_col() (*riptable.rt_accumtable.AccumTable method*), 203
_replaceitem() (*riptable.rt_struct.Struct method*), 678
_replaceitem_allnames() (*riptable.rt_struct.Struct method*), 678
_repr_html_() (*riptable.rt_dataset.Dataset method*), 278
_repr_html_() (*riptable.rt_display.DisplayString method*), 396
_repr_html_() (*riptable.rt_display.DisplayText method*), 400
_repr_html_() (*riptable.rt_meta.Doc method*), 566
_repr_html_() (*riptable.rt_meta.Info method*), 567
_repr_html_() (*riptable.rt_multiset.Multiset method*), 575
_repr_html_() (*riptable.rt_struct.Struct method*), 678
_restricted_names (*riptable.rt_struct.Struct attribute*), 673
_return_dataset() (*riptable.rt_grouping.Grouping method*), 527
_round_() (*riptable.rt_fastarray.FastArray static method*), 425
_row_numbers (*riptable.rt_pdataset.PDataset property*), 633
_row_numbers (*riptable.rt_struct.Struct property*), 671
_run_once() (*riptable.rt_struct.Struct method*), 678
_safe_reordering_of_renames() (*riptable.rt_struct.Struct method*), 678
_scalar_compiled_numba_apply() (*riptable.rt_categorical.Categorical static method*), 231
_scalar_info() (*in module riptable.Utils.rt_display_nested*), 188
_scalar_summary() (*riptable.rt_struct.Struct class method*), 678
_searchsorted() (*in module riptable.rt_numpy*), 588
_serialize_item() (*riptable.rt_struct.Struct class method*), 678
_set_anydict() (*riptable.rt_grouping.Grouping method*), 527
_set_attribute() (*riptable.rt_itemcontainer.ItemContainer method*),

541
_set_newinstance() (*riptable.rt_grouping.Grouping method*), 527
_set_timezone() (*riptable.rt_timezone.TimeZone method*), 708
_sizeof_fmt() (*riptable.rt_struct.Struct static method*), 678
_sort_column_styles() (*riptable.rt_struct.Struct method*), 678
_sort_columns (*riptable.rt_dataset.Dataset property*), 269
_sort_columns (*riptable.rt_struct.Struct property*), 671
_sort_lexsort() (*riptable.rt_dataset.Dataset method*), 278
_sort_values() (*riptable.rt_dataset.Dataset method*), 278
_stack_dataset() (*riptable.rt_accum2.Accum2 method*), 200
_std() (*riptable.rt_fastarray.FastArray static method*), 425
_steward (*riptable.rt_meta.Doc attribute*), 566
_struct_compare_check() (*riptable.rt_struct.Struct method*), 678
_styles (*riptable.rt_struct.Struct property*), 671
_substr() (*riptable.rt_str.FAString method*), 654
_sum() (*riptable.rt_fastarray.FastArray static method*), 425
_summary_len (*riptable.rt_struct.Struct attribute*), 673
_superadditem() (*riptable.rt_struct.Struct method*), 679
_tagged_as_dict() (*riptable.rt_itemcontainer.ItemContainer method*), 541
_tagged_get_dict_max() (*riptable.rt_itemcontainer.ItemContainer method*), 541
_tagged_get_names() (*riptable.rt_itemcontainer.ItemContainer method*), 541
_tagged_remove() (*riptable.rt_itemcontainer.ItemContainer method*), 541
_tagged_set_names() (*riptable.rt_itemcontainer.ItemContainer method*), 541
_temp_display() (*riptable.rt_struct.Struct method*), 679
_test_cat_ismember (*riptable.rt_categorical.Categorical attribute*), 227
_tf_spacer() (*riptable.rt_categorical.Categorical method*), 231
_timezone_str (*riptable.rt_timezone.TimeZone attribute*), 708
_timezone_str (*riptable.rt_timezone.TimeZone property*), 708
_title_color() (*riptable.rt_display.DisplayText static method*), 400
_to_tz (*riptable.rt_timezone.TimeZone attribute*), 707
_total_size (*riptable.rt_categorical.Categorical property*), 218
_transformed_scalar_compiled_numba_apply() (*riptable.rt_categorical.Categorical static method*), 231
_tree_from_sds_meta_data() (*riptable.rt_struct.Struct class method*), 679
_trim_keys() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
_type (*riptable.rt_meta.Doc attribute*), 566
_tz_offset() (*riptable.rt_timezone.TimeZone method*), 709
_unary_op() (*riptable.rt_fastarray.FastArray method*), 425
_unlock() (*riptable.rt_struct.Struct method*), 679
_update_sort() (*riptable.rt_struct.Struct method*), 679
_validate_input() (*riptable.rt_str.FAString method*), 654
_validate_names() (*riptable.rt_struct.Struct method*), 679
_var() (*riptable.rt_fastarray.FastArray static method*), 425
_view_internal() (*riptable.rt_fastarray.FastArray method*), 425
_year_splits (*riptable.rt_datetime.Date property*), 342
_yearday_splits (*riptable.rt_datetime.Date property*), 342
_yearday_splits_leap (*riptable.rt_datetime.Date property*), 342

## A

ABS (*riptable.rt_enum.MATH_OPERATION attribute*), 404
abs() (*in module riptable.rt_numpy*), 588
abs() (*riptable.rt_dataset.Dataset method*), 278
abs() (*riptable.rt_datetime.TimeSpanScalar method*), 389
abs() (*riptable.rt_fastarray.FastArray method*), 425
abs() (*riptable.rt_multiset.Multiset method*), 575
absolute() (*in module riptable.rt_numpy*), 588
accum1() (*riptable.rt_dataset.Dataset method*), 279
Accum2 (*class in riptable.rt_accum2*), 196
Accum2 (*riptable.rt_enum.TypeRegister attribute*), 409
accum2() (*riptable.rt_dataset.Dataset method*), 279
Accum2t (*riptable.rt_enum.DisplayColumnColors attribute*), 401
accum_cols() (*in module riptable.rt_accumtable*), 204
accum_ratio() (*in module riptable.rt_accumtable*), 204

accum_ratiop() (*in module riptable.rt_accumtable*), 205
ACCUM_X_MAX (*riptable.rt_accum2.Accum2 attribute*), 198
AccumTable (*class in riptable.rt_accumtable*), 202
ADD (*riptable.rt_enum.MATH_OPERATION attribute*), 404
add (*riptable.rt_fastarray.FastArray attribute*), 419
add_footer() (*riptable.rt_display.DisplayColumn method*), 395
add_matrix() (*riptable.rt_dataset.Dataset method*), 280
add_required_columns() (*riptable.rt_display.DisplayTable method*), 397
add_totals() (*riptable.rt_groupby.GroupBy method*), 477
agg() (*riptable.rt_groupbyops.GroupByOps method*), 493
AggNames (*riptable.rt_groupbyops.GroupByOps attribute*), 490
aggregate() (*riptable.rt_groupbyops.GroupByOps method*), 494
align() (*riptable.rt_categorical.Categorical class method*), 231
align_column() (*riptable.rt_display.DisplayColumn method*), 395
align_console_string() (*riptable.rt_display.DisplayColumn static method*), 395
align_html() (*riptable.rt_display.DisplayColumn static method*), 396
alignmk() (*in module riptable.rt_utils*), 710
All (*riptable.rt_enum.NumpyCharTypes attribute*), 406
all() (*in module riptable.rt_numpy*), 588
all() (*riptable.rt_dataset.Dataset method*), 280
all() (*riptable.rt_multiset.Multiset method*), 575
all() (*riptable.rt_struct.Struct method*), 679
all_columns_console() (*riptable.rt_display.DisplayTable method*), 397
all_columns_console_multiline() (*riptable.rt_display.DisplayTable method*), 397
all_unique (*riptable.rt_grouping.Grouping property*), 522
AllFloat (*riptable.rt_enum.NumpyCharTypes attribute*), 406
AllInteger (*riptable.rt_enum.NumpyCharTypes attribute*), 406
AllNames (*riptable.rt_struct.Struct attribute*), 672
AllowAnyName (*riptable.rt_struct.Struct attribute*), 673
allowed_funcs (*riptable.rt_compressedarray.CompressedArray attribute*), 265
any() (*in module riptable.rt_numpy*), 588
any() (*riptable.rt_dataset.Dataset method*), 280
any() (*riptable.rt_multiset.Multiset method*), 575
any() (*riptable.rt_struct.Struct method*), 679
AppDirs (*class in riptable.Utils.appdirs*), 168
append_dataset_dict() (*in module riptable.Utils.conversion_utils*), 176
apply() (*riptable.rt_categorical.Categorical method*), 232
apply() (*riptable.rt_dataset.Dataset method*), 280
apply() (*riptable.rt_fastarray.FastArray method*), 425
apply() (*riptable.rt_groupbyops.GroupByOps method*), 494
apply() (*riptable.rt_grouping.Grouping method*), 527
apply() (*riptable.rt_itemcontainer.ItemContainer method*), 541
apply() (*riptable.rt_multiset.Multiset method*), 576
apply() (*riptable.rt_str.FAString method*), 654
apply2d() (*riptable.rt_imatrix.IMatrix method*), 539
apply_cols() (*riptable.rt_dataset.Dataset method*), 281
apply_cols() (*riptable.rt_multiset.Multiset method*), 576
apply_helper() (*riptable.rt_grouping.Grouping method*), 529
apply_nonreduce() (*riptable.rt_categorical.Categorical method*), 232
apply_nonreduce() (*riptable.rt_groupbyops.GroupByOps method*), 494
apply_numba() (*riptable.rt_fastarray.FastArray method*), 427
apply_pandas() (*riptable.rt_fastarray.FastArray method*), 427
apply_reduce() (*riptable.rt_accum2.Accum2 method*), 200
apply_reduce() (*riptable.rt_groupbyops.GroupByOps method*), 495
apply_rows() (*riptable.rt_dataset.Dataset method*), 282
apply_rows() (*riptable.rt_multiset.Multiset method*), 576
apply_rows_numba() (*riptable.rt_dataset.Dataset method*), 283
apply_schema() (*in module riptable.rt_meta*), 568
apply_schema() (*riptable.rt_fastarray.FastArray method*), 429
apply_schema() (*riptable.rt_struct.Struct method*), 680
appname (*in module riptable.Utils.appdirs*), 173
arange() (*in module riptable.rt_numpy*), 588
argmax() (*riptable.rt_dataset.Dataset method*), 284
argmax() (*riptable.rt_fastarray.FastArray method*), 429
argmin() (*riptable.rt_dataset.Dataset method*), 284
argmin() (*riptable.rt_fastarray.FastArray method*), 429
argpartition2() (*riptable.rt_fastarray.FastArray method*), 429

argsort() (*in module riptable.rt_numpy*), 589
argsort() (*riptable.rt_categorical.Categorical method*), 232
as_categorical() (*riptable.rt_groupby.GroupBy method*), 477
as_filter() (*riptable.rt_groupbyops.GroupByOps method*), 496
as_filter() (*riptable.rt_grouping.Grouping method*), 530
as_matrix() (*riptable.rt_dataset.Dataset method*), 284
as_meta_data() (*riptable.rt_enum.TypeRegister class method*), 410
as_ordered_dictionary() (*riptable.rt_struct.Struct method*), 680
as_pandas_df() (*riptable.rt_dataset.Dataset method*), 284
as_recordarray() (*riptable.rt_dataset.Dataset method*), 284
as_singlekey() (*riptable.rt_categorical.Categorical method*), 232
as_string_array (*riptable.rt_categorical.Categorical property*), 218
as_struct() (*riptable.rt_dataset.Dataset method*), 285
asanyarray (*in module riptable.rt_numpy*), 632
asanyarray() (*in module riptable.rt_numpy*), 589
asarray (*in module riptable.rt_numpy*), 632
asarray() (*in module riptable.rt_numpy*), 589
asdict() (*riptable.rt_struct.Struct method*), 680
asrows() (*riptable.rt_dataset.Dataset method*), 285
assoc_copy() (*in module riptable.rt_numpy*), 589
assoc_index() (*in module riptable.rt_numpy*), 590
astimezone() (*riptable.rt_datetime.DateTimeNano method*), 371
astype() (*riptable.rt_dataset.Dataset method*), 285
astype() (*riptable.rt_fastarray.FastArray method*), 429
astype() (*riptable.rt_multiset.Multiset method*), 576
attribute (*riptable.Utils.rt_display_nested.AttributeTraversal attribute*), 186
AttributeTraversal (*class in riptable.Utils.rt_display_nested*), 186
auto_add_off() (*riptable.rt_categorical.Categorical method*), 232
auto_add_on() (*riptable.rt_categorical.Categorical method*), 233
autocomplete() (*in module riptable.rt_misc*), 569

## B

backfill() (*riptable.rt_groupby.GroupBy method*), 477
backtostring (*riptable.rt_str.FAString property*), 650
base_index (*riptable.rt_categorical.Categorical property*), 219
base_index (*riptable.rt_categorical.Categories property*), 258
base_index (*riptable.rt_grouping.Grouping property*), 522
between() (*riptable.rt_fastarray.FastArray method*), 430
BGColor (*riptable.rt_enum.DisplayColumnColors attribute*), 402
bincount() (*in module riptable.rt_numpy*), 590
bitcount() (*in module riptable.rt_numpy*), 590
BITWISE_AND (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_ANDNOT (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_LSHIFT (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_NOT (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_NOTAND (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_OR (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_RSHIFT (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_XOR (*riptable.rt_enum.MATH_OPERATION attribute*), 404
BITWISE_XOR_SPECIAL (*riptable.rt_enum.MATH_OPERATION attribute*), 404
Bool (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
bool_ (*class in riptable.rt_numpy*), 583
bool_to_fancy() (*in module riptable.rt_numpy*), 591
BOX_LIGHT (*in module riptable.Utils.rt_display_nested*), 188
BoxStyle (*class in riptable.Utils.rt_display_nested*), 187
build_column() (*riptable.rt_display.DisplayTable method*), 397
build_dicts_enum() (*riptable.rt_categorical.Categories class method*), 260
build_dicts_python() (*riptable.rt_categorical.Categories class method*), 260
build_final_ends() (*riptable.rt_display.DisplayTable method*), 397
build_final_headers_console() (*riptable.rt_display.DisplayTable method*), 397
build_final_headers_html() (*riptable.rt_display.DisplayTable method*), 397
build_footer() (*riptable.rt_display.DisplayColumn method*), 396

build_header() (*riptable.rt_display.DisplayColumn method*), 396
build_header_tuples() (*in module riptable.rt_misc*), 570
build_nested_html() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
build_nested_string() (*riptable.Utils.rt_display_nested.DisplayNested method*), 187
build_result_table() (*riptable.rt_display.DisplayTable method*), 397
build_result_table_new() (*riptable.rt_display.DisplayTable method*), 397
build_row_mask() (*riptable.rt_display.DisplayTable method*), 398
build_summary() (*riptable.rt_display.DisplayColumn method*), 396
build_transposed_columns() (*riptable.rt_display.DisplayTable method*), 398
Bytes (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
bytes_ (*class in riptable.rt_numpy*), 583
bytes_to_str() (*in module riptable.rt_utils*), 711

## C

cached_property (*in module riptable.Utils.common*), 175
cached_weakref_property (*class in riptable.Utils.common*), 174
Calendar (*riptable.rt_enum.TypeRegister attribute*), 409
cascade() (*riptable.rt_multiset.Multiset method*), 576
cat() (*riptable.rt_dataset.Dataset method*), 286
cat2keys() (*in module riptable.rt_numpy*), 592
cat2keys() (*riptable.rt_dataset.Dataset method*), 287
Categorical (*class in riptable.rt_categorical*), 211
Categorical (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
Categorical (*riptable.rt_enum.TypeRegister attribute*), 409
categorical_convert() (*in module riptable.rt_categorical*), 261
categorical_merge_dict() (*in module riptable.rt_categorical*), 262
Categories (*class in riptable.rt_categorical*), 257
Categories (*riptable.rt_enum.TypeRegister attribute*), 409
categories() (*riptable.rt_categorical.Categorical method*), 233
categories_as_dict() (*riptable.rt_categorical.Categories method*), 260
categories_equal() (*riptable.rt_categorical.Categorical class method*), 234
category_add() (*riptable.rt_categorical.Categorical method*), 234
category_array (*riptable.rt_categorical.Categorical property*), 219
category_codes (*riptable.rt_categorical.Categorical property*), 221
category_dict (*riptable.rt_categorical.Categorical property*), 221
category_make_unique() (*riptable.rt_categorical.Categorical method*), 234
category_mapping (*riptable.rt_categorical.Categorical property*), 221
category_mode (*riptable.rt_categorical.Categorical property*), 221
category_remove() (*riptable.rt_categorical.Categorical method*), 234
category_replace() (*riptable.rt_categorical.Categorical method*), 234
catinstance (*riptable.rt_grouping.Grouping property*), 522
CatString (*class in riptable.rt_str*), 650
CatZero() (*in module riptable.rt_categorical*), 261
CBRT (*riptable.rt_enum.MATH_OPERATION attribute*), 404
CEIL (*riptable.rt_enum.MATH_OPERATION attribute*), 404
ceil() (*in module riptable.rt_numpy*), 593
Center (*riptable.rt_enum.DisplayJustification attribute*), 402
char() (*riptable.rt_str.FAString method*), 655
Character (*riptable.rt_enum.NumpyCharTypes attribute*), 406
child_head() (*riptable.Utils.rt_display_nested.BoxStyle method*), 187
child_head() (*riptable.Utils.rt_display_nested.Style method*), 188
child_tail() (*riptable.Utils.rt_display_nested.BoxStyle method*), 187
child_tail() (*riptable.Utils.rt_display_nested.Style method*), 188
class_error() (*in module riptable.rt_stats*), 649
clear() (*riptable.rt_fastarray.Ledger static method*), 471
clear() (*riptable.rt_itemcontainer.ItemContainer method*), 541
clip_lower() (*riptable.rt_fastarray.FastArray method*), 431
clip_upper() (*riptable.rt_fastarray.FastArray method*), 431

CMP_EQ (*riptable.rt_enum.MATH_OPERATION attribute*), 404
CMP_GT (*riptable.rt_enum.MATH_OPERATION attribute*), 404
CMP_GTE (*riptable.rt_enum.MATH_OPERATION attribute*), 404
CMP_LT (*riptable.rt_enum.MATH_OPERATION attribute*), 405
CMP_LTE (*riptable.rt_enum.MATH_OPERATION attribute*), 405
CMP_NE (*riptable.rt_enum.MATH_OPERATION attribute*), 405
col_add_prefix() (*riptable.rt_struct.Struct method*), 681
col_add_suffix() (*riptable.rt_struct.Struct method*), 681
COL_ALL (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 181
col_delete (*riptable.rt_struct.Struct attribute*), 673
col_exists() (*riptable.rt_struct.Struct method*), 682
col_filter() (*riptable.rt_struct.Struct method*), 682
col_get_attribute() (*riptable.rt_struct.Struct method*), 683
col_get_len() (*riptable.rt_struct.Struct method*), 684
col_get_value() (*riptable.rt_struct.Struct method*), 684
col_map() (*riptable.rt_struct.Struct method*), 684
COL_MAX (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 181
COL_MIN (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 181
col_move() (*riptable.rt_struct.Struct method*), 685
col_move_to_back() (*riptable.rt_struct.Struct method*), 685
col_move_to_front() (*riptable.rt_struct.Struct method*), 685
col_pop() (*riptable.rt_struct.Struct method*), 686
col_remove() (*riptable.rt_struct.Struct method*), 687
col_rename() (*riptable.rt_struct.Struct method*), 689
col_replace_all() (*riptable.rt_dataset.Dataset method*), 288
col_set_attribute() (*riptable.rt_struct.Struct method*), 690
col_set_value() (*riptable.rt_struct.Struct method*), 690
col_str_match() (*riptable.rt_struct.Struct method*), 690
col_str_replace() (*riptable.rt_struct.Struct method*), 691
col_swap() (*riptable.rt_struct.Struct method*), 692
COL_T (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 181
ColHeader (*in module riptable.rt_enum*), 411
COLOR_MODE (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 181
color_mode_dict (*riptable.rt_display.DisplayCell attribute*), 394
ColorMode (*riptable.rt_display.DisplayDetect attribute*), 396
combine2groups() (*in module riptable.rt_grouping*), 536
combine2keys() (*in module riptable.rt_numpy*), 593
combine_accum1_filter() (*in module riptable.rt_numpy*), 594
combine_accum2_filter() (*in module riptable.rt_numpy*), 594
combine_filter() (*in module riptable.rt_numpy*), 595
Complex (*riptable.rt_enum.NumpyCharTypes attribute*), 406
compress_dataset_internal() (*in module riptable.rt_sds*), 641
CompressedArray (*class in riptable.rt_compressedarray*), 262
CompressPickle (*riptable.rt_fastarray.FastArray attribute*), 419
Computable (*riptable.rt_enum.NumpyCharTypes attribute*), 406
computable() (*riptable.rt_dataset.Dataset method*), 288
concat_columns() (*riptable.rt_dataset.Dataset class method*), 288
concat_rows() (*riptable.rt_dataset.Dataset class method*), 289
concat_structs() (*riptable.rt_struct.Struct class method*), 692
concatenate() (*in module riptable.rt_numpy*), 595
Console (*riptable.rt_enum.DisplayDetectModes attribute*), 402
console_detect_settings() (*riptable.rt_display.DisplayTable class method*), 398
CONSOLE_X (*riptable.Utils.display_options.DisplayOptions attribute*), 177, 181
CONSOLE_X_BUFFER (*riptable.Utils.display_options.DisplayOptions attribute*), 177, 181
CONSOLE_X_HTML (*riptable.Utils.display_options.DisplayOptions attribute*), 177, 181
console_x_offset (*riptable.rt_display.DisplayTable attribute*), 397
CONSOLE_Y (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 181
container_from_filetype() (*in module riptable.rt_sds*), 641
contains() (*riptable.rt_str.FAString method*), 655
contains_np_arrays() (*riptable.rt_groupbyops.GroupByOps static*

*method*), 496
convert_func_dict (*riptable.Utils.rt_display_properties.DisplayConvert attribute*), 189
convertBool() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
convertBytes() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
convertDefault() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
convertFloat() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
ConvertFuncCache (*riptable.Utils.rt_display_properties.DisplayConvert attribute*), 189
convertInt() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
convertMultiDims() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
convertRecord() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 189
convertString() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 190
ConvertTypeCache (*riptable.Utils.rt_display_properties.DisplayConvert attribute*), 189
copy() (*riptable.rt_categorical.Categorical method*), 234
copy() (*riptable.rt_categorical.Categories method*), 260
copy() (*riptable.rt_dataset.Dataset method*), 291
copy() (*riptable.rt_datetime.DateTimeBase method*), 363
copy() (*riptable.rt_fastarray.FastArray method*), 431
copy() (*riptable.rt_groupby.GroupBy method*), 477
copy() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
copy() (*riptable.rt_grouping.Grouping method*), 530
copy() (*riptable.rt_itemcontainer.ItemContainer method*), 541
copy() (*riptable.rt_multiset.Multiset method*), 576
copy() (*riptable.rt_pgroupby.PGroupBy method*), 639
copy() (*riptable.rt_struct.Struct method*), 693
copy() (*riptable.rt_timezone.TimeZone method*), 709
copy() (*riptable.Utils.rt_display_properties.ItemFormat method*), 190
copy_apply() (*riptable.rt_itemcontainer.ItemContainer method*), 541
copy_from() (*riptable.rt_grouping.Grouping method*), 530
copy_inplace() (*riptable.rt_itemcontainer.ItemContainer method*), 542
copy_invalid() (*riptable.rt_categorical.Categorical method*), 235
copy_invalid() (*riptable.rt_fastarray.FastArray method*), 432
CORE_COUNT (*riptable.rt_groupbynumba.GroupbyNumba attribute*), 485
count() (*riptable.rt_accum2.Accum2 method*), 201
count() (*riptable.rt_categorical.Categorical method*), 236
count() (*riptable.rt_dataset.Dataset method*), 292
count() (*riptable.rt_fastarray.FastArray method*), 433
count() (*riptable.rt_groupby.GroupBy method*), 478
count() (*riptable.rt_groupbyops.GroupByOps method*), 496
count() (*riptable.rt_grouping.Grouping method*), 530
count_uniques() (*riptable.rt_groupbyops.GroupByOps method*), 496
crc (*riptable.rt_dataset.Dataset property*), 269
crc (*riptable.rt_fastarray.FastArray property*), 417
crc32c() (*in module riptable.rt_numpy*), 595
crc64() (*in module riptable.rt_numpy*), 595
crc_match() (*in module riptable.rt_utils*), 711
css_color_classes (*riptable.rt_display.DisplayColumn attribute*), 395
css_decoration_classes (*riptable.rt_display.DisplayColumn attribute*), 395
css_justification_classes (*riptable.rt_display.DisplayColumn attribute*), 395
cumcount() (*riptable.rt_groupbyops.GroupByOps method*), 497
cummax() (*riptable.rt_groupbyops.GroupByOps method*), 497
cummin() (*riptable.rt_groupbyops.GroupByOps method*), 497
cumprod() (*riptable.rt_groupbyops.GroupByOps method*), 497
cumsum() (*in module riptable.rt_numpy*), 595
cumsum() (*riptable.rt_groupbyops.GroupByOps method*), 497
CUSTOM_COMPLETION (*riptable.Utils.display_options.DisplayOptions attribute*), 181
cut() (*in module riptable.rt_bin*), 206
cut_time() (*riptable.rt_datetime.DateTimeNano method*), 372

## D


darkbg_styles (*riptable.rt_display.DisplayCell attribute*), 394
DarkBlue (*riptable.rt_enum.DisplayColumnColors attribute*), 402
data (*riptable.rt_display.DisplayColumn property*), 395
Dataset (*class in riptable.rt_dataset*), 266
Dataset (*riptable.rt_enum.TypeRegister attribute*), 409
dataset (*riptable.rt_imatrix.IMatrix property*), 539
dataset_as_matrix() (*in module riptable.Utils.conversion_utils*), 176
dataset_as_pandas_df() (*in module riptable.Utils.pandas_utils*), 185
dataset_from_pandas_df() (*in module riptable.Utils.pandas_utils*), 185
Date (*class in riptable.rt_datetime*), 341
Date (*riptable.rt_enum.TypeRegister attribute*), 410
DateBase (*riptable.rt_enum.TypeRegister attribute*), 410
DateScalar (*class in riptable.rt_datetime*), 357
DateSpan (*class in riptable.rt_datetime*), 358
DateSpan (*riptable.rt_enum.TypeRegister attribute*), 410
DateSpanScalar (*class in riptable.rt_datetime*), 362
datestring_to_nano() (*in module riptable.rt_datetime*), 390
DateTime (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
Datetime (*riptable.rt_enum.NumpyCharTypes attribute*), 406
DATETIME_TYPES (*class in riptable.rt_enum*), 401
DateTimeBase (*class in riptable.rt_datetime*), 362
DateTimeBase (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
DateTimeBase (*riptable.rt_enum.TypeRegister attribute*), 410
DateTimeNano (*class in riptable.rt_datetime*), 364
DateTimeNano (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
DateTimeNano (*riptable.rt_enum.TypeRegister attribute*), 410
DateTimeNanoScalar (*class in riptable.rt_datetime*), 384
datetimestring_to_nano() (*in module riptable.rt_datetime*), 391
DateTimeUTC() (*in module riptable.rt_datetime*), 390
day_of_month (*riptable.rt_datetime.Date property*), 342
day_of_week (*riptable.rt_datetime.Date property*), 342
day_of_year (*riptable.rt_datetime.Date property*), 343
DebugMode (*riptable.rt_accum2.Accum2 attribute*), 198
DebugMode (*riptable.rt_categorical.Categorical attribute*), 226
DebugMode (*riptable.rt_display.DisplayTable attribute*), 396
DebugMode (*riptable.rt_groupby.GroupBy attribute*), 476
DebugMode (*riptable.rt_groupbyops.GroupByOps attribute*), 490
DebugMode (*riptable.rt_grouping.Grouping attribute*), 525
DebugMode (*riptable.rt_pgroupby.PGroupBy attribute*), 639
DebugUFunc (*riptable.rt_ledger.MathLedger attribute*), 545
decompress() (*riptable.rt_compressedarray.CompressedArray method*), 265
decompress_dataset_internal() (*in module riptable.rt_sds*), 641
Default (*riptable.rt_enum.DisplayColumnColors attribute*), 402
default_colname (*riptable.rt_categorical.Categories attribute*), 259
default_dict (*riptable.Utils.rt_metadata.MetaData attribute*), 191
DEFAULT_FORMATTER (*riptable.rt_datetime.DateTimeBase attribute*), 362
default_item_formats (*in module riptable.Utils.rt_display_properties*), 190
deref_idx() (*in module riptable.numba.indexing*), 192
deregister() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 415
deregister_array_function() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 415
deregister_array_function_type_compatibility() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 415
describe() (*in module riptable.rt_utils*), 711
describe() (*riptable.rt_dataset.Dataset method*), 292
describe() (*riptable.rt_groupbyops.GroupByOps method*), 498
describe() (*riptable.rt_multiset.Multiset method*), 576
description (*riptable.rt_meta.Info attribute*), 566
description (*riptable.rt_meta.Item attribute*), 567
detail (*riptable.rt_meta.Info attribute*), 566
dhead() (*riptable.rt_dataset.Dataset method*), 293
dict (*riptable.Utils.rt_metadata.MetaData property*), 191
dict_modes (*riptable.rt_categorical.Categories attribute*), 259
DictTraversal (*class in riptable.Utils.rt_display_nested*), 187
diff() (*in module riptable.rt_numpy*), 595
diff() (*riptable.rt_datetime.Date method*), 351
diff() (*riptable.rt_datetime.DateTimeNano method*), 372
diff() (*riptable.rt_fastarray.FastArray method*), 433
diff() (*riptable.rt_groupbyops.GroupByOps method*), 498

differs() (*riptable.rt_fastarray.FastArray method*), 434
display() (*riptable.rt_display.DisplayCell method*), 395
display_attributes() (*riptable.rt_struct.Struct method*), 693
display_convert_func() (*riptable.rt_accum2.Accum2 method*), 201
display_convert_func() (*riptable.rt_categorical.Categorical static method*), 237
display_convert_func() (*riptable.rt_datetime.Date static method*), 351
display_convert_func() (*riptable.rt_datetime.DateSpan static method*), 359
display_convert_func() (*riptable.rt_datetime.DateTimeNano static method*), 372
display_detect() (*riptable.rt_display.DisplayTable static method*), 398
display_html() (*riptable.rt_display.DisplayTable static method*), 398
display_item() (*riptable.rt_datetime.DateTimeBase method*), 364
display_item() (*riptable.rt_datetime.DateTimeNano method*), 373
display_length (*riptable.rt_datetime.DateTimeBase property*), 362
display_length (*riptable.rt_datetime.DateTimeNano property*), 368
display_precision() (*riptable.rt_display.DisplayTable static method*), 398
display_query_properties() (*riptable.rt_accum2.Accum2 method*), 201
display_query_properties() (*riptable.rt_categorical.Categorical method*), 237
display_query_properties() (*riptable.rt_datetime.DateTimeNano method*), 373
display_query_properties() (*riptable.rt_fastarray.FastArray method*), 435
display_rows() (*riptable.rt_display.DisplayTable static method*), 398
display_threshold() (*riptable.rt_display.DisplayTable static method*), 399
display_width (*riptable.rt_display.DisplayColumn property*), 395
DisplayArrayTypes (*class in riptable.rt_enum*), 401
DisplayAttributes (*riptable.rt_enum.TypeRegister attribute*), 410
DisplayCell (*class in riptable.rt_display*), 394
DisplayColumn (*class in riptable.rt_display*), 395
DisplayColumnColors (*class in riptable.rt_enum*), 401
DisplayConvert (*class in riptable.Utils.rt_display_properties*), 189
DisplayDetect (*class in riptable.rt_display*), 396
DisplayDetect (*riptable.rt_enum.TypeRegister attribute*), 410
DisplayDetectModes (*class in riptable.rt_enum*), 402
DisplayJustification (*class in riptable.rt_enum*), 402
DisplayLength (*class in riptable.rt_enum*), 402
DisplayNested (*class in riptable.Utils.rt_display_nested*), 187
DisplayOptions (*class in riptable.Utils.display_options*), 177
DisplayOptions (*riptable.rt_enum.TypeRegister attribute*), 410
DisplayString (*class in riptable.rt_display*), 396
DisplayString (*riptable.rt_enum.TypeRegister attribute*), 410
DisplayTable (*class in riptable.rt_display*), 396
DisplayTable (*riptable.rt_enum.TypeRegister attribute*), 410
DisplayText (*class in riptable.rt_display*), 399
DisplayText (*riptable.rt_enum.TypeRegister attribute*), 410
DIV (*riptable.rt_enum.MATH_OPERATION attribute*), 405
div (*riptable.rt_fastarray.FastArray attribute*), 419
Doc (*class in riptable.rt_meta*), 566
doc (*riptable.rt_fastarray.FastArray property*), 417
doc (*riptable.rt_struct.Struct property*), 672
doc() (*in module riptable.rt_meta*), 568
DOCRC (*riptable.rt_ledger.MathLedger attribute*), 545
docstring_imports() (*in module riptable.conftest*), 196
docstring_merge_datasets() (*in module riptable.conftest*), 196
double() (*in module riptable.rt_numpy*), 595
draw (*riptable.Utils.rt_display_nested.LeftAligned attribute*), 188
drop_duplicates() (*riptable.rt_dataset.Dataset method*), 293
DS_DISPLAY_TYPES (*class in riptable.rt_enum*), 401
dset_dict_to_list() (*in module riptable.Utils.conversion_utils*), 176
DT_BOOL (*riptable.rt_enum.SM_DTYPES attribute*), 408
DT_BYTE (*riptable.rt_enum.SM_DTYPES attribute*), 408
DT_BYTES (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_CHARARRAY (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_DATETIME64 (*riptable.rt_enum.SM_DTYPES attribute*), 409

DT_FLOAT16 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_FLOAT32 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_FLOAT64 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_HALF (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_INT16 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_INT32 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_INT64 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_INT8 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_INVALID (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_NPVOID (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_OBJECT (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_TIMEDELTA64 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_UINT16 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_UINT32 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_UINT64 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_UINT8 (*riptable.rt_enum.SM_DTYPES attribute*), 409
DT_UNICODE (*riptable.rt_enum.SM_DTYPES attribute*), 409
dtail() (*riptable.rt_dataset.Dataset method*), 295
dtranspose() (*riptable.rt_struct.Struct method*), 693
dtypes (*riptable.rt_dataset.Dataset property*), 270
dtypes (*riptable.rt_multiset.Multiset property*), 573
dtypes_by_group (*in module riptable.Utils.common*), 175
dump() (*riptable.rt_fastarray.Ledger static method*), 471
duplicated() (*riptable.rt_dataset.Dataset method*), 295
duplicated() (*riptable.rt_fastarray.FastArray method*), 435

## E

E_MAX (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 181
e_max() (*riptable.Utils.display_options.DisplayOptions class method*), 183
e_max() (*riptable.Utils.display_options.DisplayOptions method*), 180
E_MIN (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 181
e_min() (*riptable.Utils.display_options.DisplayOptions class method*), 183
e_min() (*riptable.Utils.display_options.DisplayOptions method*), 180
E_PRECISION (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 181
E_THRESHOLD (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 182
ema_decay() (*riptable.rt_groupbyops.GroupByOps method*), 498
ema_normal() (*riptable.rt_groupbyops.GroupByOps method*), 498
ema_weighted() (*riptable.rt_groupbyops.GroupByOps method*), 499
empty() (*in module riptable.rt_numpy*), 595
empty_like() (*in module riptable.rt_numpy*), 596
enable_custom_attribute_completion() (*in module riptable.Utils.ipython_utils*), 184
enable_numba_cache (*riptable.config.Settings attribute*), 195
endswith() (*riptable.rt_str.FAString method*), 656
eq() (*riptable.rt_fastarray.FastArray method*), 436
equals() (*riptable.rt_dataset.Dataset method*), 296
equals() (*riptable.rt_struct.Struct method*), 694
ESC (*riptable.rt_display.DisplayText attribute*), 399
EXP (*riptable.rt_enum.MATH_OPERATION attribute*), 405
EXP2 (*riptable.rt_enum.MATH_OPERATION attribute*), 405
expand_any() (*riptable.rt_categorical.Categorical method*), 237
expand_array (*riptable.rt_categorical.Categorical property*), 221
expand_dict (*riptable.rt_categorical.Categorical property*), 223
expanding() (*riptable.rt_groupby.GroupBy method*), 478
EXPM1 (*riptable.rt_enum.MATH_OPERATION attribute*), 405
extract() (*riptable.rt_str.CatString method*), 650
extract() (*riptable.rt_str.FAString method*), 656
extract_groups() (*riptable.rt_grouping.Grouping static method*), 530

## F

FABS (*riptable.rt_enum.MATH_OPERATION attribute*), 405
FastArray (*class in riptable.rt_fastarray*), 411
FastArray (*riptable.rt_enum.TypeRegister attribute*), 410
FastArray._ArrayFunctionHelper (*class in riptable.rt_fastarray*), 414
fastarray_to_pandas_series() (*in module riptable.Utils.pandas_utils*), 185
FasterUFunc (*riptable.rt_fastarray.FastArray attribute*), 419
FAString (*class in riptable.rt_str*), 650

FGColor (*riptable.rt_enum.DisplayColumnColors attribute*), 402
fill_backward() (*in module riptable.rt_fastarraynumba*), 473
fill_backward() (*riptable.rt_categorical.Categorical method*), 238
fill_backward() (*riptable.rt_groupby.GroupBy method*), 478
fill_forward() (*in module riptable.rt_fastarraynumba*), 474
fill_forward() (*riptable.rt_categorical.Categorical method*), 239
fill_forward() (*riptable.rt_groupby.GroupBy method*), 479
fill_invalid() (*riptable.rt_categorical.Categorical method*), 240
fill_invalid() (*riptable.rt_datetime.Date method*), 351
fill_invalid() (*riptable.rt_datetime.DateSpan method*), 359
fill_invalid() (*riptable.rt_datetime.DateTimeNano method*), 373
fill_invalid() (*riptable.rt_datetime.TimeSpan method*), 386
fill_invalid() (*riptable.rt_fastarray.FastArray method*), 436
fillna() (*riptable.rt_dataset.Dataset method*), 297
fillna() (*riptable.rt_fastarray.FastArray method*), 437
fillna() (*riptable.rt_multiset.Multiset method*), 576
filter() (*riptable.rt_dataset.Dataset method*), 300
filter() (*riptable.rt_fastarray.FastArray method*), 439
filtered_name (*riptable.rt_categorical.Categorical property*), 223
filtered_set_name() (*riptable.rt_categorical.Categorical method*), 240
filtered_string (*riptable.rt_categorical.Categorical property*), 223
findnth() (*riptable.rt_groupbyops.GroupByOps method*), 500
findTrueWidth() (*in module riptable.rt_utils*), 712
first() (*riptable.rt_groupbyops.GroupByOps method*), 500
first_bool (*riptable.rt_groupbyops.GroupByOps property*), 488
first_fancy (*riptable.rt_groupbyops.GroupByOps property*), 489
fit_max_columns() (*riptable.rt_display.DisplayTable method*), 399
fix_dst() (*riptable.rt_timezone.TimeZone method*), 709
fix_multiline_footers() (*riptable.rt_display.DisplayTable method*), 399
fix_multiline_headers() (*riptable.rt_display.DisplayTable method*), 399
fix_repeated_keys() (*riptable.rt_display.DisplayTable method*), 399
flatten() (*riptable.rt_multiset.Multiset method*), 576
flatten() (*riptable.rt_struct.Struct method*), 694
flatten_undo() (*riptable.rt_struct.Struct method*), 695
Float (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
Float (*riptable.rt_enum.NumpyCharTypes attribute*), 406
float32 (*class in riptable.rt_numpy*), 584
float64 (*class in riptable.rt_numpy*), 584
Float64 (*riptable.rt_enum.NumpyCharTypes attribute*), 406
FLOOR (*riptable.rt_enum.MATH_OPERATION attribute*), 405
floor() (*in module riptable.rt_numpy*), 597
FLOORDIV (*riptable.rt_enum.MATH_OPERATION attribute*), 405
floordiv (*riptable.rt_fastarray.FastArray attribute*), 419
FMOD (*riptable.rt_enum.MATH_OPERATION attribute*), 405
fmtend (*riptable.Utils.rt_display_nested.DisplayNested property*), 187
fmtstart (*riptable.Utils.rt_display_nested.DisplayNested property*), 187
footer (*riptable.rt_display.DisplayColumn property*), 395
footer_get_dict() (*riptable.rt_dataset.Dataset method*), 301
footer_get_value() (*riptable.rt_itemcontainer.ItemContainer method*), 542
footer_get_values() (*riptable.rt_dataset.Dataset method*), 302
footer_remove() (*riptable.rt_dataset.Dataset method*), 303
footer_set_value() (*riptable.rt_itemcontainer.ItemContainer method*), 542
footer_set_values() (*riptable.rt_dataset.Dataset method*), 304
footers (*riptable.rt_struct.Struct property*), 672
footers_to_string() (*riptable.rt_display.DisplayTable method*), 399
forbidden_mathops (*riptable.rt_datetime.Date attribute*), 347
forbidden_mathops (*riptable.rt_datetime.DateSpan attribute*), 358
FORCE_REPR (*riptable.rt_display.DisplayTable attribute*), 396
ForceRepr (*riptable.rt_display.DisplayDetect attribute*), 396
format_date_num() (*riptable.rt_datetime.Date static*

*method*), 352
format_date_span() (*riptable.rt_datetime.DateSpan static method*), 360
format_long (*riptable.rt_datetime.DateSpan property*), 358
format_nano_time() (*riptable.rt_datetime.DateTimeNano static method*), 374
format_scalar() (*in module riptable.Utils.rt_display_properties*), 190
format_short (*riptable.rt_datetime.DateSpan property*), 358
FrequencyStrings (*riptable.rt_datetime.DateTimeNano attribute*), 368
from_arrow() (*riptable.rt_dataset.Dataset static method*), 305
from_arrow() (*riptable.rt_fastarray.FastArray static method*), 440
from_bin() (*riptable.rt_categorical.Categorical method*), 240
from_category() (*riptable.rt_categorical.Categorical method*), 241
from_grouping() (*riptable.rt_categorical.Categories class method*), 260
from_jagged_dict() (*riptable.rt_dataset.Dataset class method*), 305
from_jagged_rows() (*riptable.rt_dataset.Dataset class method*), 306
from_meta_data() (*riptable.rt_enum.TypeRegister class method*), 410
from_pandas() (*riptable.rt_dataset.Dataset class method*), 307
from_rows() (*riptable.rt_dataset.Dataset class method*), 307
from_tagged_rows() (*riptable.rt_dataset.Dataset class method*), 308
full() (*in module riptable.rt_numpy*), 597
full() (*riptable.rt_categorical.Categorical static method*), 242
full_like() (*in module riptable.rt_numpy*), 598

## G

gb() (*riptable.rt_dataset.Dataset method*), 308
GB_CUMMAX (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_CUMMIN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_CUMNANMAX (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_CUMNANMIN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_CUMPROD (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_CUMSUM (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_EMADECAY (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_EMANORMAL (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_EMAWEIGHTED (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_FINDNTH (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_FIRST (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_FUNCTIONS (*class in riptable.rt_enum*), 403
gb_keychain (*riptable.rt_accum2.Accum2 property*), 198
gb_keychain (*riptable.rt_categorical.Categorical property*), 223
gb_keychain (*riptable.rt_groupby.GroupBy property*), 476
gb_keychain (*riptable.rt_groupbyops.GroupByOps property*), 489
GB_LAST (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_MAX (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_MEAN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_MEDIAN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_MIN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_MODE (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NANMAX (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NANMEAN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NANMIN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NANSTD (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NANSUM (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NANVAR (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_NTH (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_PREFIX (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 182
GB_QUANTILE_MULT (*riptable.rt_enum.GB_FUNCTIONS attribute*), 403
GB_ROLLING_COUNT (*riptable.rt_enum.GB_FUNCTIONS attribute*),

404
GB_ROLLING_DIFF (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_ROLLING_MEAN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_ROLLING_NANMEAN (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_ROLLING_NANSUM (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_ROLLING_QUANTILE (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_ROLLING_SHIFT (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_ROLLING_SUM (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_STD (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_SUM (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_TRIMBR (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
GB_VAR (*riptable.rt_enum.GB_FUNCTIONS attribute*), 404
gbkeys (*riptable.rt_accum2.Accum2 property*), 198
gbkeys (*riptable.rt_groupby.GroupBy property*), 476
gbkeys (*riptable.rt_groupbykeys.GroupByKeys property*), 483
gbkeys (*riptable.rt_grouping.Grouping property*), 522
gbkeys (*riptable.rt_pgroupby.PGroupBy attribute*), 639
gbkeys_filtered (*riptable.rt_groupbykeys.GroupByKeys property*), 483
gbrows() (*riptable.rt_dataset.Dataset method*), 308
gbu() (*riptable.rt_dataset.Dataset method*), 309
ge() (*riptable.rt_fastarray.FastArray method*), 440
gen() (*riptable.rt_accumtable.AccumTable method*), 203
get() (*riptable.Utils.rt_metadata.MetaData method*), 191
get_array_formatter() (*in module riptable.Utils.rt_display_properties*), 190
get_array_function() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 415
get_array_function_type_compatibility_check() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 415
get_attribute() (*riptable.rt_struct.Struct method*), 695
get_bad_color() (*riptable.rt_display.DisplayTable method*), 399
get_bin() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
get_bin_from_index() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
get_build_conf_name() (*in module riptable.Utils.teamcity_helper*), 192
get_categories() (*riptable.rt_categorical.Categories method*), 260
get_category_index() (*riptable.rt_categorical.Categories method*), 260
get_category_match_index() (*riptable.rt_categorical.Categories method*), 260
get_children() (*riptable.Utils.rt_display_nested.AttributeTraversal method*), 186
get_children() (*riptable.Utils.rt_display_nested.DictTraversal method*), 187
get_classname() (*riptable.rt_datetime.Date method*), 352
get_classname() (*riptable.rt_datetime.DateScalar method*), 357
get_classname() (*riptable.rt_datetime.DateSpan method*), 360
get_classname() (*riptable.rt_datetime.DateSpanScalar method*), 362
get_classname() (*riptable.rt_datetime.DateTimeBase method*), 364
get_classname() (*riptable.rt_datetime.DateTimeNano method*), 375
get_classname() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
get_classname() (*riptable.rt_datetime.TimeSpan method*), 387
get_classname() (*riptable.rt_datetime.TimeSpanScalar method*), 389
get_common_dtype() (*in module riptable.rt_numpy*), 599
get_default_value() (*in module riptable.rt_utils*), 712
get_dict_values() (*riptable.rt_itemcontainer.ItemContainer method*), 542
get_display_array_type() (*riptable.Utils.rt_display_properties.DisplayConvert static method*), 190
get_display_convert() (*riptable.Utils.rt_display_properties.DisplayConvert*

*static method*), 190
get_display_mode() (*riptable.rt_display.DisplayDetect static method*), 396
get_doctest_dataset_data() (*in module riptable.conftest*), 196
get_dtype() (*in module riptable.rt_numpy*), 599
get_global_settings() (*in module riptable.config*), 195
get_group() (*riptable.rt_groupby.GroupBy method*), 481
get_groupings() (*riptable.rt_groupbyops.GroupByOps method*), 501
get_header_names() (*riptable.rt_groupbyops.GroupByOps class method*), 501
get_index_from_bin() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
get_invalid() (*in module riptable.numba.invalid_values*), 194
get_item_format() (*riptable.rt_datetime.DateScalar method*), 357
get_item_format() (*riptable.rt_datetime.DateSpanScalar method*), 362
get_item_format() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
get_item_format() (*riptable.rt_datetime.TimeSpanScalar method*), 389
get_max_valid() (*in module riptable.numba.invalid_values*), 194
get_min_valid() (*in module riptable.numba.invalid_values*), 194
get_multikey_index() (*riptable.rt_categorical.Categories method*), 260
get_name() (*riptable.rt_fastarray.FastArray method*), 440
get_name() (*riptable.rt_grouping.Grouping method*), 531
get_ncols() (*riptable.rt_struct.Struct method*), 695
get_nrows() (*riptable.rt_dataset.Dataset method*), 309
get_nrows() (*riptable.rt_struct.Struct method*), 695
get_restricted_names() (*riptable.rt_struct.Struct method*), 695
get_root() (*riptable.Utils.rt_display_nested.DictTraversal method*), 187
get_row_sort_info() (*riptable.rt_dataset.Dataset method*), 310
get_scalar() (*riptable.rt_datetime.Date method*), 352
get_scalar() (*riptable.rt_datetime.DateSpan method*), 360
get_scalar() (*riptable.rt_datetime.DateTimeNano method*), 375
get_scalar() (*riptable.rt_datetime.TimeSpan method*), 387
get_sort_col_idx() (*riptable.rt_display.DisplayTable method*), 399
get_sorted_col_data() (*riptable.rt_dataset.Dataset method*), 310
get_sorted_row_index() (*riptable.rt_sort_cache.SortCache class method*), 648
get_terminal_size() (*in module riptable.Utils.terminalsize*), 192
get_text() (*riptable.Utils.rt_display_nested.DictTraversal method*), 187
GetNanoTime() (*in module riptable.rt_timers*), 704
GetTSC() (*in module riptable.rt_timers*), 704
gfx (*riptable.Utils.rt_display_nested.BoxStyle attribute*), 187
GrayItalic (*riptable.rt_enum.DisplayColumnColors attribute*), 402
GroupBy (*class in riptable.rt_groupby*), 475
Groupby (*riptable.rt_enum.DisplayColumnColors attribute*), 402
GroupBy (*riptable.rt_enum.TypeRegister attribute*), 410
groupby() (*in module riptable.rt_numpy*), 600
groupby() (*riptable.rt_dataset.Dataset method*), 310
groupby_data (*riptable.rt_categorical.Categorical property*), 223
groupby_data_clear() (*riptable.rt_categorical.Categorical method*), 242
groupby_data_set() (*riptable.rt_categorical.Categorical method*), 242
groupby_reset() (*riptable.rt_categorical.Categorical method*), 243
groupbyhash() (*in module riptable.rt_numpy*), 601
GroupByKeys (*class in riptable.rt_groupbykeys*), 482
groupbylex() (*in module riptable.rt_numpy*), 602
GroupbyNumba (*class in riptable.rt_groupbynumba*), 485
GroupByOps (*class in riptable.rt_groupbyops*), 488
groupbypack() (*in module riptable.rt_numpy*), 603
Grouping (*class in riptable.rt_grouping*), 519
grouping (*riptable.rt_categorical.Categorical property*), 224
grouping (*riptable.rt_categorical.Categories property*), 258
Grouping (*riptable.rt_enum.TypeRegister attribute*), 410
grouping (*riptable.rt_groupbyops.GroupByOps attribute*), 490
grouping_dict (*riptable.rt_categorical.Categorical*

property), 224
GroupingDebugMode (riptable.rt_categorical.Categorical attribute), 226
GroupingInit (riptable.rt_grouping.Grouping attribute), 525
groups (riptable.rt_groupbyops.GroupByOps property), 489
groupScatter() (in module riptable.rt_stats), 649
grpFillBackward() (riptable.rt_groupbynumba.GroupbyNumba method), 486
grpFillForward() (riptable.rt_groupbynumba.GroupbyNumba method), 487
grpFillForwardBackward() (riptable.rt_groupbynumba.GroupbyNumba method), 487
grpTrim() (riptable.rt_groupbynumba.GroupbyNumba method), 487
gt() (riptable.rt_fastarray.FastArray method), 441

## H

HANDLED_FUNCTIONS (riptable.rt_fastarray.FastArray._ArrayFunctionHelper attribute), 415
HANDLED_TYPE_COMPATIBILITY_CHECK (riptable.rt_fastarray.FastArray._ArrayFunctionHelper attribute), 415
has_nested_containers (riptable.rt_struct.Struct property), 672
head() (riptable.rt_dataset.Dataset method), 312
head() (riptable.rt_groupbyops.GroupByOps method), 501
HEAD_ROWS (riptable.Utils.display_options.DisplayOptions attribute), 178, 182
header (riptable.rt_display.DisplayColumn property), 395
HEADER_DARK (riptable.rt_display.DisplayText attribute), 399
header_format() (riptable.rt_display.DisplayText static method), 400
HEADER_LIGHT (riptable.rt_display.DisplayText attribute), 400
horiz_len (riptable.Utils.rt_display_nested.BoxStyle attribute), 187
hstack() (in module riptable.rt_numpy), 604
hstack() (riptable.rt_categorical.Categorical class method), 243
hstack() (riptable.rt_dataset.Dataset class method), 312
hstack() (riptable.rt_datetime.Date class method), 352
hstack() (riptable.rt_datetime.DateSpan class method), 360
hstack() (riptable.rt_datetime.DateTimeNano class method), 375
hstack() (riptable.rt_datetime.TimeSpan class method), 387
hstack() (riptable.rt_pdataset.PDataset class method), 636
hstack() (riptable.rt_struct.Struct class method), 696
hstack_any() (in module riptable.rt_hstack), 538
hstack_groupings() (in module riptable.rt_grouping), 536
hstack_test() (in module riptable.rt_grouping), 537
HTML (riptable.rt_enum.DisplayDetectModes attribute), 402
HTML (riptable.rt_enum.DS_DISPLAY_TYPES attribute), 401
HTML_DISPLAY (riptable.Utils.display_options.DisplayOptions attribute), 178, 182

## I

ifirstgroup (riptable.rt_grouping.Grouping property), 522
ifirstkey (riptable.rt_categorical.Categorical property), 224
ifirstkey (riptable.rt_groupby.GroupBy property), 476
ifirstkey (riptable.rt_grouping.Grouping property), 522
igroup (riptable.rt_grouping.Grouping property), 522
igroupby() (riptable.rt_pdataset.PDataset method), 636
igroupreverse (riptable.rt_grouping.Grouping property), 523
ikey (riptable.rt_accum2.Accum2 property), 198
ikey (riptable.rt_categorical.Categorical property), 224
ikey (riptable.rt_grouping.Grouping property), 523
ilastkey (riptable.rt_categorical.Categorical property), 224
ilastkey (riptable.rt_groupby.GroupBy property), 476
ilastkey (riptable.rt_grouping.Grouping property), 523
IMatrix (class in riptable.rt_imatrix), 539
imatrix (riptable.rt_dataset.Dataset property), 270
imatrix (riptable.rt_imatrix.IMatrix property), 539
imatrix_cls (riptable.rt_dataset.Dataset property), 271
imatrix_ds (riptable.rt_dataset.Dataset property), 271
imatrix_make() (riptable.rt_dataset.Dataset method), 312
imatrix_totals() (riptable.rt_dataset.Dataset method), 313
imatrix_xy() (riptable.rt_dataset.Dataset method), 313
imatrix_y() (riptable.rt_dataset.Dataset method), 313
indent (riptable.Utils.rt_display_nested.BoxStyle attribute), 187
index() (riptable.rt_str.FAString method), 658
index_any_of() (riptable.rt_str.FAString method), 658

inextkey (*riptable.rt_grouping.Grouping property*), 523
Info (*class in riptable.rt_meta*), 566
info() (*in module riptable.rt_meta*), 569
info() (*riptable.rt_categorical.Categorical method*), 243
info() (*riptable.rt_datetime.DateTimeNano method*), 375
info() (*riptable.rt_fastarray.FastArray method*), 441
info() (*riptable.rt_struct.Struct method*), 696
inline_svg (*riptable.Utils.rt_display_nested.DisplayNested attribute*), 187
int0 (*class in riptable.rt_numpy*), 584
int16 (*class in riptable.rt_numpy*), 585
int2strdict (*riptable.rt_categorical.Categories property*), 258
int32 (*class in riptable.rt_numpy*), 585
int64 (*class in riptable.rt_numpy*), 585
int8 (*class in riptable.rt_numpy*), 586
Integer (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
Integer (*riptable.rt_enum.NumpyCharTypes attribute*), 406
integer_range() (*in module riptable.Utils.common*), 174
integer_valid_range() (*in module riptable.Utils.common*), 174
interp() (*in module riptable.rt_numpy*), 604
interp_extrap() (*in module riptable.rt_numpy*), 605
inv (*riptable.rt_fastarray.FastArray property*), 418
inv (*riptable.rt_numpy.bool_ attribute*), 583
inv (*riptable.rt_numpy.bytes_ attribute*), 584
inv (*riptable.rt_numpy.float32 attribute*), 584
inv (*riptable.rt_numpy.float64 attribute*), 584
inv (*riptable.rt_numpy.int16 attribute*), 585
inv (*riptable.rt_numpy.int32 attribute*), 585
inv (*riptable.rt_numpy.int64 attribute*), 586
inv (*riptable.rt_numpy.int8 attribute*), 586
inv (*riptable.rt_numpy.str_ attribute*), 586
inv (*riptable.rt_numpy.uint16 attribute*), 587
inv (*riptable.rt_numpy.uint32 attribute*), 587
inv (*riptable.rt_numpy.uint64 attribute*), 588
inv (*riptable.rt_numpy.uint8 attribute*), 588
invalid_category (*riptable.rt_categorical.Categorical property*), 224
INVALID_DATA (*riptable.rt_display.DisplayTable attribute*), 397
INVALID_DICT (*in module riptable.rt_enum*), 411
invalid_set() (*riptable.rt_categorical.Categorical method*), 243
invalidate() (*riptable.rt_sort_cache.SortCache class method*), 648
invalidate_all() (*riptable.rt_sort_cache.SortCache class method*), 648
INVERT (*riptable.rt_enum.MATH_OPERATION attribute*), 405
iprevkey (*riptable.rt_grouping.Grouping property*), 523
Ipython (*riptable.rt_enum.DisplayDetectModes attribute*), 402
is_array_subclass() (*riptable.rt_enum.TypeRegister class method*), 410
is_binned_array() (*riptable.rt_enum.TypeRegister class method*), 410
is_binned_type() (*riptable.rt_enum.TypeRegister class method*), 410
is_computable() (*riptable.rt_enum.TypeRegister class method*), 411
is_datelike() (*riptable.rt_enum.TypeRegister class method*), 411
is_leapyear (*riptable.rt_datetime.Date property*), 343
is_locked() (*riptable.rt_struct.Struct method*), 697
is_running_in_teamcity() (*in module riptable.Utils.teamcity_helper*), 192
is_spanlike() (*riptable.rt_enum.TypeRegister class method*), 411
is_string_or_object() (*riptable.rt_enum.TypeRegister class method*), 411
is_valid() (*in module riptable.numba.invalid_values*), 195
is_valid_colname() (*riptable.rt_struct.Struct method*), 697
is_weekday (*riptable.rt_datetime.Date property*), 344
is_weekend (*riptable.rt_datetime.Date property*), 345
isbytes (*riptable.rt_categorical.Categories property*), 258
iscategorical (*riptable.rt_grouping.Grouping property*), 523
ischararray() (*in module riptable.rt_utils*), 712
iscomputable() (*riptable.rt_fastarray.FastArray method*), 442
isdirty (*riptable.rt_grouping.Grouping property*), 523
isdisplaysorted (*riptable.rt_grouping.Grouping property*), 524
isenum (*riptable.rt_categorical.Categorical property*), 226
isenum (*riptable.rt_categorical.Categories property*), 258
isenum (*riptable.rt_grouping.Grouping property*), 524
isfiltered() (*riptable.rt_categorical.Categorical method*), 244
ISFINITE (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isfinite() (*in module riptable.rt_numpy*), 605
isfinite() (*riptable.rt_datetime.Date method*), 352
isfinite() (*riptable.rt_datetime.DateSpanScalar method*), 362
isfinite() (*riptable.rt_datetime.DateTimeNano*

*method*), 375
isfinite() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
isfinite() (*riptable.rt_datetime.TimeSpanScalar method*), 389
isfinite() (*riptable.rt_fastarray.FastArray method*), 442
isin() (*riptable.rt_categorical.Categorical method*), 244
isin() (*riptable.rt_dataset.Dataset method*), 313
isin() (*riptable.rt_fastarray.FastArray method*), 443
isin() (*riptable.rt_grouping.Grouping method*), 531
ISINF (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isinf() (*in module riptable.rt_numpy*), 606
isinf() (*riptable.rt_fastarray.FastArray method*), 443
islogical() (*in module riptable.rt_utils*), 712
ismember() (*in module riptable.rt_numpy*), 607
ismember() (*riptable.rt_grouping.Grouping method*), 531
ismultikey (*riptable.rt_categorical.Categorical property*), 226
ismultikey (*riptable.rt_categorical.Categories property*), 258
ismultikey (*riptable.rt_grouping.Grouping property*), 524
isna() (*riptable.rt_categorical.Categorical method*), 245
isna() (*riptable.rt_fastarray.FastArray method*), 444
ISNAN (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnan() (*in module riptable.rt_numpy*), 608
isnan() (*riptable.rt_categorical.Categorical method*), 245
isnan() (*riptable.rt_datetime.Date method*), 353
isnan() (*riptable.rt_datetime.DateSpanScalar method*), 362
isnan() (*riptable.rt_datetime.DateTimeNano method*), 376
isnan() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
isnan() (*riptable.rt_datetime.TimeSpanScalar method*), 389
isnan() (*riptable.rt_fastarray.FastArray method*), 444
ISNANORZERO (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnanorzero() (*in module riptable.rt_numpy*), 609
isnanorzero() (*riptable.rt_fastarray.FastArray method*), 445
ISNORMAL (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnormal() (*riptable.rt_fastarray.FastArray method*), 446
ISNOTFINITE (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnotfinite() (*in module riptable.rt_numpy*), 609
isnotfinite() (*riptable.rt_datetime.Date method*), 353
isnotfinite() (*riptable.rt_datetime.DateSpanScalar method*), 362
isnotfinite() (*riptable.rt_datetime.DateTimeNano method*), 377
isnotfinite() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
isnotfinite() (*riptable.rt_datetime.TimeSpanScalar method*), 389
isnotfinite() (*riptable.rt_fastarray.FastArray method*), 446
ISNOTINF (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnotinf() (*in module riptable.rt_numpy*), 610
isnotinf() (*riptable.rt_fastarray.FastArray method*), 446
ISNOTNAN (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnotnan() (*in module riptable.rt_numpy*), 611
isnotnan() (*riptable.rt_categorical.Categorical method*), 246
isnotnan() (*riptable.rt_datetime.Date method*), 354
isnotnan() (*riptable.rt_datetime.DateSpanScalar method*), 362
isnotnan() (*riptable.rt_datetime.DateTimeNano method*), 377
isnotnan() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
isnotnan() (*riptable.rt_datetime.TimeSpanScalar method*), 389
isnotnan() (*riptable.rt_fastarray.FastArray method*), 447
ISNOTNORMAL (*riptable.rt_enum.MATH_OPERATION attribute*), 405
isnotnormal() (*riptable.rt_fastarray.FastArray method*), 448
isordered (*riptable.rt_grouping.Grouping property*), 524
isortrows (*riptable.rt_accum2.Accum2 property*), 198
isortrows (*riptable.rt_groupby.GroupBy property*), 476
isortrows (*riptable.rt_groupbykeys.GroupByKeys property*), 483
isortrows (*riptable.rt_grouping.Grouping property*), 524
isortrows (*riptable.rt_pgroupby.PGroupBy attribute*), 639
issinglekey (*riptable.rt_categorical.Categorical property*), 226
issinglekey (*riptable.rt_categorical.Categories property*), 258
issinglekey (*riptable.rt_grouping.Grouping property*), 524

issorted() (*in module riptable.rt_numpy*), 611
issorted() (*riptable.rt_fastarray.FastArray method*), 448
isunicode (*riptable.rt_categorical.Categories property*), 259
Item (*class in riptable.rt_meta*), 567
item_add_prefix() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_add_suffix() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_delete() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_exists() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_get_attribute() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_get_dict() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_get_len() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_get_value() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_get_values() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_move_to_back() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_move_to_front() (*riptable.rt_itemcontainer.ItemContainer method*), 542
item_rename() (*riptable.rt_itemcontainer.ItemContainer method*), 543
item_replace_all() (*riptable.rt_itemcontainer.ItemContainer method*), 543
item_set_attribute() (*riptable.rt_itemcontainer.ItemContainer method*), 543
item_set_value() (*riptable.rt_itemcontainer.ItemContainer method*), 543
item_set_value_internal() (*riptable.rt_itemcontainer.ItemContainer method*), 543
item_str_match() (*riptable.rt_itemcontainer.ItemContainer method*), 543
item_str_replace() (*riptable.rt_itemcontainer.ItemContainer method*), 543
itemclass (*riptable.Utils.rt_metadata.MetaData property*), 191
ItemContainer (*class in riptable.rt_itemcontainer*), 540
ItemFormat (*class in riptable.Utils.rt_display_properties*), 190
items (*riptable.rt_meta.Info attribute*), 567
items() (*riptable.rt_itemcontainer.ItemContainer method*), 544
items() (*riptable.rt_struct.Struct method*), 697
items_as_dict() (*riptable.rt_itemcontainer.ItemContainer method*), 544
items_tolist() (*riptable.rt_itemcontainer.ItemContainer method*), 544
iter_groups() (*riptable.rt_groupbyops.GroupByOps method*), 501
iter_values() (*riptable.rt_itemcontainer.ItemContainer method*), 544
iterrows() (*riptable.rt_dataset.Dataset method*), 314

## J

jedi_completions() (*in module riptable.rt_misc*), 570
JoinIndices (*class in riptable.rt_merge*), 546
Jupyter (*riptable.rt_enum.DisplayDetectModes attribute*), 402

## K

keep() (*riptable.rt_dataset.Dataset method*), 315
keep() (*riptable.rt_multiset.Multiset method*), 576
key_from_bin() (*riptable.rt_groupbyops.GroupByOps method*), 502
key_search() (*riptable.rt_struct.Struct method*), 698
KeyArgsConstructor (*class in riptable.Utils.rt_display_nested*), 187
keys() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
keys() (*riptable.rt_itemcontainer.ItemContainer method*), 544
keys() (*riptable.rt_struct.Struct method*), 698

## L

label_as_dict() (*riptable.rt_itemcontainer.ItemContainer method*), 544
label_as_dict() (*riptable.rt_struct.Struct method*), 698
label_filter() (*riptable.rt_struct.Struct method*), 698
label_fixup() (*riptable.rt_multiset.Multiset method*), 577

label_format (*riptable.Utils.rt_display_nested.Style attribute*), 188
label_get_names() (*riptable.rt_itemcontainer.ItemContainer method*), 544
label_get_names() (*riptable.rt_struct.Struct method*), 699
label_remove() (*riptable.rt_itemcontainer.ItemContainer method*), 544
label_remove() (*riptable.rt_struct.Struct method*), 699
label_set_names() (*riptable.rt_itemcontainer.ItemContainer method*), 544
label_set_names() (*riptable.rt_multiset.Multiset method*), 577
label_set_names() (*riptable.rt_struct.Struct method*), 699
label_space (*riptable.Utils.rt_display_nested.BoxStyle attribute*), 187
labels() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
last() (*riptable.rt_groupbyops.GroupByOps method*), 502
last_bool (*riptable.rt_groupbyops.GroupByOps property*), 489
last_child_head() (*riptable.Utils.rt_display_nested.BoxStyle method*), 187
last_child_head() (*riptable.Utils.rt_display_nested.Style method*), 188
last_child_tail() (*riptable.Utils.rt_display_nested.BoxStyle method*), 187
last_child_tail() (*riptable.Utils.rt_display_nested.Style method*), 188
last_fancy (*riptable.rt_groupbyops.GroupByOps property*), 489
le() (*riptable.rt_fastarray.FastArray method*), 448
Ledger (*class in riptable.rt_fastarray*), 471
Left (*riptable.rt_enum.DisplayJustification attribute*), 402
left_index (*riptable.rt_merge.JoinIndices attribute*), 546
LeftAligned (*class in riptable.Utils.rt_display_nested*), 188
lexsort() (*in module riptable.rt_numpy*), 612
lightbg_styles (*riptable.rt_display.DisplayCell attribute*), 394
linear_spline() (*in module riptable.rt_stats*), 649
list_modes (*riptable.rt_categorical.Categories attribute*), 259
lm() (*in module riptable.rt_stats*), 649
load() (*riptable.rt_dataset.Dataset class method*), 315
load() (*riptable.rt_struct.Struct class method*), 699
load_config() (*riptable.Utils.display_options.DisplayOptions method*), 180
load_config() (*riptable.Utils.display_options.DisplayOptions static method*), 183
load_csv_as_dataset() (*in module riptable.rt_csv*), 266
load_h5() (*in module riptable.rt_utils*), 712
load_sds() (*in module riptable.rt_sds*), 642
load_sds_mem() (*in module riptable.rt_sds*), 644
lock() (*riptable.rt_categorical.Categorical method*), 247
LOG (*riptable.rt_enum.MATH_OPERATION attribute*), 405
log() (*in module riptable.rt_numpy*), 612
LOG10 (*riptable.rt_enum.MATH_OPERATION attribute*), 405
log10() (*in module riptable.rt_numpy*), 612
LOG1P (*riptable.rt_enum.MATH_OPERATION attribute*), 405
LOG2 (*riptable.rt_enum.MATH_OPERATION attribute*), 405
logging_off() (*riptable.rt_sort_cache.SortCache class method*), 648
logging_on() (*riptable.rt_sort_cache.SortCache class method*), 648
logical() (*in module riptable.rt_numpy*), 612
LOGICAL_AND (*riptable.rt_enum.MATH_OPERATION attribute*), 405
LOGICAL_NOT (*riptable.rt_enum.MATH_OPERATION attribute*), 405
LOGICAL_OR (*riptable.rt_enum.MATH_OPERATION attribute*), 405
LOGICAL_XOR (*riptable.rt_enum.MATH_OPERATION attribute*), 405
Long (*riptable.rt_enum.DisplayLength attribute*), 402
lower (*riptable.rt_str.FAString property*), 650
lt() (*riptable.rt_fastarray.FastArray method*), 448

## M

mae() (*in module riptable.rt_stats*), 649
make_categoricals() (*riptable.rt_struct.Struct method*), 700
make_dataset() (*riptable.rt_accum2.Accum2 method*), 201
make_matlab_categoricals() (*riptable.rt_struct.Struct method*), 700
make_matlab_datetimes() (*riptable.rt_struct.Struct method*), 700
make_struct_from_categories() (*riptable.rt_struct.Struct method*), 701

make_table() (*riptable.rt_multiset.Multiset method*), 577
make_table() (*riptable.rt_struct.Struct method*), 701
makeifirst() (*in module riptable.rt_numpy*), 612
makeilast() (*in module riptable.rt_numpy*), 613
makeinext() (*in module riptable.rt_numpy*), 613
makeiprev() (*in module riptable.rt_numpy*), 613
map() (*riptable.rt_categorical.Categorical method*), 247
map() (*riptable.rt_fastarray.FastArray method*), 449
map_old() (*riptable.rt_fastarray.FastArray method*), 449
mapping_add() (*riptable.rt_categorical.Categorical method*), 248
mapping_new() (*riptable.rt_categorical.Categorical method*), 248
mapping_remove() (*riptable.rt_categorical.Categorical method*), 248
mapping_replace() (*riptable.rt_categorical.Categorical method*), 249
mask_and() (*in module riptable.rt_numpy*), 613
mask_and_isfinite() (*riptable.rt_dataset.Dataset method*), 316
mask_and_isinf() (*riptable.rt_dataset.Dataset method*), 317
mask_and_isnan() (*riptable.rt_dataset.Dataset method*), 318
mask_andi() (*in module riptable.rt_numpy*), 613
mask_andnot() (*in module riptable.rt_numpy*), 614
mask_andnoti() (*in module riptable.rt_numpy*), 614
mask_or() (*in module riptable.rt_numpy*), 614
mask_or_isfinite() (*riptable.rt_dataset.Dataset method*), 318
mask_or_isinf() (*riptable.rt_dataset.Dataset method*), 319
mask_or_isnan() (*riptable.rt_dataset.Dataset method*), 320
mask_ori() (*in module riptable.rt_numpy*), 614
mask_xor() (*in module riptable.rt_numpy*), 614
mask_xori() (*in module riptable.rt_numpy*), 614
match_str_to_category() (*riptable.rt_categorical.Categories method*), 261
MATH_OPERATION (*class in riptable.rt_enum*), 404
MathLedger (*class in riptable.rt_ledger*), 544
MathLedger (*riptable.rt_enum.TypeRegister attribute*), 410
MAX (*riptable.rt_enum.MATH_OPERATION attribute*), 405
max() (*in module riptable.rt_numpy*), 614
max() (*riptable.rt_dataset.Dataset method*), 320
max() (*riptable.rt_datetime.DateTimeNano method*), 378
max() (*riptable.rt_groupbyops.GroupByOps method*), 502
max() (*riptable.rt_multiset.Multiset method*), 577
MAX_DISPLAY_LEN (*riptable.rt_fastarray.FastArray attribute*), 419
MAX_FOOTER_WIDTH (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 182
MAX_HEADER_WIDTH (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 182
MAX_ROWS (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 182
MAX_STRING_WIDTH (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 182
maximum() (*in module riptable.rt_numpy*), 614
mbget() (*in module riptable.rt_utils*), 713
mean() (*in module riptable.rt_numpy*), 614
mean() (*riptable.rt_dataset.Dataset method*), 320
mean() (*riptable.rt_fastarray.FastArray method*), 449
mean() (*riptable.rt_groupbyops.GroupByOps method*), 502
mean() (*riptable.rt_multiset.Multiset method*), 577
median() (*in module riptable.rt_numpy*), 615
median() (*riptable.rt_dataset.Dataset method*), 320
median() (*riptable.rt_fastarray.FastArray method*), 450
median() (*riptable.rt_groupbyops.GroupByOps method*), 503
Medium (*riptable.rt_enum.DisplayLength attribute*), 403
melt() (*riptable.rt_dataset.Dataset method*), 320
memory_stats (*riptable.rt_dataset.Dataset property*), 272
merge() (*in module riptable.rt_merge*), 547
merge() (*riptable.rt_dataset.Dataset method*), 321
merge2() (*in module riptable.rt_merge*), 548
merge2() (*riptable.rt_dataset.Dataset method*), 321
merge_asof() (*in module riptable.rt_merge*), 550
merge_asof() (*riptable.rt_dataset.Dataset method*), 321
merge_asof2() (*in module riptable.rt_merge_asof*), 561
merge_cats() (*in module riptable.rt_grouping*), 537
merge_index() (*in module riptable.rt_algos*), 206
merge_indices() (*in module riptable.rt_merge*), 555
merge_lookup() (*in module riptable.rt_merge*), 557
merge_lookup() (*riptable.rt_dataset.Dataset method*), 321
merge_prebinned() (*in module riptable.rt_utils*), 714
meta_from_version() (*in module riptable.Utils.rt_metadata*), 191
META_VERSION (*in module riptable.Utils.rt_metadata*), 191
MetaData (*class in riptable.Utils.rt_metadata*), 191
MetaDefault (*riptable.rt_categorical.Categorical*

*attribute*), 226
MetaDefault (*riptable.rt_datetime.Date attribute*), 347
MetaDefault (*riptable.rt_datetime.DateSpan attribute*), 358
MetaDefault (*riptable.rt_datetime.DateTimeNano attribute*), 368
MetaVersion (*riptable.rt_categorical.Categorical attribute*), 226
MetaVersion (*riptable.rt_datetime.Date attribute*), 347
MetaVersion (*riptable.rt_datetime.DateSpan attribute*), 358
MetaVersion (*riptable.rt_datetime.DateTimeNano attribute*), 368
MIN (*riptable.rt_enum.MATH_OPERATION attribute*), 405
min() (*in module riptable.rt_numpy*), 615
min() (*riptable.rt_dataset.Dataset method*), 326
min() (*riptable.rt_datetime.DateTimeNano method*), 379
min() (*riptable.rt_groupbyops.GroupByOps method*), 503
min() (*riptable.rt_multiset.Multiset method*), 577
minimum() (*in module riptable.rt_numpy*), 615
MOD (*riptable.rt_enum.MATH_OPERATION attribute*), 406
mod (*riptable.rt_fastarray.FastArray attribute*), 419
mode (*riptable.rt_categorical.Categories property*), 259
Mode (*riptable.rt_display.DisplayDetect attribute*), 396
mode() (*riptable.rt_groupbyops.GroupByOps method*), 504
module
    riptable, 167
    riptable.config, 195
    riptable.conftest, 196
    riptable.numba, 192
    riptable.numba.indexing, 192
    riptable.numba.invalid_values, 193
    riptable.rt_accum2, 196
    riptable.rt_accumtable, 202
    riptable.rt_algos, 206
    riptable.rt_bin, 206
    riptable.rt_categorical, 211
    riptable.rt_compressedarray, 262
    riptable.rt_csv, 266
    riptable.rt_dataset, 266
    riptable.rt_datetime, 340
    riptable.rt_display, 394
    riptable.rt_ema, 400
    riptable.rt_enum, 400
    riptable.rt_fastarray, 411
    riptable.rt_fastarraynumba, 473
    riptable.rt_groupby, 475
    riptable.rt_groupbykeys, 482
    riptable.rt_groupbynumba, 485
    riptable.rt_groupbyops, 488
    riptable.rt_grouping, 518
    riptable.rt_hstack, 537
    riptable.rt_imatrix, 539
    riptable.rt_io, 539
    riptable.rt_itemcontainer, 540
    riptable.rt_ledger, 544
    riptable.rt_matplotlib, 546
    riptable.rt_merge, 546
    riptable.rt_merge_asof, 561
    riptable.rt_meta, 566
    riptable.rt_misc, 569
    riptable.rt_mlutils, 572
    riptable.rt_multiset, 572
    riptable.rt_numpy, 578
    riptable.rt_pdataset, 633
    riptable.rt_pgroupby, 639
    riptable.rt_sds, 640
    riptable.rt_sharedmemory, 648
    riptable.rt_sort_cache, 648
    riptable.rt_stats, 649
    riptable.rt_str, 650
    riptable.rt_struct, 661
    riptable.rt_timers, 704
    riptable.rt_timezone, 707
    riptable.rt_utils, 710
    riptable.Utils, 167
    riptable.Utils.appdirs, 167
    riptable.Utils.common, 173
    riptable.Utils.conversion_utils, 176
    riptable.Utils.display_options, 177
    riptable.Utils.ipython_utils, 184
    riptable.Utils.pandas_utils, 185
    riptable.Utils.rt_display_nested, 186
    riptable.Utils.rt_display_properties, 189
    riptable.Utils.rt_metadata, 190
    riptable.Utils.teamcity_helper, 192
    riptable.Utils.terminalsize, 192
month (*riptable.rt_datetime.Date property*), 345
monthyear (*riptable.rt_datetime.Date property*), 346
move_argmax() (*riptable.rt_fastarray.FastArray method*), 450
move_argmin() (*riptable.rt_fastarray.FastArray method*), 451
move_max() (*riptable.rt_fastarray.FastArray method*), 451
move_mean() (*riptable.rt_fastarray.FastArray method*), 451
move_median() (*riptable.rt_fastarray.FastArray method*), 451
move_min() (*riptable.rt_fastarray.FastArray method*), 451
move_rank() (*riptable.rt_fastarray.FastArray method*), 451

move_std() (*riptable.rt_fastarray.FastArray method*), 451
move_sum() (*riptable.rt_fastarray.FastArray method*), 451
move_var() (*riptable.rt_fastarray.FastArray method*), 451
MUL (*riptable.rt_enum.MATH_OPERATION attribute*), 406
mul (*riptable.rt_fastarray.FastArray attribute*), 420
multiget() (*riptable.rt_multiset.Multiset method*), 577
multikey (*riptable.rt_groupbykeys.GroupByKeys property*), 483
multikey_labels (*riptable.rt_groupbykeys.GroupByKeys property*), 483
multikey_labels_filtered (*riptable.rt_groupbykeys.GroupByKeys property*), 483
multikey_spacer (*riptable.rt_categorical.Categories attribute*), 259
multikeyhash() (*in module riptable.rt_numpy*), 615
Multiset (*class in riptable.rt_multiset*), 572
Multiset (*riptable.rt_enum.TypeRegister attribute*), 410
Multiset_col_a (*riptable.rt_enum.DisplayColumnColors attribute*), 402
Multiset_col_b (*riptable.rt_enum.DisplayColumnColors attribute*), 402
Multiset_head_a (*riptable.rt_enum.DisplayColumnColors attribute*), 402
Multiset_head_b (*riptable.rt_enum.DisplayColumnColors attribute*), 402

## N

n_elements (*riptable.rt_str.FAString property*), 650
name (*riptable.rt_categorical.Categories property*), 259
name (*riptable.rt_meta.Item attribute*), 567
name (*riptable.Utils.rt_metadata.MetaData property*), 191
NAN_DATE (*riptable.rt_datetime.DateSpan attribute*), 358
NAN_DATESPANSCALAR (*riptable.rt_datetime.DateSpanScalar attribute*), 362
nan_index (*riptable.rt_categorical.Categorical property*), 226
NAN_TIME (*riptable.rt_datetime.DateTimeBase attribute*), 362
nan_to_num() (*in module riptable.rt_numpy*), 616
nan_to_zero() (*in module riptable.rt_numpy*), 616
nanargmax() (*in module riptable.rt_numpy*), 616
nanargmax() (*riptable.rt_dataset.Dataset method*), 326
nanargmax() (*riptable.rt_fastarray.FastArray method*), 451
nanargmin() (*in module riptable.rt_numpy*), 616
nanargmin() (*riptable.rt_dataset.Dataset method*), 326
nanargmin() (*riptable.rt_fastarray.FastArray method*), 451
NANMAX (*riptable.rt_enum.MATH_OPERATION attribute*), 406
nanmax() (*in module riptable.rt_numpy*), 616
nanmax() (*riptable.rt_dataset.Dataset method*), 327
nanmax() (*riptable.rt_fastarray.FastArray method*), 451
nanmax() (*riptable.rt_groupbyops.GroupByOps method*), 505
nanmax() (*riptable.rt_multiset.Multiset method*), 577
nanmean() (*in module riptable.rt_numpy*), 616
nanmean() (*riptable.rt_dataset.Dataset method*), 327
nanmean() (*riptable.rt_fastarray.FastArray method*), 451
nanmean() (*riptable.rt_groupbyops.GroupByOps method*), 505
nanmean() (*riptable.rt_multiset.Multiset method*), 577
nanmedian() (*in module riptable.rt_numpy*), 617
nanmedian() (*riptable.rt_dataset.Dataset method*), 327
nanmedian() (*riptable.rt_fastarray.FastArray method*), 452
nanmedian() (*riptable.rt_groupbyops.GroupByOps method*), 506
NANMIN (*riptable.rt_enum.MATH_OPERATION attribute*), 406
nanmin() (*in module riptable.rt_numpy*), 617
nanmin() (*riptable.rt_dataset.Dataset method*), 327
nanmin() (*riptable.rt_fastarray.FastArray method*), 452
nanmin() (*riptable.rt_groupbyops.GroupByOps method*), 506
nanmin() (*riptable.rt_multiset.Multiset method*), 577
nanpercentile() (*in module riptable.rt_numpy*), 618
nanpercentile() (*riptable.rt_fastarray.FastArray method*), 452
nanpercentile() (*riptable.rt_groupbyops.GroupByOps method*), 507
nanquantile() (*riptable.rt_fastarray.FastArray method*), 452
nanquantile() (*riptable.rt_groupbyops.GroupByOps method*), 508
nanrankdata() (*riptable.rt_fastarray.FastArray method*), 452
nanstd() (*in module riptable.rt_numpy*), 618
nanstd() (*riptable.rt_dataset.Dataset method*), 327
nanstd() (*riptable.rt_fastarray.FastArray method*), 452
nanstd() (*riptable.rt_groupbyops.GroupByOps method*), 508
nanstd() (*riptable.rt_multiset.Multiset method*), 577
nansum() (*in module riptable.rt_numpy*), 619

nansum() (*riptable.rt_dataset.Dataset method*), 327
nansum() (*riptable.rt_fastarray.FastArray method*), 454
nansum() (*riptable.rt_groupbyops.GroupByOps method*), 509
nansum() (*riptable.rt_multiset.Multiset method*), 577
nanvar() (*in module riptable.rt_numpy*), 620
nanvar() (*riptable.rt_dataset.Dataset method*), 327
nanvar() (*riptable.rt_fastarray.FastArray method*), 455
nanvar() (*riptable.rt_groupbyops.GroupByOps method*), 510
nanvar() (*riptable.rt_multiset.Multiset method*), 577
nb_char (*riptable.rt_str.FAString attribute*), 652
nb_char_par (*riptable.rt_str.FAString attribute*), 652
nb_contains (*riptable.rt_str.FAString attribute*), 652
nb_contains_par (*riptable.rt_str.FAString attribute*), 652
nb_ema() (*riptable.rt_groupbynumba.GroupbyNumba method*), 487
nb_endswith (*riptable.rt_str.FAString attribute*), 652
nb_endswith_par (*riptable.rt_str.FAString attribute*), 652
nb_fill_backward() (*riptable.rt_groupbynumba.GroupbyNumba method*), 487
nb_fill_forward() (*riptable.rt_groupbynumba.GroupbyNumba method*), 488
nb_find (*riptable.rt_str.FAString attribute*), 652
nb_index (*riptable.rt_str.FAString attribute*), 652
nb_index_any_of (*riptable.rt_str.FAString attribute*), 652
nb_index_any_of_par (*riptable.rt_str.FAString attribute*), 652
nb_index_par (*riptable.rt_str.FAString attribute*), 652
nb_lower (*riptable.rt_str.FAString attribute*), 652
nb_lower_par (*riptable.rt_str.FAString attribute*), 652
nb_min() (*riptable.rt_groupbynumba.GroupbyNumba method*), 488
nb_removetrailing (*riptable.rt_str.FAString attribute*), 652
nb_removetrailing_par (*riptable.rt_str.FAString attribute*), 652
nb_replace (*riptable.rt_str.FAString attribute*), 652
nb_replace_par (*riptable.rt_str.FAString attribute*), 652
nb_reverse (*riptable.rt_str.FAString attribute*), 652
nb_reverse_inplace (*riptable.rt_str.FAString attribute*), 652
nb_reverse_inplace_par (*riptable.rt_str.FAString attribute*), 652
nb_reverse_par (*riptable.rt_str.FAString attribute*), 652
nb_startswith (*riptable.rt_str.FAString attribute*), 652
nb_startswith_par (*riptable.rt_str.FAString attribute*), 652
nb_strlen (*riptable.rt_str.FAString attribute*), 653
nb_strlen_par (*riptable.rt_str.FAString attribute*), 653
nb_substr (*riptable.rt_str.FAString attribute*), 653
nb_substr_par (*riptable.rt_str.FAString attribute*), 653
nb_sum() (*riptable.rt_groupbynumba.GroupbyNumba method*), 488
nb_sum_punt_test() (*riptable.rt_groupbynumba.GroupbyNumba method*), 488
nb_upper (*riptable.rt_str.FAString attribute*), 653
nb_upper_inplace (*riptable.rt_str.FAString attribute*), 653
nb_upper_inplace_par (*riptable.rt_str.FAString attribute*), 653
nb_upper_par (*riptable.rt_str.FAString attribute*), 653
ncols (*riptable.rt_categorical.Categories property*), 259
ncountgroup (*riptable.rt_accum2.Accum2 property*), 198
ncountgroup (*riptable.rt_grouping.Grouping property*), 524
ncountkey (*riptable.rt_accum2.Accum2 property*), 198
ncountkey (*riptable.rt_grouping.Grouping property*), 524
ne() (*riptable.rt_fastarray.FastArray method*), 457
NEG (*riptable.rt_enum.MATH_OPERATION attribute*), 406
NEGATIVE (*riptable.rt_enum.MATH_OPERATION attribute*), 406
NEW_ARRAY_FUNCTION_ENABLED (*riptable.rt_fastarray.FastArray attribute*), 419
newclassfrominstance() (*riptable.rt_categorical.Categorical class method*), 249
newclassfrominstance() (*riptable.rt_datetime.DateTimeNano class method*), 379
newclassfrominstance() (*riptable.rt_datetime.TimeSpan class method*), 387
newclassfrominstance() (*riptable.rt_enum.TypeRegister class method*), 411
newclassfrominstance() (*riptable.rt_grouping.Grouping class method*), 532
newgroupfrominstance() (*riptable.rt_grouping.Grouping method*), 532
ngroup() (*riptable.rt_groupbyops.GroupByOps method*), 510
no_colors() (*riptable.Utils.display_options.DisplayOptions method*), 180
no_colors() (*riptable.Utils.display_options.DisplayOptions static method*), 183

no_styles (*riptable.rt_display.DisplayCell attribute*), 394
NO_STYLES (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 182
node_label() (*riptable.Utils.rt_display_nested.Style method*), 188
Noncomputable (*riptable.rt_enum.NumpyCharTypes attribute*), 407
noncomputable() (*riptable.rt_dataset.Dataset method*), 327
normalize_keys() (*in module riptable.rt_utils*), 715
normalize_minmax() (*in module riptable.rt_mlutils*), 572
normalize_minmax() (*riptable.rt_dataset.Dataset method*), 327
normalize_minmax() (*riptable.rt_fastarray.FastArray method*), 457
normalize_tz_to_tzdb_name() (*riptable.rt_timezone.TimeZone static method*), 709
normalize_zscore() (*in module riptable.rt_mlutils*), 572
normalize_zscore() (*riptable.rt_dataset.Dataset method*), 327
normalize_zscore() (*riptable.rt_fastarray.FastArray method*), 457
notna() (*riptable.rt_categorical.Categorical method*), 249
notna() (*riptable.rt_fastarray.FastArray method*), 457
NoTolerance (*riptable.rt_fastarray.FastArray attribute*), 419
now() (*riptable.rt_fastarray.Recycle static method*), 471
np_quantile_mult() (*riptable.rt_groupbyops.GroupByOps static method*), 511
nrows (*riptable.rt_categorical.Categories property*), 259
nth() (*riptable.rt_categorical.Categorical method*), 249
nth() (*riptable.rt_groupby.GroupBy method*), 481
nth() (*riptable.rt_groupbyops.GroupByOps method*), 511
null() (*riptable.rt_groupbyops.GroupByOps method*), 511
numba_apply() (*riptable.rt_categorical.Categorical method*), 250
numbastring (*riptable.rt_fastarray.FastArray property*), 419
NUMBER_SEPARATOR (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 182
NUMBER_SEPARATOR_CHAR (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 182
numeric_modes (*riptable.rt_categorical.Categories attribute*), 259
numpy2d_to_dict() (*in module riptable.Utils.conversion_utils*), 176
numpy_array_to_dataset() (*in module riptable.Utils.conversion_utils*), 177
numpy_array_to_dict() (*in module riptable.Utils.conversion_utils*), 177
NumpyAggNames (*riptable.rt_groupbyops.GroupByOps attribute*), 490
NumpyCharTypes (*class in riptable.rt_enum*), 406
nunique() (*riptable.rt_categorical.Categorical method*), 251
nunique() (*riptable.rt_fastarray.FastArray method*), 457

## O

off() (*riptable.rt_fastarray.Ledger static method*), 471
off() (*riptable.rt_fastarray.Recycle static method*), 471
off() (*riptable.rt_fastarray.Threading static method*), 472
ohlc() (*riptable.rt_groupbyops.GroupByOps method*), 512
on() (*riptable.rt_fastarray.Ledger static method*), 471
on() (*riptable.rt_fastarray.Recycle static method*), 471
on() (*riptable.rt_fastarray.Threading static method*), 472
one_hot_encode() (*riptable.rt_categorical.Categorical method*), 251
one_hot_encode() (*riptable.rt_dataset.Dataset method*), 327
onedict() (*riptable.rt_grouping.Grouping method*), 532
ones() (*in module riptable.rt_numpy*), 621
ones_like() (*in module riptable.rt_numpy*), 622
options (*riptable.rt_display.DisplayTable attribute*), 397
ordered (*riptable.rt_categorical.Categorical property*), 226
ORDINAL_DATE (*riptable.rt_enum.DATETIME_TYPES attribute*), 401
os_name (*in module riptable.Utils.appdirs*), 173
outliers() (*riptable.rt_dataset.Dataset method*), 327
output_cache_flush() (*in module riptable.rt_misc*), 570
output_cache_none() (*in module riptable.rt_misc*), 570
output_cache_setsize() (*in module riptable.rt_misc*), 570

## P

P_THRESHOLD (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 182
p_threshold() (*riptable.Utils.display_options.DisplayOptions class method*), 183
p_threshold() (*riptable.Utils.display_options.DisplayOptions method*), 180

pack_by_group() (*riptable.rt_grouping.Grouping method*), 533
packed (*riptable.rt_grouping.Grouping property*), 524
pad() (*riptable.rt_groupby.GroupBy method*), 481
paint_cell() (*riptable.rt_display.DisplayCell method*), 395
paint_column() (*riptable.rt_display.DisplayColumn method*), 396
paint_highlightmax() (*riptable.rt_display.DisplayColumn method*), 396
paint_posneg() (*riptable.rt_display.DisplayColumn method*), 396
pandas_series_to_riptable() (*in module riptable.Utils.pandas_utils*), 185
parse_epoch() (*in module riptable.rt_datetime*), 391
parse_header_tuples() (*in module riptable.rt_misc*), 570
partition() (*riptable.rt_pdataset.PDataset method*), 636
partition2() (*riptable.rt_fastarray.FastArray method*), 458
pcat (*riptable.rt_pdataset.PDataset property*), 633
pcount (*riptable.rt_pdataset.PDataset property*), 633
pcutoffs (*riptable.rt_pdataset.PDataset property*), 634
PDataset (*class in riptable.rt_pdataset*), 633
PDataset (*riptable.rt_enum.TypeRegister attribute*), 410
pdict (*riptable.rt_pdataset.PDataset property*), 634
percentile() (*in module riptable.rt_numpy*), 623
percentile() (*riptable.rt_fastarray.FastArray method*), 458
percentile() (*riptable.rt_groupbyops.GroupByOps method*), 512
pgb() (*riptable.rt_pdataset.PDataset method*), 637
PGroupBy (*class in riptable.rt_pgroupby*), 639
pgroupby() (*riptable.rt_pdataset.PDataset method*), 637
Pink (*riptable.rt_enum.DisplayColumnColors attribute*), 402
piter (*riptable.rt_pdataset.PDataset property*), 634
pivot() (*riptable.rt_dataset.Dataset method*), 327
pivot() (*riptable.rt_multiset.Multiset method*), 577
plain_string_list() (*riptable.rt_display.DisplayColumn method*), 396
pload() (*riptable.rt_pdataset.PDataset class method*), 637
plot_hist() (*in module riptable.rt_stats*), 649
plotPrediction() (*in module riptable.rt_stats*), 649
pnames (*riptable.rt_pdataset.PDataset property*), 634
pop() (*riptable.rt_itemcontainer.ItemContainer method*), 544
POSITIVE (*riptable.rt_enum.MATH_OPERATION attribute*), 406
possibly_convert_tostr() (*riptable.rt_str.FAString method*), 658
possibly_invalid() (*riptable.rt_categorical.Categories method*), 261
possibly_recast() (*riptable.rt_grouping.Grouping class method*), 533
pow (*riptable.rt_fastarray.FastArray attribute*), 420
POWER (*riptable.rt_enum.MATH_OPERATION attribute*), 406
PRECISION (*riptable.rt_datetime.DateTimeBase attribute*), 362
PRECISION (*riptable.Utils.display_options.DisplayOptions attribute*), 180, 182
printh() (*in module riptable.rt_io*), 539
profile_func() (*in module riptable.rt_misc*), 570
prow_labeler() (*riptable.rt_pdataset.PDataset method*), 637
prows (*riptable.rt_pdataset.PDataset property*), 635
psave() (*riptable.rt_pdataset.PDataset method*), 637
pslice() (*riptable.rt_pdataset.PDataset method*), 637
pslices (*riptable.rt_pdataset.PDataset property*), 635
Purple (*riptable.rt_enum.DisplayColumnColors attribute*), 402
push() (*riptable.rt_fastarray.FastArray method*), 458
putmask() (*in module riptable.rt_numpy*), 623
putmask() (*riptable.rt_dataset.Dataset method*), 328
PY3 (*in module riptable.Utils.appdirs*), 173

## Q

qcut() (*in module riptable.rt_bin*), 209
quantile() (*in module riptable.rt_bin*), 210
quantile() (*riptable.rt_dataset.Dataset method*), 329
quantile() (*riptable.rt_fastarray.FastArray method*), 458
quantile() (*riptable.rt_groupbyops.GroupByOps method*), 512
quantile() (*riptable.rt_multiset.Multiset method*), 577
QUANTILE_MULTIPLIER (*riptable.rt_groupbyops.GroupByOps attribute*), 490
quantile_name_from_param() (*riptable.rt_groupbyops.GroupByOps static method*), 513

## R

r2() (*in module riptable.rt_stats*), 649
rand_floats() (*in module riptable.Utils.common*), 175
rand_integers() (*in module riptable.Utils.common*), 175
random() (*riptable.rt_datetime.DateTimeNano class method*), 379
random_invalid() (*riptable.rt_datetime.DateTimeNano class method*),

380
range() (*riptable.rt_datetime.Date class method*), 355
rank() (*riptable.rt_groupbyops.GroupByOps method*), 513
rankdata() (*riptable.rt_fastarray.FastArray method*), 458
read_dset_from_np() (*in module riptable.rt_io*), 540
rebuild() (*riptable.rt_imatrix.IMatrix method*), 539
RECIPROCAL (*riptable.rt_enum.MATH_OPERATION attribute*), 406
Record (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
Recycle (*class in riptable.rt_fastarray*), 471
Recycle (*riptable.rt_fastarray.FastArray attribute*), 419
Red (*riptable.rt_enum.DisplayColumnColors attribute*), 402
reduce() (*riptable.rt_dataset.Dataset method*), 329
REDUCE_ALL (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_ANY (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_ARGMAX (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_ARGMIN (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_FUNCTIONS (*class in riptable.rt_enum*), 407
REDUCE_MAX (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_MEAN (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_MIN (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANARGMAX (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANARGMIN (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANMAX (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANMEAN (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANMIN (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANSTD (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANSUM (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_NANVAR (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_STD (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_SUM (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
REDUCE_VAR (*riptable.rt_enum.REDUCE_FUNCTIONS attribute*), 407
regex_match() (*riptable.rt_str.FAString method*), 658
regex_replace() (*riptable.rt_str.FAString method*), 659
register_array_function() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 416
register_array_function_type_compatibility() (*riptable.rt_fastarray.FastArray._ArrayFunctionHelper class method*), 416
register_function() (*riptable.rt_fastarray.FastArray class method*), 458
register_functions() (*riptable.rt_groupbyops.GroupByOps class method*), 513
register_functions() (*riptable.rt_grouping.Grouping class method*), 533
register_null_log_handler() (*in module riptable.conftest*), 196
REGISTERED_REVERSE_TABLES (*riptable.rt_grouping.Grouping attribute*), 525
regroup() (*riptable.rt_grouping.Grouping method*), 533
reindex_fast() (*in module riptable.rt_numpy*), 623
REMAINDER (*riptable.rt_enum.MATH_OPERATION attribute*), 406
removetrailing() (*riptable.rt_str.FAString method*), 660
render() (*riptable.Utils.rt_display_nested.LeftAligned method*), 188
repeat() (*riptable.rt_datetime.DateScalar method*), 357
repeat() (*riptable.rt_datetime.DateSpanScalar method*), 362
repeat() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
repeat() (*riptable.rt_datetime.TimeSpanScalar method*), 389
repeat() (*riptable.rt_fastarray.FastArray method*), 458
replace() (*riptable.rt_fastarray.FastArray method*), 458
replace() (*riptable.rt_str.FAString method*), 660
replacena() (*riptable.rt_fastarray.FastArray method*), 458
REPR (*riptable.rt_enum.DS_DISPLAY_TYPES attribute*), 401
resample() (*riptable.rt_datetime.DateTimeNano method*), 381
resample() (*riptable.rt_groupbyops.GroupByOps method*), 514
RESET (*riptable.rt_display.DisplayText attribute*), 400
reset_config() (*riptable.Utils.display_options.DisplayOptions*

*method*), 180
reset_config() (*riptable.Utils.display_options.DisplayOptions static method*), 183
reshape() (*in module riptable.rt_numpy*), 623
reshape() (*riptable.rt_fastarray.FastArray method*), 459
result_rowcount() (*riptable.rt_merge.JoinIndices static method*), 546
reverse (*riptable.rt_str.FAString property*), 650
reverse_inplace (*riptable.rt_str.FAString property*), 651
Right (*riptable.rt_enum.DisplayJustification attribute*), 402
right_index (*riptable.rt_merge.JoinIndices attribute*), 546
right_only_rowcount (*riptable.rt_merge.JoinIndices attribute*), 546
RINT (*riptable.rt_enum.MATH_OPERATION attribute*), 406
riptable
    module, 167
riptable.config
    module, 195
riptable.conftest
    module, 196
riptable.numba
    module, 192
riptable.numba.indexing
    module, 192
riptable.numba.invalid_values
    module, 193
riptable.rt_accum2
    module, 196
riptable.rt_accumtable
    module, 202
riptable.rt_algos
    module, 206
riptable.rt_bin
    module, 206
riptable.rt_categorical
    module, 211
riptable.rt_compressedarray
    module, 262
riptable.rt_csv
    module, 266
riptable.rt_dataset
    module, 266
riptable.rt_datetime
    module, 340
riptable.rt_display
    module, 394
riptable.rt_ema
    module, 400
riptable.rt_enum
    module, 400
riptable.rt_fastarray
    module, 411
riptable.rt_fastarraynumba
    module, 473
riptable.rt_groupby
    module, 475
riptable.rt_groupbykeys
    module, 482
riptable.rt_groupbynumba
    module, 485
riptable.rt_groupbyops
    module, 488
riptable.rt_grouping
    module, 518
riptable.rt_hstack
    module, 537
riptable.rt_imatrix
    module, 539
riptable.rt_io
    module, 539
riptable.rt_itemcontainer
    module, 540
riptable.rt_ledger
    module, 544
riptable.rt_matplotlib
    module, 546
riptable.rt_merge
    module, 546
riptable.rt_merge_asof
    module, 561
riptable.rt_meta
    module, 566
riptable.rt_misc
    module, 569
riptable.rt_mlutils
    module, 572
riptable.rt_multiset
    module, 572
riptable.rt_numpy
    module, 578
riptable.rt_pdataset
    module, 633
riptable.rt_pgroupby
    module, 639
riptable.rt_sds
    module, 640
riptable.rt_sharedmemory
    module, 648
riptable.rt_sort_cache
    module, 648
riptable.rt_stats
    module, 649

riptable.rt_str
module, 650
riptable.rt_struct
module, 661
riptable.rt_timers
module, 704
riptable.rt_timezone
module, 707
riptable.rt_utils
module, 710
riptable.Utils
module, 167
riptable.Utils.appdirs
module, 167
riptable.Utils.common
module, 173
riptable.Utils.conversion_utils
module, 176
riptable.Utils.display_options
module, 177
riptable.Utils.ipython_utils
module, 184
riptable.Utils.pandas_utils
module, 185
riptable.Utils.rt_display_nested
module, 186
riptable.Utils.rt_display_properties
module, 189
riptable.Utils.rt_metadata
module, 190
riptable.Utils.teamcity_helper
module, 192
riptable.Utils.terminalsize
module, 192
rolling_count() (*riptable.rt_groupbyops.GroupByOps method*), 514
rolling_diff() (*riptable.rt_groupbyops.GroupByOps method*), 514
ROLLING_FUNCTIONS (*class in riptable.rt_enum*), 407
ROLLING_MEAN (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_mean() (*riptable.rt_fastarray.FastArray method*), 459
rolling_mean() (*riptable.rt_groupbyops.GroupByOps method*), 514
rolling_median() (*riptable.rt_groupbyops.GroupByOps method*), 514
ROLLING_NANMEAN (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_nanmean() (*riptable.rt_fastarray.FastArray method*), 459
rolling_nanmean() (*riptable.rt_groupbyops.GroupByOps method*), 514
ROLLING_NANSTD (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_nanstd() (*riptable.rt_fastarray.FastArray method*), 459
ROLLING_NANSUM (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_nansum() (*riptable.rt_fastarray.FastArray method*), 459
rolling_nansum() (*riptable.rt_groupbyops.GroupByOps method*), 514
ROLLING_NANVAR (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_nanvar() (*riptable.rt_fastarray.FastArray method*), 459
ROLLING_QUANTILE (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_quantile() (*riptable.rt_fastarray.FastArray method*), 459
rolling_quantile() (*riptable.rt_groupbyops.GroupByOps method*), 515
rolling_shift() (*riptable.rt_groupbyops.GroupByOps method*), 515
ROLLING_STD (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_std() (*riptable.rt_fastarray.FastArray method*), 460
ROLLING_SUM (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_sum() (*riptable.rt_fastarray.FastArray method*), 460
rolling_sum() (*riptable.rt_groupbyops.GroupByOps method*), 515
ROLLING_VAR (*riptable.rt_enum.ROLLING_FUNCTIONS attribute*), 408
rolling_var() (*riptable.rt_fastarray.FastArray method*), 460
ROUND (*riptable.rt_enum.MATH_OPERATION attribute*), 406
round() (*in module riptable.rt_numpy*), 623
ROW_ALL (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 182
Rownum (*riptable.rt_enum.DisplayColumnColors attribute*), 402

## S


SafeConversions (*riptable.rt_fastarray.FastArray attribute*), 419
sample() (*riptable.rt_dataset.Dataset method*), 330
sample() (*riptable.rt_fastarray.FastArray method*), 460
save() (*riptable.rt_dataset.Dataset method*), 331
save() (*riptable.rt_fastarray.FastArray method*), 461
save() (*riptable.rt_pdataset.PDataset method*), 637
save() (*riptable.rt_struct.Struct method*), 701
save_config() (*riptable.Utils.display_options.DisplayOptions method*), 180
save_config() (*riptable.Utils.display_options.DisplayOptions static method*), 184
save_sds() (*in module riptable.rt_sds*), 644
save_struct() (*in module riptable.rt_sds*), 646
scalar_or_lookup() (*in module riptable.numba.indexing*), 193
SD_CELL (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_CHAR (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_CLASS (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_DATASET (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_FUNCTIONH (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_LOGICAL (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_NUMERIC (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_NUMPY (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_PANDAS (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_SCALAR (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_STRUCT (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_TYPES (*class in riptable.rt_enum*), 408
SD_UNKNOWN (*riptable.rt_enum.SD_TYPES attribute*), 408
SD_VECTOR (*riptable.rt_enum.SD_TYPES attribute*), 408
sds_concat() (*in module riptable.rt_sds*), 646
sds_flatten() (*in module riptable.rt_sds*), 647
sds_info() (*in module riptable.rt_sds*), 647
sds_tree() (*in module riptable.rt_sds*), 647
SDSMakeDirsOff() (*in module riptable.rt_sds*), 640
SDSMakeDirsOn() (*in module riptable.rt_sds*), 640
SDSRebuildRootOff() (*in module riptable.rt_sds*), 640
SDSRebuildRootOn() (*in module riptable.rt_sds*), 640
SDSVerboseOff() (*in module riptable.rt_sds*), 640
SDSVerboseOn() (*in module riptable.rt_sds*), 640
searchsorted() (*in module riptable.rt_numpy*), 623
searchsorted() (*riptable.rt_fastarray.FastArray method*), 462
seconds_since_epoch (*riptable.rt_datetime.Date property*), 346
sem() (*riptable.rt_groupbyops.GroupByOps method*), 515
set_attribute() (*riptable.rt_struct.Struct method*), 702
set_dirty() (*riptable.rt_grouping.Grouping method*), 533
set_display_callback() (*riptable.rt_struct.Struct method*), 702
set_fast_array() (*riptable.rt_struct.Struct static method*), 702
set_footer_rows() (*riptable.rt_accumtable.AccumTable method*), 204
set_margin_columns() (*riptable.rt_accumtable.AccumTable method*), 204
set_name() (*riptable.rt_categorical.Categorical method*), 252
set_name() (*riptable.rt_fastarray.FastArray method*), 462
set_name() (*riptable.rt_grouping.Grouping method*), 533
set_pnames() (*riptable.rt_pdataset.PDataset method*), 638
set_timezone() (*riptable.rt_datetime.DateTimeNano method*), 382
set_valid() (*riptable.rt_categorical.Categorical method*), 252
setdefault() (*riptable.rt_itemcontainer.ItemContainer method*), 544
setdefault() (*riptable.Utils.rt_metadata.MetaData method*), 191
Settings (*class in riptable.config*), 195
shape (*riptable.rt_struct.Struct property*), 672
SharedMemory (*class in riptable.rt_sharedmemory*), 648
SharedMemory (*riptable.rt_enum.TypeRegister attribute*), 410
shift() (*riptable.rt_categorical.Categorical method*), 252
shift() (*riptable.rt_datetime.DateTimeNano method*), 382
shift() (*riptable.rt_fastarray.FastArray method*), 464
shift() (*riptable.rt_groupbyops.GroupByOps method*), 515
shift_cat() (*riptable.rt_categorical.Categorical method*), 255
Short (*riptable.rt_enum.DisplayLength attribute*), 403
show_all() (*riptable.rt_dataset.Dataset method*), 332
ShowEmpty (*riptable.rt_pgroupby.PGroupBy attribute*), 639
showpartitions() (*riptable.rt_pdataset.PDataset method*), 638
shrink() (*riptable.rt_categorical.Categorical method*), 255
shrink() (*riptable.rt_grouping.Grouping method*), 534
SIGN (*riptable.rt_enum.MATH_OPERATION attribute*),

406
sign() (*riptable.rt_fastarray.FastArray method*), 465
SIGNBIT (*riptable.rt_enum.MATH_OPERATION attribute*), 406
SignedInteger64 (*riptable.rt_enum.NumpyCharTypes attribute*), 407
single() (*in module riptable.rt_numpy*), 623
singlekey (*riptable.rt_groupbykeys.GroupByKeys property*), 483
site_config_dir (*riptable.Utils.appdirs.AppDirs property*), 168
site_config_dir() (*in module riptable.Utils.appdirs*), 169
site_data_dir (*riptable.Utils.appdirs.AppDirs property*), 168
site_data_dir() (*in module riptable.Utils.appdirs*), 169
size (*riptable.rt_accum2.Accum2 property*), 198
size (*riptable.rt_dataset.Dataset property*), 272
SM_DTYPES (*class in riptable.rt_enum*), 408
Sort (*riptable.rt_enum.DisplayColumnColors attribute*), 402
sort() (*in module riptable.rt_numpy*), 624
sort() (*riptable.rt_grouping.Grouping method*), 535
sort_copy() (*riptable.rt_dataset.Dataset method*), 333
sort_copy() (*riptable.rt_multiset.Multiset method*), 577
sort_gb (*riptable.rt_categorical.Categorical property*), 226
sort_gb_data (*riptable.rt_groupbykeys.GroupByKeys property*), 483
sort_inplace() (*riptable.rt_dataset.Dataset method*), 334
sort_inplace() (*riptable.rt_multiset.Multiset method*), 577
sort_view() (*riptable.rt_dataset.Dataset method*), 336
SortCache (*class in riptable.rt_sort_cache*), 648
SortCache (*riptable.rt_enum.TypeRegister attribute*), 410
sorted (*riptable.rt_categorical.Categorical property*), 226
sortinplaceindirect() (*in module riptable.rt_numpy*), 624
sorts_off() (*riptable.rt_dataset.Dataset method*), 337
sorts_on() (*riptable.rt_dataset.Dataset method*), 337
SQRT (*riptable.rt_enum.MATH_OPERATION attribute*), 406
SQUARE (*riptable.rt_enum.MATH_OPERATION attribute*), 406
squeeze() (*riptable.rt_fastarray.FastArray method*), 465
stack() (*riptable.rt_groupby.GroupBy method*), 482
stack_rows (*in module riptable.rt_hstack*), 538
start_of_month (*riptable.rt_datetime.Date property*), 346
start_of_week (*riptable.rt_datetime.Date property*), 346
startswith() (*riptable.rt_str.FAString method*), 660
statx() (*in module riptable.rt_stats*), 649
statx() (*riptable.rt_fastarray.FastArray method*), 465
std() (*in module riptable.rt_numpy*), 624
std() (*riptable.rt_dataset.Dataset method*), 337
std() (*riptable.rt_fastarray.FastArray method*), 465
std() (*riptable.rt_groupbyops.GroupByOps method*), 515
std() (*riptable.rt_multiset.Multiset method*), 577
steward (*riptable.rt_meta.Info attribute*), 567
steward (*riptable.rt_meta.Item attribute*), 568
store_sort() (*riptable.rt_sort_cache.SortCache class method*), 648
STR (*riptable.rt_enum.DS_DISPLAY_TYPES attribute*), 401
str (*riptable.rt_str.FAString property*), 651
str() (*riptable.rt_categorical.Categorical method*), 256
str() (*riptable.rt_fastarray.FastArray method*), 466
str2intdict (*riptable.rt_categorical.Categories property*), 259
str_ (*class in riptable.rt_numpy*), 586
str_append() (*riptable.rt_fastarray.FastArray method*), 467
str_to_bytes() (*in module riptable.rt_utils*), 715
strftime() (*riptable.rt_datetime.Date method*), 356
strftime() (*riptable.rt_datetime.DateScalar method*), 357
strftime() (*riptable.rt_datetime.DateSpan method*), 361
strftime() (*riptable.rt_datetime.DateTimeNano method*), 382
strftime() (*riptable.rt_datetime.DateTimeNanoScalar method*), 384
strftime() (*riptable.rt_datetime.TimeSpan method*), 387
strftime() (*riptable.rt_datetime.TimeSpanScalar method*), 389
String (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
string (*riptable.Utils.rt_metadata.MetaData property*), 191
string_modes (*riptable.rt_categorical.Categories attribute*), 259
strlen (*riptable.rt_str.FAString property*), 651
strpbrk() (*riptable.rt_str.FAString method*), 661
strptime_to_nano() (*in module riptable.rt_datetime*), 392
strstr() (*riptable.rt_str.FAString method*), 661
strstrb() (*riptable.rt_str.FAString method*), 661
Struct (*class in riptable.rt_struct*), 661
Struct (*riptable.rt_enum.TypeRegister attribute*), 410
Style (*class in riptable.Utils.rt_display_nested*), 188

style_classes_html() (*riptable.rt_display.DisplayColumn static method*), 396
style_column() (*riptable.rt_display.DisplayColumn method*), 396
styled_string_list() (*riptable.rt_display.DisplayColumn method*), 396
SUB (*riptable.rt_enum.MATH_OPERATION attribute*), 406
sub (*riptable.rt_fastarray.FastArray attribute*), 420
sub2ind() (*in module riptable.rt_misc*), 570
SUBDATES (*riptable.rt_enum.MATH_OPERATION attribute*), 406
SUBDATETIMES (*riptable.rt_enum.MATH_OPERATION attribute*), 406
substr (*riptable.rt_str.CatString property*), 650
substr (*riptable.rt_str.FAString property*), 651
substr_char_stop() (*riptable.rt_str.FAString method*), 661
sum() (*in module riptable.rt_numpy*), 625
sum() (*riptable.rt_dataset.Dataset method*), 337
sum() (*riptable.rt_groupbyops.GroupByOps method*), 516
sum() (*riptable.rt_multiset.Multiset method*), 577
summary() (*riptable.Utils.rt_display_properties.ItemFormat method*), 190
summary_as_dict() (*riptable.rt_itemcontainer.ItemContainer method*), 544
summary_as_dict() (*riptable.rt_struct.Struct method*), 702
summary_get_names() (*riptable.rt_itemcontainer.ItemContainer method*), 544
summary_get_names() (*riptable.rt_struct.Struct method*), 702
summary_remove() (*riptable.rt_itemcontainer.ItemContainer method*), 544
summary_remove() (*riptable.rt_struct.Struct method*), 702
summary_set_names() (*riptable.rt_itemcontainer.ItemContainer method*), 544
summary_set_names() (*riptable.rt_struct.Struct method*), 702
Supported (*riptable.rt_enum.NumpyCharTypes attribute*), 407
SupportedAlternate (*riptable.rt_enum.NumpyCharTypes attribute*), 407
SupportedFloat (*riptable.rt_enum.NumpyCharTypes attribute*), 407

## T

tail() (*riptable.rt_dataset.Dataset method*), 337
tail() (*riptable.rt_groupbyops.GroupByOps method*), 516
TAIL_ROWS (*riptable.Utils.display_options.DisplayOptions attribute*), 179, 182
take_groups() (*riptable.rt_grouping.Grouping static method*), 535
TestCatGb (*riptable.rt_groupby.GroupBy attribute*), 476
TestCatGb (*riptable.rt_pgroupby.PGroupBy attribute*), 639
TestFooter (*riptable.rt_display.DisplayTable attribute*), 397
TestIsMemberVerbose (*riptable.rt_categorical.Categorical attribute*), 226
text_decoration_html() (*riptable.rt_display.DisplayColumn static method*), 396
Threading (*class in riptable.rt_fastarray*), 472
threads() (*riptable.rt_fastarray.Threading static method*), 472
tic() (*in module riptable.rt_timers*), 704
ticf() (*in module riptable.rt_timers*), 704
ticp() (*in module riptable.rt_timers*), 704
ticx() (*in module riptable.rt_timers*), 705
tile() (*in module riptable.rt_numpy*), 626
tile() (*riptable.rt_datetime.DateScalar method*), 358
tile() (*riptable.rt_datetime.DateSpanScalar method*), 362
tile() (*riptable.rt_datetime.DateTimeNanoScalar method*), 385
tile() (*riptable.rt_datetime.TimeSpanScalar method*), 390
tile() (*riptable.rt_fastarray.FastArray method*), 467
timeout() (*riptable.rt_fastarray.Recycle static method*), 472
TimeSpan (*class in riptable.rt_datetime*), 385
TimeSpan (*riptable.rt_enum.DisplayArrayTypes attribute*), 401
TimeSpan (*riptable.rt_enum.TypeRegister attribute*), 410
TimeSpanScalar (*class in riptable.rt_datetime*), 388
timestring_to_nano() (*in module riptable.rt_datetime*), 393
TIMEWINDOW_FUNCTIONS (*class in riptable.rt_enum*), 409
TIMEWINDOW_PROD (*riptable.rt_enum.TIMEWINDOW_FUNCTIONS attribute*), 409
timewindow_prod() (*riptable.rt_fastarray.FastArray method*), 467
TIMEWINDOW_SUM (*riptable.rt_enum.TIMEWINDOW_FUNCTIONS attribute*), 409

timewindow_sum() (*riptable.rt_fastarray.FastArray method*), 467
TimeZone (*class in riptable.rt_timezone*), 707
TimeZone (*riptable.rt_enum.TypeRegister attribute*), 410
title (*riptable.rt_meta.Info attribute*), 567
TITLE_DARK (*riptable.rt_display.DisplayText attribute*), 400
title_format() (*riptable.rt_display.DisplayText static method*), 400
TITLE_LIGHT (*riptable.rt_display.DisplayText attribute*), 400
to_arrow() (*riptable.rt_categorical.Categorical method*), 257
to_arrow() (*riptable.rt_dataset.Dataset method*), 338
to_arrow() (*riptable.rt_datetime.Date method*), 357
to_arrow() (*riptable.rt_datetime.DateTimeNano method*), 383
to_arrow() (*riptable.rt_datetime.TimeSpan method*), 388
to_arrow() (*riptable.rt_fastarray.FastArray method*), 468
to_file() (*riptable.rt_fastarray.Ledger static method*), 471
to_iso() (*riptable.rt_datetime.DateTimeNano method*), 383
to_pandas() (*riptable.rt_dataset.Dataset method*), 338
to_str() (*in module riptable.rt_utils*), 715
to_utc() (*riptable.rt_timezone.TimeZone method*), 709
toc() (*in module riptable.rt_timers*), 705
tocf() (*in module riptable.rt_timers*), 705
tocp() (*in module riptable.rt_timers*), 705
tocx() (*in module riptable.rt_timers*), 706
tolist() (*riptable.rt_dataset.Dataset method*), 339
tolist() (*riptable.rt_struct.Struct method*), 703
total_size (*riptable.rt_dataset.Dataset property*), 273
total_sizes (*riptable.rt_struct.Struct property*), 672
transform (*riptable.rt_categorical.Categorical property*), 226
transform (*riptable.rt_groupby.GroupBy property*), 476
transitions() (*riptable.rt_fastarray.FastArray method*), 468
transpose() (*in module riptable.rt_numpy*), 627
transpose() (*riptable.rt_dataset.Dataset method*), 339
traverse (*riptable.Utils.rt_display_nested.LeftAligned attribute*), 188
tree() (*riptable.rt_struct.Struct method*), 703
treedir() (*in module riptable.Utils.rt_display_nested*), 188
trial_size() (*in module riptable.Utils.common*), 175
trim() (*riptable.rt_dataset.Dataset method*), 339
trim() (*riptable.rt_multiset.Multiset method*), 577
trim_string() (*in module riptable.Utils.rt_display_properties*), 190
trimbr() (*riptable.rt_groupbyops.GroupByOps method*), 517
TRUNC (*riptable.rt_enum.MATH_OPERATION attribute*), 406
trunc() (*in module riptable.rt_numpy*), 627
trunc() (*riptable.rt_fastarray.FastArray method*), 468
tt() (*in module riptable.rt_timers*), 706
ttx() (*in module riptable.rt_timers*), 706
type (*riptable.rt_meta.Info attribute*), 567
type (*riptable.rt_meta.Item attribute*), 568
typeid (*riptable.Utils.rt_metadata.MetaData property*), 191
TypeRegister (*class in riptable.rt_enum*), 409
tz_error_msg (*riptable.rt_timezone.TimeZone attribute*), 708

## U

uint0 (*class in riptable.rt_numpy*), 586
uint16 (*class in riptable.rt_numpy*), 587
uint32 (*class in riptable.rt_numpy*), 587
uint64 (*class in riptable.rt_numpy*), 587
uint8 (*class in riptable.rt_numpy*), 588
Undefined (*riptable.rt_enum.DisplayJustification attribute*), 402
Undefined (*riptable.rt_enum.DisplayLength attribute*), 403
unicode (*in module riptable.Utils.appdirs*), 173
unique() (*riptable.rt_fastarray.FastArray method*), 469
unique32() (*in module riptable.rt_numpy*), 627
unique_count (*riptable.rt_categorical.Categorical property*), 226
unique_count (*riptable.rt_groupbykeys.GroupByKeys property*), 483
unique_count (*riptable.rt_grouping.Grouping property*), 524
unique_repr (*riptable.rt_categorical.Categorical property*), 226
unique_unsorted() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
uniquedict (*riptable.rt_categorical.Categories property*), 259
uniquedict (*riptable.rt_grouping.Grouping property*), 524
uniquelist (*riptable.rt_categorical.Categories property*), 259
uniquelist (*riptable.rt_grouping.Grouping property*), 525
unlock() (*riptable.rt_categorical.Categorical method*), 257
UnsignedInteger (*riptable.rt_enum.NumpyCharTypes attribute*), 407
UnsignedInteger64 (*riptable.rt_enum.NumpyCharTypes attribute*), 407

unsort() (*riptable.rt_groupbykeys.GroupByKeys method*), 484
unstack() (*riptable.rt_groupby.GroupBy method*), 482
Unsupported (*riptable.rt_enum.NumpyCharTypes attribute*), 407
update() (*riptable.rt_itemcontainer.ItemContainer method*), 544
upper (*riptable.rt_str.FAString property*), 651
upper_inplace (*riptable.rt_str.FAString property*), 652
UseFastArray (*riptable.rt_struct.Struct attribute*), 673
user_cache_dir (*riptable.Utils.appdirs.AppDirs property*), 169
user_cache_dir() (*in module riptable.Utils.appdirs*), 170
user_config_dir (*riptable.Utils.appdirs.AppDirs property*), 169
user_config_dir() (*in module riptable.Utils.appdirs*), 171
user_data_dir (*riptable.Utils.appdirs.AppDirs property*), 169
user_data_dir() (*in module riptable.Utils.appdirs*), 171
user_log_dir (*riptable.Utils.appdirs.AppDirs property*), 169
user_log_dir() (*in module riptable.Utils.appdirs*), 172
user_state_dir (*riptable.Utils.appdirs.AppDirs property*), 169
user_state_dir() (*in module riptable.Utils.appdirs*), 172
utcnow() (*in module riptable.rt_timers*), 706

## V

valid_timezones (*riptable.rt_timezone.TimeZone attribute*), 708
validate_registry() (*riptable.rt_enum.TypeRegister class method*), 411
values() (*riptable.rt_itemcontainer.ItemContainer method*), 544
values() (*riptable.rt_struct.Struct method*), 703
var() (*in module riptable.rt_numpy*), 627
var() (*riptable.rt_dataset.Dataset method*), 340
var() (*riptable.rt_fastarray.FastArray method*), 469
var() (*riptable.rt_groupbyops.GroupByOps method*), 517
var() (*riptable.rt_multiset.Multiset method*), 577
Verbose (*riptable.rt_fastarray.FastArray attribute*), 419
Verbose (*riptable.rt_ledger.MathLedger attribute*), 545
Verbose (*riptable.Utils.rt_display_properties.DisplayConvert attribute*), 189
VerboseConversion (*riptable.rt_ledger.MathLedger attribute*), 545
vstack() (*in module riptable.rt_numpy*), 628

## W

WarningDict (*riptable.rt_fastarray.FastArray attribute*), 419
WarningLevel (*riptable.rt_fastarray.FastArray attribute*), 419
WarnOnInvalidNames (*riptable.rt_struct.Struct attribute*), 673
where() (*in module riptable.rt_numpy*), 629
where() (*riptable.rt_fastarray.FastArray method*), 470
winsorize() (*in module riptable.rt_stats*), 649
write_dset_to_np() (*in module riptable.rt_io*), 540

## X

X_PADDING (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 182

## Y

Y_PADDING (*riptable.Utils.display_options.DisplayOptions attribute*), 178, 182
year (*riptable.rt_datetime.Date property*), 346
yyyymmdd (*riptable.rt_datetime.Date property*), 347

## Z

zeros() (*in module riptable.rt_numpy*), 631
zeros_eager() (*in module riptable.Utils.common*), 175
zeros_like() (*in module riptable.rt_numpy*), 632