

---

# **Raccoon Documentation**

*Release 2.1.4*

**Ryan Sheftel**

**Sep 12, 2017**



---

## Contents

---

<b>1 Overview</b>	<b>1</b>
<b>2 Updates</b>	<b>3</b>
<b>3 Contents</b>	<b>7</b>
<b>4 Indices and tables</b>	<b>57</b>
<b>Python Module Index</b>	<b>59</b>



Raccoon is a lightweight DataFrame and Series implementation inspired by the phenomenal Pandas package for the one use case where Pandas is known to be sub-optimal: DataFrames and Series that grow in size by rows frequently in the code. A simple speed comparison is below in the contents.

## Source location

Hosted on GitHub: <https://github.com/rsheftel/raccoon>

## Inspiration

Pandas DataFrames and Series are excellent multi-purpose data structures for data management and analysis. One of the use cases I had was to use DataFrames as a type of in-memory database table. The issue was that this required lots of growing the rows of the DataFrame, something that is known to be slow in Pandas. The reason it is slow in Pandas is that the underlying data structure is numpy which does a complete copy of the data when the size of the array grows.

## Functionality

Raccoon implements what is needed to use the DataFrame as an in memory store of index and column data structure supporting simple and tuple indexes to mimic the hierarchical indexes of Pandas. The methods included are primarily about setting values of the data frame, growing and appending the data frame and getting values from the data frame. The raccoon DataFrame is not intended for math operations like pandas and only limited basic math methods are included.

## Underlying Data Structure

Raccoon uses the standard built in lists. There is an option on object construction to use fast blist <http://stutzbachenterprises.com/blist/> list replacement for the underlying data structure.

## Why Raccoon?

According to wikipedia some scientists believe the panda is related to the raccoon

## Future

This package serves the needs it was originally created for. Any future additions by myself will be driven by my own needs, but it is completely open source to I encourage anyone to add on and expand.

My hope is that one day Pandas solves the speed problem with growing DataFrames and this package becomes obsolete.

## Python Version

Raccoon required Python 2.7 or 3.3 or greater to run because it utilizes “yield from” which was introduced in 3.3

## Helper scripts

There is helper function to generate these docs from the source code. On windows cd into the docs directory and execute make\_docs.bat from the command line. To run the test coverage report run the coverage.sh script.

## Change Log

### 1.0.1

- Added `isin()` method

### 1.0.2

- Fixed several small bugs
- Added iterators: `iterrows()` and `itertuples()`

### 1.1.0

- Multiple bug fixes
- Speed improvements
- Removed using slices to set values which did not work
- Added sorted functionality

### 1.1.1

- Multiple bug fixes
- The `index` and `columns` data type follow the `use_blist` parameter
- Added `set_locations()` and `get_locations()` methods
- Added `as_dict()` parameter to `get_columns()`

### **1.1.2**

- Changed the default for use\_blist to False on object initialization

### **1.1.3**

- Added append\_row() method

### **1.1.4**

- Added get\_entire\_column() method and changed get() to use that when asking for only a single column

### **1.1.5**

- Bug fix where df[[columns]] would return wrong results with the column names not matching the correct column data

### **1.1.6**

- Added index parameter to iterrows() and itertuples()

### **1.1.7**

- Added reset\_index() method

### **1.1.8**

- Added methods to serialize and deserialize to JSON

### **1.1.9 (3/7/17)**

- Fixed the from\_json() for multi-index DataFrames

### **1.2 (3/8/17)**

- to\_json() will convert any non-serializable object to string representation
- Move to new version numbering scheme

### **1.2.1 (3/8/17)**

- bug fix from\_json() to work with empty DataFrames



### 1.2.2 (3/10/17)

- Added the ability to pass a function and arguments to the `assert_frame_equal` function to use when comparing data

### 1.3 (3/26/17)

- Added `Python2.7` support thanks to [tonycpsu](#)
- Fixed some logic `set_column` index check <https://github.com/rsheftel/raccoon/issues/5>
- Changed method `.print()` to `.show()` for Python2 compliance

#### 1.3.1 (3/30/17)

- Added reverse argument to `sort_columns()`

#### 1.3.2 (3/31/17)

- Added key argument to `sort_columns()`

#### 1.3.3 (4/9/17)

- Moved `from_json()` to be a class method. This breaks previous API

#### 1.3.4 (4/12/17)

- Added new `get_location()` method
- The `get()` method can now take `as_dict` parameter to pass to `get_columns`

#### 1.3.5 (4/22/17)

- Added new `get_index()` method
- Several speed up improvements

### 2.0.0 (5/1/17)

This is a major release that adds the new Series classes but importantly breaks the DataFrame API by renaming the “sorted” argument with “sort” and all associated properties and setters. This is to remove the naming conflict with the builtin sorted function

- Added new Series class
- Added new ViewSeries class
- Fix performance bug in the `select_index()` function in DataFrame
- Change sorted argument on DataFrame to sort
- Change sorted DataFrame property and setters to sort

## 2.1.0 (5/12/17)

Another potentially backwards incompatible change by making the `.index` properties to be a view and no longer a copy.

DataFrame

- Changes the `DataFrame.index` to return a view and not a copy.
- New `get_slice()` method for sorted DataFrames
- Changed `[]` on sort DataFrames to use `get_slice()` on slices
- New `set_location()` method for DataFrame
- New `append_rows()` method for DataFrame

Series

- Changed the `Series.data` and `Series.index` properties to return a view and not a copy
- New `get_slice()` method for sorted Series
- New `set_location()` method
- New `append_rows()` method for Series

## 2.1.1 (5/15/17)

- Added `columns=None` default to `get_column()` on DataFrame
- Fix bug in `get_slice` for empty DataFrames
- Fix bug in DataFrame `append` for empty DataFrames

## 2.1.2 (5/20/17)

- Added `delete_all_rows()` to DataFrame

## 2.1.3 (9/5/17)

- Added `from_series()` to `ViewSeries` to create a `ViewSeries` from a `Series`

## 2.1.4 (9/12/17)

- Made dataframe and all Series `__slots__` classes. This should reduce memory footprint and increase speed ~5%

## raccoon

### raccoon package

#### Submodules

#### raccoon.dataframe module

DataFrame class

```
class raccoon.dataframe.DataFrame (data=None, columns=None, index=None, index_name='index', use_blist=False, sort=None)
```

Bases: object

DataFrame class. The raccoon DataFrame implements a simplified version of the pandas DataFrame with the key objective difference that the raccoon DataFrame is meant for use cases where the size of the DataFrame rows is expanding frequently. This is known to be slow with Pandas due to the use of numpy as the underlying data structure. Raccoon uses BList as the underlying data structure which is quick to expand and grow the size. The DataFrame can be designated as sort, in which case the rows will be sort by index on construction, and then any addition of a new row will insert it into the DataFrame so that the index remains sort.

#### Parameters

- **data** – (optional) dictionary of lists. The keys of the dictionary will be used for the column names and the lists will be used for the column data.
- **columns** – (optional) list of column names that will define the order
- **index** – (optional) list of index values. If None then the index will be integers starting with zero
- **index\_name** – (optional) name for the index. Default is “index”
- **use\_blist** – if True then use blist() as the underlying data structure, if False use standard list()

- **sort** – if True then DataFrame will keep the index sort. If True all index values must be of same type

**add** (*left\_column, right\_column, indexes=None*)

Math helper method that adds element-wise two columns. If indexes are not None then will only perform the math on that sub-set of the columns.

**Parameters**

- **left\_column** – first column name
- **right\_column** – second column name
- **indexes** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame

**Returns** list

**append** (*data\_frame*)

Append another DataFrame to this DataFrame. If the new data\_frame has columns that are not in the current DataFrame then new columns will be created. All of the indexes in the data\_frame must be different from the current indexes or will raise an error.

**Parameters** **data\_frame** – DataFrame to append

**Returns** nothing

**append\_row** (*index, values, new\_cols=True*)

Appends a row of values to the end of the data. If there are new columns in the values and new\_cols is True they will be added. Be very careful with this function as for sort DataFrames it will not enforce sort order. Use this only for speed when needed, be careful.

**Parameters**

- **index** – value of the index
- **values** – dictionary of values
- **new\_cols** – if True add new columns in values, if False ignore

**Returns** nothing

**append\_rows** (*indexes, values, new\_cols=True*)

Appends rows of values to the end of the data. If there are new columns in the values and new\_cols is True they will be added. Be very careful with this function as for sort DataFrames it will not enforce sort order. Use this only for speed when needed, be careful.

**Parameters**

- **indexes** – list of indexes
- **values** – dictionary of values where the key is the column name and the value is a list
- **new\_cols** – if True add new columns in values, if False ignore

**Returns** nothing

**blist**

**columns**

**data**

**delete\_all\_rows** ()

Deletes the contents of all rows in the DataFrame. This function is faster than delete\_rows() to remove

all information, and at the same time it keeps the container lists for the columns and index so if there is another object that references this DataFrame, like a ViewSeries, the reference remains in tact.

**Returns** nothing

**delete\_columns** (*columns*)

Delete columns from the DataFrame

**Parameters** **columns** – list of columns to delete

**Returns** nothing

**delete\_rows** (*indexes*)

Delete rows from the DataFrame

**Parameters** **indexes** – either a list of values or list of booleans for the rows to delete

**Returns** nothing

**divide** (*left\_column, right\_column, indexes=None*)

Math helper method that divides element-wise two columns. If indexes are not None then will only perform the math on that sub-set of the columns.

**Parameters**

- **left\_column** – column name of dividend
- **right\_column** – column name of divisor
- **indexes** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame

**Returns** list

**equality** (*column, indexes=None, value=None*)

Math helper method. Given a column and optional indexes will return a list of booleans on the equality of the value for that index in the DataFrame to the value parameter.

**Parameters**

- **column** – column name to compare
- **indexes** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame
- **value** – value to compare

**Returns** list of booleans

**classmethod from\_json** (*json\_string*)

Creates and return a DataFrame from a JSON of the type created by to\_json

**Parameters** **json\_string** – JSON

**Returns** DataFrame

**get** (*indexes=None, columns=None, as\_list=False, as\_dict=False*)

Given indexes and columns will return a sub-set of the DataFrame. This method will direct to the below methods based on what types are passed in for the indexes and columns. The type of the return is determined by the types of the parameters.

**Parameters**

- **indexes** – index value, list of index values, or a list of booleans. If None then all indexes are used
- **columns** – column name or list of column names. If None then all columns are used

- **as\_list** – if True then return the values as a list, if False return a DataFrame. This is only used if the get is for a single column
- **as\_dict** – if True then return the values as a dictionary, if False return a DataFrame. This is only used if the get is for a single row

**Returns** either DataFrame, list, dict or single value. The return is a shallow copy

**get\_cell** (*index, column*)

For a single index and column value return the value of the cell

**Parameters**

- **index** – index value
- **column** – column name

**Returns** value

**get\_columns** (*index, columns=None, as\_dict=False*)

For a single index and list of column names return a DataFrame of the values in that index as either a dict or a DataFrame

**Parameters**

- **index** – single index value
- **columns** – list of column names
- **as\_dict** – if True then return the result as a dictionary

**Returns** DataFrame or dictionary

**get\_entire\_column** (*column, as\_list=False*)

Shortcut method to retrieve a single column all rows. Since this is a common use case this method will be faster than the more general method.

**Parameters**

- **column** – single column name
- **as\_list** – if True return a list, if False return DataFrame

**Returns** DataFrame is as\_list if False, a list if as\_list is True

**get\_location** (*location, columns=None, as\_dict=False, index=True*)

For an index location and list of columns return a DataFrame of the values. This is optimized for speed because it does not need to lookup the index location with a search. Also can accept relative indexing from the end of the DataFrame in standard python notation [-3, -2, -1]

**Parameters**

- **location** – index location in standard python form of positive or negative number
- **columns** – list of columns, or None to include all columns
- **as\_dict** – if True then return a dictionary
- **index** – if True then include the index in the dictionary if as\_dict=True

**Returns** DataFrame or dictionary

**get\_locations** (*locations, columns=None, \*\*kwargs*)

For list of locations and list of columns return a DataFrame of the values.

**Parameters**

- **locations** – list of index locations

- **columns** – list of column names
- **kwargs** – will pass along these parameters to the `get()` method

**Returns** DataFrame

**get\_matrix** (*indexes, columns*)

For a list of indexes and list of columns return a DataFrame of the values.

**Parameters**

- **indexes** – either a list of index values or a list of booleans with same length as all indexes
- **columns** – list of column names

**Returns** DataFrame

**get\_rows** (*indexes, column, as\_list=False*)

For a list of indexes and a single column name return the values of the indexes in that column.

**Parameters**

- **indexes** – either a list of index values or a list of booleans with same length as all indexes
- **column** – single column name
- **as\_list** – if True return a list, if False return DataFrame

**Returns** DataFrame is `as_list` if False, a list if `as_list` is True

**get\_slice** (*start\_index=None, stop\_index=None, columns=None, as\_dict=False*)

For sorted DataFrames will return either a DataFrame or dict of all of the rows where the index is greater than or equal to the `start_index` if provided and less than or equal to the `stop_index` if provided. If either the start or stop index is None then will include from the first or last element, similar to standard python slice of `[:5]` or `[-5:]`. Both end points are considered inclusive.

**Parameters**

- **start\_index** – lowest index value to include, or None to start from the first row
- **stop\_index** – highest index value to include, or None to end at the last row
- **columns** – list of column names to include, or None for all columns
- **as\_dict** – if True then return a tuple of (list of index, dict of column names: list data values)

**Returns** DataFrame or tuple

**head** (*rows*)

Return a DataFrame of the first N rows

**Parameters** **rows** – number of rows

**Returns** DataFrame

**index**

Return a view of the index as a list. Because this is a view any change to the return list from this method will corrupt the DataFrame.

**Returns** list

**index\_name**

**isin** (*column, compare\_list*)

Returns a boolean list where each elements is whether that element in the column is in the `compare_list`.

**Parameters**

- **column** – single column name, does not work for multiple columns
- **compare\_list** – list of items to compare to

**Returns** list of booleans

**iterrows** (*index=True*)

Iterates over DataFrame rows as dictionary of the values. The index will be included.

**Parameters** **index** – if True include the index in the results

**Returns** dictionary

**itertuples** (*index=True, name='Raccoon'*)

Iterates over DataFrame rows as tuple of the values.

**Parameters**

- **index** – if True then include the index
- **name** – name of the namedtuple

**Returns** namedtuple

**multiply** (*left\_column, right\_column, indexes=None*)

Math helper method that multiplies element-wise two columns. If indexes are not None then will only perform the math on that sub-set of the columns.

**Parameters**

- **left\_column** – first column name
- **right\_column** – second column name
- **indexes** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame

**Returns** list

**rename\_columns** (*rename\_dict*)

Renames the columns

**Parameters** **rename\_dict** – dict where the keys are the current column names and the values are the new names

**Returns** nothing

**reset\_index** (*drop=False*)

Resets the index of the DataFrame to simple integer list and the index name to 'index'. If drop is True then the existing index is dropped, if drop is False then the current index is made a column in the DataFrame with the index name the name of the column. If the index is a tuple multi-index then each element of the tuple is converted into a separate column. If the index name was 'index' then the column name will be 'index\_0' to not conflict on print().

**Parameters** **drop** – if True then the current index is dropped, if False then index converted to columns

**Returns** nothing

**select\_index** (*compare, result='boolean'*)

Finds the elements in the index that match the compare parameter and returns either a list of the values that match, of a boolean list the length of the index with True to each index that matches. If the indexes are tuples then the compare is a tuple where None in any field of the tuple will be treated as "\*" and match all values.

**Parameters**



- **compare** – value to compare as a singleton or tuple
- **result** – ‘boolean’ = returns a list of booleans, ‘value’ = returns a list of index values that match

**Returns** list of booleans or values

**set** (*indexes=None, columns=None, values=None*)

Given indexes and columns will set a sub-set of the DataFrame to the values provided. This method will direct to the below methods based on what types are passed in for the indexes and columns. If the indexes or columns contains values not in the DataFrame then new rows or columns will be added.

#### Parameters

- **indexes** – indexes value, list of indexes values, or a list of booleans. If None then all indexes are used
- **columns** – columns name, if None then all columns are used. Currently can only handle a single column or all columns
- **values** – value or list of values to set (index, column) to. If setting just a single row, then must be a dict where the keys are the column names. If a list then must be the same length as the indexes parameter, if indexes=None, then must be the same and length of DataFrame

**Returns** nothing

**set\_cell** (*index, column, value*)

Sets the value of a single cell. If the index and/or column is not in the current index/columns then a new index and/or column will be created.

#### Parameters

- **index** – index value
- **column** – column name
- **value** – value to set

**Returns** nothing

**set\_column** (*index=None, column=None, values=None*)

Set a column to a single value or list of values. If any of the index values are not in the current indexes then a new row will be created.

#### Parameters

- **index** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame
- **column** – column name
- **values** – either a single value or a list. The list must be the same length as the index list if the index list is values, or the length of the True values in the index list if the index list is booleans

**Returns** nothing

**set\_location** (*location, values, missing\_to\_none=False*)

Sets the column values, as given by the keys of the values dict, for the row at location to the values of the values dict. If missing\_to\_none is False then columns not in the values dict will be left unchanged, if it is True then they are set to None. This method does not add new columns and raises an error.

#### Parameters

- **location** – location
- **values** – dict of column names as keys and the value as the value to set the row for that column to
- **missing\_to\_none** – if True set any column missing in the values to None, otherwise leave unchanged

**Returns** nothing

**set\_locations** (*locations, column, values*)

For a list of locations and a column set the values.

**Parameters**

- **locations** – list of index locations
- **column** – column name
- **values** – list of values or a single value

**Returns** nothing

**set\_row** (*index, values*)

Sets the values of the columns in a single row.

**Parameters**

- **index** – index value
- **values** – dict with the keys as the column names and the values what to set that column to

**Returns** nothing

**show** (*index=True, \*\*kwargs*)

Print the contents of the DataFrame. This method uses the tabulate function from the tabulate package. Use the kwargs to pass along any arguments to the tabulate function.

**Parameters**

- **index** – If True then include the indexes as a column in the output, if False ignore the index
- **kwargs** – Parameters to pass along to the tabulate function

**Returns** output of the tabulate function

**sort**

**sort\_columns** (*column, key=None, reverse=False*)

Sort the DataFrame by one of the columns. The sort modifies the DataFrame inplace. The key and reverse parameters have the same meaning as for the built-in sort() function.

**Parameters**

- **column** – column name to use for the sort
- **key** – if not None then a function of one argument that is used to extract a comparison key from each list element
- **reverse** – if True then the list elements are sort as if each comparison were reversed.

**Returns** nothing

**sort\_index** ()

Sort the DataFrame by the index. The sort modifies the DataFrame inplace

**Returns** nothing

**subtract** (*left\_column, right\_column, indexes=None*)

Math helper method that subtracts element-wise two columns. If indexes are not None then will only perform the math on that sub-set of the columns.

**Parameters**

- **left\_column** – first column name
- **right\_column** – name of column to subtract from the left\_column
- **indexes** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame

**Returns** list

**tail** (*rows*)

Return a DataFrame of the last N rows

**Parameters** **rows** – number of rows

**Returns** DataFrame

**to\_dict** (*index=True, ordered=False*)

Returns a dict where the keys are the column names and the values are lists of the values for that column.

**Parameters**

- **index** – If True then include the index in the dict with the index\_name as the key
- **ordered** – If True then return an OrderedDict() to preserve the order of the columns in the DataFrame

**Returns** dict or OrderedDict()

**to\_json** ()

Returns a JSON of the entire DataFrame that can be reconstructed back with `raccoon.from_json(input)`. Any object that cannot be serialized will be replaced with the representation of the object using `repr()`. In that instance the DataFrame will have a string representation in place of the object and will not reconstruct exactly.

**Returns** json string

**to\_list** ()

For a single column DataFrame returns a list of the values. Raises error if more than one column.

**Returns** list

**validate\_integrity** ()

Validate the integrity of the DataFrame. This checks that the indexes, column names and internal data are not corrupted. Will raise an error if there is a problem.

**Returns** nothing

## raccoon.series module

Series class

**class** `raccoon.series.Series` (*data=None, index=None, data\_name='value', index\_name='index', use\_blist=False, sort=None*)

Bases: `raccoon.series.SeriesBase`

Series class. The raccoon Series implements a simplified version of the pandas Series with the key objective difference that the raccoon Series is meant for use cases where the size of the Series is expanding frequently. This is known to be slow with Pandas due to the use of numpy as the underlying data structure. The Series can be designated as sort, in which case the rows will be sort by index on construction, and then any addition of a new row will insert it into the Series so that the index remains sort.

### Parameters

- **data** – (optional) list of values.
- **index** – (optional) list of index values. If None then the index will be integers starting with zero
- **data\_name** – (optional) name of the data column, or will default to ‘value’
- **index\_name** – (optional) name for the index. Default is “index”
- **use\_blist** – if True then use blist() as the underlying data structure, if False use standard list()
- **sort** – if True then Series will keep the index sort. If True all index values must be of same type

### **append\_row** (*index, value*)

Appends a row of value to the end of the data. Be very careful with this function as for sorted Series it will not enforce sort order. Use this only for speed when needed, be careful.

### Parameters

- **index** – index
- **value** – value

**Returns** nothing

### **append\_rows** (*indexes, values*)

Appends values to the end of the data. Be very careful with this function as for sort DataFrames it will not enforce sort order. Use this only for speed when needed, be careful.

### Parameters

- **indexes** – list of indexes to append
- **values** – list of values to append

**Returns** nothing

### **blist**

### **data**

### **delete** (*indexes*)

Delete rows from the DataFrame

**Parameters** **indexes** – either a list of values or list of booleans for the rows to delete

**Returns** nothing

### **index**

### **reset\_index** ()

Resets the index of the Series to simple integer list and the index name to ‘index’.

**Returns** nothing

**set** (*indexes, values=None*)

Given indexes will set a sub-set of the Series to the values provided. This method will direct to the below methods based on what types are passed in for the indexes. If the indexes contains values not in the Series then new rows or columns will be added.

**Parameters**

- **indexes** – indexes value, list of indexes values, or a list of booleans.
- **values** – value or list of values to set. If a list then must be the same length as the indexes parameter.

**Returns** nothing

**set\_cell** (*index, value*)

Sets the value of a single cell. If the index is not in the current index then a new index will be created.

**Parameters**

- **index** – index value
- **value** – value to set

**Returns** nothing

**set\_location** (*location, value*)

For a location set the value

**Parameters**

- **location** – location
- **value** – value

**Returns** nothing

**set\_locations** (*locations, values*)

For a list of locations set the values.

**Parameters**

- **locations** – list of index locations
- **values** – list of values or a single value

**Returns** nothing

**set\_rows** (*index, values=None*)

Set rows to a single value or list of values. If any of the index values are not in the current indexes then a new row will be created.

**Parameters**

- **index** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the Series
- **values** – either a single value or a list. The list must be the same length as the index list if the index list is values, or the length of the True values in the index list if the index list is booleans

**Returns** nothing

**sort**

**sort\_index** ()

Sort the Series by the index. The sort modifies the Series inplace

**Returns** nothing

**class** `raccoon.series.SeriesBase`

Bases: `object`

Base Series abstract base class that concrete implementations inherit from. Note that the `.data` and `.index` property methods in Series are views to the underlying data and not copies.

No specific parameters, those are defined in the child classed

**data**

**data\_name**

**equality** (*indexes=None, value=None*)

Math helper method. Given a column and optional indexes will return a list of booleans on the equality of the value for that index in the DataFrame to the value parameter.

**Parameters**

- **indexes** – list of index values or list of booleans. If a list of booleans then the list must be the same length as the DataFrame
- **value** – value to compare

**Returns** list of booleans

**get** (*indexes, as\_list=False*)

Given indexes will return a sub-set of the Series. This method will direct to the specific methods based on what types are passed in for the indexes. The type of the return is determined by the types of the parameters.

**Parameters**

- **indexes** – index value, list of index values, or a list of booleans.
- **as\_list** – if True then return the values as a list, if False return a Series.

**Returns** either Series, list, or single value. The return is a shallow copy

**get\_cell** (*index*)

For a single index and return the value

**Parameters** **index** – index value

**Returns** value

**get\_location** (*location*)

For an index location return a dict of the index and value. This is optimized for speed because it does not need to lookup the index location with a search. Also can accept relative indexing from the end of the Series in standard python notation [-3, -2, -1]

**Parameters** **location** – index location in standard python form of positive or negative number

**Returns** dictionary

**get\_locations** (*locations, as\_list=False*)

For list of locations return a Series or list of the values.

**Parameters**

- **locations** – list of index locations
- **as\_list** – True to return a list of values

**Returns** Series or list

**get\_rows** (*indexes, as\_list=False*)

For a list of indexes return the values of the indexes in that column.

**Parameters**

- **indexes** – either a list of index values or a list of booleans with same length as all indexes
- **as\_list** – if True return a list, if False return Series

**Returns** Series if `as_list` if False, a list if `as_list` is True

**get\_slice** (*start\_index=None, stop\_index=None, as\_list=False*)

For sorted Series will return either a Series or list of all of the rows where the index is greater than or equal to the `start_index` if provided and less than or equal to the `stop_index` if provided. If either the start or stop index is None then will include from the first or last element, similar to standard python slice of `[:5]` or `[:5]`. Both end points are considered inclusive.

**Parameters**

- **start\_index** – lowest index value to include, or None to start from the first row
- **stop\_index** – highest index value to include, or None to end at the last row
- **as\_list** – if True then return a list of the indexes and values

**Returns** Series or tuple of (index list, values list)

**head** (*rows*)

Return a Series of the first N rows

**Parameters** **rows** – number of rows

**Returns** Series

**index**

**index\_name**

**isin** (*compare\_list*)

Returns a boolean list where each elements is whether that element in the column is in the `compare_list`.

**Parameters** **compare\_list** – list of items to compare to

**Returns** list of booleans

**select\_index** (*compare, result='boolean'*)

Finds the elements in the index that match the compare parameter and returns either a list of the values that match, or a boolean list the length of the index with True to each index that matches. If the indexes are tuples then the compare is a tuple where None in any field of the tuple will be treated as "\*" and match all values.

**Parameters**

- **compare** – value to compare as a singleton or tuple
- **result** – 'boolean' = returns a list of booleans, 'value' = returns a list of index values that match

**Returns** list of booleans or values

**show** (*index=True, \*\*kwargs*)

Print the contents of the Series. This method uses the `tabulate` function from the `tabulate` package. Use the `kwargs` to pass along any arguments to the `tabulate` function.

**Parameters**

- **index** – If True then include the indexes as a column in the output, if False ignore the index
- **kwargs** – Parameters to pass along to the tabulate function

**Returns** output of the tabulate function

**sort**

**tail** (*rows*)

Return a Series of the last N rows

**Parameters** **rows** – number of rows

**Returns** Series

**to\_dict** (*index=True, ordered=False*)

Returns a dict where the keys are the data and index names and the values are list of the data and index.

**Parameters**

- **index** – If True then include the index in the dict with the `index_name` as the key
- **ordered** – If True then return an `OrderedDict()` to preserve the order of the columns in the Series

**Returns** dict or `OrderedDict()`

**validate\_integrity** ()

Validate the integrity of the Series. This checks that the indexes, column names and internal data are not corrupted. Will raise an error if there is a problem.

**Returns** nothing

**class** `raccoon.series.ViewSeries` (*data=None, index=None, data\_name='value', index\_name='index', sort=False, offset=0*)

Bases: `raccoon.series.SeriesBase`

`ViewSeries` class. The `raccoon ViewSeries` implements a view only version of the `Series` object with the key objective difference that the `raccoon ViewSeries` is meant for view only use cases where the underlying index and data are modified elsewhere or static. Use this for a view into a single column of a `DataFrame`.

**Parameters**

- **data** – (optional) list of values.
- **index** – (optional) list of index values. If `None` then the index will be integers starting with zero
- **data\_name** – (optional) name of the data column, or will default to 'value'
- **index\_name** – (optional) name for the index. Default is "index"
- **sort** – if True then assumes the index is sorted for faster set/get operations
- **offset** – integer to add to location to transform to standard python list location index

**data**

**classmethod** `from_dataframe` (*dataframe, column, offset=0*)

Creates and return a `Series` from a `DataFrame` and specific column

**Parameters**

- **dataframe** – `raccoon DataFrame`
- **column** – column name



- **offset** – offset value must be provided as there is no equivalent for a DataFrame

**Returns** Series

**classmethod** `from_series` (*series*, *offset=0*)

Creates and return a Series from a Series

**Parameters**

- **series** – raccoon Series
- **offset** – offset value must be provided as there is no equivalent for a DataFrame

**Returns** Series

**index**

**offset**

**sort**

**value** (*indexes*, *int\_as\_index=False*)

Wrapper function for get. It will return a list, no index. If the indexes are integers it will be assumed that they are locations unless `int_as_index = True`. If the indexes are locations then they will be rotated to the left by offset number of locations.

**Parameters**

- **indexes** – integer location, single index, list of indexes or list of boolean
- **int\_as\_index** – if True then will treat int index values as indexes and not locations

**Returns** value or list of values

## raccoon.sort\_utils module

Utility functions for sorting and dealing with sorted Series and DataFrames

`raccoon.sort_utils.sorted_exists` (*values*, *x*)

For list, values, returns the insert position for item x and whether the item already exists in the list. This allows one function call to return either the index to overwrite an existing value in the list, or the index to insert a new item in the list and keep the list in sorted order.

**Parameters**

- **values** – list
- **x** – item

**Returns** (exists, index) tuple

`raccoon.sort_utils.sorted_index` (*values*, *x*)

For list, values, returns the index location of element x. If x does not exist will raise an error.

**Parameters**

- **values** – list
- **x** – item

**Returns** integer index

`raccoon.sort_utils.sorted_list_indexes` (*list\_to\_sort*, *key=None*, *reverse=False*)

Sorts a list but returns the order of the index values of the list for the sort and not the values themselves. For example is the list provided is ['b', 'a', 'c'] then the result will be [2, 1, 3]

### Parameters

- **list\_to\_sort** – list to sort
- **key** – if not None then a function of one argument that is used to extract a comparison key from each list element
- **reverse** – if True then the list elements are sorted as if each comparison were reversed.

**Returns** list of sorted index values

## raccoon.utils module

Raccoon utilities

`raccoon.utils.assert_frame_equal` (*left, right, data\_function=None, data\_args=None*)  
For unit testing equality of two DataFrames.

### Parameters

- **left** – first DataFrame
- **right** – second DataFrame
- **data\_function** – if provided will use this function to assert compare the df.data
- **data\_args** – arguments to pass to the data\_function

**Returns** nothing

`raccoon.utils.assert_series_equal` (*left, right, data\_function=None, data\_args=None*)  
For unit testing equality of two Series.

### Parameters

- **left** – first Series
- **right** – second Series
- **data\_function** – if provided will use this function to assert compare the df.data
- **data\_args** – arguments to pass to the data\_function

**Returns** nothing

## Module contents

## Example Usage for DataFrame

```
# remove comment to use latest development version
import sys; sys.path.insert(0, '../')
```

```
# import libraries
import raccoon as rc
```

## Initialize

```
# empty DataFrame
df = rc.DataFrame()
df
```

```
object id: 3013667704504
columns:
[]
data:
[]
index:
[]
```

```
# with columns and indexes but no data
df = rc.DataFrame(columns=['a', 'b', 'c'], index=[1, 2, 3])
df
```

```
object id: 3013667701144
columns:
['a', 'b', 'c']
data:
[[None, None, None], [None, None, None], [None, None, None]]
index:
[1, 2, 3]
```

```
# with data
df = rc.DataFrame(data={'a': [1, 2, 3], 'b': [4, 5, 6]}, index=[10, 11, 12], columns=[
→ 'a', 'b'])
df
```

```
object id: 3013667831992
columns:
['a', 'b']
data:
[[1, 2, 3], [4, 5, 6]]
index:
[10, 11, 12]
```

## Print

```
df.show()
```

```
  index  a  b
-----  -  -
     10   1  4
     11   2  5
     12   3  6
```

```
print(df)
```

```
  index  a  b
-----  -  -
     10   1  4
```

```
11  2  5
12  3  6
```

## Setters and Getters

```
# columns
df.columns
```

```
['a', 'b']
```

```
df.columns = ['first', 'second']
print(df)
```

```
  index  first  second
-----  -
10      1      4
11      2      5
12      3      6
```

```
# columns can be renamed with a dict()
df.rename_columns({'second': 'b', 'first': 'a'})
df.columns
```

```
['a', 'b']
```

```
# index
df.index
```

```
[10, 11, 12]
```

```
#indexes can be any non-repeating unique values
df.index = ['apple', 'pear', 7.7]
df.show()
```

```
index    a    b
-----  -
apple    1    4
pear     2    5
7.7      3    6
```

```
df.index = [10, 11, 12]
print(df)
```

```
  index  a  b
-----  -
10      1  4
11      2  5
12      3  6
```

```
# the index can also have a name, beault it is "index"
df.index_name
```

```
'index'
```

```
df.index_name = 'units'
df.index_name
```

```
'units'
```

```
# data is a shallow copy, be careful on how this is used
df.index_name = 'index'
df.data
```

```
[[1, 2, 3], [4, 5, 6]]
```

## Select Index

```
df.select_index(11)
```

```
[False, True, False]
```

## Set Values

```
# set a single cell
df.set(10, 'a', 100)
print(df)
```

index	a	b
10	100	4
11	2	5
12	3	6

```
# set a value outside current range creates a new row and/or column. Can also use [] ↵
↵ for setting
df[13, 'c'] = 9
df.show()
```

index	a	b	c
10	100	4	
11	2	5	
12	3	6	
13			9

```
# set column
df['b'] = 55
print(df)
```

index	a	b	c
10	100	55	

```

11    2    55
12    3    55
13           55    9

```

```

# set a subset of column
df[[10, 12], 'b'] = 66
print(df)

```

```

index    a    b    c
-----  -  -  -
10    100   66
11     2    55
12     3    66
13           55    9

```

```

# using boolean list
df.set([True, False, True, False], 'b', [88, 99])
print(df)

```

```

index    a    b    c
-----  -  -  -
10    100   88
11     2    55
12     3    99
13           55    9

```

```

# setting with slices
df[12:13, 'a'] = 33
print(df)

```

```

index    a    b    c
-----  -  -  -
10    100   88
11     2    55
12    33   99
13    33   55    9

```

```

df[10:12, 'c'] = [1, 2, 3]
print(df)

```

```

index    a    b    c
-----  -  -  -
10    100   88    1
11     2    55    2
12    33   99    3
13    33   55    9

```

```

# append a row, DANGEROUS as there is not validation checking, but can be used for_
↳ speed
df.append_row(14, {'a': 44, 'c': 100, 'd': 99})
print(df)

```

```

index    a    b    c    d
-----  -  -  -  -

```

```

10 100 88 1
11 2 55 2
12 33 99 3
13 33 55 9
14 44 100 99

```

```

# append rows, again use caution
df.append_rows([15, 16], {'a': [55, 56], 'd': [100,101]})
print(df)

```

```

index   a    b    c    d
-----  -
10  100  88   1
11   2  55   2
12  33  99   3
13  33  55   9
14  44  100  99
15  55      100
16  56      101

```

## Get Values

```

# get a single cell
df[10, 'a']

```

```
100
```

```

# get an entire column
df['c'].show()

```

```

index   c
-----  -
10     1
11     2
12     3
13     9
14  100
15
16

```

```

# get list of columns
df[['a', 'c']].show()

```

```

index   a    c
-----  -
10  100   1
11   2    2
12  33    3
13  33    9
14  44  100
15  55
16  56

```

```
# get subset of the index
df[[11, 12, 13], 'b'].show()
```

index	b
11	55
12	99
13	55

```
# get using slices
df[11:13, 'b'].show()
```

index	b
11	55
12	99
13	55

```
# get a matrix
df[10:11, ['a', 'c']].show()
```

index	a	c
10	100	1
11	2	2

```
# get a column, return as a list
df.get(columns='a', as_list=True)
```

```
[100, 2, 33, 33, 44, 55, 56]
```

```
# get a row and return as a dictionary
df.get_columns(index=13, columns=['a', 'b'], as_dict=True)
```

```
{'a': 33, 'b': 55, 'index': 13}
```

## Set and Get by Location

Locations are the index of the index, in other words the index locations from 0...len(index)

```
print(df.get_location(2))
```

index	a	b	c	d
12	33	99	3	

```
print(df.get_location(0, ['b', 'c'], as_dict=True))
```

```
{'b': 88, 'c': 1, 'index': 10}
```



```
df.get_location(-1).show()
```

```

index      a  b      c      d
-----
    16    56             101

```

```
df.get_locations(locations=[0, 2]).show()
```

```

index      a      b      c  d
-----
    10   100    88     1
    12    33    99     3

```

```
df.set_locations(locations=[0, 2], column='a', values=-9)
df.show()
```

```

index      a      b      c      d
-----
    10    -9    88     1
    11     2    55     2
    12    -9    99     3
    13    33    55     9
    14    44             100    99
    15    55             100
    16    56             101

```

## Head and Tail

```
df.head(2).show()
```

```

index      a      b      c  d
-----
    10    -9    88     1
    11     2    55     2

```

```
df.tail(2).show()
```

```

index      a  b      c      d
-----
    15    55             100
    16    56             101

```

## Delete columns and rows

```
df.delete_rows([10, 13])
print(df)
```

```

index      a      b      c  d
-----
    11     2    55     2

```

```
12  -9  99   3
14  44     100  99
15  55     100
16  56     101
```

```
df.delete_columns('b')
print(df)
```

```
index  a  c  d
-----  -  -  -
11     2  2
12    -9  3
14    44 100 99
15    55     100
16    56     101
```

## Convert

```
# return a dict
df.to_dict()
```

```
{'a': [2, -9, 44, 55, 56],
 'c': [2, 3, 100, None, None],
 'd': [None, None, 99, 100, 101],
 'index': [11, 12, 14, 15, 16]}
```

```
# exclude the index
df.to_dict(index=False)
```

```
{'a': [2, -9, 44, 55, 56],
 'c': [2, 3, 100, None, None],
 'd': [None, None, 99, 100, 101]}
```

```
# return an OrderedDict()
df.to_dict(ordered=True)
```

```
OrderedDict([('index', [11, 12, 14, 15, 16]),
             ('a', [2, -9, 44, 55, 56]),
             ('c', [2, 3, 100, None, None]),
             ('d', [None, None, 99, 100, 101])])
```

```
# return a list of just one column
df['c'].to_list()
```

```
[2, 3, 100, None, None]
```

```
# convert to JSON
string = df.to_json()
print(string)
```

```
{"data": {"a": [2, -9, 44, 55, 56], "c": [2, 3, 100, null, null], "d": [null, null, 99, 100, 101]}, "index": [11, 12, 14, 15, 16], "meta_data": {"index_name": "index", "columns": ["a", "c", "d"], "sort": false, "use_blist": false}}
```

```
# construct DataFrame from JSON
df_from_json = rc.DataFrame.from_json(string)
print(df_from_json)
```

index	a	c	d
11	2	2	
12	-9	3	
14	44	100	99
15	55		100
16	56		101

## Sort by Index and Column

```
df = rc.DataFrame({'a': [4, 3, 2, 1], 'b': [6, 7, 8, 9]}, index=[25, 24, 23, 22])
print(df)
```

index	a	b
25	4	6
24	3	7
23	2	8
22	1	9

```
# sort by index. Sorts are inplace
df.sort_index()
print(df)
```

index	a	b
22	1	9
23	2	8
24	3	7
25	4	6

```
# sort by column
df.sort_columns('b')
print(df)
```

index	a	b
25	4	6
24	3	7
23	2	8
22	1	9

```
# sort by column in reverse order
df.sort_columns('b', reverse=True)
print(df)
```

index	a	b
22	1	9
23	2	8
24	3	7
25	4	6

```
# sorting with a key function is available, see tests for examples
```

## Append

```
df1 = rc.DataFrame({'a': [1, 2], 'b': [5, 6]}, index=[1, 2])
df1.show()
```

index	a	b
1	1	5
2	2	6

```
df2 = rc.DataFrame({'b': [7, 8], 'c': [11, 12]}, index=[3, 4])
print(df2)
```

index	b	c
3	7	11
4	8	12

```
df1.append(df2)
print(df1)
```

index	a	b	c
1	1	5	
2	2	6	
3		7	11
4		8	12

## Math Methods

```
df = rc.DataFrame({'a': [1, 2, 3], 'b': [2, 8, 9]})
```

```
# test for equality
df.equality('a', value=3)
```

```
[False, False, True]
```

```
# all math methods can operate on a subset of the index
df.equality('b', indexes=[1, 2], value=2)
```

```
[False, False]
```

```
# add two columns
df.add('a', 'b')
```

```
[3, 10, 12]
```

```
# subtract
df.subtract('b', 'a')
```

```
[1, 6, 6]
```

```
# multiply
df.multiply('a', 'b', [0, 2])
```

```
[2, 27]
```

```
# divide
df.divide('b', 'a')
```

```
[2.0, 4.0, 3.0]
```

## Multi-Index

Raccoon does not have true hierarchical multi-index capabilities like Pandas, but attempts to mimic some of the capabilities with the use of tuples as the index. Raccoon does not provide any checking to make sure the indexes are all the same length or any other integrity checking.

```
tuples = [('a', 1, 3), ('a', 1, 4), ('a', 2, 3), ('b', 1, 4), ('b', 2, 1), ('b', 3, 3)]
df = rc.DataFrame({'a': [1, 2, 3, 4, 5, 6]}, index=tuples)
print(df)
```

```
index      a
-----  ---
('a', 1, 3)  1
('a', 1, 4)  2
('a', 2, 3)  3
('b', 1, 4)  4
('b', 2, 1)  5
('b', 3, 3)  6
```

The `select_index` method works with tuples by allowing the `*` to act as a wild card for matching.

```
compare = ('a', None, None)
df.select_index(compare)
```

```
[True, True, True, False, False, False]
```

```
compare = ('a', None, 3)
df.select_index(compare, 'boolean')
```

```
[True, False, True, False, False, False]
```

```
compare = (None, 2, None)
df.select_index(compare, 'value')
```

```
[('a', 2, 3), ('b', 2, 1)]
```

```
compare = (None, None, 3)
df.select_index(compare, 'value')
```

```
[('a', 1, 3), ('a', 2, 3), ('b', 3, 3)]
```

```
compare = (None, None, None)
df.select_index(compare)
```

```
[True, True, True, True, True, True]
```

## Reset Index

```
df = rc.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}, columns=['a', 'b'])
print(df)
```

index	a	b
0	1	4
1	2	5
2	3	6

```
df.reset_index()
df
```

```
object id: 3013666160880
columns:
['a', 'b', 'index_0']
data:
[[1, 2, 3], [4, 5, 6], [0, 1, 2]]
index:
[0, 1, 2]
```

```
df = rc.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}, columns=['a', 'b'], index=['x', 'y', 'z'], index_name='jelo')
print(df)
```

jelo	a	b
x	1	4
y	2	5
z	3	6

```
df.reset_index()
print(df)
```

```

index   a   b  jelo
-----  -  -  -----
0       1   4   x
1       2   5   y
2       3   6   z

```

```

df = rc.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}, columns=['a', 'b'],
                  index=[('a', 10, 'x'), ('b', 11, 'y'), ('c', 12, 'z')], index_
↳name=('melo', 'helo', 'gelo'))
print(df)

```

```

('melo', 'helo', 'gelo')    a   b
-----  -  -
('a', 10, 'x')              1   4
('b', 11, 'y')              2   5
('c', 12, 'z')              3   6

```

```

df.reset_index()
print(df)

```

```

index   a   b melo   helo  gelo
-----  -  -  -----  -----  -----
0       1   4   a      10    x
1       2   5   b      11    y
2       3   6   c      12    z

```

```

df = rc.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}, columns=['a', 'b'], index=['x', 'y
↳', 'z'], index_name='jelo')
print(df)

```

```

jelo    a   b
-----  -  -
x        1   4
y        2   5
z        3   6

```

```

df.reset_index(drop=True)
print(df)

```

```

index   a   b
-----  -  -
0       1   4
1       2   5
2       3   6

```

## Iterators

```

df = rc.DataFrame({'a': [1, 2, 'c'], 'b': [5, 6, 'd']}, index=[1, 2, 3])

```

```

for row in df.iterrows():
    print(row)

```

```
{'index': 1, 'a': 1, 'b': 5}
{'index': 2, 'a': 2, 'b': 6}
{'index': 3, 'a': 'c', 'b': 'd'}
```

```
for row in df.itertuples():
    print(row)
```

```
Raccoon(index=1, a=1, b=5)
Raccoon(index=2, a=2, b=6)
Raccoon(index=3, a='c', b='d')
```

## Sorted DataFrames

DataFrames will be set to sorted by default if no index is given at initialization. If an index is given at initialization then the parameter `sorted` must be set to `True`

```
df = rc.DataFrame({'a': [3, 5, 4], 'b': [6, 8, 7]}, index=[12, 15, 14], sort=True)
```

When `sorted=True` on initialization the data will be sorted by index to start

```
df.show()
```

index	a	b
12	3	6
14	4	7
15	5	8

```
df[16, 'b'] = 9
print(df)
```

index	a	b
12	3	6
14	4	7
15	5	8
16		9

```
df.set(indexes=13, values={'a': 3.5, 'b': 6.5})
print(df)
```

index	a	b
12	3	6
13	3.5	6.5
14	4	7
15	5	8
16		9

## List or BList

The underlying data structure can be either `blist` (default) or `list`



```
# Construct with blist=True, the default
df_blist = rc.DataFrame({'a': [1, 2, 3]}, index=[5, 6, 7], use_blist=True)
```

```
# see that the data structures are all blists
df_blist.data
```

```
blist([blist([1, 2, 3])])
```

```
df_blist.index
```

```
blist([5, 6, 7])
```

```
df_blist.columns
```

```
blist(['a'])
```

```
# now construct as blist = False and they are all lists
df_list = rc.DataFrame({'a': [1, 2, 3]}, index=[5, 6, 7], use_blist=False)
```

```
df_list.data
```

```
[[1, 2, 3]]
```

```
df_list.index
```

```
[5, 6, 7]
```

```
df_list.columns
```

```
['a']
```

## Example Usage for Series

```
# remove comment to use latest development version
import sys; sys.path.insert(0, '../')
```

```
# import libraries
import raccoon as rc
```

### Initialize

```
# empty DataFrame
srs = rc.Series()
srs
```

```
object id: 1891392163736
data:
[]
index:
[]
```

```
# with indexes but no data
srs = rc.Series(index=[1, 2, 3])
srs
```

```
object id: 1891392163568
data:
[None, None, None]
index:
[1, 2, 3]
```

```
# with data
srs = rc.Series(data=[4, 5, 6], index=[10, 11, 12])
srs
```

```
object id: 1891392217440
data:
[4, 5, 6]
index:
[10, 11, 12]
```

## Print

```
srs.show()
```

```
index  value
-----  -
    10     4
    11     5
    12     6
```

```
print(srs)
```

```
index  value
-----  -
    10     4
    11     5
    12     6
```

## Setters and Getters

```
# data_name
srs.data_name
```

```
'value'
```

```
srs.data_name = 'new_data'
print(srs)
```

index	new_data
10	4
11	5
12	6

```
# index
srs.index
```

```
[10, 11, 12]
```

```
#indexes can be any non-repeating unique values
srs.index = ['apple', 'pear', 7.7]
srs.show()
```

index	new_data
apple	4
pear	5
7.7	6

```
srs.index = [10, 11, 12]
print(srs)
```

index	new_data
10	4
11	5
12	6

```
# the index can also have a name, befault it is "index"
srs.index_name
```

```
'index'
```

```
srs.index_name = 'units'
srs.index_name
```

```
'units'
```

```
# data is a shallow copy, be careful on how this is used
srs.index_name = 'index'
srs.data
```

```
[4, 5, 6]
```

## Select Index

```
srs.select_index(11)
```

```
[False, True, False]
```

## Set Values

```
# set a single cell
srs.set(10, 100)
print(srs)
```

```
index  new_data
-----
    10      100
    11         5
    12         6
```

```
# set a value outside current range creates a new row. Can also use [] for setting
srs[13] = 9
srs.show()
```

```
index  new_data
-----
    10      100
    11         5
    12         6
    13         9
```

```
# set a subset of rows
srs[[10, 12]] = 66
print(srs)
```

```
index  new_data
-----
    10      66
    11         5
    12      66
    13         9
```

```
# using boolean list
srs.set([True, False, True, False], [88, 99])
print(srs)
```

```
index  new_data
-----
    10      88
    11         5
    12      99
    13         9
```

```
# setting with slices
srs[12:13] = 33
print(srs)
```

index	new_data
10	88
11	5
12	33
13	33

```
srs[10:12] = [1, 2, 3]
print(srs)
```

index	new_data
10	1
11	2
12	3
13	33

```
# set a location
srs.set_location(1, 22)
print(srs)
```

index	new_data
10	1
11	22
12	3
13	33

```
# set multiple locations
srs.set_locations([0, 2], [11, 27])
print(srs)
```

index	new_data
10	11
11	22
12	27
13	33

```
# append a row, DANGEROUS as there is not validation checking, but can be used for_
→speed
srs.append_row(14, 99)
print(srs)
```

index	new_data
10	11
11	22
12	27
13	33
14	99

```
# append multiple rows, again no sort check
srs.append_rows([15, 16], [100, 110])
print(srs)
```

index	new_data
10	11
11	22
12	27
13	33
14	99
15	100
16	110

### Get Values

```
# get a single cell
srs[10]
```

```
11
```

```
# get subset of the index
srs[[11, 12, 13]].show()
```

index	new_data
11	22
12	27
13	33

```
# get using slices
srs[11:13].show()
```

index	new_data
11	22
12	27
13	33

```
# return as a list
srs.get([11, 12, 13], as_list=True)
```

```
[22, 27, 33]
```

### Set and Get by Location

Locations are the index of the index, in other words the index locations from 0...len(index)

```
print(srs.get_location(2))
```

```
{'index': 12, 'new_data': 27}
```

```
srs.get_location(-1)
```

```
{'index': 16, 'new_data': 110}
```

```
srs.get_locations(locations=[0, 2]).show()
```

index	new_data
10	11
12	27

```
srs.get_locations(locations=[0, 2], as_list=True)
```

```
[11, 27]
```

```
srs.set_locations([-1, -2], values=[10, 9])
print(srs)
```

index	new_data
10	11
11	22
12	27
13	33
14	99
15	9
16	10

## Head and Tail

```
srs.head(2).show()
```

index	new_data
10	11
11	22

```
srs.tail(2).show()
```

index	new_data
15	9
16	10

## Delete rows

```
srs.delete([10, 13])
print(srs)
```

index	new_data
11	22
12	27
14	99
15	9
16	10

## Convert

```
# return a dict
srs.to_dict()
```

```
{'index': [11, 12, 14, 15, 16], 'new_data': [22, 27, 99, 9, 10]}
```

```
# exclude the index
srs.to_dict(index=False)
```

```
{'new_data': [22, 27, 99, 9, 10]}
```

```
# return an OrderedDict()
srs.to_dict(ordered=True)
```

```
OrderedDict([('index', [11, 12, 14, 15, 16]),
             ('new_data', [22, 27, 99, 9, 10])])
```

## Sort by Index

```
srs = rc.Series([6, 7, 8, 9], index=[25, 24, 23, 22])
print(srs)
```

index	value
25	6
24	7
23	8
22	9

```
# sort by index. Sorts are inplace
srs.sort_index()
print(srs)
```

index	value
22	9



```

23      8
24      7
25      6

```

## Math Methods

```
srs = rc.Series([1, 2, 3])
```

```
# test for equality
srs.equality(value=3)
```

```
[False, False, True]
```

```
# all math methods can operate on a subset of the index
srs.equality(indexes=[1, 2], value=2)
```

```
[True, False]
```

## Multi-Index

Raccoon does not have true hierarchical multi-index capabilities like Pandas, but attempts to mimic some of the capabilities with the use of tuples as the index. Raccoon does not provide any checking to make sure the indexes are all the same length or any other integrity checking.

```
tuples = [('a', 1, 3), ('a', 1, 4), ('a', 2, 3), ('b', 1, 4), ('b', 2, 1), ('b', 3, 3),
          ↪3])
srs = rc.Series([1, 2, 3, 4, 5, 6], index=tuples)
print(srs)
```

index	value
('a', 1, 3)	1
('a', 1, 4)	2
('a', 2, 3)	3
('b', 1, 4)	4
('b', 2, 1)	5
('b', 3, 3)	6

The `select_index` method works with tuples by allowing the `*` to act as a wild card for matching.

```
compare = ('a', None, None)
srs.select_index(compare)
```

```
[True, True, True, False, False, False]
```

```
compare = ('a', None, 3)
srs.select_index(compare, 'boolean')
```

```
[True, False, True, False, False, False]
```

```
compare = (None, 2, None)
srs.select_index(compare, 'value')
```

```
[('a', 2, 3), ('b', 2, 1)]
```

```
compare = (None, None, 3)
srs.select_index(compare, 'value')
```

```
[('a', 1, 3), ('a', 2, 3), ('b', 3, 3)]
```

```
compare = (None, None, None)
srs.select_index(compare)
```

```
[True, True, True, True, True, True]
```

## Reset Index

```
srs = rc.Series([1, 2, 3], index=[9, 10, 11])
print(srs)
```

```
index  value
-----  -
      9      1
     10      2
     11      3
```

```
srs.reset_index()
srs
```

```
object id: 1891392288752
data:
[1, 2, 3]
index:
[0, 1, 2]
```

```
srs = rc.Series([1, 2, 3], index=[9, 10, 11], index_name='new name')
print(srs)
```

```
new name  value
-----  -
         9      1
        10      2
        11      3
```

```
srs.reset_index()
print(srs)
```

```
index  value
-----  -
      0      1
      1      2
      2      3
```

## Sorted Series

Series will be set to sorted by default if no index is given at initialization. If an index is given at initialization then the parameter sorted must be set to True

```
srs = rc.Series([3, 5, 4], index=[12, 15, 14], sort=True)
```

When sorted=True on initialization the data will be sorted by index to start

```
srs.show()
```

index	value
12	3
14	4
15	5

```
srs[16] = 9
print(srs)
```

index	value
12	3
14	4
15	5
16	9

```
srs.set(indexes=13, values=3.5)
print(srs)
```

index	value
12	3
13	3.5
14	4
15	5
16	9

## List or BList

The underlying data structure can be either blist (default) or list

```
# Construct with blist=True, the default
srs_blist = rc.Series([1, 2, 3], index=[5, 6, 7], use_blist=True)
```

```
# see that the data structures are all blists
srs_blist.data
```

```
blist([1, 2, 3])
```

```
srs_blist.index
```

```
blist([5, 6, 7])
```

```
# now construct as blist = False and they are all lists  
srs_list = rc.Series([1, 2, 3], index=[5, 6, 7], use_blist=False)
```

```
srs_list.data
```

```
[1, 2, 3]
```

```
srs_list.index
```

```
[5, 6, 7]
```

## Convert to and from Pandas DataFrames

There are no built in methods for the conversions but these functions below should work in most basic instances.

```
import raccoon as rc  
import pandas as pd
```

### Raccoon to Pandas

```
def rc_to_pd(raccoon_dataframe):  
    """  
    Convert a raccoon dataframe to pandas dataframe  
  
    :param raccoon_dataframe: raccoon DataFrame  
    :return: pandas DataFrame  
    """  
    data_dict = raccoon_dataframe.to_dict(index=False)  
    return pd.DataFrame(data_dict, columns=raccoon_dataframe.columns, index=raccoon_  
↳ dataframe.index)
```

```
rc_df = rc.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}, columns=['a', 'b'], index=[7, 8, 9])  
↳ print(type(rc_df))  
print(rc_df)
```

```
<class 'raccoon.dataframe.DataFrame'>  
  index    a    b  
-----  ---  ---  
      7    1    4  
      8    2    5  
      9    3    6
```

```
pd_df = rc_to_pd(rc_df)  
print(type(pd_df))  
print(pd_df)
```

```
<class 'pandas.core.frame.DataFrame'>
  a  b
7  1  4
8  2  5
9  3  6
```

## Pandas to Raccoon

```
def pd_to_rc(pandas_dataframe):
    """
    Convert a pandas dataframe to raccoon dataframe

    :param pandas_dataframe: pandas DataFrame
    :return: raccoon DataFrame
    """

    columns = pandas_dataframe.columns.tolist()
    data = dict()
    pandas_data = pandas_dataframe.values.T.tolist()
    for i in range(len(columns)):
        data[columns[i]] = pandas_data[i]
    index = pandas_dataframe.index.tolist()
    index_name = pandas_dataframe.index.name
    index_name = 'index' if not index_name else index_name
    return rc.DataFrame(data=data, columns=columns, index=index, index_name=index_
↳name)
```

```
pd_df = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]}, index=[5, 6, 7], columns=['a',
↳'b'])
print(type(pd_df))
print(pd_df)
```

```
<class 'pandas.core.frame.DataFrame'>
  a  b
5  1  4
6  2  5
7  3  6
```

```
rc_df = pd_to_rc(pd_df)
print(type(rc_df))
print(rc_df)
```

```
<class 'raccoon.dataframe.DataFrame'>
  index  a  b
-----  -  -
      5  1  4
      6  2  5
      7  3  6
```

## Raccoon vs. Pandas speed test

Setup pythonpath, import libraries and initialized DataFrame to store results

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	

### Initialize 10,000 empty DataFrames

92.4 ms ± 6.36 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

2.74 s ± 234 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718

### Initialize 100 row X 100 col DataFrame()

87.8 μs ± 8.43 μs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

8.11 ms ± 779 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137

### Add 10,000 items in 1 column to empty DataFrame

43.5 ms ± 402 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)

16.2 s ± 1.53 s per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305

### Add 100 rows of 100 columns to empty DataFrame

9.32 ms ± 808 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)

184 ms ± 3.55 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766

## Append 10x10 DataFrame 1000 times

62.2 ms  $\pm$  7.32 ms per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

164 ms  $\pm$  5.16 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293

## Get

528 ms  $\pm$  64.3 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

783 ms  $\pm$  74.1 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881

36.6 ms  $\pm$  5.38 ms per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

313  $\mu$ s  $\pm$  2.39  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1000 loops each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293

get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888

465 ms ± 51.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

6 s ± 425 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531

815 ms ± 73.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

118 ms ± 10.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698

## Set

436 ms ± 59.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

1 s ± 78.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531



get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306

4.19 ms ± 748 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

12.7 ms ± 1.02 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306
set column all index	0.00376809479375936	0.011686656340490344	0.322427

269 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

22.7 s ± 244 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306
set column all index	0.00376809479375936	0.011686656340490344	0.322427
set column subset index	0.26396186901109786	22.454482046778423	0.0117554

56.3 ms ± 8.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

500 ms ± 80.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293

get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306
set column all index	0.00376809479375936	0.011686656340490344	0.322427
set column subset index	0.26396186901109786	22.454482046778423	0.0117554
set index all columns	0.05272717698682072	0.4621994576302768	0.114079

## Sort

12.4 ms  $\pm$  807  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

539  $\mu$ s  $\pm$  62.9  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1000 loops each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306
set column all index	0.00376809479375936	0.011686656340490344	0.322427
set column subset index	0.26396186901109786	22.454482046778423	0.0117554
set index all columns	0.05272717698682072	0.4621994576302768	0.114079
sort index	0.011655945561328736	0.0004977774624292124	23.416

## Iterators

24.1 ms  $\pm$  532  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

20.3 ms  $\pm$  599  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306

set column all index	0.00376809479375936	0.011686656340490344	0.322427
set column subset index	0.26396186901109786	22.454482046778423	0.0117554
set index all columns	0.05272717698682072	0.4621994576302768	0.114079
sort index	0.011655945561328736	0.0004977774624292124	23.416
iterate rows	0.02340574597695877	0.01948813583071569	1.20103

## Insert in the middle

26.4 ms  $\pm$  381  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

239 ms  $\pm$  2.81 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306
set column all index	0.00376809479375936	0.011686656340490344	0.322427
set column subset index	0.26396186901109786	22.454482046778423	0.0117554
set index all columns	0.05272717698682072	0.4621994576302768	0.114079
sort index	0.011655945561328736	0.0004977774624292124	23.416
iterate rows	0.02340574597695877	0.01948813583071569	1.20103
insert rows	0.025894068785544278	0.2348415172963314	0.110262

## Time Series Append

Simulate the recording of a stock on 1 minute intervals and appending to the DataFrame

114 ms  $\pm$  9.01 ms per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

27.2 s  $\pm$  4 s per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

index	raccoon	pandas	ratio
version	2.1.3	0.20.2	
initialize empty	0.08496840788127305	2.486505687206119	0.0341718
initialize <b>with</b> matrix	8.295801655311905e-05	0.007671599044494002	0.0108137
add rows one column	0.04288183311489533	14.87375424325954	0.00288305
add matrix	0.008299982997955908	0.1785839394495099	0.0464766
append	0.05763429718412851	0.15777540076405216	0.365293
get cell	0.4936867081052583	0.7425185427150893	0.664881
get column all index	0.032696135984224384	0.00030877898993577447	105.888
get column subset index	0.42664153398601457	5.714989510943553	0.0746531
get index all columns	0.7787561018819247	0.10775676991503928	7.22698
set cell	0.3987260515496587	0.9623946910782024	0.414306

set column all index	0.00376809479375936	0.011686656340490344	0.322427
set column subset index	0.26396186901109786	22.454482046778423	0.0117554
set index all columns	0.05272717698682072	0.4621994576302768	0.114079
sort index	0.011655945561328736	0.0004977774624292124	23.416
iterate rows	0.02340574597695877	0.01948813583071569	1.20103
insert rows	0.025894068785544278	0.2348415172963314	0.110262
time series	0.10705897210370949	23.24008671488832	0.00460665

## CHAPTER 4

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**r**

`raccoon`, [22](#)  
`raccoon.dataframe`, [7](#)  
`raccoon.series`, [15](#)  
`raccoon.sort_utils`, [21](#)  
`raccoon.utils`, [22](#)





**A**

add() (raccoon.dataframe.DataFrame method), 8  
 append() (raccoon.dataframe.DataFrame method), 8  
 append\_row() (raccoon.dataframe.DataFrame method), 8  
 append\_row() (raccoon.series.Series method), 16  
 append\_rows() (raccoon.dataframe.DataFrame method), 8  
 append\_rows() (raccoon.series.Series method), 16  
 assert\_frame\_equal() (in module raccoon.utils), 22  
 assert\_series\_equal() (in module raccoon.utils), 22

**B**

blist (raccoon.dataframe.DataFrame attribute), 8  
 blist (raccoon.series.Series attribute), 16

**C**

columns (raccoon.dataframe.DataFrame attribute), 8

**D**

data (raccoon.dataframe.DataFrame attribute), 8  
 data (raccoon.series.Series attribute), 16  
 data (raccoon.series.SeriesBase attribute), 18  
 data (raccoon.series.ViewSeries attribute), 20  
 data\_name (raccoon.series.SeriesBase attribute), 18  
 DataFrame (class in raccoon.dataframe), 7  
 delete() (raccoon.series.Series method), 16  
 delete\_all\_rows() (raccoon.dataframe.DataFrame method), 8  
 delete\_columns() (raccoon.dataframe.DataFrame method), 9  
 delete\_rows() (raccoon.dataframe.DataFrame method), 9  
 divide() (raccoon.dataframe.DataFrame method), 9

**E**

equality() (raccoon.dataframe.DataFrame method), 9  
 equality() (raccoon.series.SeriesBase method), 18

**F**

from\_dataframe() (raccoon.series.ViewSeries class method), 20

from\_json() (raccoon.dataframe.DataFrame class method), 9

from\_series() (raccoon.series.ViewSeries class method), 21

**G**

get() (raccoon.dataframe.DataFrame method), 9  
 get() (raccoon.series.SeriesBase method), 18  
 get\_cell() (raccoon.dataframe.DataFrame method), 10  
 get\_cell() (raccoon.series.SeriesBase method), 18  
 get\_columns() (raccoon.dataframe.DataFrame method), 10  
 get\_entire\_column() (raccoon.dataframe.DataFrame method), 10  
 get\_location() (raccoon.dataframe.DataFrame method), 10  
 get\_location() (raccoon.series.SeriesBase method), 18  
 get\_locations() (raccoon.dataframe.DataFrame method), 10  
 get\_locations() (raccoon.series.SeriesBase method), 18  
 get\_matrix() (raccoon.dataframe.DataFrame method), 11  
 get\_rows() (raccoon.dataframe.DataFrame method), 11  
 get\_rows() (raccoon.series.SeriesBase method), 18  
 get\_slice() (raccoon.dataframe.DataFrame method), 11  
 get\_slice() (raccoon.series.SeriesBase method), 19

**H**

head() (raccoon.dataframe.DataFrame method), 11  
 head() (raccoon.series.SeriesBase method), 19

**I**

index (raccoon.dataframe.DataFrame attribute), 11  
 index (raccoon.series.Series attribute), 16  
 index (raccoon.series.SeriesBase attribute), 19  
 index (raccoon.series.ViewSeries attribute), 21  
 index\_name (raccoon.dataframe.DataFrame attribute), 11  
 index\_name (raccoon.series.SeriesBase attribute), 19  
 isin() (raccoon.dataframe.DataFrame method), 11  
 isin() (raccoon.series.SeriesBase method), 19

iterrows() (raccoon.dataframe.DataFrame method), 12  
itertuples() (raccoon.dataframe.DataFrame method), 12

## M

multiply() (raccoon.dataframe.DataFrame method), 12

## O

offset (raccoon.series.ViewSeries attribute), 21

## R

raccoon (module), 22  
raccoon.dataframe (module), 7  
raccoon.series (module), 15  
raccoon.sort\_utils (module), 21  
raccoon.utils (module), 22  
rename\_columns() (raccoon.dataframe.DataFrame method), 12  
reset\_index() (raccoon.dataframe.DataFrame method), 12  
reset\_index() (raccoon.series.Series method), 16

## S

select\_index() (raccoon.dataframe.DataFrame method), 12  
select\_index() (raccoon.series.SeriesBase method), 19  
Series (class in raccoon.series), 15  
SeriesBase (class in raccoon.series), 18  
set() (raccoon.dataframe.DataFrame method), 13  
set() (raccoon.series.Series method), 16  
set\_cell() (raccoon.dataframe.DataFrame method), 13  
set\_cell() (raccoon.series.Series method), 17  
set\_column() (raccoon.dataframe.DataFrame method), 13  
set\_location() (raccoon.dataframe.DataFrame method), 13  
set\_location() (raccoon.series.Series method), 17  
set\_locations() (raccoon.dataframe.DataFrame method), 14  
set\_locations() (raccoon.series.Series method), 17  
set\_row() (raccoon.dataframe.DataFrame method), 14  
set\_rows() (raccoon.series.Series method), 17  
show() (raccoon.dataframe.DataFrame method), 14  
show() (raccoon.series.SeriesBase method), 19  
sort (raccoon.dataframe.DataFrame attribute), 14  
sort (raccoon.series.Series attribute), 17  
sort (raccoon.series.SeriesBase attribute), 20  
sort (raccoon.series.ViewSeries attribute), 21  
sort\_columns() (raccoon.dataframe.DataFrame method), 14  
sort\_index() (raccoon.dataframe.DataFrame method), 14  
sort\_index() (raccoon.series.Series method), 17  
sorted\_exists() (in module raccoon.sort\_utils), 21  
sorted\_index() (in module raccoon.sort\_utils), 21  
sorted\_list\_indexes() (in module raccoon.sort\_utils), 21  
subtract() (raccoon.dataframe.DataFrame method), 15

## T

tail() (raccoon.dataframe.DataFrame method), 15  
tail() (raccoon.series.SeriesBase method), 20  
to\_dict() (raccoon.dataframe.DataFrame method), 15  
to\_dict() (raccoon.series.SeriesBase method), 20  
to\_json() (raccoon.dataframe.DataFrame method), 15  
to\_list() (raccoon.dataframe.DataFrame method), 15

## V

validate\_integrity() (raccoon.dataframe.DataFrame method), 15  
validate\_integrity() (raccoon.series.SeriesBase method), 20  
value() (raccoon.series.ViewSeries method), 21  
ViewSeries (class in raccoon.series), 20