
Raccoon Documentation

Release 3.0.0

Ryan Sheftel

Aug 22, 2023

Contents

1	Documentation	3
2	Overview	5
3	Updates	7
4	Contents	13
5	Indices and tables	71
	Python Module Index	73
	Index	75

Python DataFrame with fast insert and appends

<http://raccoon.readthedocs.io/en/latest/>

1.1 Source location

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

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. Additionally Raccoon DataFrames and Series can be parametrized to be sorted so that additions to the DataFrame keep the index in sorted order to speed inserts and retrievals.

A simple speed comparison of Raccoon vs Pandas for typical functionality is located in the documentation.

2.1 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.

2.2 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.

2.3 Underlying Data Structure

Raccoon uses the standard built in lists as its default underlying data structure. There is an option on object construction to use any other drop-in replacement for lists. For example the fast blist package <http://stutzbachenterprises.com/blist/> could be used as a list replacement for the underlying data structure.

2.4 Why Raccoon?

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

2.5 Contributing

Contribution in the form of pull requests are welcome. Use pytest to run the test suite. Be sure any new additions come with accompanying tests.

2.6 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.

2.7 Python Version

Raccoon requires Python 3.4 or greater. Python 2.7 support was eliminated as of version 3.0. If you need to use raccoon with Python 2.7 use any version less than 3.0

2.8 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.

3.1 Change Log

3.1.1 1.0.1

- Added `isin()` method

3.1.2 1.0.2

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

3.1.3 1.1.0

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

3.1.4 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()`

3.1.5 1.1.2

- Changed the default for use_blist to False on object initialization

3.1.6 1.1.3

- Added append_row() method

3.1.7 1.1.4

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

3.1.8 1.1.5

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

3.1.9 1.1.6

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

3.1.10 1.1.7

- Added reset_index() method

3.1.11 1.1.8

- Added methods to serialize and deserialize to JSON

3.1.12 1.1.9 (3/7/17)

- Fixed the from_json() for multi-index DataFrames

3.1.13 1.2 (3/8/17)

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

3.1.14 1.2.1 (3/8/17)

- bug fix from_json() to work with empty DataFrames

3.1.15 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

3.1.16 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

3.1.17 1.3.1 (3/30/17)

- Added reverse argument to `sort_columns()`

3.1.18 1.3.2 (3/31/17)

- Added key argument to `sort_columns()`

3.1.19 1.3.3 (4/9/17)

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

3.1.20 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`

3.1.21 1.3.5 (4/22/17)

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

3.1.22 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

3.1.23 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

3.1.24 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

3.1.25 2.1.2 (5/20/17)

- Added `delete_all_rows()` to DataFrame

3.1.26 2.1.3 (9/5/17)

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

3.1.27 2.1.4 (9/12/17)

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

3.1.28 2.1.5 (12/30/17)

- `get_location()` method of DataFrame will now return single value if single column argument passed and not list

3.1.29 3.0.0 (10/25/19)

This is a major release with many breaking changes. If you are using Python 2.7 do not upgrade.

- Python 2.7 support is dropped. This and all future releases are Python3 only
- `.show()` method has been renamed `.print()` to be consistent with Python standards
- Major change to the API for using drop-in replacements like `blis` and removing `blis` as an installation requirement.

The refactoring was driven by two needs. The first was to consistently accommodate other drop-in list replacements. The second was that `blis` was no longer maintained and having it as a dependency for the entire `raccoon` package created difficulties with installation. Now the sole package dependency is `tabulate` and that is a pure python package.

What were the issues with blis?

- `blis` does not have a published wheel on PyPi which makes it a difficult requirement for most people to install
- the conda `blis` package does not support Python 3.7 on Windows
- **Because of the following error it will cease working in 3.8 if not resolved and there seems to be no active development:**
 - Using or importing the ABCs from ‘`collections`’ instead of from ‘`collections.abc`’ is deprecated, and in 3.8 it will stop working

Can I still use blis?

Yes the new refactoring allows any list drop-in replacement, including `blis`, to be used but just no longer makes `blis` an installation requirement.

4.1 raccoon

4.1.1 raccoon package

Submodules

raccoon.dataframe module

DataFrame class

```
class raccoon.dataframe.DataFrame(data=None, columns=None, index=None, index_name='index', sort=None, dropin=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 native lists, or any other provided drop-in replacement for lists, 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”

- **sort** – if True then DataFrame will keep the index sort. If True all index values must be of same type
- **dropin** – if supplied the drop-in replacement for list that will be used

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

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

dropin

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: str, dropin_func=None*)

Creates and return a DataFrame from a JSON of the type created by to_json.

If a dropin is in the meta data from the JSON, then the same dropin class must be provided here to allow construction as the dropin function cannot be stored with the JSON. If required use a pickle object for that.

Parameters

- **json_string** – JSON
- **dropin_func** – drop-in replacement for list that was used in the 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 either (1) list of columns return a DataFrame or dictionary of the values or (2) single column name and return the value of that cell. 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, single column name, 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 if columns is a list or value if columns is a single column name

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

print (*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

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

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 () → str

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.

If there is a dropin supplied then the output will have a string representation of the dropping func class in the meta data as the dropin function cannot be stored with the JSON.

Returns json string

to_list()

For a single column DataFrame returns a list of the values. Raises error if more then 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',
                             sort=None, dropin=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 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 native lists, or any other provided drop-in replacement for lists, as the underlying data structure which is quick to expand and grow the size. 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”
- **sort** – if True then Series will keep the index sort. If True all index values must be of same type
- **dropin** – if supplied the drop-in replacement for list that will be used

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

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

dropin

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: `abc.ABC`

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

print (*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

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

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`. There is no type checking of the data, so it is assumed the data type is list-style.

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

4.2 Example Usage for DataFrame

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

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

4.2.1 Initialize

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

```
object id: 2305959579080
columns:
[]
data:
[]
index:
[]
```

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

```
object id: 2305959578792
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: 2305959818248
columns:
['a', 'b']
data:
[[1, 2, 3], [4, 5, 6]]
index:
[10, 11, 12]
```

4.2.2 Print

```
df.print()
```

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

4.2.3 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.print()
```

```

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]]
```

4.2.4 Select Index

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

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

4.2.5 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.print()
```

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	

(continues on next page)

(continued from previous page)

```
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
```

4.2.6 Get Values

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

```
100
```

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

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

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

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'].print()
```

index	b
11	55
12	99
13	55

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

index	b
11	55
12	99
13	55

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

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}
```

4.2.7 Set and Get by Location

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

```
# get a single cell
df.get_location(2, 'a')
```

```
33
```

```
# get an entire row when the columns is None
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).print()
```

index	a	b	c	d
16	56			101

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

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

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

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

4.2.8 Head and Tail

```
df.head(2).print()
```

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

```
df.tail(2).print()
```

index	a	b	c	d
15	55			100
16	56			101

4.2.9 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

4.2.10 Convert

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

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



```
# 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, "dropin": null}}
```

```
# 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

4.2.11 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
```

4.2.12 Append

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

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

4.2.13 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]
```

4.2.14 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]
```

4.2.15 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: 2305960012584
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)
```

(<i>melo</i> , <i>helo</i> , <i>gelo</i>)	a	b
(<i>a</i> , 10, <i>x</i>)	1	4
(<i>b</i> , 11, <i>y</i>)	2	5
(<i>c</i> , 12, <i>z</i>)	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
```

4.2.16 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')
```

4.2.17 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.print()
```

```
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

4.3 Example Usage for Series

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

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

4.3.1 Initialize

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

```
object id: 1924106416536
data:
[]
index:
[]
```

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

```
object id: 1924104590056
data:
[None, None, None]
index:
[1, 2, 3]
```

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

```
object id: 1924106418216
data:
[4, 5, 6]
index:
[10, 11, 12]
```

4.3.2 Print

```
srs.print()
```

index	value
10	4
11	5
12	6

```
print(srs)
```

index	value
10	4
11	5
12	6

4.3.3 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.print()
```

```
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, beault 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]
```

4.3.4 Select Index

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

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

4.3.5 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.print()
```

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

4.3.6 Get Values

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

```
11
```

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

index	new_data
11	22
12	27
13	33

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

index	new_data
11	22
12	27
13	33

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

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

4.3.7 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]).print()
```

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

4.3.8 Head and Tail

```
srs.head(2).print()
```

index	new_data
10	11
11	22

```
srs.tail(2).print()
```

index	new_data
15	9
16	10

4.3.9 Delete rows

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

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

4.3.10 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])])
```

4.3.11 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

4.3.12 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]
```

4.3.13 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)]
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]
```

4.3.14 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: 1924106640744
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

4.3.15 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.print()
```

index	value
12	3

(continues on next page)

(continued from previous page)

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

4.4 Example Usage for Drop-in List Replacements

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

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

4.4.1 BList

The underlying data structure can be any drop-in replacement for list, in this example blist is used.

```
from blist import blist
```

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

```
# 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'])
```

```
# the dropin class  
df_blist.dropin
```

```
blist.blist
```

All the standard functionality works exactly the same

```
df_blist[6, 'a']
```

```
2
```

```
df_blist[8, 'b'] = 44  
print(df_blist)
```

```
  index    a    b  
-----  
      5     1  
      6     2  
      7     3  
      8    44
```

Works for Series as well

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

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

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

```
srs_blist.index
```

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

4.5 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.

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

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

4.5.1 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
```

4.5.2 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)
```

(continues on next page)

(continued from previous page)

```
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
```

4.6 Raccoon vs. Pandas speed test

Setup pythonpath, import libraries and initialized DataFrame to store results

```
# Use this statement to import the current development version  
import sys; sys.path.insert(0, '../')
```

```
from copy import deepcopy
```

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

4.6.1 Machine information

```
import platform  
print(platform.machine())  
print(platform.processor())  
print(platform.platform())  
print("python ", platform.python_version())
```

```
AMD64  
Intel64 Family 6 Model 142 Stepping 10, GenuineIntel  
Windows-10-10.0.18362-SP0  
python 3.7.4
```

4.6.2 Run the Speed Test

```
results = rc.DataFrame(columns=['raccoon', 'pandas', 'ratio'], sort=False)
```

```
def add_results(index):
    results[index, 'raccoon'] = res_rc.best
    results[index, 'pandas'] = res_pd.best
    results[index, 'ratio'] = res_rc.best / res_pd.best
```

```
results['version', 'raccoon'] = rc.__version__
results['version', 'pandas'] = pd.__version__
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	

4.6.3 Initialize 10,000 empty DataFrames

```
def init_rc():
    for x in range(10000):
        df = rc.DataFrame()

def init_pd():
    for x in range(10000):
        df = pd.DataFrame()
```

```
res_rc = %timeit -o init_rc()
```

```
88.7 ms ± 6.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o init_pd()
```

```
4.49 s ± 155 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('initialize empty')
```

```
results.print()
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638

4.6.4 Initialize 100 row X 100 col DataFrame()

```
data = dict()
for x in range(100):
    data['a' + str(x)] = list(range(100))
```

```
res_rc = %timeit -o df=rc.DataFrame(data=data, sort=False)
```

```
121 µs ± 7.92 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
res_pd = %timeit -o df=pd.DataFrame(data=data)
```

```
15.3 ms ± 279 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
add_results('initialize with matrix')
```

```
results.print()
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611

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

```
def one_col_add_rc():
    df = rc.DataFrame()
    for x in range(10000):
        df.set(x, 'a', x)

def one_col_add_pd():
    df = pd.DataFrame()
    for x in range(10000):
        df.at[x, 'a'] = x
```

```
res_rc = %timeit -o one_col_add_rc()
```

```
53.7 ms ± 2.67 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o one_col_add_pd()
```

```
17.1 s ± 193 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('add rows one column')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837

4.6.6 Add 100 rows of 100 columns to empty DataFrame

```
new_row = {'a' + str(x): x for x in range(100)}
columns = ['a' + str(x) for x in range(100)]
```

```
def matrix_add_rc():
    df = rc.DataFrame(columns=columns)
    for x in range(100):
        df.set(indexes=x, values=new_row)
```

```
def matrix_add_pd():
    df = pd.DataFrame(columns=columns)
    for x in range(100):
        df.loc[x] = new_row
```

```
res_rc = %timeit -o matrix_add_rc()
```

10 ms \pm 185 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

```
res_pd = %timeit -o matrix_add_pd()
```

296 ms \pm 4.94 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
add_results('add matrix')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251

4.6.7 Append 10x10 DataFrame 1000 times

```
def append_rc():
    grid = {'a' + str(x): [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] for x in range(10)}
    df = rc.DataFrame(data=deepcopy(grid), columns=list(grid.keys()))
    for x in range(100):
        index = [(y + 1) + (x + 1) * 10 for y in range(10)]
        new_grid = deepcopy(grid)
        new_df = rc.DataFrame(data=new_grid, columns=list(new_grid.keys()),
                               ↪index=index)
        df.append(new_df)

def append_pd():
    grid = {'a' + str(x): [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] for x in range(10)}
    df = pd.DataFrame(data=grid, columns=list(grid.keys()))
    for x in range(100):
        index = [(y + 1) + (x + 1) * 10 for y in range(10)]
```

(continues on next page)

(continued from previous page)

```

new_grid = deepcopy(grid)
new_df = pd.DataFrame(data=new_grid, columns=list(new_grid.keys()),
↳index=index)
df = df.append(new_df)

```

```
res_rc = %timeit -o append_rc()
```

```
77.5 ms ± 3.77 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o append_pd()
```

```
370 ms ± 19.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('append')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919

4.6.8 Get

```
# First create a 1000 row X 100 col matrix for the test. Index is [0...999]
```

```

col = [x for x in range(1000)]
grid = {'a' + str(x): col[:] for x in range(100)}

df_rc = rc.DataFrame(data=grid, columns=sorted(grid.keys()))
df_pd = pd.DataFrame(data=grid, columns=sorted(grid.keys()))

```

```

# get cell

def rc_get_cell():
    for c in df_rc.columns:
        for r in df_rc.index:
            x = df_rc.get(r, c)

def pd_get_cell():
    for c in df_pd.columns:
        for r in df_pd.index:
            x = df_pd.at[r, c]

```

```
res_rc = %timeit -o rc_get_cell()
```



```
718 ms ± 45.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
res_pd = %timeit -o pd_get_cell()
```

```
1.02 s ± 71.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('get cell')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553

```
# get column all index

def get_column_all_rc():
    for c in df_rc.columns:
        x = df_rc.get(columns=c)

def get_column_all_pd():
    for c in df_pd.columns:
        x = df_pd[c]
```

```
res_rc = %timeit -o get_column_all_rc()
```

```
42.8 ms ± 743 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o get_column_all_pd()
```

```
402 µs ± 24.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
add_results('get column all index')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18

```
# get subset of the index of the column

def get_column_subset_rc():
    for c in df_rc.columns:
        for r in range(100):
            rows = list(range(r*10, r*10 + 9))
            x = df_rc.get(indexes=rows, columns=c)

def get_column_subset_pd():
    for c in df_pd.columns:
        for r in range(100):
            rows = list(range(r*10, r*10 + 9))
            x = df_pd.loc[rows, c]
```

```
res_rc = %timeit -o get_column_subset_rc()
```

```
609 ms ± 62.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
res_pd = %timeit -o get_column_subset_pd()
```

```
7.1 s ± 40.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('get column subset index')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.867934699999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.5668462000003274	7.041264400000273	0.0805035

```
# get index all columns

def get_index_all_rc():
    for i in df_rc.index:
        x = df_rc.get(indexes=i)

def get_index_all_pd():
    for i in df_pd.index:
        x = df_pd.loc[i]
```

```
res_rc = %timeit -o get_index_all_rc()
```

```
1.07 s ± 27.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
res_pd = %timeit -o get_index_all_pd()
```

229 ms \pm 9.34 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
add_results('get index all columns')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.0504075700000015764	16.867934699999975	0.00298837
add matrix	0.0097863460000000777	0.288469300000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.56684620000003274	7.0412644000000273	0.0805035
get index all columns	1.03899180000000762	0.2065747999999985	5.02962

4.6.9 Set

```
# First create a 1000 row X 100 col matrix for the test. Index is [0...999]
```

```
col = [x for x in range(1000)]
grid = {'a' + str(x): col[:] for x in range(100)}

df_rc = rc.DataFrame(data=grid, columns=sorted(grid.keys()))
df_pd = pd.DataFrame(data=grid, columns=sorted(grid.keys()))
```

```
# set cell

def rc_set_cell():
    for c in df_rc.columns:
        for r in df_rc.index:
            df_rc.set(r, c, 99)

def pd_set_cell():
    for c in df_pd.columns:
        for r in df_pd.index:
            df_pd.at[r, c] = 99
```

```
res_rc = %timeit -o rc_set_cell()
```

578 ms \pm 33.6 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
res_pd = %timeit -o pd_set_cell()
```

1.36 s \pm 65.6 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
add_results('set cell')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.346767699999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.5668462000003274	7.041264400000273	0.0805035
get index all columns	1.0389918000000762	0.2065747999999985	5.02962
set cell	0.5282995000002302	1.26670009999998	0.417068

```
# set column all index

def set_column_all_rc():
    for c in df_rc.columns:
        x = df_rc.set(columns=c, values=99)

def set_column_all_pd():
    for c in df_pd.columns:
        x = df_pd[c] = 99
```

```
res_rc = %timeit -o set_column_all_rc()
```

```
5.84 ms ± 512 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
res_pd = %timeit -o set_column_all_pd()
```

```
17.2 ms ± 516 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
add_results('set column all index')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.346767699999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.5668462000003274	7.041264400000273	0.0805035
get index all columns	1.0389918000000762	0.2065747999999985	5.02962
set cell	0.5282995000002302	1.26670009999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036

```
# set subset of the index of the column

def set_column_subset_rc():
    for c in df_rc.columns:
        for r in range(100):
            rows = list(range(r*10, r*10 + 10))
            x = df_rc.set(indexes=rows, columns=c, values=list(range(10)))

def set_column_subset_pd():
    for c in df_pd.columns:
        for r in range(100):
            rows = list(range(r*10, r*10 + 10))
            x = df_pd.loc[rows, c] = list(range(10))
```

```
res_rc = %timeit -o set_column_subset_rc()
```

```
380 ms ± 7.21 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
res_pd = %timeit -o set_column_subset_pd()
```

```
59 s ± 732 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('set column subset index')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.867934699999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.56684620000003274	7.0412644000000273	0.0805035
get index all columns	1.0389918000000762	0.2065747999999985	5.02962
set cell	0.5282995000002302	1.266700099999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036
set column subset index	0.372898399999994	58.03955229999974	0.0064249

```
row = {x:x for x in grid.keys() }
```

```
# set index all columns

def set_index_all_rc():
    for i in df_rc.index:
        x = df_rc.set(indexes=i, values=row)

def set_index_all_pd():
    for i in df_pd.index:
        x = df_pd.loc[i] = row
```

```
res_rc = %timeit -o set_index_all_rc()
```

```
64.4 ms ± 513 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o set_index_all_pd()
```

```
1.41 s ± 15.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('set index all columns')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.0504075700000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.56684620000003274	7.0412644000000273	0.0805035
get index all columns	1.03899180000000762	0.2065747999999985	5.02962
set cell	0.52829950000002302	1.266700099999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036
set column subset index	0.372898399999994	58.03955229999974	0.0064249
set index all columns	0.06374265999997988	1.3900373000000131	0.0458568

4.6.10 Sort

```
# make a dataframe 1000x100 with index in reverse order
```

```
rev = list(reversed(range(1000)))
```

```
df_rc = rc.DataFrame(data=grid, index=rev)
```

```
df_pd = pd.DataFrame(grid, index=rev)
```

```
res_rc = %timeit -o df_rc.sort_index()
```

```
16 ms ± 953 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
res_pd = %timeit -o df_pd.sort_index()
```

```
859 µs ± 12.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
add_results('sort index')
```

```
print(results)
```

index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.346767699999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.5668462000003274	7.041264400000273	0.0805035
get index all columns	1.0389918000000762	0.2065747999999985	5.02962
set cell	0.5282995000002302	1.26670009999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036
set column subset index	0.37289839999994	58.03955229999974	0.0064249
set index all columns	0.06374265999997988	1.390037300000131	0.0458568
sort index	0.014900102999999944	0.0008343847000001006	17.8576

4.6.11 Iterators

```
# First create a 1000 row X 100 col matrix for the test. Index is [0...999]
```

```
col = [x for x in range(1000)]
grid = {'a' + str(x): col[:] for x in range(100)}
```

```
df_rc = rc.DataFrame(data=grid, columns=sorted(grid.keys()))
df_pd = pd.DataFrame(data=grid, columns=sorted(grid.keys()))
```

```
# iterate over the rows
```

```
def iter_rc():
    for row in df_rc.iterrows():
        x = row

def iter_pd():
    for row in df_pd.itertuples():
        x = row
```

```
res_rc = %timeit -o iter_rc()
```

```
27.2 ms ± 381 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o iter_pd()
```

```
34 ms ± 532 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
add_results('iterate rows')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----

(continues on next page)

(continued from previous page)

version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.5668462000003274	7.041264400000273	0.0805035
get index all columns	1.0389918000000762	0.2065747999999985	5.02962
set cell	0.5282995000002302	1.266700099999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036
set column subset index	0.372898399999994	58.03955229999974	0.0064249
set index all columns	0.06374265999997988	1.390037300000131	0.0458568
sort index	0.014900102999999944	0.0008343847000001006	17.8576
iterate rows	0.026519559999997	0.03318497000000207	0.799144

4.6.12 Insert in the middle

```
# First create a 500 row X 100 col matrix for the test. Index is [1, 3, 5, 7,...500]
↳every other
```

```
col = [x for x in range(1, 1000, 2)]
grid = {'a' + str(x): col[:] for x in range(100)}

df_rc = rc.DataFrame(data=grid, columns=sorted(grid.keys()), sort=True)
df_pd = pd.DataFrame(data=grid, columns=sorted(grid.keys()))
```

```
row = {x:x for x in grid.keys() }
```

```
# set index all columns

def insert_rows_rc():
    for i in range(0, 999, 2):
        x = df_rc.set(indexes=i, values=row)

def insert_rows_pd():
    for i in range(0, 999, 2):
        x = df_pd.loc[i] = row
```

```
res_rc = %timeit -o insert_rows_rc()
```

```
33 ms ± 3.08 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o insert_rows_pd()
```

```
723 ms ± 30.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('insert rows')
```

```
print(results)
```


index	raccoon	pandas	ratio
version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.3467676999999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.56684620000003274	7.0412644000000273	0.0805035
get index all columns	1.03899180000000762	0.2065747999999985	5.02962
set cell	0.52829950000002302	1.266700099999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036
set column subset index	0.372898399999994	58.03955229999974	0.0064249
set index all columns	0.06374265999997988	1.3900373000000131	0.0458568
sort index	0.014900102999999944	0.00083438470000001006	17.8576
iterate rows	0.026519559999997	0.03318497000000207	0.799144
insert rows	0.0306854700000025816	0.6862636999999268	0.0447138

4.6.13 Time Series Append

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

```
data_row = {'open': 100, 'high': 101, 'low': 99, 'close': 100.5, 'volume': 999}

dates = pd.date_range('2010-01-01 09:30:00', periods=10000, freq='1min')

def time_series_rc():
    ts = rc.DataFrame(columns=['open', 'high', 'low', 'close', 'volume'], index_name=
→ 'datetime', sort=True)
    for date in dates:
        ts.set_row(date, data_row)

def time_series_pd():
    ts = pd.DataFrame(columns=['open', 'high', 'low', 'close', 'volume'])
    for date in dates:
        ts.loc[date] = data_row
```

```
res_rc = %timeit -o time_series_rc()
```

```
134 ms ± 4.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
res_pd = %timeit -o time_series_pd()
```

```
29.4 s ± 124 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
add_results('time series')
```

```
print(results)
```

index	raccoon	pandas	ratio
-----	-----	-----	-----

(continues on next page)

(continued from previous page)

version	3.0.0	0.25.2	
initialize empty	0.08248082999998588	4.237655099999756	0.0194638
initialize with matrix	0.0001130529800000204	0.014922300999996878	0.00757611
add rows one column	0.050407570000015764	16.86793469999975	0.00298837
add matrix	0.009786346000000777	0.28846930000008797	0.0339251
append	0.07348676999999952	0.346767699999873	0.211919
get cell	0.6728431999999884	0.9376910000000862	0.717553
get column all index	0.041298069999993456	0.0003524336000000403	117.18
get column subset index	0.5668462000003274	7.041264400000273	0.0805035
get index all columns	1.0389918000000762	0.2065747999999985	5.02962
set cell	0.5282995000002302	1.26670009999998	0.417068
set column all index	0.00548794899999848	0.01662835399999949	0.330036
set column subset index	0.37289839999994	58.03955229999974	0.0064249
set index all columns	0.06374265999997988	1.390037300000131	0.0458568
sort index	0.014900102999999944	0.0008343847000001006	17.8576
iterate rows	0.026519559999997	0.03318497000000207	0.799144
insert rows	0.030685470000025816	0.6862636999999268	0.0447138
time series	0.13054121000000124	29.182044600000154	0.00447334

CHAPTER 5

Indices and tables

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

r

- `raccoon`, [29](#)
- `raccoon.dataframe`, [13](#)
- `raccoon.series`, [22](#)
- `raccoon.sort_utils`, [27](#)
- `raccoon.utils`, [28](#)

A

`add()` (*raccoon.dataframe.DataFrame* method), 14
`append()` (*raccoon.dataframe.DataFrame* method), 14
`append_row()` (*raccoon.dataframe.DataFrame* method), 14
`append_row()` (*raccoon.series.Series* method), 22
`append_rows()` (*raccoon.dataframe.DataFrame* method), 14
`append_rows()` (*raccoon.series.Series* method), 22
`assert_frame_equal()` (in module *raccoon.utils*), 28
`assert_series_equal()` (in module *raccoon.utils*), 28

C

`columns` (*raccoon.dataframe.DataFrame* attribute), 14

D

`data` (*raccoon.dataframe.DataFrame* attribute), 14
`data` (*raccoon.series.Series* attribute), 23
`data` (*raccoon.series.SeriesBase* attribute), 24
`data` (*raccoon.series.ViewSeries* attribute), 27
`data_name` (*raccoon.series.SeriesBase* attribute), 24
`DataFrame` (class in *raccoon.dataframe*), 13
`delete()` (*raccoon.series.Series* method), 23
`delete_all_rows()` (*raccoon.dataframe.DataFrame* method), 14
`delete_columns()` (*raccoon.dataframe.DataFrame* method), 15
`delete_rows()` (*raccoon.dataframe.DataFrame* method), 15
`divide()` (*raccoon.dataframe.DataFrame* method), 15
`dropin` (*raccoon.dataframe.DataFrame* attribute), 15
`dropin` (*raccoon.series.Series* attribute), 23

E

`equality()` (*raccoon.dataframe.DataFrame* method), 15
`equality()` (*raccoon.series.SeriesBase* method), 24

F

`from_dataframe()` (*raccoon.series.ViewSeries* class method), 27
`from_json()` (*raccoon.dataframe.DataFrame* class method), 15
`from_series()` (*raccoon.series.ViewSeries* class method), 27

G

`get()` (*raccoon.dataframe.DataFrame* method), 15
`get()` (*raccoon.series.SeriesBase* method), 24
`get_cell()` (*raccoon.dataframe.DataFrame* method), 16
`get_cell()` (*raccoon.series.SeriesBase* method), 24
`get_columns()` (*raccoon.dataframe.DataFrame* method), 16
`get_entire_column()` (*raccoon.dataframe.DataFrame* method), 16
`get_location()` (*raccoon.dataframe.DataFrame* method), 16
`get_location()` (*raccoon.series.SeriesBase* method), 24
`get_locations()` (*raccoon.dataframe.DataFrame* method), 17
`get_locations()` (*raccoon.series.SeriesBase* method), 25
`get_matrix()` (*raccoon.dataframe.DataFrame* method), 17
`get_rows()` (*raccoon.dataframe.DataFrame* method), 17
`get_rows()` (*raccoon.series.SeriesBase* method), 25
`get_slice()` (*raccoon.dataframe.DataFrame* method), 17
`get_slice()` (*raccoon.series.SeriesBase* method), 25

H

`head()` (*raccoon.dataframe.DataFrame* method), 17
`head()` (*raccoon.series.SeriesBase* method), 25

I

[index \(raccoon.dataframe.DataFrame attribute\)](#), 18
[index \(raccoon.series.Series attribute\)](#), 23
[index \(raccoon.series.SeriesBase attribute\)](#), 25
[index \(raccoon.series.ViewSeries attribute\)](#), 27
[index_name \(raccoon.dataframe.DataFrame attribute\)](#), 18
[index_name \(raccoon.series.SeriesBase attribute\)](#), 25
[isin\(\) \(raccoon.dataframe.DataFrame method\)](#), 18
[isin\(\) \(raccoon.series.SeriesBase method\)](#), 25
[iterrows\(\) \(raccoon.dataframe.DataFrame method\)](#), 18
[itertuples\(\) \(raccoon.dataframe.DataFrame method\)](#), 18

M

[multiply\(\) \(raccoon.dataframe.DataFrame method\)](#), 18

O

[offset \(raccoon.series.ViewSeries attribute\)](#), 27

P

[print\(\) \(raccoon.dataframe.DataFrame method\)](#), 18
[print\(\) \(raccoon.series.SeriesBase method\)](#), 25

R

[raccoon \(module\)](#), 29
[raccoon.dataframe \(module\)](#), 13
[raccoon.series \(module\)](#), 22
[raccoon.sort_utils \(module\)](#), 27
[raccoon.utils \(module\)](#), 28
[rename_columns\(\) \(raccoon.dataframe.DataFrame method\)](#), 18
[reset_index\(\) \(raccoon.dataframe.DataFrame method\)](#), 19
[reset_index\(\) \(raccoon.series.Series method\)](#), 23

S

[select_index\(\) \(raccoon.dataframe.DataFrame method\)](#), 19
[select_index\(\) \(raccoon.series.SeriesBase method\)](#), 26
[Series \(class in raccoon.series\)](#), 22
[SeriesBase \(class in raccoon.series\)](#), 24
[set\(\) \(raccoon.dataframe.DataFrame method\)](#), 19
[set\(\) \(raccoon.series.Series method\)](#), 23
[set_cell\(\) \(raccoon.dataframe.DataFrame method\)](#), 19
[set_cell\(\) \(raccoon.series.Series method\)](#), 23
[set_column\(\) \(raccoon.dataframe.DataFrame method\)](#), 20

[set_location\(\) \(raccoon.dataframe.DataFrame method\)](#), 20
[set_location\(\) \(raccoon.series.Series method\)](#), 23
[set_locations\(\) \(raccoon.dataframe.DataFrame method\)](#), 20
[set_locations\(\) \(raccoon.series.Series method\)](#), 23
[set_row\(\) \(raccoon.dataframe.DataFrame method\)](#), 20
[set_rows\(\) \(raccoon.series.Series method\)](#), 23
[sort \(raccoon.dataframe.DataFrame attribute\)](#), 20
[sort \(raccoon.series.Series attribute\)](#), 24
[sort \(raccoon.series.SeriesBase attribute\)](#), 26
[sort \(raccoon.series.ViewSeries attribute\)](#), 27
[sort_columns\(\) \(raccoon.dataframe.DataFrame method\)](#), 20
[sort_index\(\) \(raccoon.dataframe.DataFrame method\)](#), 21
[sort_index\(\) \(raccoon.series.Series method\)](#), 24
[sorted_exists\(\) \(in module raccoon.sort_utils\)](#), 27
[sorted_index\(\) \(in module raccoon.sort_utils\)](#), 28
[sorted_list_indexes\(\) \(in module raccoon.sort_utils\)](#), 28
[subtract\(\) \(raccoon.dataframe.DataFrame method\)](#), 21

T

[tail\(\) \(raccoon.dataframe.DataFrame method\)](#), 21
[tail\(\) \(raccoon.series.SeriesBase method\)](#), 26
[to_dict\(\) \(raccoon.dataframe.DataFrame method\)](#), 21
[to_dict\(\) \(raccoon.series.SeriesBase method\)](#), 26
[to_json\(\) \(raccoon.dataframe.DataFrame method\)](#), 21
[to_list\(\) \(raccoon.dataframe.DataFrame method\)](#), 21

V

[validate_integrity\(\) \(raccoon.dataframe.DataFrame method\)](#), 22
[validate_integrity\(\) \(raccoon.series.SeriesBase method\)](#), 26
[value\(\) \(raccoon.series.ViewSeries method\)](#), 27
[ViewSeries \(class in raccoon.series\)](#), 26