BabyPandas

Release 1.0

Sep 12, 2022

Contents

Release 1.0

Date Sep 12, 2022

`babypandas` is a `pandas` data-analysis module with a restricted API.

This module is a simplified version of `pandas` with only methods deemed necessary for an introductory course. `babypandas` methods also only contain the necessary arguments needed for basic table manipulation and visualization, reducing the complication found in `pandas`. Writing this module as a restricted version of `pandas` allows for an easy transition into `pandas`.

CHAPTER 1

Reference

1.1 DataFrame

Summary of `DataFrame` methods for `babyandas`. `DataFrame` are a two-dimensional tabular data structure with labeled rows (indices) and columns.

Click a method to see its documentation.

Creation

DataFrame.__init__(**kwargs)	Create an empty DataFrame.
DataFrame.from_dict(data)	Construct DataFrame from dict of array-like or dicts.
DataFrame.from_records(data, *[, columns])	Convert structured or record ndarray to DataFrame.
DataFrame.assign(**kwargs)	Assign new columns to a DataFrame.

1.1.1 bpd.DataFrame.__init__

DataFrame.**__init__**(***kwargs*)

Create an empty DataFrame.

1.1.2 bpd.DataFrame.from_dict

classmethod DataFrame.**from_dict**(*data*)

Construct DataFrame from dict of array-like or dicts.

Parameters **data** (*dict*) – Of the form {field : array-like} or {field : dict}.

Returns

Return type DataFrame

1.1.3 bpd.DataFrame.from_records

classmethod DataFrame.**from_records**(*data*, *, *columns=None*)

Convert structured or record ndarray to DataFrame.

Parameters

- **data** (*ndarray (structured dtype), list of tuples, dict, or DataFrame*) –
- **columns** (*sequence, default None, keyword-only*) – Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

Returns

Return type DataFrame

1.1.4 bpd.DataFrame.assign

DataFrame.**assign**(***kwargs*)

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

Parameters ****kwargs** (*dict of {str: callable or Series}*) – The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns **df_with_cols** – A new DataFrame with the new columns in addition to all the existing columns.

Return type DataFrame

Raises ValueError – If columns have different lengths or if new columns have different lengths than the existing DataFrame

Examples

```
>>> df = bpd.DataFrame().assign(flower=['sunflower', 'rose'])
>>> df.assign(color=['yellow', 'red'])
      flower   color
0  sunflower  yellow
1       rose     red
```

Selection

DataFrame.take(indices)	Return the rows in the given *positional* indices.
DataFrame.drop(*[, columns])	Remove columns by specifying column names.
DataFrame.sample([n, replace, random_state])	Return a random sample of rows from a data frame.
DataFrame.get(key)	Return column or columns from data frame.

1.1.5 bpd.DataFrame.take

DataFrame.**take**(*indices*)

Return the rows in the given *positional* indices.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters **indices** (*array-like*) – An array of ints indicating which positions to take.

Returns **taken** – An DataFrame containing the elements taken from the object.

Return type DataFrame

Raises IndexError – If any *indices* are out of bounds with respect to DataFrame length.

Examples

```
>>> df = bpd.DataFrame().assign(name=['falcon', 'parrot', 'lion'],
...                             kind=['bird', 'bird', 'mammal'])
>>> df
     name    kind
0  falcon    bird
1  parrot    bird
2    lion  mammal
>>> df.take([0, 2])
     name    kind
0  falcon    bird
2    lion  mammal
```

1.1.6 bpd.DataFrame.drop

DataFrame.**drop**(*, *columns=None*)

Remove columns by specifying column names.

Parameters **columns** (*single label or list-like*) – Column names to drop.

Returns **df** – DataFrame with the dropped columns.

Return type DataFrame

Raises KeyError – If none of the column labels are found.

Examples

```
>>> df = bpd.DataFrame().assign(A=[0, 4, 8],
...                             B=[1, 5, 9],
...                             C=[2, 6, 10],
...                             D=[3, 7, 11])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
>>> df.drop(columns=['B', 'C'])
   A   D
```

(continues on next page)

(continued from previous page)

```
0  0   3
1  4   7
2  8  11
```

1.1.7 bpd.DataFrame.sample

DataFrame.**sample**(*n=None*, *, *replace=False*, *random_state=None*)

Return a random sample of rows from a data frame.

You can use *random_state* for reproducibility.

Parameters

- **n** (*None or int, optional*) – Number of rows to return. None corresponds to 1.
- **replace** (*{False, True}, optional, keyword only.*) – Sample with or without replacement.
- **random_state** (*int or numpy.random.RandomState, optional, keyword only*) – Seed for the random number generator (if int), or numpy RandomState object.

Returns **s_df** – A new DataFrame containing *n* items randomly sampled from the caller object.

Return type DataFrame

Raises ValueError – If a sample larger than the length of the DataFrame is taken without replacement.

Examples

```
>>> df = bpd.DataFrame().assign(letter=['a', 'b', 'c'],
...                             count=[9, 3, 3],
...                             points=[1, 2, 2])
>>> df.sample(1, random_state=0)
    letter  count  points
2      c      3       2
```

1.1.8 bpd.DataFrame.get

DataFrame.**get**(*key*)

Return column or columns from data frame.

Parameters **key** (*str or iterable of strings*) – Column label or iterable of column labels

Returns **series_or_df** – Series with the corresponding label or DataFrame with the corresponding column labels.

Return type Series or DataFrame

Raises KeyError – If any column named in *key* not found in columns.

Examples

```
>>> df = bpd.DataFrame().assign(letter=['a', 'b', 'c'],
...                             count=[9, 3, 3],
...                             points=[1, 2, 2])
>>> df.get('letter')
0    a
1    b
2    c
Name: letter, dtype: object
>>> df.get(['count', 'points'])
   count  points
0      9       1
1      3       2
2      3       2
```

Transformation

DataFrame.apply(func[, axis])	Apply a function along an axis of the DataFrame.
DataFrame.sort_values(by, *[, ascending])	Sort by the values in column(s) named in *by*.
DataFrame.describe()	Generate descriptive statistics.
DataFrame.groupby(by)	Group DataFrame by values in columns specified in *by*.
DataFrame.reset_index(*[, drop])	Reset the index.
DataFrame.set_index(keys[, drop])	Set the DataFrame index using existing columns.

1.1.9 bpd.DataFrame.apply

DataFrame.**apply**(*func*, *axis=0*)

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). The final return type is inferred from the return type of the applied function.

Parameters

- **func** (*function*) – Function to apply to each column or row.
- **axis** (*{0 or 'index', 1 or 'columns'}, default 0*) – Axis along which the function is applied:
  - 0 or 'index': apply function to each column.
  - 1 or 'columns': apply function to each row.

Returns **applied** – Result of applying `func` along the given axis of the DataFrame.

Return type Series or DataFrame

Examples

```
>>> def add_two(row):
...     return row + 2
>>> df = bpd.DataFrame(A=[1, 1],
...                    B=[2, 2])
>>> df.apply(add_two)
   A  B
```

(continues on next page)

(continued from previous page)

```
0  3  4
1  3  4
```

1.1.10 bpd.DataFrame.sort_values

DataFrame.**sort_values**(*by*, *, *ascending=True*)

Sort by the values in column(s) named in *by*.

Parameters

- **by** (*str or list of str*) – Name or list of column names to sort by.
- **ascending** (*{True, False} or list of bool, keyword only*) – Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the *by*. Default is True.

Returns **sorted_obj**

Return type DataFrame

Raises KeyError – If *by* not found in columns.

Examples

```
>>> df = bpd.DataFrame().assign(name=['Sally', 'George', 'Bill', 'Ann'],
...                             age=[21, 25, 18, 28],
...                             height_cm=[161, 168, 171, 149])
>>> df.sort_values(by='age')
     name  age  height_cm
2    Bill   18        171
0   Sally   21        161
1  George   25        168
3     Ann   28        149
>>> df.sort_values(by='height_cm', ascending=False)
     name  age  height_cm
2    Bill   18        171
1  George   25        168
0   Sally   21        161
3     Ann   28        149
```

1.1.11 bpd.DataFrame.describe

DataFrame.**describe**()

Generate descriptive statistics.

Statistics summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types.

Parameters **None** –

Returns **descr** – Summary statistics of the Dataframe provided.

Return type DataFrame

Examples

```
>>> df = bpd.DataFrame().assign(A=[0, 10, 20],
...                             B=[1, 2, 3])
>>> df.describe()
          A    B
count   3.0  3.0
mean   10.0  2.0
std    10.0  1.0
min     0.0  1.0
25%     5.0  1.5
50%    10.0  2.0
75%    15.0  2.5
max    20.0  3.0
```

1.1.12 bpd.DataFrame.groupby

DataFrame.**groupby**(*by*)

Group DataFrame by values in columns specified in *by*.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. this can be used to group large amounts of data and compute operations on these groups.

Parameters **by** (*label, or list of labels*) – Used to determine the groups for the groupby. Should be a label or list of labels that will group by the named columns in `self`. Notice that a tuple is interpreted a (single) key.

Returns **df_gb** – groupby object that contains information about the groups.

Return type DataFrameGroupBy

Raises `KeyError` – If *by* not found in columns

Examples

```
>>> df =bpd.DataFrame(animal=['Falcon', 'Falcon', 'Parrot', 'Parrot'],
...                   max_speed=[380, 370, 24, 26])
>>> df.groupby('animal').mean()
        max_speed
animal
Falcon      375.0
Parrot       25.0
```

1.1.13 bpd.DataFrame.reset_index

DataFrame.**reset_index**(*, *drop=False*)

Reset the index.

Reset the index of the DataFrame, and use the default one instead.

Parameters **drop** (*bool, default False, keyword only*) – Do not try to insert index into dataframe columns. This resets the index to the default integer index.

Returns

- *DataFrame* – DataFrame with the new index.

- *Reset the index of the DataFrame, and use the default one*
- *instead. If the DataFrame has a MultiIndex, this method can*
- *remove one or more levels.*

Examples

```
>>> df = bpd.DataFrame().assign(name=['Sally', 'George', 'Bill', 'Ann'],
...                             age=[21, 25, 18, 28],
...                             height_cm=[161, 168, 171, 149])
>>> sorted = df.sort_values(by='age')
>>> sorted
     name  age  height_cm
2    Bill   18        171
0   Sally   21        161
1  George   25        168
3     Ann   28        149
>>> sorted.reset_index(drop=True)
     name  age  height_cm
0    Bill   18        171
1   Sally   21        161
2  George   25        168
3     Ann   28        149
```

1.1.14 bpd.DataFrame.set_index

`DataFrame.`**`set_index`**(*keys*, *drop=True*)

Set the DataFrame index using existing columns.

Set the DataFrame index (row labels) using one or more existing columns or arrays (of the correct length). The index replaces the existing index.

Parameters

- **`keys`** (*`label or array-like or list of labels/arrays`*) – This parameter can be either a single column key, a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of column keys and arrays. Here, "array" encompasses `Series`, `Index` and `np.ndarray`.
- **`drop`** (*`bool, default True`*) – Delete columns to be used as the new index.

Returns Data frame with changed row labels.

Return type DataFrame

Raises `KeyError` – If *keys* not found in columns.

Examples

```
>>> df = bpd.DataFrame().assign(name=['Sally', 'George', 'Bill', 'Ann'],
...                             age=[21, 25, 18, 28],
...                             height_cm=[161, 168, 171, 149])
>>> df.set_index('name')
        age  height_cm
name
```

(continues on next page)

(continued from previous page)

```
Sally    21        161
George   25        168
Bill     18        171
Ann      28        149
```

Combining

DataFrame.merge(right[, how, on, left_on, ...])	Merge DataFrame or named Series objects with a database-style join.
DataFrame.append(other[, ignore_index])	Append rows of *other* to the end of caller, returning a new object.

1.1.15 bpd.DataFrame.merge

DataFrame.**merge**(*right*, *how='inner'*, *on=None*, *left_on=None*, *right_on=None*, *left_index=False*, *right_index=False*)

Merge DataFrame or named Series objects with a database-style join.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

Parameters

- **right** (*DataFrame or named Series*) – Object to merge with.
- **how** (*{'left', 'right', 'outer', 'inner'}, default 'inner'*) – Type of merge to be performed.

  *** left: use only keys from left frame, similar to a SQL left outer** join; preserve key order.

  *** right: use only keys from right frame, similar to a SQL right** outer join; preserve key order.

  *** outer: use union of keys from both frames, similar to a SQL full** outer join; sort keys lexicographically.

  *** inner: use intersection of keys from both frames, similar to a** SQL inner join; preserve the order of the left keys.

- **on** (*label or list*) – Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.
- **left_on** (*label or list, or array-like*) – Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.
- **right_on** (*label or list, or array-like*) –

  Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right

  left_index [boolean, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

- **right_index** (*boolean, default False*) – Use the index from the right DataFrame as the join key. Same caveats as left_index DataFrame. These arrays are treated as if they are columns.

Returns A DataFrame of the two merged objects.

Return type DataFrame

Raises `KeyError` – If any input labels are not found in the corresponding DataFrame's columns.

Examples

```
>>> df1 = bpd.DataFrame().assign(pet=['dog', 'cat', 'lizard', 'turtle'],
...                              kind=['mammal', 'mammal', 'reptile', 'reptile'])
>>> df2 = bpd.DataFrame().assign(kind=['mammal', 'reptile', 'amphibian'],
...                              abr=['m', 'r', 'a'])
>>> df1.merge(df2, on='kind')
      pet     kind abr
0     dog   mammal   m
1     cat   mammal   m
2  lizard  reptile   r
3  turtle  reptile   r
```

1.1.16 bpd.DataFrame.append

`DataFrame.`**append**(*other*, *ignore_index=False*)

Append rows of *other* to the end of caller, returning a new object.

Columns in *other* that are not in the caller are added as new columns.

Parameters

- **other** (*DataFrame or Series/dict-like object, or list of these*) – The data to append.
- **ignore_index** (*boolean, default False*) – If True, do not use the index labels.

Returns **a_df** – DataFrame with appended rows.

Return type DataFrame

Plotting

DataFrame.plot(*args, **kwargs)	DataFrame plotting accessor and method

1.1.17 bpd.DataFrame.plot

`DataFrame.`**plot**(**args*, ***kwargs*)

DataFrame plotting accessor and method

Examples

```
>>> df.plot.line()
>>> df.plot.scatter('x', 'y')
>>> df.plot.hexbin()
```

IO

DataFrame.to_csv([path_or_buf, index])	Write object to a comma-separated values (csv) file.
DataFrame.to_numpy()	Convert the DataFrame to a NumPy array.

1.1.18 bpd.DataFrame.to_csv

DataFrame.**to_csv**(*path_or_buf=None*, *, *index=True*)

Write object to a comma-separated values (csv) file.

Parameters

- **path_or_buf** (*str or file handle, default None*) – File path or object, if None is provided the result is returned as a string.
- **index** (*bool, default True*) – Write row names (index).

Returns If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

Return type None or str

1.1.19 bpd.DataFrame.to_numpy

DataFrame.**to_numpy**()

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are `float16` and `float32`, the results dtype will be `float32`. This may require copying data and coercing values, which may be expensive.

Parameters **None** –

Returns **df_arr** – DataFrame as a NumPy array.

Return type numpy.ndarray

1.2 Series

Summary of `Series` methods for `babypandas`. `Series` are one-dimensional arrays with labeled indices.

Click a method to see its documentation.

Creation

Series.__init__(**kwargs)	Create an empty Series.

1.2.1 bpd.Series.__init__

Series.**__init__**(***kwargs*)

Create an empty Series.

Selection

Series.take(indices)	Return the elements in the given *positional* indices.
Series.sample([n, replace, random_state])	Return a random sample of elements from a Series.

1.2.2 bpd.Series.take

Series.**take**(*indices*)

Return the elements in the given *positional* indices.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

Parameters **indices** (*array-like*) – An array of ints indicating which positions to take.

Returns **taken** – A Series containing the elements taken from the object.

Return type Series

Raises IndexError – If any *indices* are out of bounds with respect to Series length.

Examples

```
>>> s = bpd.Series(data=[1, 2, 3], index=['A', 'B', 'C'])
>>> s.take([0, 3])
A    1
C    3
dtype: int64
>>> s.take(np.arange(2))
A    1
B    2
dtype: int64
```

1.2.3 bpd.Series.sample

Series.**sample**(*n=None*, *replace=False*, *random_state=None*)

Return a random sample of elements from a Series.

You can use *random_state* for reproducibility.

Parameters

- **n** (*None or int, optional*) – Number of elements to return. None corresponds to 1.
- **replace** (*{False, True}, optional, keyword only.*) – Sample with or without replacement.
- **random_state** (*int or numpy.random.RandomState, optional, keyword only*) – Seed for the random number generator (if int), or numpy RandomState object.

Returns **s_series** – A new Series containing *n* items randomly sampled from the caller object.

Return type Series

Raises ValueError – If a sample larger than the length of the Series is taken without replacement.

Examples

```
>>> s = bpd.Series(data=[1, 2, 3, 4, 5])
>>> s.sample(3, random_state=0)
2    3
0    1
1    2
dtype: int64
>>> s.sample(7, replace=True, random_state=10)
1    2
4    5
0    1
1    2
3    4
4    5
1    2
dtype: int64
```

Transformation

Series.apply(func)	Invoke function on values of Series.
Series.sort_values(*[, ascending])	Sort by the values.
Series.describe()	Generate descriptive statistics.
Series.reset_index(*[, drop])	Reset the index.

1.2.4 bpd.Series.apply

Series.**apply**(*func*)

Invoke function on values of Series.

Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values.

Parameters **func** (*function*) – Python function or NumPy ufunc to apply.

Returns **a_obj** – If func returns a Series object the result will be a DataFrame.

Return type Series or DataFrame

Examples

```
>>> def cut_off_5(val):
...     if val > 5:
...         return 5
...     else:
...         return val
>>> s = bpd.Series(data=[1, 3, 5, 7, 9]
>>> s.apply(cut_off_5)
0    1
1    3
2    5
3    5
4    5
dtype: int64
```

1.2.5 bpd.Series.sort_values

Series.**sort_values**(*, *ascending=True*)

Sort by the values.

Sort a Series in ascending or descending order.

Parameters **ascending** (*bool, default True, keyword only*) – If True, sort values in ascending order, otherwise descending.

Returns **s_series** – Series ordered by values.

Return type Series

Example

```
>>> s = bpd.Series(data=[6, 4, 3, 9, 5])
>>> s.sort_values()
2    3
1    4
4    5
0    6
3    9
dtype: int64
>>> s.sort_values(ascending=False)
3    9
0    6
4    5
1    4
2    3
dtype: int64
```

1.2.6 bpd.Series.describe

Series.**describe**()

Generate descriptive statistics.

Statistics summarize the central tendency, dispersion and shape of a Series' distribution, excluding NaN values.

Analyzes both numeric and object series.

Parameters **None** –

Returns **descr** – Summary statistics of the Series provided.

Return type Series

Examples

```
>>> s = bpd.Series(data=[6, 7, 7, 5, 9, 5, 1])
>>> s.describe()
count    7.000000
mean     5.714286
std      2.497618
min      1.000000
25%      5.000000
```

(continues on next page)

(continued from previous page)

```
50%       6.000000
75%       7.000000
max       9.000000
dtype: float64
```

1.2.7 bpd.Series.reset_index

Series.**reset_index**(*, *drop=False*)

Reset the index.

This is useful when the index is meaningless and needs to be reset to the default before another operation.

Parameters **drop** (*bool, default False, keyword only*) – When True, do not try to insert index into dataframe columns. This resets the index to the default integer index. If False, then turn input Series into DataFrame, adding original index as column.

Returns When *drop* is False (the default), a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values. When *drop* is True, a *Series* is returned.

Return type Series or DataFrame

Examples

```
>>> s = bpd.Series([6, 4, 3, 9, 5])
>>> sorted = s.sort_values()
>>> sorted.reset_index()
   index  0
0      2  3
1      1  4
2      4  5
3      0  6
4      3  9
>>> sorted.reset_index(drop=True)
0    3
1    4
2    5
3    6
4    9
dtype: int64
```

Plotting

Series.plot(*args, **kwargs)	Series plotting accessor and method.

1.2.8 bpd.Series.plot

Series.**plot**(**args*, ***kwargs*)

Series plotting accessor and method.

Examples

```
>>> s.plot.line()
>>> s.plot.bar()
>>> s.plot.hist()
```

IO

Series.to_csv([path_or_buf, index])	Write object to a comma-separated values (csv) file.
Series.to_numpy()	A NumPy ndarray representing the values in this Series or Index.

1.2.9 bpd.Series.to_csv

Series.**to_csv**(*path_or_buf=None*, *index=True*)

Write object to a comma-separated values (csv) file.

Parameters

- **path_or_buf** (*str or file handle, default None*) – File path or object, if None is provided the result is returned as a string.
- **index** (*bool, default True*) – Write row names (index).

Returns If path_or_buf is None, returns the resulting csv format as a string. Otherwise returns None.

Return type None or str

1.2.10 bpd.Series.to_numpy

Series.**to_numpy**()

A NumPy ndarray representing the values in this Series or Index.

Parameters **None** –

Returns **arr**

Return type numpy.ndarray

Calculations

Series.count()	Return number of non-NA/null observations in the Series.
Series.mean()	Return the mean of the values for the requested axis.
Series.median()	Return the median of the values for the requested axis.
Series.min()	Return the minimum of the values in the Series.
Series.max()	Return the maximum of the values in the Series.
Series.sum()	Return the sum of the values in the Series.
Series.abs()	Return a Series with absolute numeric value of each element.

1.2.11 bpd.Series.count

`Series.`**`count`**`()`
Return number of non-NA/null observations in the Series.

1.2.12 bpd.Series.mean

`Series.`**`mean`**`()`
Return the mean of the values for the requested axis.

1.2.13 bpd.Series.median

`Series.`**`median`**`()`
Return the median of the values for the requested axis.

1.2.14 bpd.Series.min

`Series.`**`min`**`()`
Return the minimum of the values in the Series.

1.2.15 bpd.Series.max

`Series.`**`max`**`()`
Return the maximum of the values in the Series.

1.2.16 bpd.Series.sum

`Series.`**`sum`**`()`
Return the sum of the values in the Series.

1.2.17 bpd.Series.abs

`Series.`**`abs`**`()`
Return a Series with absolute numeric value of each element.

1.3 GroupBy

Summary of `DataFrameGroupBy` methods for `babypandas`. `DataFrameGroupBy` objects are returned by `groupby` calls: *`DataFrame.groupby()`*

Click a method to see its documentation.

Calculations

`DataFrameGroupBy.count`()	Compute count of group.
`DataFrameGroupBy.mean`()	Compute mean of group.
`DataFrameGroupBy.median`()	Compute median of group.

Continued on next page

Table 13 – continued from previous page

DataFrameGroupBy.min()	Compute min of group.
DataFrameGroupBy.max()	Compute max of group.
DataFrameGroupBy.sum()	Compute sum of group.
DataFrameGroupBy.size()	Compute group sizes.

1.3.1 bpd.DataFrameGroupBy.count

DataFrameGroupBy.**count**()
Compute count of group.

1.3.2 bpd.DataFrameGroupBy.mean

DataFrameGroupBy.**mean**()
Compute mean of group.

1.3.3 bpd.DataFrameGroupBy.median

DataFrameGroupBy.**median**()
Compute median of group.

1.3.4 bpd.DataFrameGroupBy.min

DataFrameGroupBy.**min**()
Compute min of group.

1.3.5 bpd.DataFrameGroupBy.max

DataFrameGroupBy.**max**()
Compute max of group.

1.3.6 bpd.DataFrameGroupBy.sum

DataFrameGroupBy.**sum**()
Compute sum of group.

1.3.7 bpd.DataFrameGroupBy.size

DataFrameGroupBy.**size**()
Compute group sizes.

Index