Namespace `data-pypeline` {#data-pypeline}

Sub-modules

- data-pypeline.DataModel
- data-pypeline.IO
- data-pypeline.Pipeline
- data-pypeline.Summarize
- data-pypeline.Transform
- data-pypeline.main
- data-pypeline.setup

Module `data-pypeline.DataModel` {#data-pypeline.DataModel}

Sub-modules

- data-pypeline.DataModel.dataframe
- data-pypeline.DataModel.test

Module `data-pypeline.DataModel.dataframe` {#data-pypeline.DataModel.dataframe}

Classes

Class `DataFrame` {#data-pypeline.DataModel.dataframe.DataFrame}

```
class DataFrame(
    data=None
)
```

A 2 dimensions DataFrame.

...

Attributes

self.__groups : list Ordered list of variables names defining the group structure of the DataFrame. self.__columns : list Ordered list of the variables names on the DataFrame. self.__data : dict Data container, with variable names as keys, and lists as values.

Methods

init(data=None) Create an empty (if data is None) or already filled DataFrame (if data is a dict). **str** Gives a string representation of the DataFrame, displaying it's group structure ans 5 first data lines. **len** : int Returns the number of lines in the DataFrame. **getitem**(item) : Object Returns a column (if item is a string or an int), a line (if item is a tuple with None as first value and an int as second value) or a value (if item is a tuple with a string or an int as first value and an int as second value) of the DataFrame. **setitem**(key, value) Sets to value a column (if key is a string or an int), a line (if key is a tuple with None as first value and an int as second value) or a value (if key is a tuple with a string or an int as first value and an int as second value) of the DataFrame. **iter** : Iterator Gets an iterator on the lines of the DataFrame. **next** : list Gets the next line of the DataFrame. dict : dict Returns a dict representation of the DataFrame. vars : list Returns the list of the variables names in the DataFrame. shape : (int, int) Returns a tuple with the number of columns and number of lines of the DataFrame. groups : list Returns a list with the group identifier of each line of the DataFrame. groups_vars : list Returns an ordered list of the variables representing the group structure of the DataFrame. groups_df : [DataFrame] Returns a list of DataFrames, with one DataFrame per group. add_column(name, content=None, after=None, before=None) Insert a new column in the DataFrame, with name as name and a list of None (if content is None) or content as content. If after or before is specified with a string or an int, the column is inserted after or before the specified column. Otherwise, the column is inserted at the end of the DataFrame. add_row(content=None, after=None, before=None) Insert a new line in the DataFrame, with a list of None (if content is None) or content as content. If after or before is specified with an int, the line is inserted after or before the specified line. Otherwise, the line is inserted at the end of the DataFrame. add_group(name, level=None) Insert the name variable in the DataFrame's group structure. If level is None, the variable is inserted at the last level on the group structure, otherwise, the variable is inserted at the level index. rename_column(old_name, new_name) Changes the name of the old_name column to new_name. del_column(name) Delete the column with the specified name. del_row(index) Delete the line with the specified index. del_group(name) Removes the name variable of the group structure. row_as_dict(index, with_lag=True, with_lead=True) Returns the contents of the line of the specified index as a dict, with the variables names as keys and the values on the line as values. If with_lag is True, returns also in the same dict the variables of the previous line with "variable_lag" as keys and the values of the previous line as values, or None as values if there isn't any previous line. If with_lead is True, returns also in the same dict the variables of the next line with "variable_lead" as keys and the values of the next line as values, or None as values if there isn't any next line.

Create a new DataFrame object from scratch or based on a dict with one key per column

Parameters

`data` : `dict`, optional : Dictionary with one key per column and data as list for each column on values

Instance variables

Variable `dict` {#data-pypeline.DataModel.dataframe.DataFrame.dict}

Returns

`dict` : Dictionary representation of the DataFrame

Variable `groups` {#data-pypeline.DataModel.dataframe.DataFrame.groups}

Gives the group index of each row of the DataFrame, based on the group structure

Returns

`list` : Group index of each row

Variable `groups_df` {#data-pypeline.DataModel.dataframe.DataFrame.groups_df}

Variable `groups_vars` {#data-pypeline.DataModel.dataframe.DataFrame.groups_vars}

Returns

`list` : Hierarchical representation of the group structure

Variable `shape` {#data-pypeline.DataModel.dataframe.DataFrame.shape}

Gives the shape of the DataFrame

Returns

(int, int) Number of columns and number of lines

Variable `vars` {#data-pypeline.DataModel.dataframe.DataFrame.vars}

Returns

`list` : List of the variables names

Methods

Method `add_column` {#data-pypeline.DataModel.dataframe.DataFrame.add_column}

```
def add_column(
    self,
    name,
    content=None,
    after=None,
    before=None
)
```

Method `add_group` {#data-pypeline.DataModel.dataframe.DataFrame.add_group}

```
def add_group(
    self,
    name,
    level=None
)
```

Method `add_row` {#data-pypeline.DataModel.dataframe.DataFrame.add_row}

```
def add_row(
    self,
    content=None,
    after=None,
    before=None
)
```

Method `del_column` {#data-pypeline.DataModel.dataframe.DataFrame.del_column}

```
def del_column(
    self,
    name
)
```

Method `del_group` {#data-pypeline.DataModel.dataframe.DataFrame.del_group}

```
def del_group(
    self,
    name
)
```

Method `del_row` {#data-pypeline.DataModel.dataframe.DataFrame.del_row}

```
def del_row(
    self,
```

```
    index
)
```

Method `rename_column` {#data-pypeline.DataModel.dataframe.DataFrame.rename_column}

```
def rename_column(
    self,
    old_name,
    new_name
)
```

Method `row_as_dict` {#data-pypeline.DataModel.dataframe.DataFrame.row_as_dict}

```
def row_as_dict(
    self,
    index,
    with_lag=True,
    with_lead=True
)
```

Module `data-pypeline.DataModel.test` {#data-pypeline.DataModel.test}

Sub-modules

- data-pypeline.DataModel.test.test_dataframe

Module `data-pypeline.DataModel.test.test_dataframe` {#data-pypeline.DataModel.test.test_dataframe}

Classes

Class `TestDataFrame` {#data-pypeline.DataModel.test.test_dataframe.TestDataFrame}

```
class TestDataFrame(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.DataModel.test.test_dataframe.TestDataFrame.setUp}

```
def setUp(
    self
```

```
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_getItem_singleColumn_DataFrame` {#data-pypeline.DataModel.test.test_dataframe.TestDataFrame.test_getItem_singleColumn_DataFrame}

```
def test_getItem_singleColumn_DataFrame(
    self
)
```

Method `test_init_emptyDataFrame` {#data-pypeline.DataModel.test.test_dataframe.TestDataFrame.test_init_emptyDataFrame}

```
def test_init_emptyDataFrame(
    self
)
```

Method `test_lenDataFrame` {#data-pypeline.DataModel.test.test_dataframe.TestDataFrame.test_lenDataFrame}

```
def test_lenDataFrame(
    self
)
```

Method `test_shapeDataFrame` {#data-pypeline.DataModel.test.test_dataframe.TestDataFrame.test_shapeDataFrame}

```
def test_shapeDataFrame(
    self
)
```

Module `data-pypeline.IO` {#data-pypeline.IO}

Sub-modules

- data-pypeline.IO.exportcsv
- data-pypeline.IO.exportmapfrmetro
- data-pypeline.IO.st_import

Module `data-pypeline.IO.exportcsv` {#data-pypeline.IO.exportcsv}

Classes

Class `ExportCSV` {#data-pypeline.IO.exportcsv.ExportCSV}

```
class ExportCSV(
    path,
    headers=True,
    delimiter=';',
    encoding='ISO-8859-1'
)
```

Exports a DataFrame as CSV file.

...

Attributes

self.__path : str Absolute or relative path of the CSV file to generate self.__headers : bool Indicates if the headers of the DataFrame will be exported or not self.__delimiter : str Specify the delimiter character on the CSV file self.__encoding : str Specify the encoding of the CSV file

Methods

init(path, headers=True, delimiter=";", encoding='ISO-8859-1') Create an ExportCSV pipelinable object and define the specifications of the future CSV file. apply(df) : DataFrame Takes the DataFrame df and exports it as CSV file

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.IO.exportcsv.ExportCSV.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.IO.exportmapfrmetro` {#data-pypeline.IO.exportmapfrmetro}

Classes

Class `ExportMapFRMetro` {#data-pypeline.IO.exportmapfrmetro.ExportMapFRMetro}

```
class ExportMapFRMetro(
    path,
    type_geo,
    var_geocode,
    var_tomap,
    title=None,
    color_scale='viridis',
    display_labels=True,
    display_scale=True
)
```

Exports a DataFrame as a Metropolitan France map.

...

Attributes

self.__path : str Absolute or relative path of the map image file to generate self.__json_path : str Relative path of the GeoJSON file with the correct administrative division self.__var_geocode : str Name of the variable with the geographic codes in the DataFrame self.__var_tomap : str Name of the vairable with the values to display in the map self.__title : str Title of the map self.__color_scale : str Name of the matplotlib's color scale to use self.__display_lables : bool Displays the geographic codes on the map if True self.__display_scale : bool Displays the colors scale right to the map if True

Methods

init(path, type_geo, var_geocode, var_tomap, title=None, color_scale='viridis', display_labels=True, display_scale=True) Create an ExportMapFRMetro pipelinable object and

define the specifications of the future map. type_geo must be set to "dep" or "departement" for a map at departmental scale or to "reg" or "region" for a map at regional scale.

apply(df) : DataFrame Takes the DataFrame df and exports it as map image file

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.IO.exportmapfrmetro.ExportMapFRMetro.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.IO.st_import` {#data-pypeline.IO.st_import}

Classes

Class `Import` {#data-pypeline.IO.st_import.Import}

```
class Import
```

Static methods

`Method import_csv` {#data-pypeline.IO.st_import.Import.import_csv}

```
def import_csv(
    path,
    headers=True,
    delimiter=';',
    encoding='ISO-8859-1'
)
```

Imports a CSV file as DataFrame

Parameters

`path` : `str` : Absolute or relative path to the CSV file to import

`headers` : `bool = True` : Specify if the file have headers

`delimiter` : `str = ";"` : Specify the file's delimiter

`encoding` : `str = 'ISO-8859-1'` : Specify the file's encoding

Returns

`DataFrame` : A DataFrame with the contents of the CSV file

`Method import_json` {#data-pypeline.IO.st_import.Import.import_json}

```
def import_json(
    path,
    root=None
)
```

Imports a JSON file as DataFrame.

Parameters

`path` : `str` : Absolute or relative path to the JSON file to import

`root` : `str = None` : Name of the root's node to import ; if None, imports the first root node of the file

Returns

`DataFrame` : A DataFrame with the contents of the JSON file

Module `data-pypeline.Pipeline` {#data-pypeline.Pipeline}

Sub-modules

- data-pypeline.Pipeline.ongroups
- data-pypeline.Pipeline.onvars
- data-pypeline.Pipeline.pipeline
- data-pypeline.Pipeline.pipelineable

Module `data-pypeline.Pipeline.ongroups` {#data-pypeline.Pipeline.ongroups}

Classes

Class `OnGroups` {#data-pypeline.Pipeline.ongroups.OnGroups}

```
class OnGroups
```

Abstract class for pipelinable operations on groups.

...

Methods

_operation(group_df) Abstract method to implement, performs the operation on a group DataFrame

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Pipeline.onvars` {#data-pypeline.Pipeline.onvars}

Classes

Class `OnVars` {#data-pypeline.Pipeline.onvars.OnVars}

```
class OnVars(
    *on_vars
)
```

Abstract class for pipelinable operations taking variable names as parameters.

...

Attributes

self._vars : list List of variables to use on the apply or _operation method

Methods

init(*on_vars) Base constructor for child classes, handling the variables names specification *add_vars(*on_vars) Add new variables to the object after it's creation vars : list Returns the list of currently specified variables del_vars(*on_vars) Remove variables to the object after it's creation _secure_add_vars(on_vars): Handles the type checking on adding variables operations

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Instance variables

Variable `vars` {#data-pypeline.Pipeline.onvars.OnVars.vars}

Methods

Method `add_vars` {#data-pypeline.Pipeline.onvars.OnVars.add_vars}

```
def add_vars(
    self,
    *on_vars
)
```

Method `del_vars` {#data-pypeline.Pipeline.onvars.OnVars.del_vars}

```
def del_vars(
    self,
    *on_vars
)
```

Module `data-pypeline.Pipeline.pipeline` {#data-pypeline.Pipeline.pipeline}

Classes

Class `Pipeline` {#data-pypeline.Pipeline.pipeline.Pipeline}

```
class Pipeline(
    *operations
)
```

A Pipeline of pipelineable operations to apply on a DataFrame.

Is itself a pipelinable operation.

...

Attributes

self.__operations : list List of pipelineable operations to apply on a DataFrame

Methods

init(*operations) Create a Pipeline with the specified operations add_operations(*operations) Add new operations to the Pipeline after it's creation operations : list Returns the list of currently specified operations del_operations(*operations) Remove operations to the Pipeline after it's creation apply(df) : DataFrame Apply the operations of the PipeLine to the DataFrame df and returns the final DataFrame _secure_add_operations(operations): Handles the pipelineable checking when adding operations

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Instance variables

Variable `operations` {#data-pypeline.Pipeline.pipeline.Pipeline.operations}

Methods

Method `add_operations` {#data-pypeline.Pipeline.pipeline.Pipeline.add_operations}

```
def add_operations(
    self,
    *operations
)
```

Method `apply` {#data-pypeline.Pipeline.pipeline.Pipeline.apply}

```
def apply(
    self,
```

```
    df
)
```

Method `del_operations` {#data-pypeline.Pipeline.pipeline.Pipeline.del_operations}

```
def del_operations(
    self,
    *operations
)
```

Module `data-pypeline.Pipeline.pipelineable` {#data-pypeline.Pipeline.pipelineable}

Classes

Class `Pipelineable` {#data-pypeline.Pipeline.pipelineable.Pipelineable}

```
class Pipelineable
```

Abstract class for pipelinable operations.

...

Methods

apply(group_df) Abstract method to implement, performs the operation on a DataFrame

Ancestors (in MRO)

- abc.ABC

Methods

Method `apply` {#data-pypeline.Pipeline.pipelineable.Pipelineable.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Summarize` {#data-pypeline.Summarize}

Sub-modules

- data-pypeline.Summarize.average
- data-pypeline.Summarize.count
- data-pypeline.Summarize.max
- data-pypeline.Summarize.min
- data-pypeline.Summarize.sum
- data-pypeline.Summarize.summarizeongroups
- data-pypeline.Summarize.test
- data-pypeline.Summarize.variance

Module `data-pypeline.Summarize.average` {#data-pypeline.Summarize.average}

Classes

Class `Average` {#data-pypeline.Summarize.average.Average}

```
class Average(
    *on_vars,
    delete_na=True,
    delete_nan=True
)
```

Calculates the arithmetic average for a column (of a DataFrame).

...

Methods

_operation(col) : dict Calculates the average on the col object, assuming this DataFrame represents a single group The output variable is called "Variable_Average" (if col = Variable).

Ancestors (in MRO)

- Summarize.summarizeongroups.SummarizeOnGroups
- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Summarize.count` {#data-pypeline.Summarize.count}

Classes

Class `Count` {#data-pypeline.Summarize.count.Count}

```
class Count(
    *on_vars,
    delete_na=True,
    delete_nan=True
)
```

Calculates the number of observations of a column (of a DataFrame).

...

Methods

_operation(col) : dict Calculates the number of observations on the col object, assuming this DataFrame represents a single group. The number of observations is given by the length of the input. The output variable is called "Variable_Count" (if col = Variable).

Ancestors (in MRO)

- Summarize.summarizeongroups.SummarizeOnGroups
- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Summarize.max` {#data-pypeline.Summarize.max}

Classes

Class `Max` {#data-pypeline.Summarize.max.Max}

```
class Max(
    *on_vars,
    delete_na=True,
    delete_nan=True
)
```

Calculates the maximum of a column (of a DataFrame).

...

Methods

_operation(col) : dict Calculates the number maximum on the col object, assuming this DataFrame represents a single group. The output variable is called "Variable_Max" (if col = Variable).

Ancestors (in MRO)

- Summarize.summarizeongroups.SummarizeOnGroups
- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Summarize.min` {#data-pypeline.Summarize.min}

Classes

Class `Min` {#data-pypeline.Summarize.min.Min}

```
class Min(
    *on_vars,
    delete_na=True,
    delete_nan=True
)
```

Calculates the minimum of a column (of a DataFrame).

...

Methods

_operation(col) : dict Calculates the number minimum on the col object, assuming this DataFrame represents a single group. The output variable is called "Variable_Min" (if col = Variable).

Ancestors (in MRO)

- Summarize.summarizeongroups.SummarizeOnGroups
- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Summarize.sum` {#data-pypeline.Summarize.sum}

Classes

Class `Sum` {#data-pypeline.Summarize.sum.Sum}

```
class Sum(
    *on_vars,
    delete_na=True,
    delete_nan=True
)
```

Calculates the sum of a column (of a DataFrame).

...

Methods

_operation(col) : dict Calculates the total of all observations on the col object, assuming this DataFrame represents a single group. The output variable is called "Variable_Sum" (if col = Variable).

Ancestors (in MRO)

- Summarize.summarizeongroups.SummarizeOnGroups
- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Summarize.summarizeongroups` {#data-pypeline.Summarize.summarizeongroups}

Classes

Class `SummarizeOnGroups` {#data-pypeline.Summarize.summarizeongroups.SummarizeOnGroups}

```
class SummarizeOnGroups(
    *on_vars,
    delete_na=True,
    delete_nan=True
)
```

Splits data according to groups (if specified) then paste results together in the form of a DataFrame.

...

Attributes

`__del_na` : `bool` : indicates if we delete NAs or not, by default NA are removed

`__del_nan` : `bool` : indicates if we delete NaNs or not, by default NaNs are removed

Methods

apply(df) : DataFrame The apply method manages the splitting of the data according to the group provided by the user. If no group is specified, we assume that the input data represents a single group. Data are transmitted to the _operation method which carries out the calculations needed (average, min, max and so on). Finally, the apply method reassembles the results before returning them in the form of a DataFrame.

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Summarize.summarizeongroups.SummarizeOnGroups.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Summarize.test` {#data-pypeline.Summarize.test}

Sub-modules

- data-pypeline.Summarize.test.test_average
- data-pypeline.Summarize.test.test_count
- data-pypeline.Summarize.test.test_max
- data-pypeline.Summarize.test.test_min
- data-pypeline.Summarize.test.test_sum
- data-pypeline.Summarize.test.test_variance

Module `data-pypeline.Summarize.test.test_average`

{#data-pypeline.Summarize.test.test_average}

Classes

Class `TestAverage` {#data-pypeline.Summarize.test.test_average.TestAverage}

```
class TestAverage(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Summarize.test.test_average.TestAverage.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_average_var1` {#data-pypeline.Summarize.test.test_average.TestAverage.test_average_var1}

```
def test_average_var1(
    self
)
```

Method `test_average_var1_group_by` {#data-pypeline.Summarize.test.test_average.TestAverage.test_average_var1_group_by}

```
def test_average_var1_group_by(
    self
)
```

Method `test_average_var2` {#data-pypeline.Summarize.test.test_average.TestAverage.test_average_var2}

```
def test_average_var2(
    self
)
```

Method `test_average_var2_group_by` {#data-pypeline.Summarize.test.test_average.TestAverage.test_average_var2_group_by}

```
def test_average_var2_group_by(
    self
)
```

Module `data-pypeline.Summarize.test.test_count` {#data-

pypeline.Summarize.test.test_count}

Classes

Class `TestCount` {#data-pypeline.Summarize.test.test_count.TestCount}

```
class TestCount(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Summarize.test.test_count.TestCount.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_count_var1` {#data-pypeline.Summarize.test.test_count.TestCount.test_count_var1}

```
def test_count_var1(
    self
)
```

Method `test_count_var1_group_by` {#data-pypeline.Summarize.test.test_count.TestCount.test_count_var1_group_by}

```
def test_count_var1_group_by(
    self
)
```

Method `test_count_var2` {#data-pypeline.Summarize.test.test_count.TestCount.test_count_var2}

```
def test_count_var2(
    self
)
```

Method `test_count_var2_group_by` {#data-pypeline.Summarize.test.test_count.TestCount.test_count_var2_group_by}

```
def test_count_var2_group_by(
    self
)
```

Method `test_count_var3_with_nan_na` {#data-pypeline.Summarize.test.test_count.TestCount.test_count_var3_with_nan_na}

```
def test_count_var3_with_nan_na(
    self
```

```
)
```

Method `test_count_var3_without_nan_na` {#data-pypeline.Summarize.test.test_count.TestCount.test_count_var3_without_nan_na}

```
def test_count_var3_without_nan_na(
    self
)
```

Module `data-pypeline.Summarize.test.test_max` {#data-pypeline.Summarize.test.test_max}

Classes

Class `TestMax` {#data-pypeline.Summarize.test.test_max.TestMax}

```
class TestMax(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion

methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Summarize.test.test_max.TestMax.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_max_var1` {#data-pypeline.Summarize.test.test_max.TestMax.test_max_var1}

```
def test_max_var1(
    self
)
```

Method `test_max_var1_group_by` {#data-pypeline.Summarize.test.test_max.TestMax.test_max_var1_group_by}

```
def test_max_var1_group_by(
    self
)
```

Method `test_max_var2` {#data-pypeline.Summarize.test.test_max.TestMax.test_max_var2}

```
def test_max_var2(
    self
)
```

Method `test_max_var2_group_by` {#data-pypeline.Summarize.test.test_max.TestMax.test_max_var2_group_by}

```
def test_max_var2_group_by(
    self
)
```

Module `data-pypeline.Summarize.test.test_min` {#data-pypeline.Summarize.test.test_min}

Classes

Class `TestMin` {#data-pypeline.Summarize.test.test_min.TestMin}

```
class TestMin(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Summarize.test.test_min.TestMin.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_min_var1` {#data-pypeline.Summarize.test.test_min.TestMin.test_min_var1}

```
def test_min_var1(
    self
)
```

Method `test_min_var1_group_by` {#data-pypeline.Summarize.test.test_min.TestMin.test_min_var1_group_by}

```
def test_min_var1_group_by(
    self
)
```

Method `test_min_var2` {#data-pypeline.Summarize.test.test_min.TestMin.test_min_var2}

```
def test_min_var2(
    self
)
```

Method `test_min_var2_group_by` {#data-

pypeline.Summarize.test.test_min.TestMin.test_min_var2_group_by}

```
def test_min_var2_group_by(
    self
)
```

Module `data-pypeline.Summarize.test.test_sum` {#data-pypeline.Summarize.test.test_sum}

Classes

Class `TestSum` {#data-pypeline.Summarize.test.test_sum.TestSum}

```
class TestSum(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert

methods) will be printed on failure in *addition* to any explicit message passed.

- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Summarize.test.test_sum.TestSum.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_sum_var1` {#data-pypeline.Summarize.test.test_sum.TestSum.test_sum_var1}

```
def test_sum_var1(
    self
)
```

Method `test_sum_var1_group_by` {#data-pypeline.Summarize.test.test_sum.TestSum.test_sum_var1_group_by}

```
def test_sum_var1_group_by(
    self
)
```

Method `test_sum_var2` {#data-pypeline.Summarize.test.test_sum.TestSum.test_sum_var2}

```
def test_sum_var2(
    self
)
```

Method `test_sum_var2_group_by` {#data-pypeline.Summarize.test.test_sum.TestSum.test_sum_var2_group_by}

```
def test_sum_var2_group_by(
    self
)
```

Module `data-pypeline.Summarize.test.test_variance` {#data-pypeline.Summarize.test.test_variance}

Classes

Class `TestVariance` {#data-pypeline.Summarize.test.test_variance.TestVariance}

```
class TestVariance(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert

methods) will be printed on failure in *addition* to any explicit message passed.

- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Summarize.test.test_variance.TestVariance.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_sd_var1_without_variance` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_sd_var1_without_variance}

```
def test_sd_var1_without_variance(
    self
)
```

Method `test_sd_var1_without_variance_group_by` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_sd_var1_without_variance_group_by}

```
def test_sd_var1_without_variance_group_by(
    self
)
```

Method `test_sd_var2_without_variance` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_sd_var2_without_variance}

```
def test_sd_var2_without_variance(
    self
)
```

Method `test_sd_var2_without_variance_group_by` {#data-

pypeline.Summarize.test.test_variance.TestVariance.test_sd_var2_without_variance_group_by}

```
def test_sd_var2_without_variance_group_by(
    self
)
```

Method `test_variance_var1_without_sd` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_variance_var1_without_sd}

```
def test_variance_var1_without_sd(
    self
)
```

Method `test_variance_var1_without_sd_group_by` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_variance_var1_without_sd_group_by}

```
def test_variance_var1_without_sd_group_by(
    self
)
```

Method `test_variance_var2_without_sd` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_variance_var2_without_sd}

```
def test_variance_var2_without_sd(
    self
)
```

Method `test_variance_var2_without_sd_group_by` {#data-pypeline.Summarize.test.test_variance.TestVariance.test_variance_var2_without_sd_group_by}

```
def test_variance_var2_without_sd_group_by(
    self
)
```

Module `data-pypeline.Summarize.variance` {#data-pypeline.Summarize.variance}

Classes

Class `Variance` {#data-pypeline.Summarize.variance.Variance}

```
class Variance(
    *on_vars,
    delete_na=True,
    delete_nan=True,
    get_var=True,
    get_sd=True
)
```

Calculates the maximum of a column (of a DataFrame).

...

Attributes

`__get_var` : `bool` : boolean indicating if variance is calculated or not

`__get_sd` : `bool` : boolean indicating if standard deviation is calculated or not

Methods

_operation(col) : dict Calculates the variance and/or standard deviation on the col object, assuming this DataFrame represents a single group. The output variable is called "Variable_Var" if the variance is calculated and called "Variable_SD" for the standard deviation (if col = Variable).

Ancestors (in MRO)

- Summarize.summarizeongroups.SummarizeOnGroups
- Pipeline.onvars.OnVars
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Transform` {#data-pypeline.Transform}

Sub-modules

- data-pypeline.Transform.asnumeric
- data-pypeline.Transform.filter
- data-pypeline.Transform.groupby
- data-pypeline.Transform.join

- data-pypeline.Transform.kmeans
- data-pypeline.Transform.movingaverage
- data-pypeline.Transform.mutate
- data-pypeline.Transform.normalize
- data-pypeline.Transform.rename
- data-pypeline.Transform.select
- data-pypeline.Transform.sort
- data-pypeline.Transform.test
- data-pypeline.Transform.transformongroups
- data-pypeline.Transform.ungroup

Module `data-pypeline.Transform.asnumeric` {#data-pypeline.Transform.asnumeric}

Classes

Class `AsNumeric` {#data-pypeline.Transform.asnumeric.AsNumeric}

```
class AsNumeric(
    *on_vars
)
```

Transform strings to numeric values.

...

Methods

apply(df) : DataFrame Transforms each specified variable of the DataFrame df in a numeric form Raises Exception if any value can't be transformed into an int or a float

__num(val) : int or float Returns val as int if possible, or as float Raises ValueError if val can't be converted into int or float

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.asnumeric.AsNumeric.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Transform.filter` {#data-pypeline.Transform.filter}

Classes

Class `Filter` {#data-pypeline.Transform.filter.Filter}

```
class Filter(
    **criteria
)
```

Filter a DataFrame by taking into account one or more criteria and the group structure.

...

Attributes

self.__criteria : dict Dict of filtering criteria, with keys as variables and values as criteria.

Methods

init(**criteria): Create a Filter object with criteria as key/values pairs. On each pair, the key is the name of the variable and the value is a string of the exact Python syntax of the test, such as "==1" to test if a variable is equal to numeric 1, or "=='Cat'" to test if a variable is equal to string 'Cat'. All of the critera must be checked in order to select a line. Lagged and leaded values are available.

_operation(df): DataFrame Apply the filter to a group DataFrame, and returns a new DataFrame, keeping only the matching rows.

Ancestors (in MRO)

- Transform.transformongroups.TransformOnGroups

- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Transform.groupby` {#data-pypeline.Transform.groupby}

Classes

Class `GroupBy` {#data-pypeline.Transform.groupby.GroupBy}

```
class GroupBy(
    *on_vars
)
```

Add variables to the group structure of a DataFrame.

...

Methods

apply(df) : DataFrame Add the variables of the GroupBy object to the DataFrame df, and returns the new DataFrame

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.groupby.GroupBy.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Transform.join`

{#data-pypeline.Transform.join}

Classes

Class `Join` {#data-pypeline.Transform.join.Join}

```
class Join(
    other,
    **matches
)
```

Performs a left join transformation between two DataFrames.

The right join should be performed by inverting left hand and right hand, the full join by performing both left and right join and computing the union between the results and the inner join by performing both left and right joint and computing the intersection between the results.

...

Methods

init(other, **matches) Create the Join object, with other as right hand DataFrame, and the matching criteria as key/values pairs. The key is the name of a variable in the right hand, and the value is the name of the matching variable in the left hand as string.

apply(df): DataFrame Execute the left join operation with DataFrame df as left hand. Returns a new DataFrame, with all left hand variables and the non matching variables of the right hand DataFrame. If a variable exists in both, the right hand one will be renamed as "Y_Variable".

Define the left join transformation

Parameters

`other` : `DataFrame` : Right hand DataFrame for the juncture

`matches` : `kwargs` : Pairs of matching variables, on the form : RightHand="LeftHand"

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.join.Join.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Transform.kmeans` {#data-pypeline.Transform.kmeans}

Classes

Class `KMeans` {#data-pypeline.Transform.kmeans.KMeans}

```
class KMeans(
    clusters,
    *on_vars,
    normalize=True,
    max_iter=1000,
    random_seed=None
)
```

Use the Lloyd algorithm to perform a K-Means clustering on a DataFrame, by taking into account the group structure.

...

Attributes

self.__clusters : int Number of clusters to define on the partition self.__normalize : bool If true, will normalize data before performing the algorithm self.__max_iter : int Sets the maximum number of iterations for the Lloyd algorithm self.__seed : int If not None, fixes the random seed for the class centers initialization

Methods

init(clusters, *on_vars, normalize=True, max_iter=1000, random_seed=None) Setup the KMeans by defining the numbers of clusters, the variables to use for the classification, and the normalization, iterations and random_seed options. _operation(group_df) : DataFrame Performs the Lloyd algorithm on the group_df DataFrame, and returns a new DataFrame with a column Partition, indicating the partition index (starting at 0) of each row. __euclidean_distance(point_x, point_y) : float Returns the euclidean distance between point_x

and point_y

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Transform.transformongroups.TransformOnGroups
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Transform.movingaverage` {#data-pypeline.Transform.movingaverage}

Classes

Class `MovingAverage` {#data-pypeline.Transform.movingaverage.MovingAverage}

```
class MovingAverage(
    window,
    time_var,
    *on_vars
)
```

Compute a moving average for a list of variables on a DataFrame, by taking into account it's group structure.

...

Attributes

self.__window : int Window size for the moving average, as a number of lines self.__time_var : str Name of the variable giving the time information, to sort the DataFrame

Methods

init(window, time_var, *on_vars) Setup the MovingAverage computation by defining window size, the time variable (time_var) and the variables for which the moving average should be calculated.

_operation(group_df): DataFrame Performs the computation on each group_df DataFrame, by sorting them with respect of the time_var variable, and inserts columns named "Variable_MA" with the results, for example "Val_MA5" for a moving average of window 5 on the variable Val. When a value can't be computed by lack of data, a None value is inserted.

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Transform.transformongroups.TransformOnGroups
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Transform.mutate` {#data-pypeline.Transform.mutate}

Classes

Class `Mutate` {#data-pypeline.Transform.mutate.Mutate}

```
class Mutate(
    **expressions
)
```

Create new variables in a DataFrame by taking into account the group structure.

...

Attributes

self.__expressions : dict Dict of expressions, with keys as new variables names and values as formulas to compute.

Methods

init(**expressions): Create a Mutate object with expressions as key/values pairs. On each pair, the key is the name of the new variable and the value is a string of the exact Python syntax of the formula to create the variable, such as "Var+2" or, "(Var-lag_Var)/lag_Var". Lagged and leaded values are available with prefixes "lag_" and "_lead". The row index value is also available with the variable name "row_index".

_operation(df): DataFrame Apply to a group DataFrame, and returns a new DataFrame, with the new variables added and computed.

Ancestors (in MRO)

- Transform.transformongroups.TransformOnGroups
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Transform.normalize` {#data-pypeline.Transform.normalize}

Classes

Class `Normalize` {#data-pypeline.Transform.normalize.Normalize}

```
class Normalize(
    *on_vars,
    center=True,
    reduce=True
)
```

Normalize variables in a DataFrame, by taking into account the group structure.

...

Attributes

self.__center : bool If True, will center the variable self.__reduce : bool If True, will reduce the variable

Methods

init(*on_vars, center=True, reduce=True) Setup the Normalization process on all listed variables, and define if the variables will be centred and reduced. If both (default), the variable will be normalized. _operation(group_df) : DataFrame Performs the normalization on group_df DataFrame for specified variables and return a new DataFrame with the normalized variables inserted after the original ones, and named "Variable_Std"

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Transform.transformongroups.TransformOnGroups
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Module `data-pypeline.Transform.rename` {#data-pypeline.Transform.rename}

Classes

Class `Rename` {#data-pypeline.Transform.rename.Rename}

```
class Rename(
    **names
)
```

Rename variables in a DataFrame.

...

Attributes

self.__names : dict Dict of renames operations, with keys as new names and values as old names

Methods

init(**names): Create a Rename object with new/old names as key/values pairs. On each pair, the key is the new name of the variable and the value is the current name of the variable as a string.

apply(df): DataFrame Apply the Rename to the DataFrame df, renaming in place each variable from their old name to new name.

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.rename.Rename.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Transform.select` {#data-pypeline.Transform.select}

Classes

Class `Select` {#data-pypeline.Transform.select.Select}

```
class Select(
    *on_vars
)
```

Select some variables from a DataFrame

...

Methods

apply(df) : DataFrame Create and return a new DataFrame containing only the selected variables of df, ordered in the order they were added on the Select object

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.select.Select.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Transform.sort` {#data-pypeline.Transform.sort}

Classes

Class `Sort` {#data-pypeline.Transform.sort.Sort}

```
class Sort(
    *on_vars
)
```

Implements the merge sort algorithm for DataFrame transformation.

...

Attributes

self._*vars : str Ordered list of the sorting vars criteria. Use the "desc*" prefix before the name of a variable to perform a descending sorting on this variable.

Methods

_operation(group_df) : DataFrame Execute the sorting algorithm on the group_df object, assuming this DataFrame represents a single group

__merge_sort(nested_list, index_criteria) : list Entry point for the merge sort algorithm based on a nested list (first level : rows, second level : columns) and the list of sorting criteria columns indexes. This list is shifted of 1 (i.e. for the real index 5, the index_critera will be 6) and negative for descending sorting (i.e. a descending sorting on the column 0 will be coded -1).

__merge(nested_list_a, nested_list_b, index_criteria) : list Performs the merge step of the merge sort algorithm between two ordered nested lists.

__compare_lists(list_a, list_b, index_criteria) : bool Will be True if list_a should be inserted before list_b and False on the contrary, based on the variables defined by index_criteria

Ancestors (in MRO)

- Pipeline.onvars.OnVars
- Transform.transformongroups.TransformOnGroups
- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable

- abc.ABC

Module `data-pypeline.Transform.test` {#data-pypeline.Transform.test}

Sub-modules

- data-pypeline.Transform.test.test_asnumeric
- data-pypeline.Transform.test.test_filter
- data-pypeline.Transform.test.test_groupby
- data-pypeline.Transform.test.test_join
- data-pypeline.Transform.test.test_kmeans
- data-pypeline.Transform.test.test_movingaverage
- data-pypeline.Transform.test.test_mutate
- data-pypeline.Transform.test.test_normalize
- data-pypeline.Transform.test.test_rename
- data-pypeline.Transform.test.test_select
- data-pypeline.Transform.test.test_sort
- data-pypeline.Transform.test.test_ungroup

Module `data-pypeline.Transform.test.test_asnumeric` {#data-pypeline.Transform.test.test_asnumeric}

Classes

Class `TestAsNumeric` {#data-pypeline.Transform.test.test_asnumeric.TestAsNumeric}

```
class TestAsNumeric(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_asnumeric.TestAsNumeric.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_mixedVar` {#data-pypeline.Transform.test.test_asnumeric.TestAsNumeric.test_mixedVar}

```
def test_mixedVar(
    self
```

```
)
```

Module `data-pypeline.Transform.test.test_filter` {#data-pypeline.Transform.test.test_filter}

Classes

Class `TestFilter` {#data-pypeline.Transform.test.test_filter.TestFilter}

```
class TestFilter(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if

required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_filter.TestFilter.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_multipleEqualFilter` {#data-pypeline.Transform.test.test_filter.TestFilter.test_multipleEqualFilter}

```
def test_multipleEqualFilter(
    self
)
```

Method `test_simpleEqualFilter` {#data-pypeline.Transform.test.test_filter.TestFilter.test_simpleEqualFilter}

```
def test_simpleEqualFilter(
    self
)
```

Method `test_simpleNonEqualFilter` {#data-pypeline.Transform.test.test_filter.TestFilter.test_simpleNonEqualFilter}

```
def test_simpleNonEqualFilter(
    self
)
```

Module `data-pypeline.Transform.test.test_groupby`

{#data-pypeline.Transform.test.test_groupby}

Classes

Class `TestGroupBy` {#data-pypeline.Transform.test.test_groupby.TestGroupBy}

```
class TestGroupBy(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_groupby.TestGroupBy.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_setOneGroup` {#data-pypeline.Transform.test.test_groupby.TestGroupBy.test_setOneGroup}

```
def test_setOneGroup(
    self
)
```

Method `test_setTwoGroups` {#data-pypeline.Transform.test.test_groupby.TestGroupBy.test_setTwoGroups}

```
def test_setTwoGroups(
    self
)
```

Module `data-pypeline.Transform.test.test_join` {#data-pypeline.Transform.test.test_join}

Classes

Class `TestJoin` {#data-pypeline.Transform.test.test_join.TestJoin}

```
class TestJoin(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_join.TestJoin.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_joinTables` {#data-pypeline.Transform.test.test_join.TestJoin.test_joinTables}

```
def test_joinTables(
    self
)
```

Module `data-pypeline.Transform.test.test_kmeans` {#data-pypeline.Transform.test.test_kmeans}

Classes

Class `TestKMeans` {#data-pypeline.Transform.test.test_kmeans.TestKMeans}

```
class TestKMeans(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert

methods) will be printed on failure in *addition* to any explicit message passed.

- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_kmeans.TestKMeans.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_kmeans` {#data-pypeline.Transform.test.test_kmeans.TestKMeans.test_kmeans}

```
def test_kmeans(
    self
)
```

Module `data-pypeline.Transform.test.test_movingaverage` {#data-pypeline.Transform.test.test_movingaverage}

Classes

Class `TestMovingAverage` {#data-pypeline.Transform.test.test_movingaverage.TestMovingAverage}

```
class TestMovingAverage(
```

```
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_movingaverage.TestMovingAverage.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_group_ma` {#data-pypeline.Transform.test.test_movingaverage.TestMovingAverage.test_group_ma}

```
def test_group_ma(
    self
)
```

Method `test_ungroup_ma` {#data-pypeline.Transform.test.test_movingaverage.TestMovingAverage.test_ungroup_ma}

```
def test_ungroup_ma(
    self
)
```

Module `data-pypeline.Transform.test.test_mutate` {#data-pypeline.Transform.test.test_mutate}

Classes

Class `TestMutate` {#data-pypeline.Transform.test.test_mutate.TestMutate}

```
class TestMutate(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called.

It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_mutate.TestMutate.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_compute_var` {#data-pypeline.Transform.test.test_mutate.TestMutate.test_compute_var}

```
def test_compute_var(
    self
)
```

Method `test_leads_with_groups` {#data-pypeline.Transform.test.test_mutate.TestMutate.test_leads_with_groups}

```
def test_leads_with_groups(
    self
)
```

Method `test_vars_with_lag` {#data-pypeline.Transform.test.test_mutate.TestMutate.test_vars_with_lag}

```
def test_vars_with_lag(
    self
)
```

Module `data-pypeline.Transform.test.test_normalize` {#data-pypeline.Transform.test.test_normalize}

Classes

Class `TestNormalize` {#data-pypeline.Transform.test.test_normalize.TestNormalize}

```
class TestNormalize(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than

'errored'.

- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_normalize.TestNormalize.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_center` {#data-pypeline.Transform.test.test_normalize.TestNormalize.test_center}

```
def test_center(
    self
)
```

Method `test_normalize` {#data-pypeline.Transform.test.test_normalize.TestNormalize.test_normalize}

```
def test_normalize(
    self
)
```

Method `test_reduce` {#data-pypeline.Transform.test.test_normalize.TestNormalize.test_reduce}

```
def test_reduce(
    self
)
```

Module `data-pypeline.Transform.test.test_rename` {#data-pypeline.Transform.test.test_rename}

Classes

Class `TestRename` {#data-pypeline.Transform.test.test_rename.TestRename}

```
class TestRename(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_rename.TestRename.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_rename` {#data-pypeline.Transform.test.test_rename.TestRename.test_rename}

```
def test_rename(
    self
)
```

Module `data-pypeline.Transform.test.test_select` {#data-pypeline.Transform.test.test_select}

Classes

Class `TestSelect` {#data-pypeline.Transform.test.test_select.TestSelect}

```
class TestSelect(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed.

When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_select.TestSelect.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_multiple_vars` {#data-pypeline.Transform.test.test_select.TestSelect.test_multiple_vars}

```
def test_multiple_vars(
    self
```

```
)
```

Method `test_one_var` {#data-pypeline.Transform.test.test_select.TestSelect.test_one_var}

```
def test_one_var(
    self
)
```

Module `data-pypeline.Transform.test.test_sort` {#data-pypeline.Transform.test.test_sort}

Classes

Class `TestMutate` {#data-pypeline.Transform.test.test_sort.TestMutate}

```
class TestMutate(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion

methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.

- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_sort.TestMutate.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_multiple_sort` {#data-pypeline.Transform.test.test_sort.TestMutate.test_multiple_sort}

```
def test_multiple_sort(
    self
)
```

Method `test_simple_desc_sort` {#data-pypeline.Transform.test.test_sort.TestMutate.test_simple_desc_sort}

```
def test_simple_desc_sort(
    self
)
```

Method `test_simple_sort` {#data-pypeline.Transform.test.test_sort.TestMutate.test_simple_sort}

```
def test_simple_sort(
    self
)
```

Module `data-pypeline.Transform.test.test_ungroup` {#data-pypeline.Transform.test.test_ungroup}

Classes

Class `TestUngroup` {#data-pypeline.Transform.test.test_ungroup.TestUngroup}

```
class TestUngroup(
    methodName='runTest'
)
```

A class whose instances are single test cases.

By default, the test code itself should be placed in a method named 'runTest'.

If the fixture may be used for many test cases, create as many test methods as are needed. When instantiating such a TestCase subclass, specify in the constructor arguments the name of the test method that the instance is to execute.

Test authors should subclass TestCase for their own tests. Construction and deconstruction of the test's environment ('fixture') can be implemented by overriding the 'setUp' and 'tearDown' methods respectively.

If it is necessary to override the **init** method, the base class **init** method must always be called. It is important that subclasses should not change the signature of their **init** method, since instances of the classes are instantiated automatically by parts of the framework in order to be run.

When subclassing TestCase, you can set these attributes:

- failureException: determines which exception will be raised when the instance's assertion methods fail; test methods raising this exception will be deemed to have 'failed' rather than 'errored'.
- longMessage: determines whether long messages (including repr of objects used in assert methods) will be printed on failure in *addition* to any explicit message passed.
- maxDiff: sets the maximum length of a diff in failure messages by assert methods using difflib. It is looked up as an instance attribute so can be configured by individual tests if required.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Ancestors (in MRO)

- unittest.case.TestCase

Methods

Method `setUp` {#data-pypeline.Transform.test.test_ungroup.TestUngroup.setUp}

```
def setUp(
    self
)
```

Hook method for setting up the test fixture before exercising it.

Method `test_ungroup` {#data-pypeline.Transform.test.test_ungroup.TestUngroup.test_ungroup}

```
def test_ungroup(
    self
)
```

Module `data-pypeline.Transform.transformongroups` {#data-pypeline.Transform.transformongroups}

Classes

Class `TransformOnGroups` {#data-pypeline.Transform.transformongroups.TransformOnGroups}

```
class TransformOnGroups
```

Splits data according to groups (if specified) then paste results together in the form of a DataFrame.

...

Methods

apply(df) : DataFrame The apply method manages the splitting of the data according to the group provided by the user. If no group is specified, we assume that the input data represents a single group. Data are transmitted to the _operation method which carries out the calculations needed. Finally, the apply method reassembles the results before returning them in the form of a single DataFrame.

Ancestors (in MRO)

- Pipeline.ongroups.OnGroups
- Pipeline.pipelineable.Pipelineable
- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.transformongroups.TransformOnGroups.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.Transform.ungroup` {#data-pypeline.Transform.ungroup}

Classes

Class `Ungroup` {#data-pypeline.Transform.ungroup.Ungroup}

```
class Ungroup
```

Removes the group structure of a DataFrame.

...

Methods

apply(df) : DataFrame Returns a copy of the df DataFrame without any group structure

Ancestors (in MRO)

- Pipeline.pipelineable.Pipelineable

- abc.ABC

Methods

Method `apply` {#data-pypeline.Transform.ungroup.Ungroup.apply}

```
def apply(
    self,
    df
)
```

Module `data-pypeline.main` {#data-pypeline.main}

Module `data-pypeline.setup` {#data-pypeline.setup}

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).