Dal.io

Release 0.1.1

Renato Zimmermann

Jul 15, 2020

USER MODULES:

CHAPTER

ONE

DALIO.EXTERNAL PACKAGE

1.1 Submodules

1.2 dalio.external.external

Define abstract External class

External instances manage connections between your environment and an external source. Class instacnes will often be redundant with existing connection handlers, but at least subclasses will allow for more integrated connection handling and collection, so that you can have a single connection object for each external connection.

class `dalio.external.external.`**`External`**`(`*config=None*`)`

Bases: `dalio.base.node._Node`

Represents external data input or output

External instances have one external input and one internal output or one internal input and one external output.

`_connection`

connection with outside source of data

`_config`

authentication settings for outside sources

Type dict

`authenticate``()`

Establish a connection with the source.

Returns True if authenication is successful or if it is already existent False if the authentication fails.

`check``()`

Check if connection is ready to request data

Returns Whether data is ready to be requested

`request``(`***kwargs*`)`

Request data to or from an external source

`update_config``(`*new_conf*`)`

Update configuration dict with new data

Parameters **`new_conf`** – dictionary with new configurations or file containing configuration settings translatable to a dictionary

Raises **`TypeError`** – if config is a non-existent file or not a dict.

1.3 dalio.external.file

Define File IO classes

Files are external sources of data that can be processed in several ways as raw data used in a graph.

class `dalio.external.file.`**`FileWriter`**(*out_file=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Bases: *`dalio.external.external.External`*

File string writer

`_connection`

any file instance that can be written on

`check`()

Check if there is an open file as the connection

`request`(***kwargs*)

Write a request string onto a file

`set_connection`(*new_connection*)

Set current connection

Set connection to opened file or open a new file given the path to one.

Parameters **`new_connection`** – open file instance or path to an existing file.

Raises

- **`IOError`** – if specified path does not exist.
- **`TypeError`** – if specified "new_connection" argument is of an invalid type

class `dalio.external.file.`**`PandasInFile`**(*in_file*)

Bases: *`dalio.external.external.External`*

Get data from a file using the pandas package

`_connection`

path to a file that can be read by some pandas function.

Type str

`check`()

Check if connection is ready to request data

Returns Whether data is ready to be requested

`request`(***kwargs*)

Get data input from a file according to its extension

Parameters **`**kwargs`** – arguments to the inport function.

1.4 dalio.external.image

Define classes for image pieces

Images, be it a plot, picture or video are considered external outputs as the figure itself is not contained in the python session, and must be shown in a screen or server.

class `dalio.external.image.`**`PyPfOptGraph`**(*figsize=None*)

Bases: `dalio.external.image.PyPlotGraph`

Graphs data from the PyPfOpt package

`plot`(*data*, *coords=None*, *kind=None*, ***kwargs*)

Graph data from pypfopt

Parameters **`data`** – plottable data from pypfopt package

Raises **`TypeError`** – if data is not of a plottable class from pypfopt

class `dalio.external.image.`**`PyPlotGraph`**(*figsize=None*)

Bases: `dalio.external.image._Figure`

Figure from the matplotlib.pyplot package.

`_connection`

graph figure

Type matplotlib.pyplot.Figure

`_axes`

figure axis

Type matplotlib.axes._subplots.AxesSubplot

`plot`(*data*, *kind=None*, ***graph_opts*)

Plot x onto the x-axis and y onto the y-axis, if applicable.

Parameters

- **`data`** (`matrix or array like`) – either data to be plotted on the x axis or a tuple of x and y data to be plotted or the x and y axis.
- **`kind`** (`str`) – kind of graph.
- **`**graph_opts`** – plt plotting arguments for this kind of graph.

`request`(***kwargs*)

Processed request for data.

This adds the SHOW request to the base class implementation

`reset`()

Set connection and axes to a single figure and axis

class `dalio.external.image.`**`PySubplotGraph`**(*rows*, *cols*, *figsize=None*)

Bases: `dalio.external.image._MultiFigure`

A matplotlib.pyplot.Figure containing multiple subplots.

This has a set number of axes, rows and columns which can be accessed individually to have data plotted on. These will often be used inside of applications that require more than one subplot all contained in the same instance.

`_rows`

number of rows in the subplot

Type int

`_cols`
number of columns in the subplot

Type int

`_loc`
array of the figure's axes

Type np.array

`get_loc`(*coords*)
Gets a specific axis from the _loc attribute at given coordinates

`make_manager`(*coords*)
Create a SubPlotManager to manage this instance's subplots

`plot`(*data*, *coords=None*, *kind=None*, ***graph_opts*)
Plot on a specified subplot axis

Parameters **`coords`** (*`tuple`*) – tuple of subplot coordinates to plot data

Raises **`ValueError`** – if coordinates are out of range.

`reset`()
Resets figure and all axes

class `dalio.external.image.`**`SubplotManager`**(*subplot*, *coords*)
Bases: *`dalio.external.image.PyPlotGraph`*

A manager object for treating a subplot axis like a single plot.

Applications will often take in single plots and have their functionality catered to such. Subplots, while useful, will often be used for specific applications. A subplot manager allows you to create multiple subplots and pass each one individually onto applications that take a single subplot axis and still have access to the underlying figure.

`reset`()
Set connection and axes to a single figure and axis

1.5 dalio.external.web

Define web external request classes

class `dalio.external.web.`**`QuandlAPI`**(*config=None*)
Bases: *`dalio.external.external.External`*

Set up the Quandl API and request table data from quandl.

`_quandl_conf`
Quandl API config object

`authenticate`()
Set the api key if it is available in the config dictionary

Returns True if key was successfully set, False otherwise

`check`()
Check if the api key is set

`request`(***kwargs*)
Request table data from quandl

Parameters `**kwargs` – keyword arguments for quandl request. query: table to get data from. filter: dictionary of filters for data. Depends on table. columns: columns to select.

Raises

- `IOError` – if api key is not set.
- `ValueError` – if filters kwarg is not a dict.

class `dalio.external.web.`**`YahooDR`**`(`*config=None*`)`

Bases: `dalio.external.web._PDR`

Represents financial data from Yahoo! Finance

`request``(`***kwargs*`)`

Get data from specified tickers

DALIO.TRANSLATOR PACKAGE

2.1 Submodules

2.2 dalio.translator.file

Translator for common file imports

These will often be very specific to the file being imported, but should strive to still be as flexible as possible. These will often hold the format translated to constant and try being adaptable with the data to fit it. So it is more importat to begin with the output and then adapt to the input, not the other way.

class `dalio.translator.file.`**`StockStreamFileTranslator`**(*date_col=None*, *att_name=None*)

Bases: *`dalio.translator.translator.Translator`*

Create a DataFrame conforming to the STOCK_STREAM validator preset.

The STOCK_STREAM preset includes:

a) having a time series index,

b) being a dataframe,

c) **having a multiindex column with levels named ATTRIBUTE and TICKER.** Such that an imported excel file will have column names renamed that or assume a single column name row is of ticker names.

`date_col`

column name to get date data from.

Type str

`att_name`

name of the attribute column if imported dataframe column has only one level.

Type str

`copy`(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy

- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

`run`(***kwargs*)
Request pandas data from file and format it into a dataframe that complies with the STOCK_STREAM validator preset

Parameters **`**kwargs`** – Optional request arguments TICKER: single ticker or iterable of tickers to filter for

in data.

`translations = None`

2.3 dalio.translator.pdr

Define translators for data from the pandas_datareader package

class `dalio.translator.pdr.`**`YahooStockTranslator`**
Bases: *`dalio.translator.translator.Translator`*

Translate stock data gathered from Yahoo! Finance

`run`(***kwargs*)
Request data subset and translate columns

Parameters **`**kwargs`** – optional run arguments. TICKER: ticker to get data from.

`translations = None`

2.4 dalio.translator.quandl

Define Translator instances for data imported from quandl.

These should be designed with both input and output in mind as quandl inputs can, for a good extent, known from the table and query, both of which are known from the time of request. This means that these translators should be designed to be more specific to the query instead of being flexible.

class `dalio.translator.quandl.`**`QuandlSharadarSF1Translator`**
Bases: *`dalio.translator.translator.Translator`*

Import and translate data from the SHARADAR/SF1 table

`run`(***kwargs*)
Get input from quandl's SHARADAR/SF1 table, and format according to the STOCK_STREAM validator preset.

`translations = None`

class `dalio.translator.quandl.`**`QuandlTickerInfoTranslator`**
Bases: *`dalio.translator.translator.Translator`*

Import and translate data from the SHARADAR/TICKERS table

`run`(***kwargs*)
Get input from quandl's SHARADAR/TICKER table, and format according to the STOCK_INFO validator preset.

translations = None

2.5 dalio.translator.translator

Define Translator class

Translators are the root of all data that feeds your graph. Objects of this take in data from some external source then "translates" it into a format that can be used universaly by other elements in this package. Please consult the translation manual to make this as usabel as possible and make extensive use of the base tools to build translations.

class `dalio.translator.translator.`**`Translator`**

Bases: `dalio.base.transformer._Transformer`

`_source`

Connection used to retrieve raw data from outide source.

`translations`

dictionary of translations from vocabulaary used in the data source to base constants. These should be created from initialization and kept unmodified. This is to ensure data coming through a translator is though of before usage to ensure integrity.

`copy`(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

`set_input`(*new_input*)

See base class

`translate_item`(*item*)

Translate all items of an iterable

Parameters **`item`** (*`dict, any`*) – item or iterator of items to translate.

Returns A list with the translated names.

`translations: Dict[str, str] = None`

`update_translations`(*new_translations*)

Update translations dictionary with new dictrionary

`with_input`(*new_input*)

See base class

CHAPTER

THREE

DALIO.PIPE PACKAGE

3.1 Submodules

3.2 dalio.pipe.builders

Builder Pipes

class dalio.pipe.builders.**CovShrink**(*frequency=252*)

Bases: *dalio.pipe.pipe.PipeBuilder*

Perform Covariance Shrinkage on data

Builder with a single piece: shirnkage. Shrinkage defines what kind of shrinkage to apply on a resultant covariance matrix. If none is set, covariance will not be shrunk.

frequency

data time period frequency

Type int

build_model(*data*, ***kwargs*)

Builds Covariance Srhinkage object and returns selected shrinkage strategy

Returns Function fitted on the data.

check_name(*param*, *name*)

Check if name and parameter combination is valid.

This will always be called upon setting a new piece to ensure this piece is present dictionary and that the name is valid. Subclasses will often override this method to implement the name checks in accordance to their specific name parameter combination options. Notice that checks cannot be done on arguments before running the _Builder. This also can be called from outside of a _Builder instance to check for the validity of settings.

Parameters

- **piece** (*str*) – name of the key in the piece dictionary.
- **name** (*str*) – name option to be set to the piece.

copy(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- ***args** – Positional arguments to be passed to initialize copy
- ****kwargs** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

frequency: int = None

transform(*data*, ***kwargs*)
Build model using data get results.

Returns A covariance matrix

class `dalio.pipe.builders.`**ExpectedReturns**
Bases: *dalio.pipe.pipe.PipeBuilder*

Get stock's time series expected returns.

Builder with a single piece: return_model. return_model is what model to get the expected returns from.

build_model(*data*, ***kwargs*)
Assemble pieces into a model given some data

The data will opten be optional, but several builder models will require it to be fitted on initialization. Which further shows why builders are necessary for context-agnostic graphs.

Parameters

- **data** – data that might be used to build the model.
- ****kwargs** – any additional argument used in building

check_name(*param*, *name*)
Check if name and parameter combination is valid.

This will always be called upon setting a new piece to ensure this piece is present dictionary and that the name is valid. Subclasses will often override this method to implement the name checks in accordance to their specific name parameter combination options. Notice that checks cannot be done on arguments before running the _Builder. This also can be called from outside of a _Builder instance to check for the validity of settings.

Parameters

- **piece** (*str*) – name of the key in the piece dictionary.
- **name** (*str*) – name option to be set to the piece.

transform(*data*, ***kwargs*)
Builds model using data and gets expected returns from it

class `dalio.pipe.builders.`**ExpectedShortfall**(*quantiles=None*)
Bases: *dalio.pipe.builders.ValueAtRisk*

Get expected shortfal for given quantiles

See base class for more in depth explanation.

transform(*data*, ***kwargs*)
Get the value at risk given by an arch model and calculate the expected shortfall at given quantiles.

class `dalio.pipe.builders.`**MakeARCH**
Bases: *dalio.pipe.pipe.PipeBuilder*

Build arch model and make it based on input data.

This class allows for the creation of arch models by configuring three pieces: the mean, volatility and distribution. These are set after initialization through the _Builder interface.

`_piece`
see _Builder class.

Type list

`assimilate`(*model*)
Assimilate core pieces of an existent ARCH Model.

Assimilation means setting this model's' pieces in accordance to an existing model's pieces. Assimilation is shallow, so only the main pieces are assimilated, not their parameters.

Parameters **`model`** (*`ARCHModel`*) – Existing ARCH Model.

`build_model`(*data*, ***kwargs*)
Build ARCH Model using data, set pieces and their arguments

Returns A built arch model from the arch package.

`transform`(*data*, ***kwargs*)
Build model with sourced data

class `dalio.pipe.builders.`**`OptimumWeights`**
Bases: *`dalio.pipe.pipe.PipeBuilder`*

Get optimum portfolio weights from an efficient frontier or CLA. This is also a builder with one piece: strategy. The strategy piece refers to the optimization strategy.

`build_model`(*data*, ***kwargs*)
Assemble pieces into a model given some data

The data will opten be optional, but several builder models will require it to be fitted on initialization. Which further shows why builders are necessary for context-agnostic graphs.

Parameters

- **`data`** – data that might be used to build the model.
- **`**kwargs`** – any additional argument used in building

`check_name`(*param*, *name*)
Check if name and parameter combination is valid.

This will always be called upon setting a new piece to ensure this piece is present dictionary and that the name is valid. Subclasses will often override this method to implement the name checks in accordance to their specific name parameter combination options. Notice that checks cannot be done on arguments before running the _Builder. This also can be called from outside of a _Builder instance to check for the validity of settings.

Parameters

- **`piece`** (*`str`*) – name of the key in the piece dictionary.
- **`name`** (*`str`*) – name option to be set to the piece.

`transform`(*data*, ***kwargs*)
Get efficient frontier, fit it to model and get weights

class `dalio.pipe.builders.`**`PandasLinearModel`**
Bases: *`dalio.pipe.pipe.PipeBuilder`*

Create a linear model from input pandas dataframe, using its index as the X value.

This builder is made up of a single piece: strategy. This piece sets which linear model should be used to fit the data.

build_model(*data*, ***kwargs*)
Build model by returning the chosen model and initialization parameters

Returns Unfitted linear model

transform(*data*, ***kwargs*)
Set up fitting parameters and fit built model.

Returns Fitted linear model

class dalio.pipe.builders.**StockComps**(*strategy='sic_code'*, *max_ticks=6*)
Bases: *dalio.pipe.pipe.Pipe*

Get a list of a ticker's comparable stocks

This can utilize any strategy of getting stock comparative companies and return up to a certain ammount of comps.

_strategy
comparisson strategy name or function.

Type str, callable

max_ticks
maximum number of tickers to return.

Type int

copy(**args*, ***kwargs*)
Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- ***args** – Positional arguments to be passed to initialize copy
- ****kwargs** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

max_ticks: int = None

run(***kwargs*)
Gets ticker argument and passes an empty ticker request to transform.

Empty ticker requests are supposed to return all tickers available in a source, so this allows the compariisson to be made in all stocks from a certain source.

Raises **ValueError** – if ticker is more than a single symbol.

transform(*data*, ***kwargs*)
Get comps according to the set strategy

class dalio.pipe.builders.**ValueAtRisk**(*quantiles=None*)
Bases: *dalio.pipe.pipe.Pipe*

Get the value at risk for data based on an ARHC Model

This takes in an ARCH Model maker, not data, which might be unintuitive, yet necessary, as this allows users to modify the ARCH model generating these values separately. A useful strategy that allows for this is using a pipeline with an arch model as its first input and a ValueAtRisk instance as its second layer. This allows us to treat the PipeLine as a data input with VaR output and still have control over the ARCH Model pieces (given you left a local variable for it behind.)

`_quantiles`
list of quantiles to check the value at risk for.

Type list

`copy`(**args*, ***kwargs*)
Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

`transform`(*data*, ***kwargs*)
Get values at risk at each quantile and each results maximum exedence from the mean.

The maximum exedence columns tells which quantile the loss is placed on. The word "maximum" might be misleading as it is compared to the minimum quantile, however, this definition is accurate as the column essentially answers the question: "what quantile furthest away from the mean does the data exeed?"

Thank you for the creators of the arch package for the beautiful visualizations and ideas!

Raises

- **`ValueError`** – if ARCH model does not have returns. This is often the case for unfitted models. Ensure your graph is complete.
- **`TypeError`** – if ARCH model has unsuported distribution parameter.

3.3 dalio.pipe.col_generation

Implement transformations that generates new colums from exising ones

class `dalio.pipe.col_generation.`**`Bin`**(*bin_map*, **args*, *bin_strat='normal'*, *columns=None*, *new_cols=None*, *drop=True*, *reintegrate=False*, ***kwargs*)
Bases: *`dalio.pipe.col_generation.Custom`*

A pipeline stage that adds a binned version of a column or columns.

If drop is set to True the new columns retain the names of the source columns; otherwise, the resulting column gain the suffix '_bin'

`bin_map`
implicitly projects a left-most bin containing all elements smaller than the left-most end point and a right-most bin containing all elements larger that the right-most end point. For example, the list [0, 5, 8] is interpreted as the bins ($-\infty$, 0), [0-5), [5-8) and [8, ∞).

Type array-like

bin_strat
binning strategy to use. "normal" uses the default binning strategy per a list of value separations or number of bins. "quantile" uses a list of quantiles or a preset quantile range (4 for quartiles and 10 for deciles).

Type str, default "normal"

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[-3],[4],[5], [9]], [1,2,3, 4], ['speed'])
>>> pdp.Bin({'speed': [5]}, drop=False).apply(df)
   speed speed_bin
1     -3        <5
2      4        <5
3      5         5
4      9         5
>>> pdp.Bin({'speed': [0,5,8]}, drop=False).apply(df)
   speed speed_bin
1     -3        <0
2      4       0-5
3      5       5-8
4      9         8
```

class dalio.pipe.col_generation.**BoxCox**(**args, columns=None, new_cols=None, non_neg=False, const_shift=None, drop=True, reintegrate=False, **kwargs*)
Bases: *dalio.pipe.col_generation.Custom*

A pipeline stage that applies the BoxCox transformation on data.

const_shift
If given, each transformed column is first shifted by this constant. If non_neg is True then that transformation is applied first, and only then is the column shifted by this constant.

Type int, optional

class dalio.pipe.col_generation.**Change**(**args, strategy='diff', columns=None, new_cols=None, drop=True, reintegrate=False, **kwargs*)
Bases: dalio.pipe.col_generation._ColGeneration

Perform item-by-item change

This has two main forms, percentage change and absolute change (difference).

_strategy
change strategy.

Type str, callable

copy(**args, **kwargs*)
Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

class `dalio.pipe.col_generation.`**`Custom`**(*func*, **args*, *columns=None*, *new_cols=None*, *strategy='apply'*, *axis=0*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: `dalio.pipe.col_generation._ColGeneration`

Apply custom function.

`strategy`

strategy for applying value function. One of [“apply”, “transform”, “agg”, “pipe”]

Type str, default “pipe”

`copy`(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class’ copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

class `dalio.pipe.col_generation.`**`CustomByCols`**(*func*, **args*, *strategy='apply'*, *columns=None*, *new_cols=None*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: *`dalio.pipe.col_generation.Custom`*

A pipeline stage applying a function to individual columns iteratively.

`func`

The function to be applied to each element of the given columns.

Type function

`strategy`

Application strategy. Different from Custom class’ strategy parameter (which here is kept at “apply”) as this will now be done on a series (each column). Extra care should be taken to ensure resulting column lengths match.

Type str

Example

```
>>> import pandas as pd; import pdpipe as pdp; import math;
>>> data = [[3.2, "acd"], [7.2, "alk"], [12.1, "alk"]]
>>> df = pd.DataFrame(data, [1,2,3], ["ph","lbl"])
>>> round_ph = pdp.ApplyByCols("ph", math.ceil)
>>> round_ph(df)
   ph  lbl
1   4  acd
2   8  alk
3  13  alk
```

class `dalio.pipe.col_generation.`**`Index`**(*index_at*, **args*, *columns=None*, *new_cols=None*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: `dalio.pipe.col_generation._ColGeneration`

`copy`(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

class `dalio.pipe.col_generation.`**`Log`**(**args*, *columns=None*, *new_cols=None*, *non_neg=False*, *const_shift=None*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: *`dalio.pipe.col_generation.Custom`*

A pipeline stage that log-transforms numeric data.

`non_neg`

If True, each transformed column is first shifted by smallest negative value it includes (non-negative columns are thus not shifted).

Type bool, default False

`const_shift`

If given, each transformed column is first shifted by this constant. If non_neg is True then that transformation is applied first, and only then is the column shifted by this constant.

Type int, optional

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> data = [[3.2, "acd"], [7.2, "alk"], [12.1, "alk"]]
>>> df = pd.DataFrame(data, [1,2,3], ["ph","lbl"])
>>> log_stage = pdp.Log("ph", drop=True)
>>> log_stage(df)
         ph  lbl
1  1.163151  acd
2  1.974081  alk
3  2.493205  alk
```

class `dalio.pipe.col_generation.`**`MapColVals`**(*value_map*, **args*, *columns=None*, *new_cols=None*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: *`dalio.pipe.col_generation.Custom`*

A pipeline stage that reintegrates the values of a column by a map.

`value_map`

A dictionary mapping existing values to new ones. Values not in the dictionary as keys will be converted to NaN. If a function is given, it is applied element-wise to given columns. If a Series is given, values are mapped by its index to its values.

Type dict, function or pandas.Series

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1], [3], [2]], ['UK', 'USSR', 'US'], ['Medal'])
>>> value_map = {1: 'Gold', 2: 'Silver', 3: 'Bronze'}
>>> pdp.MapColVals('Medal', value_map).apply(df)
       Medal
UK      Gold
USSR  Bronze
US    Silver
```

class `dalio.pipe.col_generation.`**`Period`**(*period*, **args*, *agg_func=<function mean>*, *columns=None*, *new_cols=None*, *axis=0*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: `dalio.pipe.col_generation._ColGeneration`

Resample input time series data to a different period

Attributes: agg_func (callable): function to aggregate data to one period.

# Quandl Input

Default set to np.mean.

_period (str): period to resample data to. Can be either daily, monthly, quarterly or yearly.

`agg_func: Callable[[Iterable], Any] = None`

`copy`(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

class `dalio.pipe.col_generation.`**`Rolling`**(*func*, **args*, *columns=None*, *new_cols=None*, *rolling_window=2*, *axis=0*, *drop=True*, *reintegrate=False*, ***kwargs*)

Bases: `dalio.pipe.col_generation._ColGeneration`

Apply rolling function

`rolling_window`

rolling window to apply function. If none, no rolling window is applied.

Type int, defailt None

`copy`(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

class `dalio.pipe.col_generation.`**`StockReturns`**(*columns=None*, *new_cols=None*, *drop=True*, *reintegrate=False*)

Bases: `dalio.pipe.col_generation._ColGeneration`

Perform percent change and minor aesthetic changes to data

3.4 dalio.pipe.custom

3.5 dalio.pipe.forecast

Transformations makes forecasts based on data

class `dalio.pipe.forecast.`**`Forecast`**(*horizon=10*)

Bases: *`dalio.pipe.pipe.Pipe`*

Generalized forecasting class.

This should be used mostly for subclassing or very generic forecasting interfaces.

`horizon`

how many steps ahead to forecast

Type int

horizon: int = None

transform(*data*, ***kwargs*)
Return forecast of data

class dalio.pipe.forecast.**GARCHForecast**(*start=None*, *horizon=1*)
Bases: *dalio.pipe.forecast.Forecast*

Forecast data based on a fitted GARCH model

_start
forecast start time and date.

Type pd.Timestamp

transform(*data*, ***kwargs*)
Make a mean, variance and residual variance forecast.

Forecast will be made for the specified horizon starting at the specified time. This means that will only get data for the steps starting at the specified start date and the steps after it.

Returns A DataFrame with the columns MEAN, VARIANCE and RESIDUAL_VARIANCE for the time horizon after the start date.

3.6 dalio.pipe.pipe

Defines the Pipe and PipeLine classes

Pipes are perhaps the most common classes in graphs and represent any transformation with one input and one output. Pipes` main functionality revolves around the .transform() method, which actually applies a transformation to data retrieved from a source. Pipes must also implement propper data checks by adding descriptions to their source.

class dalio.pipe.pipe.**Pipe**
Bases: dalio.base.transformer._Transformer

Pipes represend data modifications with one internal input and one internal output.

_source
input data definition

Type _DataDef

copy(**args*, ***kwargs*)
Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- ***args** – Positional arguments to be passed to initialize copy
- ****kwargs** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

get_input()
Return the input transformer

pipeline(**args*)
Returns a PipeLine instance with self as the input source and any other Pipe instances as part of its pipeline.

Parameters **`*args`** – any additional Pipe to be added to the pipeline, in that order.

run(***kwargs*)
Get data from source, transform it, and return it

This will often be left alone unless there are specific keyword arguments or checks done in addition to the actual transformation. Keep in mind this is rare, as keyword arguments are often required by Translators, and checks are performed by DataDefs.

set_input(*new_input*)
Set the input data source in place.

Parameters **`new_input`** (*`_Transformer`*) – new transformer to be set as input to source connection.

Raises **`TypeError`** – if new_input is not an instance of _Transformer.

transform(*data*, ***kwargs*)
Apply a transformation to data returned from source.

This is where the bulk of funtionality in a Pipe lies. And allows it to be highly customizable. This will often be the only method needed to be overwriten in subclasses.

Parameters **`data`** – data returned by source.

with_input(*new_input*)
Return copy of this transformer with the new data source.

class `dalio.pipe.pipe.`**PipeBuilder**
Bases: *`dalio.pipe.pipe.Pipe`*, `dalio.base.builder._Builder`

Hybrid builder type for complementing _Transformer instances.

These specify extra methods implemented by _Transformer instances.

copy(**args*, ***kwargs*)
Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- **`*args`** – Positional arguments to be passed to initialize copy
- **`**kwargs`** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

with_piece(*param*, *name*, **args*, ***kwargs*)
Copy self and return with a new piece set

class `dalio.pipe.pipe.`**PipeLine**(**args*)
Bases: *`dalio.pipe.pipe.Pipe`*

Collection of Pipe transformations.

PipeLine instances represent multiple Pipe transformations being performed consecutively. Pipelines essentially execute multiple transformations one after the other, and thus do not check for data integrity in between them; so keep in mind that order matters and only the first data definition will be enforced.

`pipeline`
list of Pipe instaces this pipeline is composed of

Type list

`copy`(**args*, ***kwargs*)
Make a copy of this Pipeline

`extend`(**args*, *deep=False*)
Extend existing pipeline with one or more Pipe instances

Keep in mind that this will not mean that

`transform`(*data*, ***kwargs*)
Pass data sourced from first pipe through every Pipe`s .transform() method in order.

Parameters **`data`** – data sourced and checked from first source.

3.7 dalio.pipe.selection

CHAPTER

FOUR

DALIO.MODEL PACKAGE

4.1 Submodules

4.2 dalio.model.financial

Define comps analysis models

class dalio.model.financial.**CompsData**

Bases: *dalio.model.model.Model*

Get a ticker's comps and their data.

This model has two sources: comps_in and data_in. comps_in gets a ticker's comparative stocks. data_in sources ticker data given a "TICKER" keyword argument.

run(***kwargs*)

Run model.

This will be the bulk of subclass functionality. It is where all data is sourced and processed.

class dalio.model.financial.**CompsFinancials**

Bases: *dalio.model.financial.CompsData*

Subclass to CompsData for getting stock price information

class dalio.model.financial.**CompsInfo**

Bases: *dalio.model.financial.CompsData*

Subclass to CompsData for getting comps stock information

class dalio.model.financial.**MakeCriticalLine**(*weight_bounds=(-1, 1)*)

Bases: *dalio.model.model.Model*

Fit a critical line algorithm This model takes in two sources: sample_covariance and expected_returns. These are self-explanatory. The model calculates the algorithm for a set of weight bounds. .. attribute:: weight_bounds

lower and upper bound for portfolio weights.

type tuple

run(***kwargs*)

Get source data and create critical line algorithm

weight_bounds: Tuple[int] = None

class dalio.model.financial.**MakeEfficientFrontier**(*weight_bounds=(0, 1), gamma=0*)

Bases: *dalio.model.financial.MakeCriticalLine*

Make an efficient frontier algorithm. :param gamma: gamma optimization parameter. :type gamma: int

add_constraint(*new_constraint*)

Wrapper to PyPortfolioOpt BaseConvexOptimizer function Add a new constraint to the optimisation problem. This constraint must be linear and must be either an equality or simple inequality. :param new_constraint: the constraint to be added :type new_constraint: callable

Raises **AttributeError** – if new objective is not callable.

add_objective(*new_objective*, **args*, ***kwargs*)

Wrapper to PyPortfolioOpt BaseConvexOptimizer function Add a new term into the objective function. This term must be convex, and built from cvxpy atomic functions. :param new_objective: the objective to be added :type new_objective: cp.Expression

Raises

- **ValueError** – if the new objective is not supported.
- **AttributeError** – if new objective is not callable.

add_sector_definitions(*sector_defs=None*, ***kwargs*)

add_sector_weight_constraint(*sector=None*, *constraint='is'*, *weight=0.5*)

add_stock_weight_constraint(*ticker=None*, *comparisson='is'*, *weight=0.5*)

Wrapper to add_constraint method. Adds constraing on a named ticker. This is a much more intuitive interface to add constraints, as these will often be stocks of an unknown order in a dataframe. :param ticker: stock ticker or location to be constrained. :type ticker: str, int :param comparisson: constraing comparisson. :type comparisson: str :param weight: weight to constrain. :type weight: float

Raises **TypeError** – if any of the arguments are of an invalid type

copy()

Copy superclass, objectives and constraints.

gamma: int = None

run(***kwargs*)

Make efficient frontier. Create efficient frontier given a set of weight constraints.

weight_bounds: Tuple[int] = None

class dalio.model.financial.**OptimumPortfolio**

Bases: *dalio.model.model.Model*

Create optimum portfolio of stocks given dictionary of weights. This model has two sources: weights_in and data_in. The weights_in source gets optimum weights for a set of tickers. The data_in source gets price data for these same tickers.

run(***kwargs*)

Gets weights and uses them to create portfolio prices if weights were kept constant.

4.3 dalio.model.model

Define Model class

Models are transformers that take in multiple inputs and has a single output. Model instance can be much more flexible with additional options for differen strategies of data processing and collection.

class dalio.model.model.**Model**

Bases: dalio.base.transformer._Transformer

Models represent data modification with multiple internal inputs and a single internal output.

_source
dictionary of input data definitions

copy(**args*, ***kwargs*)
Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- ***args** – Positional arguments to be passed to initialize copy
- ****kwargs** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

run(***kwargs*)
Run model.

This will be the bulk of subclass functionality. It is where all data is sourced and processed.

set_input(*source_name*, *new_input*)
Set a new connection to a data definition in dictionary entry matching the key name.

Parameters

- **source_name** (`str`) – initialized item in sources dict.
- **new_input** – new source connection.

Raise: KeyError: if input name is not present in sources dict.

with_input(*source_name*, *new_input*)
Return a copy of this model with the specified data definition connection changed

Parameters

- **source_name** (`str`) – initialized item in sources dict.
- **new_input** – new source connection.

CHAPTER

FIVE

DALIO.APPLICATION PACKAGE

5.1 Submodules

5.2 dalio.application.application

Define the Application class

While Models are normally the last stage of the processing chain, it still has a single output, which might have limited value in itself. Applications are tools used for the interpretation of some input and outisde outputs. These can have a broad range of uses, from graphing to real-time trading. The main functionality is in the .run() method, which gets input data and interprets it as needed.

class dalio.application.application.**Application**

Bases: *dalio.model.model.Model*

Represent final representation of graph data through external entities.

Applications are transformations with one or more internal inputs and one or more external outputs.

_out

dictionary of outisde output connections

Type dict

copy(**args*, ***kwargs*)

Makes a copy of transformer, copying its attributes to a new instance.

This copy should essentially create a new transformation node, not an entire new graph, so the _source attribute of the returned instance should be assigned without being copied. This is also made to be built upon by subclasses, such that only new attributes need to be added to a class' copy method.

Parameters

- ***args** – Positional arguments to be passed to initialize copy
- ****kwargs** – Keyword arguments to be passed to initialize copy

Returns A copy of this _Transformer instance with copies of necessary attributes and empty input.

run(***kwargs*)

Run application.

This will be the bulk of subclass functionality. It is where all data is sourced, processed and output.

set_output(*output_name*, *new_output*)

Set a new output to data definition in dictionary entry matching the name

Parameters

- **output_name** (*str*) – the name of the output from the output dict.
- **new_output** – new External source to be set as the output.

Raises

- **KeyError** – if name is not in the output dict.
- **ValueError** – if the new output is not an instance of External.

with_output(*output_name*, *new_output*)
Return a copy of this model with the specified data definition output changed

Parameters

- **output_name** (*str*) – the name of the output from the output dict.
- **new_output** – new External source to be set as the output.

5.3 dalio.application.graphers

Applications based on graphing input data

class dalio.application.graphers.**ForecastGrapher**
Bases: *dalio.application.graphers.Grapher*

Application to graph data and a forecast horizon

This Application has two sources data_in and forecast_in. The data-in source is explained in Grapher. The forecast_in source gets a forecast data to be graphed.

run(***kwargs*)
Get data, its forecast and plot both

class dalio.application.graphers.**Grapher**
Bases: *dalio.application.application.Application*

Base grapher class.

Does basic graphing, assuming data does not require any processing before being passed onto an external grapher.

This Application has one source: data_in. The data_in source gets internal data to be graphed.

This Application has one output: data_out. The data_out output represents an external graph.

reset_out()
Reset the output graph. Figure instances should implement the .reset() method.

run(***kwargs*)
Gets data input and plots it

class dalio.application.graphers.**LMGrapher**(*x=None*, *y=None*, *legend=None*)
Bases: *dalio.application.graphers.PandasXYGrapher*

Application to graph data and a linear model fitted to it.

This Application has two sources data_in and linear_model. The data-in source is explained in Grapher. The linear_model source is a fitted linear model with intercept and coefficient data.

_legend
legend position on graph.

Type str, None

run(***kwargs*)
Get data, its fitted coefficients and intercepts and graph them.

class dalio.application.graphers.**MultiGrapher**(*rows*, *cols*)
Bases: *dalio.application.application.Application*, dalio.base.builder._Builder

Grapher for multiple inputs taking in the same keyword arguments.

This is useful to greate subplots of the same data processed in different ways. Sources are the data inputs and pieces are their kinds, args and kwargs.

This applicaiton can N sources and pieces, where N is the total number of graphs.

build_model(*data*, ***kwargs*)
Return data unprocessed

run(***kwargs*)
Gets data input from each source and plots it using the set information in each piece

class dalio.application.graphers.**PandasMultiGrapher**(*rows*, *cols*)
Bases: *dalio.application.graphers.MultiGrapher*

Multigrapher with column selection mechanisms

In this MultiGrapher, you can select any x, y and z columns as piece kwargs and they will be interpreted during the run. Keep in mind that this allows for any combination of these layered one on top of each other regardless of name. If you specify an "x" and a "z", the "z" column will be treated like a "y" column.

There are also no interpretations of what is to be graphed, and thus all wanted columns should be specified.

There is one case for indexes, where the x_index, y_index or z_index keyword arguments can be set to True.

build_model(*data*, ***kwargs*)
Process data columns

class dalio.application.graphers.**PandasTSGrapher**(*y=None*, *legend=None*)
Bases: *dalio.application.graphers.PandasXYGrapher*

Graphs a pandas time series

Same functionality as parent class with stricter inputs.

class dalio.application.graphers.**PandasXYGrapher**(*x=None*, *y=None*, *legend=None*)
Bases: *dalio.application.graphers.Grapher*

Graph data from a pandas dataframe with option of selecting columns used as axis

_x
name of column to be used for x-axis.

Type str

_y
name of column to be used for y-axis.

Type str

_legend
legend position. None by default

Type str, None

run(***kwargs*)
Get data, separate columns and feed it to data output graph

class dalio.application.graphers.**VaRGrapher**

Bases: *dalio.application.graphers.Grapher*

Application to visualize Value at Risk

run(***kwargs*)

Get value at risk data, plot returns, value at risk lines and exceptions at their maximum exedence.

Thank you for the creators of the arch package for the amazing visulaization idea!

5.4 dalio.application.printers

Print data onto an external output

class dalio.application.printers.**FilePrinter**

Bases: *dalio.application.application.Application*

Application to print data onto a file

This application has one source: data_in. The data_in source is the data to be printed.

This application has one output: data_out. The data_out output is the external output to print the data to.

run(***kwargs*)

Gets data and prints it

CHAPTER

SIX

DALIO.OPS MODULE

Define various operations

dalio.ops.**get_comps_by_sic**(*data*, *ticker*, *max_ticks=None*)

Get an equity's comps based on market cap and sic code similarity

This has the major flaw of getting too many comps for common industries.

Parameters

- **data** (*pd.DataFrame*) – data containing all possible comparisson candidates.
- **ticker** (*str*) – ticker of main stock.
- **max_ticks** (*int*) – maximum number of tickers to return.

Raises **KeyError** – if stock is not present in data.

dalio.ops.**index_cols**(*df*, *i=100*)

Index columns at some value

dalio.ops.**risk_metrics**(*data*, *lam*, *ignore_first=True*)

Apply the basic RiskMetrics (EWMA) continuous volatility measure to a a dataframe

Parameters

- **lam** (*float*) – lambda parameter
- **ignore_first** (*bool*) – whether to ignore the first row. This is often the case after a change pipe.

Returns A copy of data with the continuous volatility of each value

CHAPTER

SEVEN

DALIO.BASE

7.1 Submodules

7.2 dalio.base.builder module

Define extra utility classes used throughout the package

These classes implement certain interfaces used in specific cases and are not constrained an object's parent class.

7.3 dalio.base.constants module

Define constant terms

In order to maintain name integrity throughout graphs, constants are used instead of any string name for variables that were created or will be usued in any _Transformer instance before or after the current one. These are often column names for pandas DataFrames, though can be anything that is or will be used to identify data throughout the graph.

7.4 dalio.base.datadef module

Defines DataDef base class

DataDef instances describe data inputs throughout the graph and ensure the integrity of data continuously. These are composed of various validators that serve both to describe approved data and check for whether data passes a test.

7.5 dalio.base.node module

Defines Node abstract class

Nodes are the key building blocks of your model as they represent any data that passes thorugh it. These are usued in subsequent classes to describe and manage data.

7.6 dalio.base.transformer module

Define Transformer class

Transformers are a base class that represents any kind of data modification. These interact with DataOrigin instances as they are key to their input and output integrity. A set_source() method sets the source of the input, the .run() method cannot be executed if the input"s source is not set.

7.7 Module contents

import classes

CHAPTER

EIGHT

DALIO.VALIDATOR

8.1 Submodules

8.2 dalio.validator.array_val module

Definte validators applied to array-like inputs

class dalio.validator.array_val.**HAS_DIMS**(*dims*, *comparisson='=='*)

Bases: *dalio.validator.validator.Validator*

Check if an array has a number of dimensions

_dims

number of dimensions

Type int

_comparisson

which comparisson to perform

Type str

validate(*data*)

Validate data

Check if data fits a certain description.

Returns A description of any errors in the data according to this specific validation condition, and None if data is valid.

8.3 dalio.validator.base_val module

Define Validators used for general python objects

class dalio.validator.base_val.**ELEMS_TYPE**(*t*)

Bases: *dalio.validator.base_val.HAS_ATTR*

Checks if all elements of an iterator is of a certain type.

_t

type to check iterator's elements for

Type type, tuple

validate(*data*)

Validates data if it is an iterable with all elements of type self._t

class dalio.validator.base_val.**HAS_ATTR**(*attr*)
Bases: *dalio.validator.validator.Validator*

Checks if data has an attribute

_attr
attribute to check for

Type str

validate(*data*)
Validates data if it contains attribute self._attr

class dalio.validator.base_val.**IS_TYPE**(*t*)
Bases: *dalio.validator.validator.Validator*

Checks if data is of a certain type

Attribute: t (type): type of data to check for

validate(*data*)
Validates data if it is of type self._t

8.4 dalio.validator.pandas_val module

class dalio.validator.pandas_val.**HAS_COLS**(*cols*, *level=None*)
Bases: *dalio.validator.pandas_val.IS_PD_DF*

Checks if data has certain column names

_cols
list of column names to check

validate(*data*)
Validates data if all the columns in self._cols is present in the dataframe

class dalio.validator.pandas_val.**HAS_INDEX_NAMES**(*names*, *axis=0*)
Bases: *dalio.validator.pandas_val.IS_PD_DF*

Checks if an axis has specified names

_names
names to check for

_axis
axis to check for names

validate(*data*)
Validates data if specified axis has the specified names

class dalio.validator.pandas_val.**HAS_IN_COLS**(*items*, *cols=None*)
Bases: *dalio.validator.pandas_val.HAS_COLS*

Check if certain items are present in certain columns

_cols
See base class

_items
items that must be present in each of the specified columns

validate (*data*)
Validates data if items in self._items are not present in specified columns. Specified columns are all columns if self._cols is None.

class `dalio.validator.pandas_val.`**HAS_LEVELS** (*levels*, *axis=0*, *comparisson='<='*)
Bases: `dalio.validator.pandas_val.IS_PD_DF`

validate (*data*)
Validates data if it is of type self._t

class `dalio.validator.pandas_val.`**IS_PD_DF**
Bases: `dalio.validator.base_val.IS_TYPE`

Checks if data is a pandas dataframe

See base class

class `dalio.validator.pandas_val.`**IS_PD_TS**
Bases: `dalio.validator.base_val.IS_TYPE`

Checks if data is a pandas time series

validate (*data*)
Validates data if it's index is of type pandas.DateTimeIndex

8.5 dalio.validator.presets module

Define Validator collection presets

These are useful to describe very specific data characteristics commonly used in some analysis.

8.6 dalio.validator.validator module

Define Validator class

Validators are the building blocks of data integrity in the graph. As modularity is key, validators ensure that the data that enters a node is what it is mean to be or that errors are targeted to make debugging easier.

class `dalio.validator.validator.`**Validator** (*fatal=True*)
Bases: `object`

Check for some characteristic of a piece of data

Validators can have any attribute needed, but functionality is stored in u the .validate function, which returns any errors in the data.

fatal
Whether if invalid data is fatal. Decides whether invalid data can still be passed on (with a warning) or if it is grounds to stop the execution of the graph. False by default.

Type bool

test_desc
Description of tests performed on data

Type str

fatal: bool = None

fatal_off()
Turn fatal off and return self

fatal_on()
Turn fatal on and return self

is_on: bool = None

test_desc: str = None

validate(*data*)
Validate data

Check if data fits a certain description.

Returns A description of any errors in the data according to this specific validation condition, and None if data is valid.

8.7 Module contents

CHAPTER

NINE

DALIO.UTIL

9.1 Submodules

9.2 dalio.util.plotting_utils module

Plotting utilities

Thank you for the creators of pypfopt for the wonderful code!

`dalio.util.plotting_utils.`**`plot_covariance`**(*cov_matrix*, *plot_correlation=False*, *show_tickers=True*, *ax=None*)

Generate a basic plot of the covariance (or correlation) matrix, given a covariance matrix.

Parameters

- **`cov_matrix`** (*`pd.DataFrame, np.ndarray`*) – covariance matrix
- **`plot_correlation`** (*`bool`*) – whether to plot the correlation matrix instead, defaults to False. Optional.
- **`show_tickers`** (*`bool`*) – whether to use tickers as labels (not recommended for large portfolios). Optional. Defaults to True.
- **`ax`** (*`matplolib.axis, None`*) – Axis to plot on. Optional. New axis will be created if none is specified.

Returns matplotlib axis

`dalio.util.plotting_utils.`**`plot_dendrogram`**(*hrp*, *show_tickers=True*, *ax=None*, ***kwargs*)

Plot the clusters in the form of a dendrogram.

Parameters

- **`hrp`** – HRPpt object that has already been optimized.
- **`show_tickers`** (*`bool`*) – whether to use tickers as labels (not recommended for large portfolios). Optional. Defaults to True.
- **`ax`** (*`matplolib.axis, None`*) – Axis to plot on. Optional. New axis will be created if none is specified.
- **`**kwargs`** – optional parameters for main graph.

Returns matplotlib axis

`dalio.util.plotting_utils.`**`plot_efficient_frontier`**(*cla*, *points=100*, *visible=25*, *show_assets=True*, *ax=None*, ***kwargs*)

Plot the efficient frontier based on a CLA object

Parameters

- **points** (*int*) – number of points to plot. Optional. Defaults to 100
- **show_assets** (*bool*) – whether we should plot the asset risks/returns also. Optional. Defaults to True.
- **ax** (*matplolib.axis, None*) – Axis to plot on. Optional. New axis will be created if none is specified.
- ****kwargs** – optional parameters for main graph.

Returns matplotlib axis

`dalio.util.plotting_utils.`**`plot_weights`**(*weights*, *ax=None*, ***kwargs*)

Plot the portfolio weights as a horizontal bar chart

Parameters

- **weights** (*dict*) – the weights outputted by any PyPortfolioOpt optimiser.
- **ax** (*matplolib.axis, None*) – Axis to plot on. Optional. New axis will be created if none is specified.
- ****kwargs** – optional parameters for main graph.

Returns matplotlib axis

9.3 dalio.util.processing_utils module

Data processing utilities

`dalio.util.processing_utils.`**`list_str`**(*listi*)

`dalio.util.processing_utils.`**`process_cols`**(*cols*)

Standardize input columns

`dalio.util.processing_utils.`**`process_date`**(*date*)

Standardize input date

Raises **TypeError** – if the type of the date parameter cannot be converted to a pandas timestamp

`dalio.util.processing_utils.`**`process_new_colnames`**(*cols*, *new_cols*)

Get new column names based on the column parameter

`dalio.util.processing_utils.`**`process_new_df`**(*df1*, *df2*, *cols*, *new_cols*)

Process new dataframe given columns and new column names

Parameters

- **df1** (*pd.DataFrame*) – first dataframe.
- **df2** (*pd.DataFrame*) – dataframe to join or get columns from
- **cols** (*iterable*) – iterable of columns being targetted.
- **new_cols** (*iterable*) – iterable of new column names.

9.4 dalio.util.translation_utils module

Translation utilities

`dalio.util.translation_utils.`**`get_numeric_column_names`**(*df*)

Return the names of all columns of numeric type.

Parameters **`df`** (*`pandas.DataFrame`*) – The dataframe to get numeric column names for.

Returns The names of all columns of numeric type.

Return type list of str

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> data = [[2, 3.2, "acd"], [1, 7.2, "alk"], [8, 12.1, "alk"]]
>>> df = pd.DataFrame(data, [1,2,3], ["rank", "ph","lbl"])
>>> sorted(get_numeric_column_names(df))
['ph', 'rank']
```

`dalio.util.translation_utils.`**`translate_df`**(*translator*, *df*, *inplace=False*)

Translate dataframe column and index names in accordance to translator dictionary.

Parameters

- **`translator`** (*`dict`*) – dictionary of {original: translated} key value pairs.
- **`df`** (*`pd.DataFrame`*) – dataframe to have rows and columns translated.
- **`inplace`** (*`bool`*) – whether to perform operation inplace or return a translated copy. Optional. Defaults to False.

9.5 Module contents

`dalio.util.`**`extract_level_names_dict`**(*df*)

Extract all column names in a dataframe as (level: **names_** dicitonar7

Parameters **`df`** (*`pd.DataFrame`*) – dataframe whose columns will be extracted

`dalio.util.`**`filter_levels`**(*levels*, *filters*)

Filter columns in levels to either be equal to specified columns or a filtering function

Parameters

- **`levels`** (*`dict`*) – all column names in a (level: names) dict
- **`filters`** (*`str, list, callable, dict`*) – either columns to place on a specified level or filter functions to select columns there.

`dalio.util.`**`extract_cols`**(*df*, *cols*)

Extract columns from a dataframe

Parameters

- **`df`** (*`pd.DataFrame`*) – dataframe containing the columns
- **`cols`** (*`hashable, iterable, dict`*) – single column, list of columnst or dict with the level as keys and column(s) as values.

Raises **`KeyError`** – if columns are not in dataframe

`dalio.util.`**`insert_cols`**(*df*, *new_data*, *cols*)

Insert new data into specified existing columns

Parameters

- **`df`** (*`pd.DataFrame`*) – dataframe to insert data into.
- **`new_data`** (*`any`*) – new data to be inserted
- **`cols`** (*`hashable, iterable, dict`*) – existing columns in data.

Raises

- **`KeyError`** – if columns are not in dataframe
- **`Exception`** – if new data doesn't fit cols dimensions

`dalio.util.`**`drop_cols`**(*df*, *cols*)

Drop selected columns from levels

Parameters

- **`df`** (*`pd.DataFrame`*) – dataframe to have columns dropped.
- **`cols`** (*`hashable, iterable, dict`*) – column selection

`dalio.util.`**`get_slice_from_dict`**(*df*, *cols*)

Get a tuple of slices that locate the specified (level: column) combination.

Parameters

- **`df`** (*`pd.DataFrame`*) – dataframe with multiindex
- **`cols`** (*`dict`*) – (level: column) dictionary

Raises

- **`ValueError`** – if any of the level keys are not integers
- **`KeyError`** – if any level key is out of bounds or if columns are not in the dataframe

`dalio.util.`**`mi_join`**(*df1*, *df2*, **args*, ***kwargs*)

Join two dataframes and sort their columns

Parameters

- **`df2`** (*`df1,`*) – dataframes to join
- **`**kwargs`** (*`*args,`*) – arguments for join function (called from df1)

Raises **`ValueError if number of levels don't match`** –

`dalio.util.`**`add_suffix`**(*all_cols*, *cols*, *suffix*)

Add suffix to appropriate level in a given column index.

Parameters

- **`all_cols`** (*`pd.Index, pd.MultiIndex`*) – all columns from an index. This is only relevent when the columns at hand are a multindex, as each tuple element will contain elements from all levels (not only the selected ones)
- **`cols`** (*`str, list, dict`*) – selected columns
- **`suffix`** (*`str`*) – the suffix to add to the selected columns.

`dalio.util.`**`out_of_place_col_insert`**(*df*, *series*, *loc*, *column_name=None*)

Returns a new dataframe with given column inserted at given location.

Parameters

- **df** (*pandas.DataFrame*) – The dataframe into which to insert the column.
- **series** (*pandas.Series*) – The pandas series to be inserted.
- **loc** (*int*) – The location into which to insert the new column.
- **column_name** (*str, default None*) – The name to assign the new column. If None, the given series name attribute is attempted; if the given series is missing the name attribute a ValueError exception will be raised.

Returns The resulting dataframe.

Return type pandas.DataFrame

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1, 'a'], [4, 'b']], columns=['a', 'g'])
>>> ser = pd.Series([7, 5])
>>> out_of_place_col_insert(df, ser, 1, 'n')
   a  n  g
0  1  7  a
1  4  5  b
```

dalio.util.**translate_df**(*translator*, *df*, *inplace=False*)
Translate dataframe column and index names in accordance to translator dictionary.

Parameters

- **translator** (*dict*) – dictionary of {original: translated} key value pairs.
- **df** (*pd.DataFrame*) – dataframe to have rows and columns translated.
- **inplace** (*bool*) – whether to perform operation inplace or return a translated copy. Optional. Defaults to False.

dalio.util.**get_numeric_column_names**(*df*)
Return the names of all columns of numeric type.

Parameters **df** (*pandas.DataFrame*) – The dataframe to get numeric column names for.

Returns The names of all columns of numeric type.

Return type list of str

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> data = [[2, 3.2, "acd"], [1, 7.2, "alk"], [8, 12.1, "alk"]]
>>> df = pd.DataFrame(data, [1,2,3], ["rank", "ph","lbl"])
>>> sorted(get_numeric_column_names(df))
['ph', 'rank']
```

dalio.util.**process_cols**(*cols*)
Standardize input columns

dalio.util.**process_new_colnames**(*cols*, *new_cols*)
Get new column names based on the column parameter

`dalio.util.`**`process_date`**`(`*date*`)`

Standardize input date

Raises **`TypeError`** – if the type of the date parameter cannot be converted to a pandas timestamp

`dalio.util.`**`process_new_df`**`(`*df1*, *df2*, *cols*, *new_cols*`)`

Process new dataframe given columns and new column names

Parameters

- **`df1`** (*`pd.DataFrame`*) – first dataframe.
- **`df2`** (*`pd.DataFrame`*) – dataframe to join or get columns from
- **`cols`** (*`iterable`*) – iterable of columns being targetted.
- **`new_cols`** (*`iterable`*) – iterable of new column names.

`dalio.util.`**`translate_df`**`(`*translator*, *df*, *inplace=False*`)`

Translate dataframe column and index names in accordance to translator dictionary.

Parameters

- **`translator`** (*`dict`*) – dictionary of {original: translated} key value pairs.
- **`df`** (*`pd.DataFrame`*) – dataframe to have rows and columns translated.
- **`inplace`** (*`bool`*) – whether to perform operation inplace or return a translated copy. Optional. Defaults to False.

`dalio.util.`**`plot_efficient_frontier`**`(`*cla*, *points=100*, *visible=25*, *show_assets=True*, *ax=None*, ***kwargs*`)`

Plot the efficient frontier based on a CLA object

Parameters

- **`points`** (*`int`*) – number of points to plot. Optional. Defaults to 100
- **`show_assets`** (*`bool`*) – whether we should plot the asset risks/returns also. Optional. Defaults to True.
- **`ax`** (*`matplolib.axis, None`*) – Axis to plot on. Optional. New axis will be created if none is specified.
- **`**kwargs`** – optional parameters for main graph.

Returns matplotlib axis

`dalio.util.`**`plot_covariance`**`(`*cov_matrix*, *plot_correlation=False*, *show_tickers=True*, *ax=None*`)`

Generate a basic plot of the covariance (or correlation) matrix, given a covariance matrix.

Parameters

- **`cov_matrix`** (*`pd.DataFrame, np.ndarray`*) – covariance matrix
- **`plot_correlation`** (*`bool`*) – whether to plot the correlation matrix instead, defaults to False. Optional.
- **`show_tickers`** (*`bool`*) – whether to use tickers as labels (not recommended for large portfolios). Optional. Defaults to True.
- **`ax`** (*`matplolib.axis, None`*) – Axis to plot on. Optional. New axis will be created if none is specified.

Returns matplotlib axis

`dalio.util.`**`plot_weights`**`(`*weights*, *ax=None*, ***kwargs*`)`

Plot the portfolio weights as a horizontal bar chart

Parameters

- **`weights`** (*`dict`*) – the weights outputted by any PyPortfolioOpt optimiser.
- **`ax`** (*`matplotlib.axis, None`*) – Axis to plot on. Optional. New axis will be created if none is specified.
- **`**kwargs`** – optional parameters for main graph.

Returns matplotlib axis

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX