Dal.io

Release 0.0.1

Renato Mateus Zimmermann

Jul 19, 2020

MODULES

CHAPTER

ONE

QUICK LINKS

1.1 User Modules

1.1.1 dalio.external package

Submodules

dalio.external.external module

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.

dalio.external.file module

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.

dalio.external.image module

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

dalio.external.web module

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

Module contents

class dalio.external.**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.**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.

class dalio.external.**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.**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.**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.**YahooDR**(*config=None*)
Bases: dalio.external.web._PDR

Represents financial data from Yahoo! Finance

request(***kwargs*)
Get data from specified tickers

class dalio.external.**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.

1.1.2 dalio.translator package

Submodules

dalio.translator.file module

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

dalio.translator.pdr module

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

dalio.translator.quandl module

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

dalio.translator.translator module

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

Module contents

class dalio.translator.**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.**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

class dalio.translator.**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

class dalio.translator.**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

1.1.3 dalio.pipe package

Submodules

dalio.pipe.builders module

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.

dalio.pipe.col_generation module

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 (-∞, 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"

Example

```
>>> import pandas as pd; from dalio.pipe import Custom;
>>> data = [[3, 2143], [10, 1321], [7, 1255]]
>>> df = pd.DataFrame(data, [1,2,3], ['years', 'avg_revenue'])
>>> total_rev = lambda row: row['years'] * row['avg_revenue']
>>> add_total_rev = Custom(total_rev, 'total_revenue', axis=1)
>>> add_total_rev.transform(df)
   years  avg_revenue  total_revenue
1      3         2143           6429
2     10         1321          13210
3      7         1255           8785
>>> def halfer(row):
...     new = {'year/2': row['years']/2,
...            'rev/2': row['avg_revenue']/2}
...     return pd.Series(new)
>>> half_cols = Custom(halfer, axis=1, drop=False)
>>> half_cols.transform(df)
   years  avg_revenue   rev/2  year/2
1      3         2143  1071.5     1.5
2     10         1321   660.5     5.0
3      7         1255   627.5     3.5
```

```
>>> data = [[3, 3], [2, 4], [1, 5]]
>>> df = pd.DataFrame(data, [1,2,3], ["A","B"])
>>> func = lambda df: df['A'] == df['B']
>>> add_equal = Custom(func, "A==B", strategy="pipe", drop=False)
>>> add_equal.transform(df)
   A  B   A==B
1  3  3   True
2  2  4  False
3  1  5  False
```

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

dalio.pipe.forecast module

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.

dalio.pipe.pipe module

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.

dalio.pipe.select module

Defines various ways of getting a subset of data based on some condition

class dalio.pipe.select.**ColDrop**(*columns*)
Bases: dalio.pipe.select._ColSelection

A pipeline stage that drops columns by name.

Parameters **columns** (*str, iterable or callable*) – The label, or an iterable of labels, of columns to drop. Alternatively, columns can be assigned a callable returning bool values for pandas.Series objects; if this is the case, every column for which it return True will be dropped.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[8,'a'],[5,'b']], [1,2], ['num', 'char'])
>>> pdp.ColDrop('num').apply(df)
  char
1    a
2    b
```

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.

class dalio.pipe.select.**ColRename**(*map_dict*)
Bases: *dalio.pipe.pipe.Pipe*

A pipeline stage that renames a column or columns.

rename_map
Maps old column names to new ones.

Type dict

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[8,'a'],[5,'b']], [1,2], ['num', 'char'])
>>> pdp.ColRename({'num': 'len', 'char': 'initial'}).apply(df)
   len initial
1    8       a
2    5       b
```

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*)
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.

class dalio.pipe.select.**ColReorder**(*map_dict*, *level=0*)
Bases: dalio.pipe.select._ColSelection

A pipeline stage that reorders columns.

positions
A mapping of column names to their desired positions after reordering Columns not included in the mapping will maintain their relative positions over the non-mapped colums.

Type dict

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[8,4,3,7]], columns=['a', 'b', 'c', 'd'])
>>> pdp.ColReorder({'b': 0, 'c': 3}).apply(df)
   b  a  d  c
0  4  8  7  3
```

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*)
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.

class dalio.pipe.select.**ColSelect** (*columns*)
Bases: dalio.pipe.select._ColSelection

Select columns

transform (*data*, ***kwargs*)
Selects the specified columns or returns data as is if no column was specified.

Returns Data of the same format as before but only only containing the specified columns.

class dalio.pipe.select.**DateSelect** (*start=None*, *end=None*)
Bases: *dalio.pipe.pipe.Pipe*

Select a date range.

This is commonly left as a local variable to control date range being used at a piece of a graph.

_start
start date.

Type pd.Timestamp

_end
end date.

Type pd.Timestamp

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_end(*end*)
Set the _end attribute

set_start(*start*)
Set the _start attribute

transform(*data*, ***kwargs*)
Slices time series into selected date range.

Returns Time series of the same format as input containing a subset of the original dates.

class dalio.pipe.select.**DropNa**(***kwargs*)
Bases: *dalio.pipe.pipe.Pipe*

A pipeline stage that drops null values.

Supports all parameter supported by pandas.dropna function.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,None],[1,11]], [1,2,3], ['a','b'])
>>> pdp.DropNa().apply(df)
   a     b
1  1   4.0
3  1  11.0
```

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.

class dalio.pipe.select.**FreqDrop**(*values*, *columns=None*)
Bases: dalio.pipe.select._ColValSelection

A pipeline stage that drops rows by value frequency.

Parameters

- **threshold** (*int*) – The minimum frequency required for a value to be kept.
- **column** (*str*) – The name of the colums to check for the given value frequency.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[1,11]], [1,2,3], ['a','b'])
>>> pdp.FreqDrop(2, 'a').apply(df)
   a   b
1  1   4
3  1  11
```

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.

class dalio.pipe.select.**RowDrop**(*conditions*, *columns=None*, *reduce_strat=None*)

Bases: dalio.pipe.select._ColSelection

A pipeline stage that drop rows by callable conditions.

Parameters

- **conditions** (*list-like or dict*) – The list of conditions that make a row eligible to be dropped. Each condition must be a callable that take a cell value and return a bool value. If a list of callables is given, the conditions are checked for each column value of each row. If a dict mapping column labels to callables is given, then each condition is only checked for the column values of the designated column.
- **reduce** (*'any', 'all' or 'xor', default 'any'*) – Determines how row conditions are reduced. If set to 'all', a row must satisfy all given conditions to be dropped. If set to 'any', rows satisfying at least one of the conditions are dropped. If set to 'xor', rows satisfying exactly one of the conditions will be dropped. Set to 'any' by default.
- **columns** (*str or iterable, optional*) – The label, or an iterable of labels, of columns. Optional. If given, input conditions will be applied to the sub-dataframe made up of these columns to determine which rows to drop.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[5,11]], [1,2,3], ['a','b'])
>>> pdp.RowDrop([lambda x: x < 2]).apply(df)
   a   b
2  4   5
3  5  11
>>> pdp.RowDrop({'a': lambda x: x == 4}).apply(df)
   a   b
1  1   4
3  5  11
```

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.

class dalio.pipe.select.**ValDrop**(*values*, *columns=None*)
Bases: dalio.pipe.select._ColValSelection

A pipeline stage that drops rows by value.

Parameters

- **values** (*list-like*) – A list of the values to drop.
- **columns** (*str or list-like, default None*) – The name, or an iterable of names, of columns to check for the given values. If set to None, all columns are checked.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[18,11]], [1,2,3], ['a','b'])
>>> pdp.ValDrop([4], 'a').apply(df)
    a   b
1   1   4
3  18  11
>>> pdp.ValDrop([4]).apply(df)
    a   b
3  18  11
```

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.

class dalio.pipe.select.**ValKeep**(*values*, *columns=None*)
Bases: dalio.pipe.select._ColValSelection

A pipeline stage that keeps rows by value.

Parameters

- **values** (*list-like*) – A list of the values to keep.
- **columns** (*str or list-like, default None*) – The name, or an iterable of names, of columns to check for the given values. If set to None, all columns are checked.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[5,11]], [1,2,3], ['a','b'])
>>> pdp.ValKeep([4, 5], 'a').apply(df)
   a   b
2  4   5
3  5  11
>>> pdp.ValKeep([4, 5]).apply(df)
   a  b
2  4  5
```

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.

Module contents

class dalio.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.

class dalio.pipe.**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”

Example

```
>>> import pandas as pd; from dalio.pipe import Custom;
>>> data = [[3, 2143], [10, 1321], [7, 1255]]
>>> df = pd.DataFrame(data, [1,2,3], ['years', 'avg_revenue'])
>>> total_rev = lambda row: row['years'] * row['avg_revenue']
>>> add_total_rev = Custom(total_rev, 'total_revenue', axis=1)
>>> add_total_rev.transform(df)
   years  avg_revenue  total_revenue
1      3         2143           6429
2     10         1321          13210
3      7         1255           8785
>>> def halfer(row):
...     new = {'year/2': row['years']/2,
...            'rev/2': row['avg_revenue']/2}
```

(continues on next page)

(continued from previous page)

```
...     return pd.Series(new)
>>> half_cols = Custom(halfer, axis=1, drop=False)
>>> half_cols.transform(df)
   years  avg_revenue   rev/2  year/2
1      3         2143  1071.5     1.5
2     10         1321   660.5     5.0
3      7         1255   627.5     3.5
```

```
>>> data = [[3, 3], [2, 4], [1, 5]]
>>> df = pd.DataFrame(data, [1,2,3], ["A","B"])
>>> func = lambda df: df['A'] == df['B']
>>> add_equal = Custom(func, "A==B", strategy="pipe", drop=False)
>>> add_equal.transform(df)
   A  B   A==B
1  3  3   True
2  2  4  False
3  1  5  False
```

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.**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.**ColSelect**(*columns*)
Bases: dalio.pipe.select._ColSelection

Select columns

transform(*data*, ***kwargs*)
Selects the specified columns or returns data as is if no column was specified.

Returns Data of the same format as before but only only containing the specified columns.

class dalio.pipe.**DateSelect**(*start=None*, *end=None*)
Bases: *dalio.pipe.pipe.Pipe*

Select a date range.

This is commonly left as a local variable to control date range being used at a piece of a graph.

_start
start date.

Type pd.Timestamp

_end
end date.

Type pd.Timestamp

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_end(*end*)
Set the _end attribute

set_start(*start*)
Set the _start attribute

transform(*data*, ***kwargs*)
Slices time series into selected date range.

Returns Time series of the same format as input containing a subset of the original dates.

class dalio.pipe.**ColDrop**(*columns*)
Bases: dalio.pipe.select._ColSelection

A pipeline stage that drops columns by name.

Parameters **columns** (*str, iterable or callable*) – The label, or an iterable of labels, of columns to drop. Alternatively, columns can be assigned a callable returning bool values for pandas.Series objects; if this is the case, every column for which it return True will be dropped.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[8,'a'],[5,'b']], [1,2], ['num', 'char'])
>>> pdp.ColDrop('num').apply(df)
  char
1    a
2    b
```

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.

class dalio.pipe.**ValDrop**(*values*, *columns=None*)

Bases: dalio.pipe.select._ColValSelection

A pipeline stage that drops rows by value.

Parameters

- **values** (*list-like*) – A list of the values to drop.
- **columns** (*str or list-like, default None*) – The name, or an iterable of names, of columns to check for the given values. If set to None, all columns are checked.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[18,11]], [1,2,3], ['a','b'])
>>> pdp.ValDrop([4], 'a').apply(df)
    a   b
1   1   4
3  18  11
>>> pdp.ValDrop([4]).apply(df)
    a   b
3  18  11
```

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.

class dalio.pipe.**ValKeep**(*values*, *columns=None*)

Bases: dalio.pipe.select._ColValSelection

A pipeline stage that keeps rows by value.

Parameters

- **values** (*list-like*) – A list of the values to keep.
- **columns** (*str or list-like, default None*) – The name, or an iterable of names, of columns to check for the given values. If set to None, all columns are checked.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[5,11]], [1,2,3], ['a','b'])
>>> pdp.ValKeep([4, 5], 'a').apply(df)
   a   b
2  4   5
3  5  11
>>> pdp.ValKeep([4, 5]).apply(df)
   a  b
2  4  5
```

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.

class dalio.pipe.**ColRename**(*map_dict*)

Bases: *dalio.pipe.pipe.Pipe*

A pipeline stage that renames a column or columns.

rename_map

Maps old column names to new ones.

Type dict

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[8,'a'],[5,'b']], [1,2], ['num', 'char'])
>>> pdp.ColRename({'num': 'len', 'char': 'initial'}).apply(df)
   len initial
1    8       a
2    5       b
```

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*)

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.

class dalio.pipe.**DropNa**(***kwargs*)
Bases: *dalio.pipe.pipe.Pipe*

A pipeline stage that drops null values.

Supports all parameter supported by pandas.dropna function.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,None],[1,11]], [1,2,3], ['a','b'])
>>> pdp.DropNa().apply(df)
   a     b
1  1   4.0
3  1  11.0
```

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.

class dalio.pipe.**FreqDrop**(*values*, *columns=None*)
Bases: dalio.pipe.select._ColValSelection

A pipeline stage that drops rows by value frequency.

Parameters

- **threshold** (*int*) – The minimum frequency required for a value to be kept.
- **column** (*str*) – The name of the colums to check for the given value frequency.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[1,11]], [1,2,3], ['a','b'])
>>> pdp.FreqDrop(2, 'a').apply(df)
   a   b
1  1   4
3  1  11
```

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.

class dalio.pipe.**ColReorder**(*map_dict*, *level=0*)
Bases: dalio.pipe.select._ColSelection

A pipeline stage that reorders columns.

positions
A mapping of column names to their desired positions after reordering Columns not included in the mapping will maintain their relative positions over the non-mapped colums.

Type dict

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[8,4,3,7]], columns=['a', 'b', 'c', 'd'])
>>> pdp.ColReorder({'b': 0, 'c': 3}).apply(df)
   b  a  d  c
0  4  8  7  3
```

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*)

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.

class dalio.pipe.**RowDrop** (*conditions*, *columns=None*, *reduce_strat=None*)

Bases: dalio.pipe.select._ColSelection

A pipeline stage that drop rows by callable conditions.

Parameters

- **conditions** (*list-like or dict*) – The list of conditions that make a row eligible to be dropped. Each condition must be a callable that take a cell value and return a bool value. If a list of callables is given, the conditions are checked for each column value of each row. If a dict mapping column labels to callables is given, then each condition is only checked for the column values of the designated column.
- **reduce** (*'any', 'all' or 'xor', default 'any'*) – Determines how row conditions are reduced. If set to 'all', a row must satisfy all given conditions to be dropped. If set to 'any', rows satisfying at least one of the conditions are dropped. If set to 'xor', rows satisfying exactly one of the conditions will be dropped. Set to 'any' by default.
- **columns** (*str or iterable, optional*) – The label, or an iterable of labels, of columns. Optional. If given, input conditions will be applied to the sub-dataframe made up of these columns to determine which rows to drop.

Example

```
>>> import pandas as pd; import pdpipe as pdp;
>>> df = pd.DataFrame([[1,4],[4,5],[5,11]], [1,2,3], ['a','b'])
>>> pdp.RowDrop([lambda x: x < 2]).apply(df)
   a   b
2  4   5
3  5  11
>>> pdp.RowDrop({'a': lambda x: x == 4}).apply(df)
   a   b
1  1   4
3  5  11
```

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.

class dalio.pipe.**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.**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

class dalio.pipe.**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.**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.**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 (-∞, 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.**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.**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.**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.**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.**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.**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.**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.**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.**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.

class dalio.pipe.**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.**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.**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

1.1.4 dalio.model package

Submodules

dalio.model.basic module

Define basic models

class dalio.model.basic.**Join**(***kwargs*)
Bases: *dalio.model.model.Model*

Join two dataframes on index.

This model has two sources: left and right.

_kwargs
optional keyword arguments for pd.join

Type dict

run(***kwargs*)
Get left and right side data and join

dalio.model.financial module

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.

dalio.model.model module

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.

dalio.model.statistical module

Define statistical models

class dalio.model.statistical.**XYLinearModel**
Bases: *dalio.model.model.Model*, dalio.base.builder._Builder

Generalized Linear model for arrays from two sources.

This Model has two sources, x and y.

This Builder has one piece. the linear model strategy.

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

Returns Unfitted linear model

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*)
Get data from both sources, transform them into np.arrays and fit the built model

Module contents

class dalio.model.**Join**(***kwargs*)
Bases: *dalio.model.model.Model*

Join two dataframes on index.

This model has two sources: left and right.

_kwargs
optional keyword arguments for pd.join

Type dict

run(***kwargs*)
Get left and right side data and join

class dalio.model.**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.**CompsFinancials**
Bases: *dalio.model.financial.CompsData*

Subclass to CompsData for getting stock price information

class dalio.model.**CompsInfo**
Bases: *dalio.model.financial.CompsData*

Subclass to CompsData for getting comps stock information

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

class dalio.model.**XYLinearModel**
Bases: *dalio.model.model.Model*, dalio.base.builder._Builder

Generalized Linear model for arrays from two sources.

This Model has two sources, x and y.

This Builder has one piece. the linear model strategy.

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

Returns Unfitted linear model

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*)
Get data from both sources, transform them into np.arrays and fit the built model

1.1.5 dalio.application package

Submodules

dalio.application.application module

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.

dalio.application.graphers module

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!

dalio.application.printers module

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

Module contents

class dalio.application.**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

class dalio.application.**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.**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.**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.**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.**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.**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!

class dalio.application.**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.

1.2 Developer Modules

1.2.1 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

1.2.2 dalio.base package

Submodules

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.

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.

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.

dalio.base.memory module

Defines memory transformers

class dalio.base.memory.**LazyRunner**(*mem_type*, *args=None*, *kwargs=None*, *buff=1*, *update=False*)

Bases: dalio.base.transformer._Transformer

Memory manager created to set memory input of an object after executing a transformer with given kwargs.

This is useful when you want to store data sourced from a transformer but doesn't know which kwarg requests will be used. This object waits for a run request to source data from a transformer and set the source of one or more memory objects (in order) when data does arrive, and reutilize it if data is requested with the same kwargs.

For every new and valid evaluation, a new Memory instance is created and saved as a value in the _memory dictionary. Failed memory storage will result in no new Memory instance being created. This is done instead of simply setting inputs to Memory instances created upon initialization in order to reduce the memory usage of LazyRunner instances.

KEEP IN MIND that this does not check if the actual input is the same to relay the data, only the kwargs (for speed's sake). This creates the risk that inputs or transformer attributes are changed (keeping kwarg requests the same) and the old data is retrieved. Use the pop() or clear() methods to solve this.

KEEP IN MIND there is a risk of having very different inputs being retrieved (from different external sources or date filters, for example) only based on kwarg requests. This is only relevant if _buff > 1.

_source
transformer to source data from. No data definitions are used as this should be performed by the memory type uppon setting an input.

Type _Transformer

_mem_type
type object for generating new memory instances.

Type type

_args
tuple of arguments for new _mem_type instance initialization

Type tuple

_kwargs
dict of keyword arguments for new _mem_type instance initialization

Type dict

_memory
deque containing one Memory instance for every unique kwargs ran with a (kwarg, Memory) tuple structure.

Type deque

_buff
Maximum number of Memory instances to be stored at any point. Positive numbers will be this limit, -1 represent no buffer limits. This option should be used with caution, as it can be highly memory-inneficient. Subclasses can create new methods of managing this limit.

Type int, -1 or >0

_update
Whether _memory dict should be updated if a new set of kwargs is ran after reaching maximum capacity (as defined by the _buff attribute). If set to True, the last element of the _memory dict will be substituted.

Type bool

clear()
Clear memory

copy(**args*, ***kwargs*)
Return a copy of this instance with a shallow memory dict copy

run(***kwargs*)
Compare kwargs with existing keys, update or set _memory in accordance to _update and _buff attributes.

Raises **BufferError** – if new kwargs, buffer is full and update set to False

set_buff(*buff*)
Set the _buff attribute

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

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

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

set_update(*update*)
Set the _update attribute

with_input(*new_input*)
Return copy of this transformer with the new input connection.

Returns Copy of self with new input.

class dalio.base.memory.**LocalMemory**
Bases: *dalio.base.memory.Memory*

Stores memory in the local session

clear()
Clear memory

run(***kwargs*)
Return data stored in source variable

If data can be coppied, it will. This might not be memory efficient, but it makes behaviour from the Memory._source attribute more consistent with external memory sources.

set_input(*new_input*)
Store input data into source variable

class dalio.base.memory.**Memory**
Bases: dalio.base.transformer._Transformer

Implement mechanics to store and retrieve input data.

This is a pseudo-transformer, as it is supposed to behave like on on the surface (implementing all needed methods) but not actually performing any actual transformation.

This is used in pipes that heavily reutilize the same external data source using the same kwarg requests. Implementations store and retrieive data through different methods and locations, and might implement certain requirements that must be met by input data in order for it to be stored.

_def
Connection-less data definition that checks for required characteristics of of input data.

Type _DataDef

_source
Memory source. Implementations will often have additional attributes to manage this source.

Type any

clear()
Clear memory

copy(**args*, ***kwargs*)
Create new instance and memory source

run(***kwargs*)
Check if location is set and return stored data accordingly

set_input(*new_input*)
Store input data

with_input(*new_input*)
Return copy of this transformer with the new input connection.

Returns Copy of self with new input.

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.

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.

Module contents

import classes

1.2.3 dalio.validator package

Submodules

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.

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

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

dalio.validator.presets module

Define Validator collection presets

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

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.

Module contents

1.2.4 dalio.util package

Submodules

dalio.util.level_utils module

Utilities for dealing with DataFrame index or column levels

dalio.util.level_utils.**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.level_utils.**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.level_utils.**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.level_utils.**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.level_utils.**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.level_utils.**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.level_utils.**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.level_utils.**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.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

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.

dalio.util.transformation_utils module

dalio.util.transformation_utils.**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.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.

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** (*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

1.3 Understanding Graphs

1.3.1 What do I mean by "graphical structure"?

In a graphical structures data is represented as nodes and operations as edges. Think of it as a way to represent many inter-connected transformations and their input and output data.

1.3.2 Progressive Disclosure of Complexity

The main philosophy behind the graphical structure of Dal-io come from the Deep Learning library Keras. In their documentation, they state that "A core principle of Keras is **progressive disclosure of complexity**. You should always be able to get into lower-level workflows in a gradual way. You shouldn't fall off a cliff if the high-level functionality doesn't exactly match your use case. You should be able to gain more control over the small details while retaining a commensurate amount of high-level convenience."

So you are familiar with Keras, you will understand that they provide users with a plethora of pre-implemented classes (layers and models) that fit into each other, though the user is also free to create subclasses of their own that can be integrated into the Deep Neural Network and interact with it as just another layer.

Likewise, all of the classes described below where made with the objective of being easily customized by more experienced users. After all, the great majority of objects you will be using where implemented like that! Once you feel like you got a hang of Dal-io and want to build your own pieces, check out the source code or the *Core Classes and Concepts*.

1.3.3 Why is a graphical structure optimal for financial modeling?

- Modern automated financial models retrieve data, clean and dirty, from various sources and through cleaning and integration are able to join them, further process this product and finally derive insights. The problem is that as these models utilize more and more data from various sources, created models tend to become confusing for both technical and non technical people. Also, as there is no unified workflow to deal with these, created models tend to become highly inflexible and lacking portability (onto other models or projects.) A graphical architecture offers an intuitive workflow for working with data, where inputs can have a unified translation, data can be constantly checked for validity and outputs can be used in flexible ways as parts of a bigger system or drive actions.
- Utilizing large amounts of data can also end up being highly memory-inefficient when data sources are varied and outputs are as simple as a buy/sell command. As in the TensorFlow graphical architecture, using these constructs allow for automatic parallelization of models to better use modern hardware. Applications can also be built to fit multiple models, and updated independently from the rest of the system.
- Graphs are easy to interpret visually, which is useful for understanding the flow of data and interpreting output or bugs. They are also highly flexible, allowing users to modify pieces or generate new connections while keeping an enforceable system of data integrity.
- Perhaps most importantly, these graphs are extremely lightweight and portable, which is key for widespread distribution and access. While every piece can be accessed and tested on-the-go for better ease of development, they are ultimately just pieces of a bigger structure, where data flows continuously and leftover data is discarded automatically, keeping the memory and processing burden at a minimum when dealing with massive datasets.

1.4 Base Classes

These are the classes you will use throughout an analysis, or rather a class that implements their functionality. Getting to know them is important as it makes it easier to identify one when you see one and make it easier to search for one when you don't really remember where to find it.

1.4.1 External <_Node>

Manage connections between your environment and an external source.

Every model requires an origin to the data it uses, and often wants to send this data out again once it's processed. Subclasses of `External` will implement systems to manage the input and output of data to and from an external sources. An external source is any data or application located outside of your python environment. Two common examples are files and graphs. While these can be manipulated from the python environemt, the actual data is stored outside.

`External` class instances 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 supplicant object for each external connection.

As a child class of `_Node`,:code:*External* implements the `.request(**kwargs)` method, which takes in requests and executed valid ones on their external connections.

While this method is responsible for the main requests to and from the data, subclasses will often have other methods to perform more specific actions on it. Additionally, the `**kwargs` parameter will rarely be the same as the one relayed through the `_Transformer.run()` as `Translator` and `Application` instances will often curate these to be more generalizable to multiple `External` implementations.

What to Look For:

- What the external source is.
- Is it reliant on configuration? If so, what configuration parameters are required/considered?

1.4.2 _Transformer

Represent data transformations.

`_Transformer` instances are defined by their inputs and outputs. IO can be limited to one or more sources and the source can be either internal or external (as defined in *External <_Node>*).

All `_Transformer` instances implement the `.run(**kwargs)` method to:

1. Request source data from a `_Node` instance.
2. Apply specific transformations to the sourced data.
3. Return the transformed data.

This process will vary depending on the subclass, though the one thing to keep in mind is that the output of this method is what will be fed onto the next node on the graph, so it's a powerful tool for debugging.

`_Transformer` instances also define each input in their initialization by using `Validator` instances. You can find more about these in the developers-guide section on the *Validator* but for now, you can use the `_Transformer.describe()` method to get an idea of what kind of inputs this piece requires or prefers.

You won't be using these directly in your analyses, but will definitely use one of its subclasses.

What to Look For:

- Number of input and outputs.
- Sources/destinations of inputs and outputs.
- Input descriptions.

1.4.3 Translator <_Transformer>

Request and standardize external data.

One external input, one internal output

While `External` instances are the origin of all data, `Translator` instances are the root of all *clean and standardized* data. Objects of this class have `External` instances as their source and are tasked with creating requests understandable by that instance and standardize the response data into an acceptable format.

For more information on the Dal-io formatting standards, check out formatting.

All `Translator` instances implement the `.run(**kwargs)` method to:

1. Source data from an `External` instance.
2. Translate the data into a format as specified by the formatting guide.
3. Return the translated data.

These also tend to be the PipeLine stages where `kwargs` source from.

What to Look For:

- Compatible `External` instances.
- What translation format is being used and how will the output contain.
- What are the keyword arguments it can interpret.

1.4.4 Pipe <_Transformer>

Transform a single input into a single output.

One internal input, one internal output

Pipes will compose the majority of data wranging and processing in your graphs, and are designed to be easily extendable by users.

All pipes must implement the `.transform(data, **kwargs)` method, which takes in the output from sourced data and returns it transformed. This has three main purposes.

1. Subclasses can more objectively focus on transforming and outputting the `data` parameter instead of having to deal with sourcing it.
2. It makes it possible to use `Pipe` instances to transform data outside of the Dal-io library directly, which is useful for applications outside of the library's scope or for testing the transformation.
3. More efficient compatibility with *PipeLine <Pipe>* objects.

All `Pipe` instances implement the `.run(**kwargs)` method to:

1. Define input requirements.
2. Source data from another `_Transformer` instance, applying integrity checks.
3. Pass it as the `data` parameter to the `.transform()` method.
4. Return the transformed data.

While the default implementation of the `.run()` method simply sources data and passes into `.transform`, it is often changed to modify keyword arguments passed onto the source node and the .transform() call.

What to Look For:

- What are the input requirements.
- What the `.transform` method does.
- What are changeable attributes that affect the data processing.

1.4.5 Model <_Transformer>

Utilize multiple input sources to get one output.

Multiple internal inputs, one internal output

`Model` instances are a lot like `Pipe` instances as their main task it to transform inputs to get an output. Though taking in multiple inputs might not seem like enough to warrant a whole different class, the key differences come from all the extra considerations needed when creating a `Model` instance.

There are two main uses for `Model` instances:

1. Getting multiple inputs and joining them to form a single output.
2. Using the output of one of the inputs to format a request to another input.

These objectives thus require a lot more flexibility when it comes to sourcing the inputs, which is why, unlike `Pipe` instances, `Model` instances do not have a `.transform()` method, and instead rely solely on their `run()` method to:

1. Source data from inputs.
2. Process and transform data.
3. (Possibly) source more data given the above transformations.
4. (Possibly) join all sourced data.
5. Return the final product.

What to Look For:

1. All the input names and what they represent.
2. The requirements for each input.
3. How the `.run()` method deals with each input piece.
4. What changeable attributes affect the data processing.

1.4.6 Application <Model>

Act on external sources *Multiple internal inputs, zero or more external or internal outputs*

While you might be using Dal-io mostly for processing data for further use in your python session, `Application` instances offer methods of using this processed data to interact with external sources. These will be managed by `External` instances which are called by the application with data it sources from its inputs. These interactions can take a broad range of forms, from simple printing to the console to graphing, executing trade orders or actively requesting more data from the inputs. Ultimately, `Application` instances offer the greatest set of possibilities for users wanting to implement their own, as it is not bound by the scope of what the library can do.

All `Application` instances implement the `.run(**kwargs)` method to:

1. Source, validate, process and/or combine data from different inputs.
2. Use processed input data to send a request to an external source.
3. Get responses from external sources and further interactions.

What to Look For:

1. All the input names and what they represent.
2. The requirements for each input.
3. All the output names and what they represent.
4. How the `.run()` method deals with each input piece and how will it be transmitted to the output.

1.5 Extra Classes and Concepts

Now that we've seen what will make your models work, lets jump into what will make your models **work incredibly.**

1.5.1 PipeLine <Pipe>

As Pipe instances implement a normally small operation and have only one input and one output, you are able to join them together, through the __add__() internal method (which overrides the + operator) to create a sequence of transformations linked one after the other. These simply pass the output of one Pipe instance's .transform() method as the input to another, which can be a significant speed boost, though you should be careful with data integrity here.

KEEP IN MIND that good alternatives to these is just linking Pipe instances together in order to validate the data at every stage of the pipeline. This will have the same output as a PipeLine, but compromise on speed and possibly aesthetics.

1.5.2 Memory <_Transformer>

When using APIs to fetch online data, there is often a delay that ranges from a few to a few dozen seconds. This might be completely fine if data will only pass through your model once to feed an application, for example, but will become a problem if you are also performing analyses on several pieces of the model or have several Model instances in your graph (which call on an input once for every source). The solution to this lies in Memory instances that temporarily save model inputs to some location and retrieves it when ran.

Notice that Memory inherits from a _Transformer, which makes it compatible as input to any piece of your graph and behaves like any other input (most closely resembling a Pipe.)

Subclasses will implement different storage strategies for different locations. These will have their own data requirements and storage and retrieval logic - imagine the different in data structure, storage and retrieval required for storing data on a database vs on the local python session.

One thing to keep in mind is that these only store one piece of memory, so if you, for example, want to vary your .run() kwargs, this might not be the best option beyond building and debugging your model. If you still want the speed advantages of Memory while allowing for more runtime argument flexibility, check out the LazyRunner class below.

1.5.3 LazyRunner <_Transformer>

These objects are the solution to storing multiple Memory instances for different runtime kwargs that pass through the instance. These do not store the data itself, but rather the memory instances that do. This allows for more flexibility, as any single Memory subclass can be used to store the data. These are created when a new keyword argument is seen, and it does so by getting the data from a _Transformer input and setting its result as the source of a new Memory instance. The Memory type and initialization arguments are all specified in the LazyRunner initialization.

KEEP IN MIND that these could mean a significant memory burden, if you are widely saving data from different inputs with several kwargs combinations passed on to them.

The solution to the memory problem comes in the buffer= initialization argument of the LazyRunner. These will limit the number of Memory instances that are saved at any point. This also comes with the update= initialization argument for whether or not stored Memory instances should be updated in FIFO order once the buffer is full or whether an error should be thrown.

KEEP IN MIND that this will not notice if its source data input has any sort of input changes itself (this could be a change in date range, for example or data source.) This will become a problem as changes will not be relayed if the runtime kwargs are the same as before a change. This happens as the LazyRunner will assume that nothing changed, see the kwarg and return the (old) saved version of the response. This can be solved by calling the .clear() method to reset the memory dictionary.

1.5.4 Keyword Arguments

Just like data propagates forward in the network through nodes and transformers, requests propagate backwards through `.run()` and `request()` keyword arguments. Though often you won't need them (and much less often need to implement a new one), keyword arguments (aka kwargs) are a way on which a front piece of your graph can communicate with pieces before them at runtime. In essence, kwargs are passed from run to request over and over until they reach a node that can use them. These nodes can use these kwargs in different ways.They can:

- Use them to filter sourced data.
- Use them to create another request, based on previously-unknown information.

Though they might seem like an amazing way of making your graph act more like a function, adding new kwarg requirements should be done very rarely and done with full knowledge of what are the taken kwarg names, as conflicting names will certainly cause several unforeseen bugs.

1.6 Tips and Tricks

1.6.1 Importing

As any other python (or any other programming language) workflow, we start with imports. Dal-io will often require several pieces to be used in a workflow, each of which is located within a submodule named after the base classes we have seen above. This means that importing the whole `dalio` package and instantiating piece by piece will often create unappealing code, which is why the following techniques are preferred.

Import submodules with an alias:

```
import dalio.external as de
import dalio.translator as dt
import dalio.pipe as dp
import dalio.model as dm
import dalio.application as da
```

This technique might not be the most standard or space efficient, but is very useful when you are still testing out models and architectures. For most worflows where you want to try out new paths and strategies, having these imports will give you all the core functionality you need while keeping your code clean.

Import specific pieces from each submodule:

```
from dalio.external import YahooDR, PyPlotGraph

from dalio.translator import YahooStockTranslator

from dalio.pipe import (
    Change,
    ColSelect,
    Custom,
    DateSelect,
)

from dalio.model import (
    OptimumWeights,
    OptimumPortfolio,
)

from dalio.application import Grapher
```

This doing this is more standard to match common workflows like those in `keras` and `sklearn` though can easily grow out of hand ind a Dal-io workflow, especially when trying to experiment with new inputs and pieces.

This is preferred once you have created a graph you are happy with and is ready for use. Importing all pieces explicitly not only makes your code more readable, it also makes the used pieces more explicit to the ones reading your implementation.

A use hybrid approach:

```
from dalio.external import YahooDR, PyPlotGraph
from dalio.translator import YahooStockTranslator

import dalio.pipe as dp
import dalio.model as dm

from dalio.application import Grapher
```

This approach is a great way of reconciling both importing workflows, as it keeps the most relevant pieces of the graph explicit (the original input, the application and the final output) while giving you flexibility of accessing all `Pipe` and `Model` pieces available for testing.

1.6.2 The Basic Workflow

Now that you are familiar with the most common parent classes used in the Dal-io system and ways of importing them, we can start talking about how a basic workflow with them will tend to look like.

We will separate our basic workflow into the following steps.

1. Set up imports.
2. Set up core data sources.
3. Data wrangling and processing.
4. Application set-up.

Set up imports:

This is the stage where you use set up and configure any `External` object instances and set them as inputs to a Translator. This defines the core of the data that will be sent to the rest of your graph, so it is always positive to have test runs of this raw input.

Set up core data sources:

Now that you have your inputs, perform any sort of transformations which will further standardize it to your specific needs. These can be selecting specific columns (like only the "close" column if your source gets OCHLV data) or joining sources.

This is an optional, yet often relevant step, and you should see this as a preparation to the data that will feed every step following this.

If we were to picture a graph with various nodes and edges which source data from a single node, this step is setting up a few nodes between the source and the actual first node that other pieces often get data from. In other words, no other pieces but the ones used in this step will be interacting with the pieces that come before it.

Data wrangling and processing:

This is the most general step and is all about setting up processing pipelines for your data. This might involve performing transformations, joining sources into models and maybe even setting up different diagnostic applications midway. Theres no overwhelming structure to these other than setting up the inputs that will feed your last nodes.

Application set-up:

While applications are not a requirement for a graph, they are often the very last nodes in one. Above that, `Application` instances often have the largest burden of setup, so deciding all of their pieces and putting together inputs is a common last step.

Once applications are set up, the following analysis will be for the most part a process of actually using it or optimizing your results by tweaking some of the steps done previously.

1.6.3 When Reading the Docs

I find it that reading the docs can be a completely different experience depending on the package I am researching. Whether you want to find out whether a specific process currently exists in the Dal-io library or if you just want to get more specifications on a single piece you know exists, there are a couple of breadcrumbs left as part of the documentation structure that where placed to guide you there.

Know how your piece fits:

As you have seen throughout the beginners guide, every Dal-io piece inherits from a base class, which represents a certain state of data or transformation. Knowing well what you are looking for in terms of these states or transformations can go a long way on trying to find the submodule to look for the piece.

You can ask questions like:

- Is this a transformation on data or a representation of data?
- How many inputs does this transformation have?
- Are there any external inputs or outputs involved in this specific piece?

Beyond the base class submodules, these are further organized into different script folders to ensure there is further separation of what the base class implementations do. Definitely see what are the current available submodule "categories" to further narrow your search. The good thing is that while there are separated into links in the user-modules page, they are all joined together into the same specific submodule page.

Know how to explore your piece:

Once you have pinpointed your piece, explore its definition or source code to know how to fully utilize it in your specific case. While one could argue that only by going through the source could one fully understand an implementation's full potential, this is often a tedious approach and definitely not beginner-friendly.

If you want to cut to the chase when it comes to knowing a function, look for the things specified under the "What to look for" sessions on each of the base class descriptions above.

1.6.4 Must-Know Classes

Now that you are fully armed with the knowledge needed to venture into the package, let's get you introduced to a couple of pieces the development team (currently composed of one) has used with frequently.

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

Apply custom function.

strategy

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

Type str, default "pipe"

Example

```
>>> import pandas as pd; from dalio.pipe import Custom;
>>> data = [[3, 2143], [10, 1321], [7, 1255]]
>>> df = pd.DataFrame(data, [1,2,3], ['years', 'avg_revenue'])
>>> total_rev = lambda row: row['years'] * row['avg_revenue']
>>> add_total_rev = Custom(total_rev, 'total_revenue', axis=1)
>>> add_total_rev.transform(df)
   years  avg_revenue  total_revenue
1      3         2143           6429
2     10         1321          13210
3      7         1255           8785
>>> def halfer(row):
...     new = {'year/2': row['years']/2,
...            'rev/2': row['avg_revenue']/2}
...     return pd.Series(new)
>>> half_cols = Custom(halfer, axis=1, drop=False)
>>> half_cols.transform(df)
   years  avg_revenue   rev/2  year/2
1      3         2143  1071.5     1.5
2     10         1321   660.5     5.0
3      7         1255   627.5     3.5
```

```
>>> data = [[3, 3], [2, 4], [1, 5]]
>>> df = pd.DataFrame(data, [1,2,3], ["A","B"])
>>> func = lambda df: df['A'] == df['B']
>>> add_equal = Custom(func, "A==B", strategy="pipe", drop=False)
>>> add_equal.transform(df)
   A  B   A==B
1  3  3   True
2  2  4  False
3  1  5  False
```

__init__ (*func*, **args*, *columns=None*, *new_cols=None*, *strategy='apply'*, *axis=0*, *drop=True*, *reintegrate=False*, ***kwargs*)
Initialize instance and set up input DataDef.

In Pipe instance initializations, data definitions are described and attributes are checked.

The `Custom` pipe does what the name implies: it applies a custom transformation to an input `pandas.DataFrame` instance. It inherits most of its functionality from the :code"*_ColGeneration* abstract class, so reading its description will help you understand how flexible your transformations can be when it comes to reintegrating it back into the original dataframe while keeping its column structure intact.

class dalio.pipe.col_generation.**_ColGeneration**(**args*, *columns=None*, *new_cols=None*, *axis=0*, *drop=True*, *reintegrate=False*, ***kwargs*)
Generate column based on a selection from a dataframe.

These are very useful for simple operations or for testing, as no additional class definitions or understanding of the documentation is requred. .. attribute:: columns

Column labels in the DataFrame to be mapped.

type single label or list-like

func
The function to be applied to each row of the processed DataFrame.

Type callable

result_columns
If list-like, labels for the new columns resulting from the mapping operation. Must be of the same length as columns. If str, the suffix mapped columns gain if no new column labels are given. If None, behavior depends on the replace parameter.

Type str or list-like, default None

axis
axis to apply value funciton to. Irrelevant if strategy = "pipe".

Type int, default 1

drop
If set to True, source columns are dropped after being mapped.

Type bool, default True

reintegrate
If set to False, modified version is returned without being placed back into original dataframe. If set to True, an insertion is attemtped; if the transformation changes the data's shape, a RuntimeError will be raised.

Type bool, default False

_args
arguments to be passed onto the function at execution time.

_kwargs
keyword arguments to be passed onto the function at execution time.

__init__ (**args*, *columns=None*, *new_cols=None*, *axis=0*, *drop=True*, *reintegrate=False*, ***kwargs*)
Initialize instance and set up input DataDef.

In Pipe instance initializations, data definitions are described and attributes are checked.

transform(*data*, ***kwargs*)
Apply custom transformation and insert back as specified

This applies the transformation in three main steps: 1. Extract specified columns 2. Apply modification 3. Insert columns if needed or return modified dataframe

These steps have further details for dealing with levels.

Raises **RuntimeError** – if transformed data is to be reintegrated but has a different shape than data being reintegrated on the dataframe.

This class is extremely important as it essentially the user's first point of entry into creating their custom transformations. `Custom` pipes work by applying your specified function to either the dataframe's rows or columns (specified through the :code"*axis* parameter).

The application itself is divided into different pandas strategies (specified through the :code"*strategy* parameter, set to `\"apply\"` by default.) The strategies correspond to `pandas.DataFrame` methods, really, so if you want to get to the specifics of its, just read the pandas documentaion for the `\"apply\"`, `\"transform\"`, `\"agg\"` and `\"pipe\"` descriptions. But for most cases, you will be using two strategies.

- `\"apply\"`: here, each row or column is passed onto the custom function as `pd.Series` instances. This is the most generic strategy and should used the most often.
- `\"pipe\"`: unlike `\"apply\"`, here the whole dataframe is passed onto your custom function at once, which can be useful for experimenting with specific functions you might want to implement as a piece later.

class dalio.pipe.select.**DateSelect**(*start=None*, *end=None*)
Select a date range.

This is commonly left as a local variable to control date range being used at a piece of a graph.

_start
start date.

Type pd.Timestamp

_end
end date.

Type pd.Timestamp

set_end(*end*)
Set the _end attribute

set_start(*start*)
Set the _start attribute

This piece also has a name as intuitive as what it does. It essentially takes in a time series `pandas.DataFrame` (one which has a `pandas.DatetimeIndex` as its index) and returns a subset of its dates. What makes it so powerful is its use as a "remote control" for your input time interval.

This effectively gives you an adjustable "filter" that can be adjusted at any point of your analysis to decide what section of the data to perform it on, which is crucial in various kinds of time series analyses.

For an interesting use case of this, check out the backtesting cookbook!

1.7 Core Classes and Concepts

1.7.1 Validator

Validators are the building blocks of data integrity in the graph. As modularity is key, validators ensure that data sourced from a `_DataDef` is what it is mean to be or that errors are targeted to make debugging easier. Validators can have any attribute needed, but functionality is stored in the .validate function, which either passes warning data on or stops execution with an error. These can and should be reused with multiple `_DataDef` instances.

1.7.2 _Node

Node instances represent data. They have a connection to some data input, internal or external, and make requests to this data as well as ensure their integrity. These form the basis for External and `_DataDef` classes.

1.7.3 _DataDef <Node>

`_DataDef` instances are sources of data and implement mechanisms to ensure the integrity of that data, as input from sources is uncertain.

KEEP IN MIND that this is a tool only used by developers while creating new transformations, actual users do not enter in contact with neither `Validator` nor `_DataDef` instances.

Validation: In order to hold descriptions true, the data is validated by a chain of `Validator` functions before returning any actual data, in order to ensure that if data is actually returned, it is accurate to its descriptions and won't break the subsequent transformation. These are referred to as descriptions inside `_DataDef` instances and are added to them upon initialization of a Transformer instance.

Speed Concerns: While it's understandable that these might pose a significant speed burden to applications, they are designed to reduce these by as much as possible. Firstly, validations are not dependent on each other and can thus be parallelized. Also, they can be turned off as needed, though this must be done with caution.

1.7.4 _Builder

Builders are a solution to the problem of standardizing several package workflows into something more consistent to the inexperienced user.

Take the `MakeARCH` builder as an example. In the arch package, users have to assemble an ARCH model starting with an arch.mean model initialized _with_ the data, followed by setting arch.variance and arch.distribution objects, each with their own respective parameters. Keeping this interface would have been highly inflexible and required the user to essentially learn how to use the package from scratch. Inheriting from `_Builder` allowed the `MakeARCH` pipe to maintain this flexibility of setting different pieces as well as creating the model's structure before actually having any data (which wouldn't be possible with the original package).

1.8 Development Notes on Base Classes

1.8.1 External <_Node>

Configuration: Sources often require additional ids, secrets or paths in order to access their data. The .config attribute aims to summarize all key configuration details and data needed to access a resource. Additional functions can be added as needed to facilitate one-time connection needs.

Factories: Sources, typically web APIs, will give users various functionalities with the same base configurations. The .make() method can be implemented to return subclasses that inherit parent processing and configuration.

1.8.2 Translator <_Transformer>

1.8.3 Pipe <_Transformer>

1.8.4 Model <_Transformer>

1.8.5 Applications <Model>

1.9 Key Concepts, Differences and Philosophy

1.9.1 running vs requesting

You might have notices that classes that inherit from <Pipe> have .run() methods, classes that inherit from <Node> have .request() methods, both of which return some form of data. While these two essentially have the same output functionality, they differ in implementation, where .run() methods get data from a source and modifies is while .request() methods get data, also from some source, and validates it. Thus, the idea of a `_DataDef` compared to a Pipe becomes clearer.

1.9.2 describing vs tagging

The `.tags` and `.desc` attributes might seem to be redundant, as both are used to describe some sort of data passing by them and both can be used to search for nodes in the graph. Firstly, and most importantly, the `.desc` attribute is common to all `_DataDef` instances that inherit from another `_DataDef`, while the .tag attribute is unique to that node, unless it is also present on the parent `_DataDef` or shared with other DataDefs upon instantiation.

They also do defer in "strictness," as tags will not be checked for truthfulness, while descriptions will be tested on the data, unless, of course, users turn checking off. Tags are included as a feature to allow more flexible, personalizeable descriptions that describe groups or structures within the graph rather than a certain functionality.

CHAPTER

TWO

TABLE OF CONTENTS

CHAPTER

THREE

INTRODUCTION

Dal-io is a financial modeling package for python aiming to facilitate the gathering, wrangling and analysis of financial data. The library uses **graphical object structures** and **progressive display of complexity** to make workflows suit the user's specific proficiency in python without making efficiency sacrifices.

The core library implements common workflows from well-supported packages and the means to flexibly interlink them, and aims to continue adding relevant features. However, the user is not constrained by these features, and is free to extend pieces through inheritance in order to implement extra functionality that can be used with the rest of the package. See *Core Classes and Concepts* for more information on extending core features.

CHAPTER

FOUR

INSTALLATION

You can clone this repository from git using

```
git clone https://github.com/renatomatz/Dal-io
```

If you are using Windows, make sure you are in the package folder to use the functionality and that you run the following command before importing the modules.

```
import sys
sys.path.append("/path-to-dalio/Dal-io")
```

For Linux and Mac, you can access the package contents from your python environment anywhere with

```
export PYTHONPATH=$PYTHONPATH:"path/to/Dal-io"
```

CHAPTER

FIVE

A GUIDED EXAMPLE

Let's go through a quick example of what Dal-io can do. We'll build a simple portfolio optimization workflow and test it out with some sample stocks.

This example will be fairly dry, so if you want to jump right into it with some understanding of the Dal-io mechanics, you can go through the *Understanding Graphs* first. If you just want to see what the library is capable of, let's get right to it.

We'll start off by importing the Dal-io pieces

```
import numpy as np

import dalio.external as de
import dalio.translator as dt
import dalio.pipe as dp
import dalio.model as dm
import dalio.application as da
```

Specific pieces can also be imported individually, though for testing this sub-module import structure is preferred.

Now lets set up our stock data input from Yahoo! Finance.

```
tickers = ["GOOG", "MSFT", "ATVI", "TTWO", "GM", "FORD", "SPY"]

stocks = dt.YahooStockTranslator()\
    .set_input(de.YahooDR())
```

Easy right? Notice that the stock input is composed of one external source (in this case `de.YahooDR`) and one translator (`dt.YahooStockTranslator`). This is the case for any input, with one piece getting raw data from an external source and another one translating it to a format friendly to Dal-io pieces. For more on formatting, go to formatting.

Notice the `.set_input` call that took in the YahooDR object. Every all translators, pipes, models and applications share this method that allows them to plug the output of another object as their own input. This idea of connecting different objects like nodes in a graph is at the core of the **graphical object design**.

At this point you can try out running the model with `stocks.run(ticker=tickers)` which will get the OHLCV data for the ticker symbols assigned to :code"*tickers*, though you can specify any ticker available in Yahoo! Finance. Notice that the column names where standardized to be all lower-case with underscores (_) instead of spaces. This is performed as part of the translation step to ensure all imported data can be referenced with common string representations.

Now lets create a data processing pipeline for our input data.

```
time_conf = dp.DateSelect()
```

(continues on next page)

(continued from previous page)

```
close = dp.PipeLine(
    dp.ColSelect(columns="close"),
    time_conf
)(stocks)

annual_rets = close + \
    dp.Period("Y", agg_func=lambda x: x[-1]) + \
    dp.Change(strategy="pct_change")

cov = dp.Custom(lambda df: df.cov(), strategy="pipe")\
    .with_input(annual_rets)

exp_rets = annual_rets + dp.Custom(np.mean)
```

That was a bit more challenging! Let's take it step by step.

We started off defining a `DateSelect` pipe (which we will use later) and passing it into a pipeline with other pipes to get a company's annual returns. Pipelines aggregate zero or more Pipe objects and pass in a common input through all of their transformations. This skips data integrity checking while still allowing users to control pipes inside the pipeline from the outside (as we will with :code"*time_conf*)

We then added a custom pipe that applies the np.mean function to the annual returns to get the expected returns for each stock.

Finally, we did the exact same thing but with a lambda that calls the pd.DataFrame internal method .cov() to get the dataframe's covariance. As we will be passing the whole dataframe to the function at once, we set the Custom strategy to "pipe".

Notice how we didn't use `.set_input()` as we did before, that's because we utilized alternative ways of establishing this same node-to-node connection.

We can connect nodes with:

1. `p1.set_input(p2)` set p1's input to p2.
2. `p1.with_input(p2)` create a copy of p1 and set its input to p2.
3. `p1(p2)` same as `p1.with_input(p2)`.
4. `pL + p2` set p2 as the last transformation in the PipeLine pL.

Now let's set up our efficient frontier model, get the optimal weights and finally create our optimal portfolio model.

```
ef = dm.MakeEfficientFrontier(weight_bounds=(-0.5, 1))\
    .set_input("sample_covariance", cov)\
    .set_input("expected_returns", exp_rets)\

weights = dp.OptimumWeights()(ef)\
    .set_piece("strategy", "max_sharpe", risk_free_rate=0.0)

opt_port = dm.OptimumPortfolio()\
    .set_input("weights_in", weights)\
    .set_input("data_in", close)
```

And those are two examples of Dal-io Models! As you can see, models can have multiple named inputs, which can be set the same way as you would in a pipe but also having to specify their name. You also saw an example of a Builder, which has pieces (that can be set with the `.set_piece()`) method which allow for more modular flexibility when deciding characteristics of certain pipes or models.We could go into what each source and pieces represents, but that can be better done through the documentation.

Now, as a final step, lets graph the performance of the optimal portfolio.

```
graph = da.PandasXYGrapher(x=None, y="close", legend="upper_right")\
    .set_input("data_in", dp.Index(100)(opt_port))\
    .set_output("data_out", de.PyPlotGraph(figsize=(12, 8)))
```

Additionally, you can change the time range of the whole model at any point using the `time_conf` object we created all the way in the beginning. Below is an example of setting the dates from 2016 to 2020.

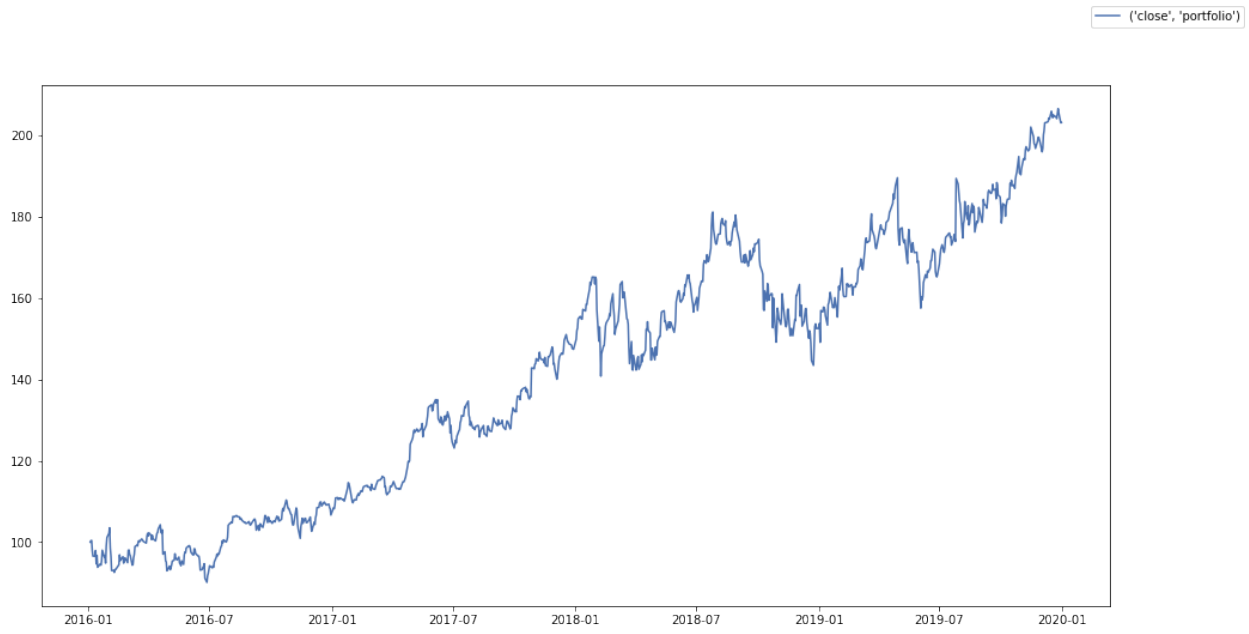
```
time_conf.set_start("2016-01-01")
time_conf.set_end("2019-12-31")
```

And that's it!

All that you have to do now is run the model with `graph.run(ticker=tickers)` to

1. Get stock data from Yahoo! Finance
2. Process data
3. Optimize portfolio weights
4. Get an optimum portfolio
5. Graph optimum portfolio

Which yields this figure:



Notice how this `.run()` call was the same as you did all the way back when you only had your imported data. This method is also common to all translators, pipes, models and applications, and it gives you the piece's output.

This means you can get information of any of the stages you created like this, and for any stock that you'd like. For example, we can run the `weights` object we created to get the weights associated with the portfolio we just plotted.

```
weights.run(ticker=tickers)
```

```
{'GOOG': 0.45514,
 'MSFT': 0.82602,
```

(continues on next page)

(continued from previous page)

```
'ATVI': -0.49995,
'TTWO': 0.29241,
'GM': -0.43788,
'FORD': 0.38413,
'SPY': -0.01986}
```

Also, every time you run a set of stocks or time intervals, the new run will be automatically layered with the old one and indexed at 100, which can be great for comparing how multiple portfolios would have fared! To clear this, just re-define the graph.

Hope this example was enough to show how you can create clean and powerful models using just a few lines of code!

CHAPTER

SIX

NEXT STEPS

If you read and enjoyed the example above, that's great! Now comes the part where you get to understand its various pieces, workflows and internal logic for you to start creating your own models with Dal-io.

A good first step, if you haven't already is reading the *Understanding Graphs*.

If you understood these core concepts well and are ready for some more examples, check out the cookbook.

For those who want to adventure into creating your own pieces (and hopefully contributing to the library) can read the *Core Classes and Concepts* as well as the formatting.

And as always, you can check the full breakdown of the modules with the ol' reliable *User Modules*.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y