SmartBetas Code Documentation

Release 0.1

Elena Pfefferlé

May 25, 2019

Table of Contents

CHAPTER 1

SmartBetas Investing

This program is aimed at testing Smart Betas strategies. It allows users to build portfolios and virtually invest in them. Users can also verify if these strategies work by asking the program to compute returns on their virtual investments.

`main.py` is the first module loaded when a user starts the program, it initiates the application.

The program is built on a database, the communications between the program and the db is managed by *pydal* [1] (Python Database Abstraction Layer). The module checks if *pydal* is installed on the host; if not, the program outputs a message and closes.

The module will also check if the host is connected to the internet, this is required because the data used to compute portfolios is pulled from a web API. If the host is not connected to the internet, the program will output a message and the program will close.

The connectivity test is performed by trying to open a socket [2] on google website's TCP port 80 (*www.google.com:80*).

This module also clears the command line window prior starting the console.

Note To start the program, one must call python on the folder containing the *__main__.py* module.

---

1 pyDal's API documentation: https://pydal.readthedocs.io/en/latest/
2 Python socket - Low-level netorking interface: https://docs.python.org/3/library/socket.html

CHAPTER 2

Database Layer

`db.py` manages the database where the application is storing operations, tickers, portfolios and reports.

The python layer that manages the database is provided by *pyDal* [3], this is an open source python package which acts as an abstraction layer between an application and a database.

pyDal can handle multiple types of database, in application there are no intensive database operations therefore we can use sqlite3 [4].

When imported the module is checking if the *db* folder (located at the same level than the application folder) is present. The folder is created if not found.

The database is initialized with the following statement:

```
folder = os.getcwd()+('/db' if os.name != 'nt' else '\db')
db = DAL('sqlite://storage.sqlite', folder=folder, migrate_enabled=True)
```

Where *folder* is the database location and *migrate_enabled* indicates to *pyDal* that it should not assume that the tables already exists, but rather check and compare against the *define_table* statements.

For this project we defined the following tables:

Investments: Contains the investments made by the user, each entry features a *date*, a *name*, a volatility based portfolio *vol*, a momentum based portfolio *cmr*, a composite based portfolio *cmp* and a register containing the prices of all the stocks listed in each portfolio (the portfolios and the price register are stored in *json*).

```
{
    "ACB": 1832,  //ticker: quantity
    "AMD": 597,
    "KEYW": 1487,
    ...
}
```

Portfolios: Contains the portfolios that the user chose to save after computing them. Each entry features a *date*, a *name* a volatility based portfolio *vol*, a momentum based portfolio *cmr*, a composite based portfolio *cmp* and a register containing the prices of all the securities listed in each portfolio (the portfolios and the price register are stored in *json*).

```
{   //example with momentum
    "AAPL": 6.5261228230980795,   //ticker: momentum
    "ACB": 27.887788778877898,
    "AMD": 149.33058702368692,
    ...
}
```

Symbols: Contains the tickers and their corresponding names, each entry features a *ticker* and a *name*.

3 pyDal's API documentation: https://pydal.readthedocs.io/en/latest/
4 SQLite3 documentation: https://www.sqlite.org/draft/version3.html

Checks: Contains the portfolios returns reports generated by users and saved in the database. Each entry features a *name*, a volatility based portfolio *vol*, a momentum based portfolio *cmr*, a composite based portfolio *cmp* and *date*. (the portfolios are stored in *json*).

The *json* structure for *checks* table is as follow:

```
{
"ACB": {
    "abs": -109.92,                // absolute returns
    "date": "2019-04-22 16:57:42", // data of
    "new": 9.04,
    "old": 9.1,
    "qty": 1832,
    "rel": -0.66
},
...
```

CHAPTER 3

Tickers - User Input

The main user's data inputs are tickers which will be used to form portfolios. These inputs must be thoroughly checked and stored locally (in a database) for future usage.

`tickers.py` is handles operations related to tickers.

`tickers.`**`save`** (*ticks*)

Saves toclers names into a global variable and into the database.

Parameters:

- *ticks* : `list` of `string` , each one is a tickert.

The function will check for each ticker if an entry already exists in the database, if not, it will attempt to fetch the ticker's name from a web API.
If the fetch is successfull, the result is written into the database. This way next time the ticker is used, the application will not need to pull data from the web API.
The function also sets the global *TICKERS* and *NAMES* `list` with the tickers and their corresponding names.
When a ticker can not be identified on the web API, it is not stored in the global variable nor in the database, but in a list that will be returned.
Returns a `list` containing tickers that caused errors.

`tickers.`**`save_portfolio`** (*vol*, *cmr*, *cmp*, *prices*, *name*)

Writes the computed portfolios into the database.

Parameters:

- *vol* : `dict` volatility portfolio.
- *cmr* : `dict` momentum portfolio.
- *cmp* : `dict` composite portfolio.
- *prices* : `dict` price register.
- *name* : `string` name of the portfolio.

Saves into the database the items passed as argument.
Returns *True* if it succeeded.

`tickers.`**`validate`** (*raw*)

Checks the content of the data provided by the user.
Users provide tickers to the application by writing them into a file that is loaded through the console interface with the <load filename> command.
We expect the file to be filled with coma separated tickers `string`.

Parameters:

- *raw* : `string` content of the user provided file.

The function strips the raw data from spaces, carrier returns and split the content around comas. It will also check if there are trailing comas or if the user mistakenly put two comas instead of one between tickers.

Returns a `list` of sanitized tickers

CHAPTER 4

Web API Interface

A web API [5] is an Application Programming Interface for a web server or a web browser. On the server side, it consists or one or more publicly exposed endpoints accessible through *http* requests. The content is typically expressed in JSON (JavaScript Object Notation) or XML (eXtensible Markup Language).

`smartbetas` relies on two APIs. One is used to convert tickers to company names (As example: AAPL -> Apple Inc), the second one is used to get information about a stock (daily open, daily high, daily low, daily close and daily volume).

The ticker API is totally free and open source, it is maintained by a single contributor (Kambala Yashwanth), this API simply takes a ticker as input and returns the name of the stock if found [6].

Alpha Vantage provides the stock data API. This is not a free service, however they provide a free limited access to their data. The following constraints are applicable when using a free API Key [7]:

- 5 API requests per minute
- 500 API requests per day

Note that Alpha Vantage requires free users to register to get their API key. Our key is stored in `__gbl__`.

`api.py` handles communications between the host and the web APIs.

`api.`**`symbol`** (*tick*)

Determines the name of a stock based on a ticker.

Parameters:

- *tick* : `string` stock's ticker.

Constructs the request URL with the ticker passed as argument then makes an *http* request and parses the data received from the API.
Json Data sample from the API: `[{"symbol":"AAPL","name":"Apple Inc."}]`
Returns a `string` contraining the name of the sock OR `None` if the API failed to return data for the ticker (as example, if a wrong ticker was provided).

`api.`**`tsd`** (*tick*)

Fetches a stock daily data from 20 years ago till now.

Parameters:

- *tick* : `string` security ticker.

The function builds the URL, makes the *http* request to the web API and parses the received JSON, then to assert that data was indeed returned, it checks the length of the data:

5 Wikipedia Web API: https://en.wikipedia.org/wiki/Web_API
6 Ticker Search API doc: https://github.com/yashwanth2804/TickerSymbol
7 Alpha Vantage API Documentation: https://www.alphavantage.co/documentation/

- length = 2 : Data returned
- length < 2 : No data returned

If no data is returned, the function assumes that the maximum amount of calls per minute is reached; it will wait for 15 seconds and retry; if it fails again it will wait again till data is returned.

Note If the maximum amount of daily request is reached, the function will loop forever.

Data sample:

```
{
"Meta Data": {
    "1. Information": "Daily Prices (open, high, low, close) Volumes",
    "2. Symbol": "AAPL",
    "3. Last Refreshed": "2019-04-26",
    "4. Output Size": "Full size",
    "5. Time Zone": "US/Eastern"
},
"Time Series (Daily)": {
    "2019-04-26": {
        "1. open": "204.9000",
        "2. high": "205.0000",
        "3. low": "202.1200",
        "4. close": "204.3000",
        "5. volume": "18611948"
    },  // max 20 years of values
    ...
```

Once the data was obtained, the function will create a `list` of `tuple` containing in each position a datetime object and the closing price of the stock for that date. (List starts with the date closest to now)
List sample : `[(datetime.datetime(2019, 4, 18, 0, 0), 203.86)]`
Returns a `list` of `tuple` containing dates and closing prices.

CHAPTER 5

Building Portfolios

This modules uses data returned from the web API (*api.tsd()*) to compute the volatility and momentum of each stock.

Based on the stock ranking it also builds a composite portfolio *composite.composite()*.

To expedite things, this module uses python's *concurrent.futures* [8] to fetch data from the API in a parallel fashion. Executing *http* requests in parallel saves a considerable amount of time.

`compute.py` is used to compute volatility, momentum and composite based porfolio.

`compute.`**`compute`** (*tick*)

Returns the volatility, momentum and price of a stock.
This function gets the tickers data with *api.tsd()*, and uses this data to compute the volatility with `volatility.volability()` and the momentum with the module *cumulative.ret()*.

- Gets data from the web API - *api.tsd()*
- Computes volatility - *volatility.volatility()*
- Computes momentum - *cumulative.ret()*

Returns a `list` containing the volatility, the momentum and the price of a ticker.

`compute.`**`smartbetas`** (*tickers*)

Builds three portfolios based on volatility, momentum and a composite score.
Parameters:

- *tickers* : `list` of tickers to compute

Uses `ProcessPoolExecutor.map` to launch parallel processes running *compute.compute()* on each stock passed as argument.
The output of the parallel processing is stored in a `list` containing for each ticker a `list` of `tuples`.
Sample:

```
[[('t1', volatility), ('t1', momentum), ('t1', price)],
[('t2', volatility), ('t2', momentum), ('t2', price)]]
```

The volability portfolio is built by sorting the `list` in an ascending fashion and extracting only the tickers and their volatility.
The momentum portfolio is built by sorting the `list` in a descending fashion and extracting the tickers and their momentum.
The prices are also extracted and stored to allow for future comparisons against the market prices.
The composite portfolio is computed with *composite.composite()*.
The results are stored in the global variables.
Returns nothing

8 Python `concurrent.futures`: https://docs.python.org/3/library/concurrent.futures.html

5.1 Volatility

In this application, volatility is calculated over a period of two years (between two years ago and now).

To compute the volatility, a standard deviation needs to be performed. To do so we use the standard python's statistics [9] package.

`volatility.py` is used to compute the volatility of a stock.

`volatility.`**`volatility`** (*data*)

Computes the volatility of a stock.

Parameters:

- *data* : `list` of `tuples`

Note The function's argument is provided by *`api.tsd()`*, each item in this list is structured as follow: `[(datetime.datetime(2019, 4, 18, 0, 0), 203.86)]`

The function uses `datetime` to determine the dates between which the volatility shall be computed. A new `list` in which the daily returns between the right time frame is built.
We then use *statistics.stdev* to compute the standard deviation of the list.
Returns a the volatility of the security (`float`)

5.2 Momentum

We define momentum as the cumulative returns over a period of 10 months (between 12 months ago and 2 months ago).

All the date and time operations are handled with python's standard *datetime* module [10].

`cumulative.py` is used to compute cumulative returns; also known as momentum.

`cumulative.`**`momentum`** (*vector*)

Computes the momentum of a vector.

Parameters:

- *vector* : `list` of `floats`

Moentum is computed as follow:

- momentum = 100*(new - old)/old

Returns the momentum of the vector (`float`)

`cumulative.`**`ret`** (*data*)

Computes the cumulative returns (momentum) of a security.

Parameters:

- *data* : `list` of `tuples`

Note The function argument is built by *`api.tsd()`*, it is structured as follow: `[(datetime.datetime(2019, 4, 18, 0, 0), 203.86)]`.

---

9 Python statistics: https://docs.python.org/3/library/statistics.html
10 Python datetime: https://docs.python.org/3/library/datetime.html

The function uses `datetime` to determine the dates between which the momentum shall be calculated. A new `list` containing only the values in the right time frame is built and passed to *`cumulative.momentum()`*.
Returns the momentum of the security (`float`)

5.3 Composite

In this application we define a composite portfolio as a portfolio based on the ranking of stocks in the volatility portfolio and in the momentum portfolio.

The score of a stock is the sum of its index (rank) in the two other portfolio. The stocks are then sorted in an ascending fasion.

`composite.py` is used to build a composite portfolio.

`composite.`**`composite`** (*vol*, *cmr*)

Ranks securities in a composite fashion.

Parameters:

- *vol* : `dict` volatility portfolio.
- *cmr* : `dict` momentum portfolio.

Note at this point, the same tickers are present in both portfolios. Their ranking only is different.

The function builds a `dict` with the tickers and set their score to zero; sample {'ticker': 0}. Then it adds to the ticker score their index in volatility and momentum portfolio.
The tickers are then sorted ascendingly, after having been transformed into a `tuple`.
Returns a `dict` containing tickers and their score.

Investing in the Portfolios

In the application, after a user requested the program to generate portfolios with the tickers he provided, he can choose to invest money in each of them.

`money.py` handles investments in the computed portfolios.

`money.`**`invest`** (*p_size*, *p_vol*, *p_cmr*, *p_cmp*, *prices*, *s_name*, *stack=100000*)

Invests 100‘000 USD in each computed portfolio.

Parameters:

- *p_size* : `int` amount of tickers per portfolio.
- *p_vol* : `dict` volatility portfolio.
- *p_cmr* : `dict` momentum portfolio.
- *p_cmp* : `dict` composite portfolio.
- *prices* : `dict` price register.
- *s_name* : `string` name of the investment.
- *stack* : `int` defaults to 100000, money to invest.

In each portfolio, the function determines how many shares of stock can be purchased. A `dict` containing the stocks and the quantity of shares purchased is built.
The function then stores the outcomes into the database, the prices of the securities are also stored for future usage.
Returns nothing.

CHAPTER 7

Measuring Results

Using the latest prices of the stocks (fetched from the web API), this module will compute the returns of each portfolios in which the user invested.

To expedite things, this module uses python's *concurrent.futures* to fetch data from the API in a parallel fashion. Executing *http* requests in parallel saves a considerable amount of time.

`check.py` is used to determine the returns of an investment.

`check.`**`get_price`** (*tick*)

Gets the current price of a stock.

Parameters:

- *tick* : `string` stock ticker

Fetches the latest price of the stock from the web API. The result is stored in a `tuple` : `` `('ticker', 149) ``

Returns a `tuple` containing the ticker and its latest price.

`check.`**`returns`** (*inv*, *pfl*, *prices*)

Computes the returns of a portfolio.

Parameters:

- *inv* : `list` past investment session row from the db
- *pfl* : `string` name of the portfolio (can be *vol*, *cmr* or *cmp*)
- *prices* : `dict` latest investment's ticker prices

Computes the absolute change and the returns for each stock in the portfolio. These values are calculated as follow:

- Absolute change : (*qty`*`new*) - (*qty`*`old*) (*rounded at 2 digits*)
- Returns : [(*new-old*)/*old`*`100*] (*rounded at 2 digits*)

The total absolute change and return is handled by *`check.t_returns()`*.

Returns a `dict` containing for each stock the initial price, the new price, the absolute change, the returns, the quantity of shares and the date of the intial purchase.

`check.`**`sessions`** (*id*)

Computes returns of an investment session.

Parameters:

- *id* : `integer` investment session database id.

Uses `ProcessPoolExecutor.map` to launch parallel processes running `check.get_prices()` on each stock passed as argument.

The returns of the portfolios are calculated by *`check.returns()`*.

Once the returns are computed, the results are saved in the database, this is to allow users to consults the results later without having the re-compute the returns or query any data from the API.

Returns:

- *r_vol* : `dict` Returns of the volability portfolio.
- *r_cmr* : `dict` Returns of the momentum portfolio.
- *r_cmp* : `dict` Returns of the composite portfolio.
- *inv['name']* : `string` investment session name.

`check.`**`t_returns`** (*inv*, *pfl*, *prices*, *date*)

Computes the total return of a portfolio.

Parameters:

- *inv* : `list` investment session *db* row
- *pfl* : `string` name of the portfolio
- *prices* : `dict` latest investment's ticker prices
- *date* : `string` date of the purchase

Computes the sum of the shares when the invesment was made to the sum of the shares now. The absolute change and returns are calculated with the same formulas as in *`check.returns()`*

Returns a `dict` containing the total initial price, the new price, the absolute change, the returns and the date of the purchase.

CHAPTER 8

User Interface

The user interface is built on top of python's standard *cmd* [11] module with a class *betacmd*.

betacmd inherits everything from the standard cmd *class*.

The following parameters are modified:

- *intro* : 'Smart Betas Investing - Requires an internet connection'
- *prompt* : (beta)

A new parameter is introduced:

- 'out' : '–>', this string precedes outputs on the console

console provides a simple yet useful command line interface.

class console.**betacmd** (*completekey='tab', stdin=None, stdout=None*)

do_bye (*arg*)
Terminates the program.
usage: bye

do_check (*arg*)
Generates a report about investment returns.
usage: check

do_compute (*arg*)
Build portfolis with the provided tickers.
usage: compute

do_invest (*arg*)
Invest 100'000 USD into each of the computed portfolio.
usage: invest

do_load (*arg*)
Extracts tickers from a file.
usage: load <FILENAME>
Tickers in the file should be coma separated.

do_portfolios (*arg*)
Displays stored portfolios
Users can select a portfolio, view its stocks and/or invest in it.
usage: portfolios

11 Python cmd module: https://docs.python.org/3/library/cmd.html

do_report (*arg*)
Displays previously generated reports
usage: report

do_show (*arg*)
Displays information to the user.
tickers: shows the list of tickers loaded to build a portfolio.
portfolio: shows the session portfolio (after <compute>)
invest: shows the investment sessions performed by the user.
usage: show <tickers/portfolio/invest>

do_symbols (*arg*)
Shows tickers and their associated names stored in the db.
usage: symbols

emptyline ( )
Overriden to prevent default

CHAPTER 9

Input / Outputs

i_o.py handles inputs and outputs on the console.

i_o.**investment** (*qty*)
Interactive prompt to invest 100'000 USD into the computed portfolios.

```
(beta): invest
 Invest 100'000 USD in each portefolio (y/n): y
 Invest in the [x] top securities (1): 1
 Name of the investment : Glorious Investment
```

Returns nothing.

i_o.**portfolio** (*vol, cmr, cmp, prices*)
Displays three computed portfolios.

```
-------------------------------------------------------------------------
| Pos |     Volatility     |      Momentum      |     Composite      |
-------------------------------------------------------------------------
|  1  |   AAPL     21.98   |   AAPL     5.88 %  |   AAPL    204.3 $  |
|  2  |   TSLA     29.98   |   TSLA     0.3 %   |   TSLA    235.14 $ |
-------------------------------------------------------------------------
```

Returns nothing.

i_o.**portfolios** (*p_flo*)
Displays a table listing all the saved portfolios.

```
-------------------------------------------------------------------
|  id |   Date   |      Name     |              Tickers             |
-------------------------------------------------------------------
|  1  |2019-04-23|   Apple Test  |               AAPL               |
|  2  |2019-04-23|   Test Apple  |               AAPL               |
|  3  |2019-04-23|      Test     |               AAPL               |
|  4  |2019-04-23| 14 Tickers ftw|      AMD, ACB, GWW, GOOGL...     |
|  5  |2019-04-27|   Apple Only  |               AAPL               |
|  6  |2019-04-27|  Apple Tesla  |           AAPL, TSLA...          |
-------------------------------------------------------------------
```

Returns nothing.

i_o.**reports** (*rep*)
Displays a table listing all the saved reports.

```
-------------------------------------------------------------------
| Pos |    Id    |          Date          |          Name          |
-------------------------------------------------------------------
|  1  |    1     |       2019-04-26       |       6 Big Stuffs     |
|  2  |    2     |       2019-04-27       |       6 Big Stuffs     |
```

```
|  3  |    3     |        2019-04-27       |           Test          |
|  4  |    4     |        2019-04-27       |       6 Big Stuffs      |
|  5  |    5     |        2019-04-28       |         One One         |
-------------------------------------------------------------------
```

Returns nothing.

i_o.**returns** (*vol*, *cmr*, *cmp*, *name*)

Displays the returns report of an investment session.

```
-------------------------------------------------------------------------------
                                One One - Report
-------------------------------------------------------------------------------
                            Volatility Based Portfolio
-------------------------------------------------------------------------------
| Ticker |N Shares| Purchase Date |  Initial  |  Current  | Abs Change|Returns
 |
-------------------------------------------------------------------------------
|  MBRX  | 12270  |   2019-04-27  |   1.63 $  |   1.42 $  | -2576.7 $ |-12.88
%|
|  RAD   |  1992  |   2019-04-27  | 10.0401 $ |   9.08 $  | -1912.52 $|-9.56 %
 |
|  KEYW  |  1784  |   2019-04-27  |  11.21 $  |   11.3 $  |  160.56 $ | 0.8 %
 |
|  ACB   |  2212  |   2019-04-27  |  9.0396 $ |   9.04 $  |   0.88 $  | 0.0 %
 |
|  STLD  |  609   |   2019-04-27  |  32.86 $  |  31.58 $  | -779.52 $ | -3.9 %
 |
| Total  |   NA   |   2019-04-27  | 100006.0 $| 94898.7 $ | -5107.3 $ |-5.11 %
 |
-------------------------------------------------------------------------------
                             Momentum Based Portfolio
-------------------------------------------------------------------------------
| Ticker |N Shares| Purchase Date |  Initial  |  Current  | Abs Change|Returns
 |
-------------------------------------------------------------------------------
|  AMD   |  717   |   2019-04-27  |  27.89 $  |  27.88 $  |  -7.17 $  |-0.04 %
 |
|  ACB   |  2212  |   2019-04-27  |  9.0396 $ |   9.04 $  |   0.88 $  | 0.0 %
 |
|  GWW   |   69   |   2019-04-27  | 291.015 $ |  291.91 $ |  61.76 $  | 0.31 %
 |
| GOOGL  |   16   |   2019-04-27  | 1264.28 $ | 1277.42 $ |  210.24 $ | 1.04 %
 |
|  BABA  |  107   |   2019-04-27  | 186.7425 $|  187.09 $ |  37.18 $  | 0.19 %
 |
| Total  |   NA   |   2019-04-27  | 100282.7 $| 100585.6 $|  302.9 $  | 0.3 %
 |
-------------------------------------------------------------------------------
                            Composite Based Portfolio
-------------------------------------------------------------------------------
| Ticker |N Shares| Purchase Date |  Initial  |  Current  | Abs Change|Returns
 |
-------------------------------------------------------------------------------
|  ACB   |  2212  |   2019-04-27  |  9.0396 $ |   9.04 $  |   0.88 $  | 0.0 %
 |
|  AMD   |  717   |   2019-04-27  |  27.89 $  |  27.88 $  |  -7.17 $  |-0.04 %
 |
|  MBRX  | 12270  |   2019-04-27  |   1.63 $  |   1.42 $  | -2576.7 $ |-12.88
%|
|  KEYW  |  1784  |   2019-04-27  |  11.21 $  |   11.3 $  |  160.56 $ | 0.8 %
 |
|  BABA  |  107   |   2019-04-27  | 186.7425 $|  187.09 $ |  37.18 $  | 0.19 %
 |
```

```
| Total  |   NA   |   2019-04-27  | 99972.9 $ | 97587.7 $ | -2385.2 $ |-2.39 %
 |
-------------------------------------------------------------------------------
```

Returns nothing.

i_o.**sessions** (*inv*)

Displays a table listing the investments made by the user.

```
-------------------------------------------------------------------
| Pos |    Id    |           Date          |           Name          |
-------------------------------------------------------------------
|  1  |    1     |   2019-04-22 16:57:42   |       6 Big Stuffs      |
|  2  |    2     |   2019-04-27 13:59:42   |           Test          |
|  3  |    3     |   2019-04-27 14:05:07   |         One One         |
|  4  |    4     |   2019-04-28 12:31:58   |   Glorious Investment   |
|  5  |    5     |   2019-04-28 12:32:38   |   Glorious Investment   |
-------------------------------------------------------------------
```

Returns nothing.

i_o.**tickers** (*row*)

Lists all the tickers and names saved in the database.

```
(beta): symbols
-->   0: AAPL -  Apple Inc.
-->   1: TSLA -  Tesla Inc.
-->   2: GOOGL -  Alphabet Inc.
-->   3: BABA -  Alibaba Group Holding Limited
-->   4: GWW -  W.W. Grainger Inc.
-->   5: CVX -  Chevron Corporation
-->   6: HAL -  Bank Nova Scotia Halifax Pfd 3
-->   7: RAD -  Credit Suisse X-Links Monthly Pay 2xLeveraged Alerian MLP
Index Exchange Traded Notes due May 16 2036
-->   8: MBRX -  Moleculin Biotech Inc.
-->   9: KEYW -  The KEYW Holding Corporation
-->  10: BBBY -  Bed Bath & Beyond Inc.
-->  11: AMD -  Advanced Micro Devices Inc.
-->  12: STLD -  Steel Dynamics Inc.
-->  13: ACB -  Aurora Cannabis Inc.
-->  14: ATHEN -  Athene Holding Ltd. Class A
```

Returns nothing.

CHAPTER 10

Global Variables

In this application we need to make some pieces of data available for multiple modules, to do so, we use global scope variables. For the sake of clarity, they are declared on a separated module.

The following global variables are declared:

- *TICKERS* : `list` storing tickers to compute
- *NAMES* : `list` storing *TICKERS*'s name
- *P_VOL* : `dict` storing the volatility portfolio
- *P_CMR* : `dict` storing the momentum portfolio
- *P_CMP* : `dict` storing the composite portfolio
- *PRICES* : `dict` storing the prices register
- *API_KEY* : `string` storing the API key
- *API_URL* : `string` storing the API URL

`gbl.py` is used to list/access/set variables with a global scope.

CHAPTER 11

Test Modules

The functions used to compute the volatility and the momentum can be tested with test modules. Developers are encouraged to test the modified modules with these test modules.

`test_cumulative.py` tests the function that computes the volatility of a vector in *`volatility.volatility()`*. This module uses *unittest* to assert the correct output of the tested module.

This module can be exectuted with `python test_volatility.py`

class `test_volatility.`**`TestVolatility`** (*methodName='runTest'*)

`test_cumulative` tests the function that computes the momentum of a vector in *`cumulative.momentum()`*. This module uses *unittest* to assert the correct output of the tested module.

This module can be exectuted with `python test_cumulative.py`

class `test_cumulative.`**`TestVolatility`** (*methodName='runTest'*)

a

c

g

i

m

t

v