# The Quant Testing Framework I use to create profitable strategies: Part 2

This is the next part of explaining my current quant testing framework. The vital class StrategyHelper will be explained.

Hierarchy of classes/modules for reference

- Yggdrasil:
  - StockAnalyzer
  - StrategyHelper
  - market_analy
  - useful_methods

Below, I post the module containg the class StrategyHelper and explain in detail.

- Module: strategy_helper.py
  - Contains the class StrategyHelper
  - class: StrategyHelper
    - This creates an instance of a strategy itself, which is independent of the stocks it is acting on.
    - Instance variables:
      - num_points: Approximate number of data points which the strategy will be executed on.
      - custom: Boolean of whether the data used will be custom (previously exported from Yahoo Finance via yfinance module) or it will be extracted live via the alpaca api
      - freq: Frequency of trading and data which the strategy is acting on.
      - shift: Number of days to shift the data given the number of data points. Note that this is done in days regardless of the freq being used.
      - data: DataFrame of the whole trading strategy. Includes the rolling parameters such as the mean, rolling z score, the entry and exit points of when trades were executed and completed, the closing prices and return of each of the stocks in the portfolio, the closing prices and return of the base, and the date-time of all days in the data set. See the example given below.
      - results: Dictionary of all the aspects, or metrics, of the trading strategy. Includes quantities like the Sharpe Ratio, profit percentage, beta, and average alpha.
      - strategy: List with the name of the strategy being used and the parameters of that strategy.

        - base: Dataframe of the data of the base which the strategy is being compared to, (in this case I just use SP 500)
        - start: the start date of the data set. Mostly for reference.
        - end: the end date of the data set. Mostly for reference.
    - Instance methods:
        - reset_params
            - Can reset, or use different, parameters for the given instance of the StrategyHelper object.
            - Parameters are merely freq, num_points, shift, and custom.
        - get_params
            - returns a list containing the parameters of the strategy and the resulting metrics. Mostly for convenience.
            - run_method
            - Runs the strategy on a given set of stocks. Mostly for use in the Yggdrasil class constructor.
            - Parameters
                - stock_uni: List of stocks in the portfolio which the strategy will be executed on.
        - __ mean_rev __
            - Private method to execute the logic of mean reversion with the given parameters if that is the strategy which is chosen in the given instance of a test which is defined in Yggdrasil. Executed in run_method.
            - Parameters:
                - stock_uni: the list of stocks the strategy is being tested on
                - roll_value: the value of the size of the rolling data which will be used in generating the signal for mean reversion. Default value is 20.
                - cutoff_value: the value of the z-score which will be used as the signal to enter and exit. Default value is 1.5.
- free code:

    - This executes the strategy of mean reversion with parameters of the rolling window having a value of 25 and the signal being a z score of 1.3. It is executed on hourly, custom data of 'AAPL', with the default size of $\approx$ 500 data points being generated. (Note in this case the actual number is 395. The method of data extraction will be made more accurate, but the initial goal was merely to obtain a large enough sample size, regardless of specifics, in order to accurately test the given strategy on different data sets.)
    - The results of the strategy being evaluated with a set of metrics is given below the code, and the actual documentation of the backtest is given further below. Note that this example is merely for demonstration purposes of the backtesting and strategy generation framework, and the actual example strategy was not used for real alpha generation purposes.

```
import pandas as pd
import numpy as np
from pandas.core.interchange.dataframe_protocol import DataFrame

from market_analy import get_time_period,full_stocks
import useful_methods

pd.set_option('display.max_rows', None)
pd.set_option('display.max_column', None)
pd.set_option('display.width', 0)

class StrategyHelper:
    def __init__(self,freq: str,strategy:list, num_points = 500, shift =
0, custom = False):
        self.num_points = num_points
        self.custom = custom
        self.freq = freq
        self.shift = shift
        self.data = None
        self.results = {}
        self.strategy = strategy[0]
        self.params = strategy[len(strategy) - 2:len(strategy)]
        self.base = get_time_period('base',
num_data_points=self.num_points, custom_data=self.custom, freq=self.freq,
                                    shift=self.shift)

        self.start = ''
        self.end = ''


    def reset_params(self, freq: str, num_points = 500, shift = 0, custom
= False):
        self.num_points = num_points
        self.custom = custom
        self.freq = freq
        self.shift = shift

    def get_params(self):
        param_list ={'num points': self.num_points, 'custom':
self.custom,'shift': self.shift,'freq': self.freq,'param data in terms of
time attributes': self.params, 'start of the data set': str(self.start),
'end of the data set': str(self.end), 'strategy': self.strategy}

        return [param_list,self.results]

    def __mean_rev(self,stock_uni: list, roll_value = 20,cutoff_value =
1.1):


        #Data Extraction occurs first
        allocation = []
        for x in range(0,len(stock_uni)):
            allocation.append(1/len(stock_uni))
        port = pd.DataFrame({'Stock': stock_uni, 'Allocation':
allocation})

        base = self.base
        tab = []
        skip_stock = False
        for i in stock_uni:
            data_org =
get_time_period(i,num_data_points=self.num_points,custom_data=self.custom,fre
            index_common = [x for x in data_org.index if x in base.index]
            data_org = data_org.loc[index_common]
            base = base.loc[index_common]
            data_org.reset_index(inplace=True)
            base.reset_index(inplace=True)
            tab.append(data_org)
        if skip_stock:
            return None
        data_compare = pd.DataFrame()
        u = 0
        roll = roll_value
        for i in tab:
             data_compare['Date_Time'] = i['Datetime']
             data_compare['close ' + str(stock_uni[u])] = i['close']
             data_compare['Daily_return None ' + str(stock_uni[u])] =
(data_compare['close ' + str(stock_uni[u])] - data_compare['close ' +
str(stock_uni[u])].shift(1))/data_compare['close ' +
str(stock_uni[u])].shift(1)
             data_compare['Rolling_Mean ' + str(stock_uni[u])] =
data_compare['close ' +
str(stock_uni[u])].rolling(window=roll).mean().fillna(0)
             data_compare['Rolling_STD ' + str(stock_uni[u])] =
data_compare[
                ['close ' +
str(stock_uni[u])]].rolling(window=roll).std().fillna(0)
             data_compare['Z Score ' + str(stock_uni[u])] =
(data_compare['close ' + str(stock_uni[u])] - data_compare['Rolling_Mean '
+ str(stock_uni[u])])/data_compare['Rolling_STD ' + str(stock_uni[u])]
             index_drop = data_compare.index[list(range(0,roll))]
             data_compare.drop(index_drop,inplace=True)
             u+=1
        data_compare.fillna(0.00,inplace=True)
        index_common = [x for x in data_compare.index if x in base.index]
        data_compare = data_compare.loc[index_common]
        base = base.loc[index_common]

        base_ = base.copy()
        base_.reset_index(inplace=True)
        data_compare.reset_index(inplace=True,drop=True)


        #The initial investment is given
        input_money_initial = 10000

        #The strategy is set up ready to execute with the given parameters
        upper_bound = cutoff_value
        lower_bound = -cutoff_value
        base_data = base.loc[:,'close']
        cur_money = input_money_initial
        columns_ = []
        for stock in port['Stock']:
            data_compare['Sell ' + stock] = data_compare['Z Score ' +
stock] > upper_bound
            data_compare['Buy ' + stock] = data_compare['Z Score ' +
stock] < lower_bound
            columns_.append('Sell ' + stock)
            columns_.append('Buy ' + stock)
            columns_.append('close '+stock)

        buy_counter = [0]*len(allocation)
        sell_counter = [0]*len(allocation)
        result_trades = pd.DataFrame({'Action':
[np.full(len(stock_uni),'Hold')], 'Current Amount': cur_money, 'Action/DAY
Number': int(0)})
        trades_pct, trades_queue = [], []
        trades_duration, trades_duration_final = [],[]
        duration_counter = 0
        base_track,base_queue = [], []
        completed_trades = 0
        temp = data_compare[columns_]
        alloc = [input_money_initial * abs(allo) for allo in allocation]
        sell_index = [x for x in list(range(0, 3 * len(allocation))) if x
% 3 == 0]
        buy_index = [x for x in list(range(0, 3 * len(allocation))) if x %
3 == 1]
        price_inde_x = [x for x in list(range(0, 3 * len(allocation))) if
x % 3 == 2]


        #This is the execution of the strategy across the given data set
        for i in range(0,len(data_compare.index)):
            index = 0
            value_sell_list = temp.iloc[i,sell_index]
            value_buy_list = temp.iloc[i,buy_index]
            price_list = temp.iloc[i,price_inde_x]
            duration_counter +=1
            buy = []
            for value_buy,value_sell,price in zip(value_buy_list,
value_sell_list,price_list):
                alloc_ = allocation[index]
                cur_money = alloc[index]
                if alloc_ < 0:
                    if value_sell and cur_money - price > 0 and price != 0
:
```

```
                        buy.append('Sell_short')
                        cur_money = cur_money + price
                        alloc[index] = cur_money
                        sell_counter[index] += 1
                        trades_queue.append(price)
                        trades_duration.append(duration_counter)
                        base_queue.append(base_data.iloc[i])
                    elif value_buy and sell_counter[index] > 0 :
                        buy.append('Buy_short')
                        cur_money = cur_money - price
                        alloc[index] = cur_money
                        sell_counter[index] -= 1
                        value = trades_queue.pop(0)
                        trades_pct.append((price - value) / value)
                        trades_duration_final.append(duration_counter -
trades_duration.pop(0))
                        val = base_queue.pop(0)
                        base_track.append((base_data.iloc[i] - val) / val)
                        completed_trades += 1
                    else:
                         alloc[index] = cur_money
                         buy.append('Hold_short')
                if alloc_ > 0:
                    if value_sell and buy_counter[index] > 0:
                        buy.append('Sell')
                        cur_money = cur_money + price
                        alloc[index] = cur_money
                        buy_counter[index]-=1
                        value = trades_queue.pop(0)
                        trades_pct.append((price - value) / value)
                        trades_duration_final.append(duration_counter -
trades_duration.pop(0))
                        val = base_queue.pop(0)
                        base_track.append((base_data.iloc[i] - val)/val)
                        completed_trades+=1
                    elif value_buy and cur_money - price > 0 and price !=
0  :

                        buy.append('Buy')
                        cur_money = cur_money - price
                        alloc[index] = cur_money
                        buy_counter[index]+=1
                        trades_queue.append(price)
                        trades_duration.append(duration_counter)
                        base_queue.append(base_data.iloc[i])
                    else:
                         alloc[index] = cur_money
                         buy.append('Hold')
                index+=1
             result_trades.loc[len(result_trades)] = [[buy], sum(alloc), i
+ 1]

        #The results of the completed simulation are stored and computed
```

```
if needed
        if trades_duration_final:
            avg_duration = np.mean(trades_duration_final)
        else:
            avg_duration = 0
        non_cash_profit = 0
        for buy_count,price in zip(buy_counter,price_list):
            if buy_count > 0:
                non_cash_profit += price * buy_count
        profit_pct = (sum(alloc) + non_cash_profit - input_money_initial)
* 100/input_money_initial
        input_values_port = {'Upper Bound': upper_bound, 'Lower Bound':
lower_bound, 'Number of Completed Trades': completed_trades,
                           'Average Duration': avg_duration, 'Remaining
Trades': sum(buy_counter),'Final_money' : sum(alloc), 'Profit':sum(alloc)
+ non_cash_profit - input_money_initial, 'Profit %': profit_pct}
        result_trades.drop(0,inplace=True)
        result_trades.reset_index(drop=True,inplace=True)
        cur = pd.concat([data_compare,result_trades],axis=1)
        data = [input_values_port, cur]
        self.start = data_compare['Date_Time'].iloc[0]
        self.end = data_compare['Date_Time'].iloc[-1]

        #The data is analyzed and certain metrics which can be used to
track its performance are calculated and stored
        x = data
        x[1]['Current Amount'] = x[1]['Current
Amount']/input_money_initial
        x[1]['Portfolio Return'] = x[1]['Current Amount'].pct_change()
        x[1]['Base Adj_close '] = base_.get('close')
        x[1]['Base Return'] = base_.get('close').pct_change()
        buy_tracker,count, sell_tracker = 0,1,0
        total = len(x[1]['Action'])
        index = []
        x[1].drop(0,inplace=True)
        for y in x[1]['Action']:
            track = 0
            for i in range(0,len(y[0])):
                if allocation[i] >= 0:
                   if y[0][i] == 'Buy':
                      buy_tracker+=1
                   if y[0][i] == 'Sell' and buy_tracker != 0:
                      buy_tracker-=1
                   if buy_tracker == 0:
                      track+=1
                else:
                    if y[0][i] == 'Buy' and sell_tracker != 0:
                        sell_tracker -= 1
                    if y[0][i] == 'Sell':
                        sell_tracker += 1
                    if sell_tracker == 0:
                        track += 1
            if track == len(stock_uni):
               index.append(count)
            count+=1
        values = {'leng before': len(x[1])}
        x[1] = x[1].drop(index)
        values['leng after'] = len(x[1])

        if x[1].empty:
             useful_methods.format_(y=stock_uni[0])
             return None
        x[1].reset_index(inplace=True)
        if x[1]['Portfolio Return'].empty:
            useful_methods.format_('N',stock)
            return None
        pct_exposure = (1 - len(index)/total) * 100
        avg_port_return, avg_port_std = np.mean(x[1]['Portfolio Return']),
np.std(x[1]['Portfolio Return'])
        val = x[1]['Portfolio Return']
        max_list = val.cummax()
        draw_ = (val - max_list)/max_list
        draw = draw_.copy()
        draw = draw[~draw.isin([-np.inf, np.inf])]
        max_draw = draw.min()
        x[1]['Draw down'] = draw_
        if avg_port_return == 0 or avg_port_std == 0: return None
        sharpe_ratio_ = (avg_port_return)/avg_port_std
        if self.freq == '1d':
            length = 252
        elif self.freq == '1h':
            length = 252*7
        elif self.freq == '5m':
            length = 252*7*12
        sharpe_ratio_ = sharpe_ratio_ * np.sqrt(length)
        cov_ = np.cov(x[1].loc[0:len(x[1]['Portfolio Return'] )-
1,'Portfolio Return'],
                       x[1].loc[0:len(x[1]['Base Return'] )- 1,'Base
Return'])
        beta_ = cov_[0][1]/cov_[1][1]

        alpha_ = x[1]['Portfolio Return'] - beta_*x[1]['Base Return'] - (1
- beta_)* .0002
        values_ = {'beta': beta_,'sharpe ratio': sharpe_ratio_, 'avg
alpha': np.mean(alpha_)*len(x[1].index),
                    'avg std values': avg_port_std,
                           'mean': avg_port_return, 'max draw': max_draw,
'exposure pct': pct_exposure}
        values_ = values | values_
        x[1]['alpha'] = alpha_
        x[0] = x[0]|values_
        for metric in x[0] :
            x[0][metric] = float(x[0][metric])
```

```
        #The resulting data of all the completed trades and the results of
the performance of this strategy on the given data set are stored to this
instance of a test.
        self.data = data[1]
        self.results = data[0]
        return data


    def run_method(self,stock_uni: list):
        if self.strategy == 'Mean Rev':
            self.__mean_rev(stock_uni,self.params[0],self.params[1])


#free code
ygg_ex = Yggdrasil('h',['Mean Rev',25,1.3],True,stock_list = ['AAPL'])
useful_methods.format_('Data', y=ygg_ex.cur_strat.data)
#A display method for convenience
values =
list(zip(ygg_ex.cur_strat.results.keys(),ygg_ex.cur_strat.results.values()))
useful_methods.format_output(title='Results',i=values)
#Another display method for convenience
```

## Results and Data for the Example given above

Results

1. ['Upper Bound', '1.3']
2. ['Lower Bound', '-1.3']
3. ['Number of Completed Trades', '85.0']
4. ['Average Duration', '80.3529']
5. ['Remaining Trades', '13.0']
6. ['Final_money', '7,622.3442']
7. ['Profit', '291.2442']
8. ['Profit %', '2.9124']
9. ['leng before', '471.0']
10. ['leng after', '396.0']
11. ['beta', '0.8832']
12. ['sharpe ratio', '-1.0839']
13. ['avg alpha', '-0.1516']
14. ['avg std values', '0.0248']
15. ['mean', '-0.0004']
16. ['max draw', '-2.4838']
17. ['exposure pct', '84.1102']

Data: ⟶ Link