The Quant Testing Framework I use to create profitable strategies: Part 1

Here is my setup for creating profitable strategies in quant. The framework is quite large and still expanding, meaning that I will post each of its aspects individually, and explain how they all fit together.

The modules I have created are:

- yggdrasil.py: contains the class Yggdrasil, which is the root of all my code; it creates an instance of a strategy which acts on a set of stocks. Combines StockAnalyzer and StrategyHelper (see below).
- market_analy: contains a detailed retrieval method which obtains either live data or previously stored personal data given a set of parameters including trading frequency and number of data points
- useful_shi: this is a module of helper functions which helps display code and various results easier and helps in analysis
- strategy_analy: contains the class StockAnalyzer which can run various types of analysis, given a set of stocks
- strategy_helper.py: contains the class StrategyHelper which creates an instance of a working strategy depending on which strategy is picked; does not contain a stock list but merely the strategy logic
- test_live_helper.py: runs the actual live trading bot based on the tested strategy logic of the other modules.

Below, I post the following module and explain in detail.

- Module: yggdrasil.py #Contains the class Yggdrasil
  - class: Yggdrasil #This is the root of all my strategies and analysis. Creates a general instance of a test.
    - Instance variables:
      - stock_universe: StockUniverse object which contains the list of stocks that will be used in this test.
      - cur_strat: StrategyHelper object which sets and executes the basic framework of the given strategy in this test
      - strat_list: List of available executable strategies; mostly for convenience.
      - analysis: Various aspects of the stock list and the method being used; mostly for convenience.
      - title: Defines the various aspects of the strategy being used; mostly for convenience.
    - Constructor:

        - __ init __ (self, freq, strategy_,custom,shift_= 0, num_points = 500, stock_list = None)
            - Creates an instance of a test which is made of a strategy and a list of stocks. Executes this test upon instantiation.
                - Parameters: - freq: the frequency of the strategy being used - strategy_: the type of strategy being used - custom: the type of data being used, if custom or live - shift: how much the data is being shifted in its extraction. Measured in days. Default value of 0. - num_points: number of data points in the extracted data. Note that this does not always return the exact number passed, but has a minimal error range in most cases (± 10). Default value is 500. - stock_list: the list of stocks the strategy as defined by the previous parameters will act on
    - Instance methods:
        - set_title(self): #Sets the title of the strategy if needed
- free code: Saves the results of an example test run in a JSON file. The set of parameters is defined first and then the strategy is tested upon a list of various stocks. In the given example, the set of parameters is the shift of the data and the parameters of the strategy being used, which is Mean Reversion in this case defined by two metrics. These are the size of the rolling data used to compute metrics like the mean, and the cutoff z-score used to enter/exit the strategy.
    - Note that in this example, I'm currently in the process of creating viable classes for stock classification, meaning that each allocation of stocks being tested as the last parameter is merely just each individual stock only. The classification in the example given is simply that the number of completed trades is above 30 in the given data set and the profit is positive.

```
import json

import useful_shi
from market_analy import full_stocks
from strategy_analy import StockAnalyzer
from strategy_helper import StrategyHelper


class Yggdrasil:
    def __init__(self, freq, strategy_, custom, shift_=0, num_points=500,
stock_list=None):
        self.strat_list = []
        self.cur_strat = StrategyHelper(freq, strategy_, custom=custom,
shift=shift_, num_points=num_points)
        self.stock_universe = StockAnalyzer(stock_list, self.cur_strat)
        self.cur_strat.run_method(stock_list)
        self.analysis = None
        self.title = ''
```

```
    def set_title(self):
        title = ('(freq: ' + self.cur_strat.freq + ', ' + 'num of data
points: ' + str(self.cur_strat.num_points) +
                 ', custom: ' + str(self.cur_strat.custom) + ', shift: ' +
str(self.cur_strat.shift)
                 + ', start date: ' + str(self.cur_strat.start) + ', end
date: ' + str(self.cur_strat.end)
                 + ', roll:' + str(roll) + ', cutoff:' + str(cutoff) +
')')
        self.title = title


para_test = []
for shift in range(1):
    for roll in range(10, 21, 5):
        for cutoff in range(11, 13, 1):
            for stock in full_stocks:
                para_test.append([stock, shift, roll, cutoff / 10])

stock_pass_test = []
stock_pass_condition = False
for x in para_test[0:len(full_stocks) *2]:
    strategy = ['Mean Rev'] + x[2:4]
    ygg = Yggdrasil('5m', strategy, True, shift_=x[1], num_points=100,
stock_list=[x[0]])
    # useful_shi.format_(y=
[ygg.cur_strat.results,ygg.cur_strat.strategy,ygg.cur_strat.params])
    # useful_shi.break_pt()
    results = [ygg.cur_strat.results[x] for x in ['Profit', 'Number of
Completed Trades']]
    if results[0] > 0 and results[1] > 30:
        stock_pass_test.append(ygg.cur_strat.get_params())
with open('stuff_type_shi.json', 'w') as pdf:
    json.dump(stock_pass_test, pdf, indent=2)
```