Introduction

This is a worksheet for the nineth COMP3071 lab.

This week, we will have a look at the Bristol Stock Exchange simulation. Nabil will discuss this topic in detail during tomorrow's lecture. Nevertheless, the agenda for today's lab is to understand how to add new trader types to the system and simulate the effects of noise and time delays on the system.

Getting Started

Download the Bristol Stock Exchange simulator from https://github.com/davecliff/BristolStockExchange

(i.e. go to the green "Code" button and select "Download ZIP"), then unzip the file, and the file processResults.py from the module Moodle page. Copy processResults.py into the folder that you downloaded.

You might find it useful to briefly skim through `BSEguide1.2e.pdf` in your downloaded folder.

You will come across the following statement if you refer to page 3 of the `BSEguide1.2e.pdf`.

> BSE.py has been written to be easy to understand; it is certainly not going to win any prizes for efficiency; probably not for elegance either. It's roughly 1000 lines of code, all in one long file rather than split across multiple files.

Since there are multiple classes and methods within the BSE.py file, I initially have difficulty in understanding the connection between them. Instead, to improve readability and code understanding, I decompose[1] the classes and methods in separate files. This approach may help to better focus the functionality each of code block without being distracted by the other parts of the block. Also, this approach makes debugging easier and improves code organization.

Refactor the bse.py

Let's first investigate the `classes` and `methods` available under the `BSE.py`. If you skim the `BSE.py`, you will find several classes namely;

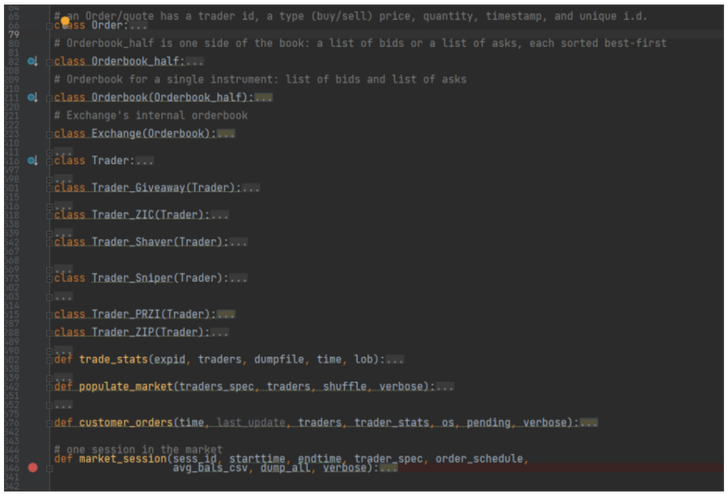
`Order`, `Orderbook_half, Orderbook`, `Exchange`, `Trader`, `Trader_Giveaway`, `Trader_ZIC`, `Trader_Shaver`, `Trader_Sniper`, `Trader_PRZI`, `Trader_ZIP`,

In addition, there are several Python methods, including ;

`trade_stats`, `populate_market`, `customer_orders`, and `market_session`.

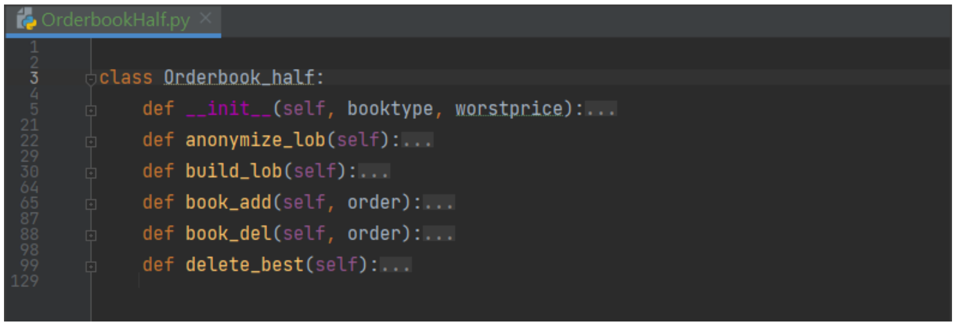
The collapsed code below provides a bird's-eye view of the original code.

[1] See the following link for more information about the advantage of decompose classes and methods into separate files https://teclado.com/30-days-of-python/python-30-day-21-multiple-files/



```
# an Order/quote has a trader id, a type (buy/sell) price, quantity, timestamp, and unique i.d.
class Order:...

# Orderbook_half is one side of the book: a list of bids or a list of asks, each sorted best-first

class Orderbook_half:...

# Orderbook for a single instrument: list of bids and list of asks

class Orderbook(Orderbook_half):...

# Exchange's internal orderbook

class Exchange(Orderbook):...

...
class Trader:...

...
class Trader_Giveaway(Trader):...

...
class Trader_ZIC(Trader):...

...
class Trader_Shaver(Trader):...


...
class Trader_Sniper(Trader):...

...

class Trader_PRZI(Trader):...
class Trader_ZIP(Trader):...

...
def trade_stats(expid, traders, dumpfile, time, lob):...

...
def populate_market(traders_spec, traders, shuffle, verbose):...

...

def customer_orders(time, last_update, traders, trader_stats, os, pending, verbose):...

# one session in the market
def market_session(sess_id, starttime, endtime, trader_spec, order_schedule,
                   avg_bals_csv, dump_all, verbose):...
```

Let's refactor the code by, for example, copying all the syntax under the class `Orderbook` and pasting it into a new Python file named `OrderbookHalf.py`, as shown below:



OrderbookHalf.py

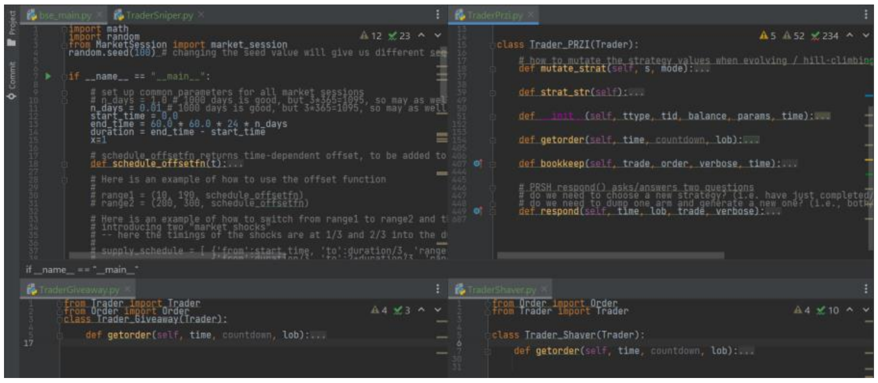
```python

class Orderbook_half:
    def __init__(self, booktype, worstprice):...
    def anonymize_lob(self):...
    def build_lob(self):...
    def book_add(self, order):...
    def book_del(self, order):...
    def delete_best(self):...
```

Similarly, repeat the same procedure for each class and function. For convenience, use the following `.py` names for each `class` and `method`, as shown in the table below:

No	type	Class or method name	.py name
1.	class	Exhange	Exchange.py
2.	class	Order	Order
3.	class	Orderbook	Orderbook
4.	class	Orderbook_half	OrderbookHalf
5.	class	Trader	Trader
6.	class	Trader_ZIP	Trader_zip
7.	class	Trader_Giveaway	TraderGiveaway
8.	class	Trader_PRZI	TraderPrzi
9.	class	Trader_Shaver	TraderShaver
10.	class	Trader_Sniper	TraderSniper
11.	class	Trader_ZIC	TraderZic
12.	class	Trader_ZIP	TraderZip
13.	def	customer_orders	CustomerOrders.py
14.	def	market_session	MarketSession
15.	def	populate_market	PopulateMarket
16.	def	Trade_stats	TradeStats

Lets also refactored all the lines under the `__name__ == "__main__"` from the original `bse.py` to a new file named `bse_main.py`.

If you perform the refactoring correctly, the decomposed code will look something like the example below, where each code block resides in its own `.py` file:



bse_main.py   TraderSniper.py

```
import math
import random
from MarketSession import market_session
random.seed(100) # changing the seed value will give us different see

if __name__ == "__main__":

    # set up common parameters for all market sessions
    # n_days = 1.0 # 1000 days is good, but 3*365=1095, so may as wel
    n_days = 0.01 # 1000 days is good, but 3*365=1095, so may as well
    start_time = 0.0
    end_time = 60.0 * 60.0 * 24 * n_days
    duration = end_time - start_time
    x=1

    # schedule_offsetfn returns time-dependent offset, to be added to
    def schedule_offsetfn(t):...

    # Here is an example of how to use the offset function
    #
    # range1 = (10, 190, schedule_offsetfn)
    # range2 = (200, 300, schedule_offsetfn)

    # Here is an example of how to switch from range1 to range2 and t
    # introducing two "market shocks"
    # -- here the timings of the shocks are at 1/3 and 2/3 into the d
    #
    # supply_schedule = [ {'from':start_time, 'to':duration/3, 'range
```

if __name__ == "__main__"

TraderPrzi.py

```
class Trader_PRZI(Trader):

    # how to mutate the strategy values when evolving / hill-climbing
    def mutate_strat(self, s, mode):...

    def strat_str(self):...

    def __init__(self, ttype, tid, balance, params, time):...

    def getorder(self, time, countdown, lob):...

    def bookkeep(self, trade, order, verbose, time):...

    # PRSH respond() asks/answers two questions
    # do we need to choose a new strategy? (i.e. have just completed/
    # do we need to dump one arm and generate a new one? (i.e., both
    def respond(self, time, lob, trade, verbose):...
```

TraderGiveaway.py

```
from Trader import Trader
from Order import Order
class Trader_Giveaway(Trader):

    def getorder(self, time, countdown, lob):...
```

TraderShaver.py

```
from Order import Order
from Trader import Trader

class Trader_Shaver(Trader):

    def getorder(self, time, countdown, lob):...
```

Debugging trick

To speed up the simulation, change the variables `n_trials` and `n_trials_recorded` to both equal `1`. This will mean that the simulation will only run once.

Remark

Run `bse_main.py`, this will generate several files.

The one we are interested in is `avg_balance.csv`. Open this in Excel or similar. Each row represents one trade in the system. `Columns C` and D represent the best bid and ask at the time of that trade, then there are blocks of four columns, representing a kind of automated trader and then the total profit for that trader type, number of traders of that type, and average profit for that trader type.

Commented [b1]: This is detailed in the comments around lines 1893-1904 of the code.

Run `processResults.py`. This should generate a plot showing the profit over time for each type of trader. Make sure that you understand how the graphing process works.

Commented [b2]: From moodle

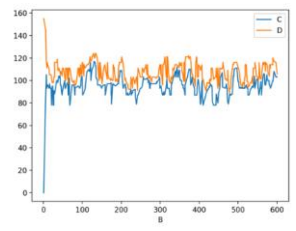
If you want the program to print out details of what it is doing during the run, there are various "`verbose`" flags under the method `market_session` that you can set to True. This makes the program print out (a lot) of information about its various data structures as it runs.

You may consider to have a fix random seed for development purposes. This ensures that the generated output is remain consistent throughout the development stage.

Tasks

1. Add a single line to `processResults.py`, which does a line plot of `column B` against `columns C` and `D`. You can model this on the other plot command in that file.

You should see a plot of the buy/sell prices converging (somewhat) to an equilibrium price around 100, like this:



So, agents are interacting, and quite rapidly the market is settling on the average price between the prices demanded by sellers and buyers (100 pence).

2. Experiment with changing the numbers of traders (i.e., `buyers_spec` and `selects_spec`) of different types in the market.

- What happens to profits and equilibrium price if you say, up the number of ZIP traders from 10 to 100?
- What happens if there are more seller agents in the system than buyer agents—say, 50 of each type of buyer?
- In particular, how does that influence the equilibrium price?
- Remember that when you re-run the code, you need to re-run `bse_main.py` first then `processResults.py` to see the changes.

3. Add a new kind of agent to the system—an "`insider trading`" agent that knows that the equilibrium price is 100 pence (a real agent would, of course, not know this information). This has the following rules:

- for a Bid offer, check the customer price, and if it is greater than 100, set the quote price at 100
- for an Ask offer, check the customer price, and if it is less than 100, set the quote price to 100

i.e. if the customer is asking for "too high" a price for buying, or "too low" a price for selling, set it to the equilibrium price.

You will find it useful to start by making a copy of the `Trader_Giveaway` class, and adding new code to the getorder method in that. To see how to do different actions depending on whether it is a Bid or Ask offer, see e.g. `Trader_ZIC`.

Add 10 of these to the trader population. You will need to add your new trader type to `populate_market` and then add them to `buyers_spec` and `sellers_spec`.

(harder) Of course, in practice it is impossible (or illegal) to have access to such "inside information". But, you could estimate it as you go. Add a response method to `Trader_Insider` that builds up an estimate of the equilibrium price by monitoring each trade and keeping a running average of the actual traded price, then use that number in place of "100" in the code.

(even harder) Change the customer orders so that there is a "shock" half way through the trading period that makes a sudden change to the equilibrium price. Does your method still work? You might want to put a limit on the number of prices included in the average (a "moving window" average) so that the estimate of the equilibrium is based only on recent data.

Extensions

1. Deal with delays in a more realistic way. That is, a trader makes an order at a particular time, but that order doesn't appear on the order-book until after a delay.
2. Extend the simulation to trading of quantities of shares more than one. This would involve you getting to know the code fairly well.
3. Implement a GUI that allows you to run simulations with different numbers of traders, different amounts of noise, different delays etc. and visualises the results.
4. Write some code that runs the simulation a number of times, each with a different level of noise. Modify the plotting code so that rather than plotting all four agent types on one graph, it generates four graphs, each of which shows one agent-type for (say) four different levels of noise (including none).
5. Add a simple model of time delays into the system. The simplest (if rather unrealistic) is to add code around line 1860 so that rather than each trader having an equal chance of being chosen to add its order to the order book, this probability is weighted—some traders are "close to the exchange" and have a higher probability of being selected than those that are further away. How does this change the behaviour of the system? You might need to graph "nearby" versus "further away" traders rather than trader types to get a proper understanding of this.
6. Add noise to the price quoted. Before doing this, comment out the checks in lines 1866-1869 and 429-433 of BSE.py. Then, add a line in the "getOrder" method of each trader, so that just before the line beginning

   order = Order(…

   you modify "quoteprice" by adding on a random (integer) number between, say, -5 and 5 (alternatively, you could add some code around line 1872 so that this noise is inserted after the order is processed—this is a little harder to do, but it means that this will work for all traders, including any new ones that you write). What effect does this noise have on the equilibrium price? Are all traders equally sensitive to this noise?