CSSE1001 / 7030

Semester 1, 2017
Assignment 2
20 Marks

Due: Friday 5 May 2017 at 9:30pm

Introduction

This assignment will use a simple data analysis application to assess object-oriented concepts. Your program will analyse data about the stock market. You will use real stock data and techniques employed by commercial applications. Your program will only perform some simple data analysis but the underlying techniques can be used for sophisticated analysis.

The most important concept of object-oriented programming is inheritance and polymorphism. These provide a mechanism to easily extend a program. Your program will make use of polymorphism to perform the data analysis and consequently would allow it to do more sophisticated analysis in the future.

Design

You will need several classes and two inheritance hierarchies to implement this program. You are provided with a file **`stocks.py`** that contains a set of support classes for the assignment. The provided classes are:

- **Stock** – stores details for a single stock listed on the market.
- **TradingData** – stores data for a single day of trading in one stock.
- **StockCollection** – stores the data for all stocks in the application and manages access to Stock objects.
- **Loader** – an abstract class that defines the process of loading stock market data.
- **Analyser** – an abstract class that defines the interface for analysing stock data.
- **AverageVolume** – analyses a single stock's data to determine its average trading volume.

Loading Data

Stock market data from different sources can be formatted in different ways. Your program will need to cater for two data formats. You will need to implement two subclasses of **Loader** called **LoadCSV** and **LoadTriplet**. Each of these classes will implement the functionality of loading data from one of the file formats. (For those who are interested, this approach for dealing with different options is called the Strategy design pattern.)

LoadCSV loads stock market data from files that are in a comma-separate format. The format of the data in the file is:

```
stock_code,date,opening_price,high_price,low_price,closing_price,volume
```

For example:

```
ACB,20170327,0.058,0.058,0.058,0.058,175116
ACG,20170327,0.06,0.06,0.059,0.059,88351
```

You will need to implement the **_process** method that is inherited from **Loader**. It will need to iterate through the file, extracting the data from a line. A **TradingData** object will need to be created to store the data from a line. This **TradingData** object will need to added to the appropriate stock object. You can look up the **Stock** object in the **StockCollection** using the stock_code.

LoadTriplet loads stock market data from files that are in a triplet key-coded format. The format of the data in the file is:

```
stock_code:key:data
```

Where the keys and their corresponding data are:

```
DA:date
OP:opening_price
HI:high_price
LO:low_price
CL:closing_price
VO:volume
```

For example:

```
1AD:DA:20170327
1AD:OP:0.22
1AD:HI:0.22
1AD:LO:0.22
1AD:CL:0.22
1AD:VO:17500
```

The data for a stock can be in any order (e.g. volume before opening price) but all data for a single stock on a single date is contiguous in the file. The **_process** method will need to iterate through the file, extracting the data line by line. Once the data for a stock on one date has been read it will need to be stored in a **TradingData** object that is added to the appropriate stock object.

If either of these **Loader** subclasses encounter an error while reading data from a file (e.g. incorrect data formats) they should raise a **RuntimeError**. Your main program should handle these exceptions gracefully.

Analysing Data

There are many types of analysis that can be performed on stock market data. Your program will implement three simple types of analysis. You will need to implement three subclasses of **Analyser** called **HighLow**, **MovingAverage** and **GapUp**. Each of these classes will implement one type of analysis. You are provided with the **AverageVolume** class as an example of implementing an **Analyser** subclass and getting it to work with objects of the **Stock** class. **AverageVolume** calculates the average trading volume of a stock. (This approach of processing data is called the Visitor design pattern.)

The **process** method in the **Analyser** class takes a **TradingData** object as a parameter and performs part of the analysis on that data. The **analyse** method in the **Stock** class takes an **Analyser** object as a parameter and iterates over all of the stock's trading data calling the **Analyser** object's **process** method on each day's data. This allows the **Analyser** object to progressively collect all of the trading data for the stock and perform its analysis. The **result** method in the **Analyser** class returns the data resulting from the analysis. The **reset** method re-initialises the Analyser object so that it can perform a new analysis.

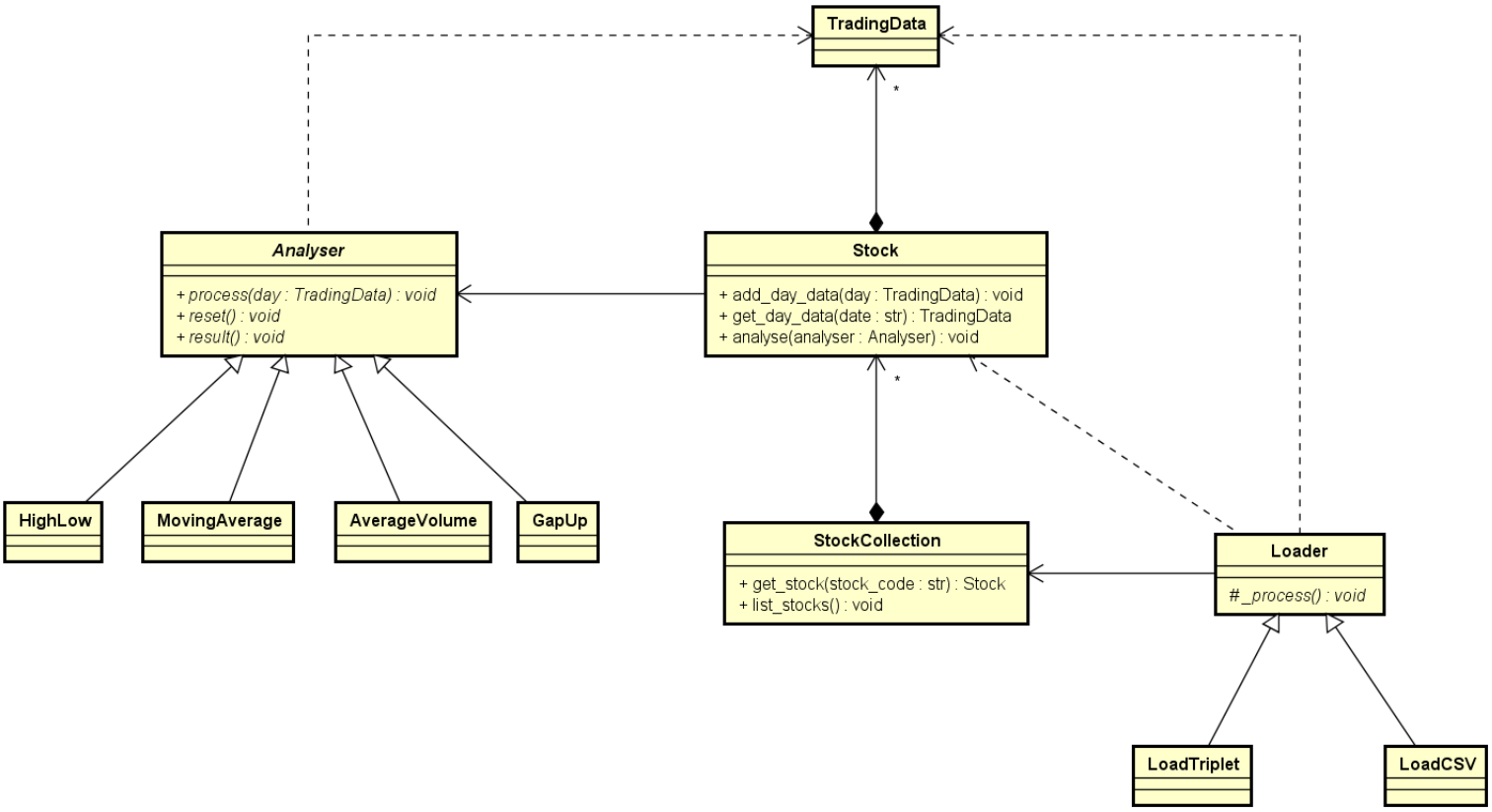
HighLow determines the highest and lowest prices paid for a stock across all of the data stored for the stock. These values are returned as a tuple from **result**. The tuple that is returned should store the high value and then the low value (e.g. (1.23, 0.99)).

MovingAverage calculates the average closing price of a stock over a specified period of time. The **__init__** method will take a parameter called **num_days** that is the number of days over which to calculate the average. This will be a simple moving average value, which is just the average closing price over the last **num_days** of trading data.

GapUp finds the most recent day in the trading data where the stock's opening price was significantly higher than its previous day's closing price. The **__init__** method will need to be passed a **delta** value to be used to determine if the price difference is significant or not. The **TradingData** object corresponding to the date on which this gap up was found is returned from **result**, or **None** if no gap up occurred in the data.

If one of the **Analyser** subclasses encounter an error while processing **TradingData** (e.g. invalid data) they should raise a **ValueError**. Your main program should handle these exceptions gracefully.

Class Diagram



Data Formats

You are provided with text files containing some sample Australian stock market data. Files with the extension ".csv" contain comma-separated data as described above. Files ending with the extension ".trp" contain triplet key-coded data as described above. Each file contains one week's worth of data. The csv files correspond to the five weeks with Friday's occurring in March. The trp files correspond to the four weeks with Friday's occurring in February. There is also a march5.trp file that is a conversion of the march5.csv file into the trp format. You may find this useful for testing purposes.

Your final program should be able to load data from all nine files feb1-4.trp and march1-5.csv and then perform its analyses across all of this data.

Example Interaction

The following is the sample output from the program you need to implement. It shows the use of the four types of analysers to produce a simple output. The code that produced this output is provided in an example function in the supplied **stock_analysis.py** file. This code will not work until you have implemented the **Loader** and **Analyser** subclasses required for this assignment.

```
Average Volume of ADV is 3629160
Highest & Lowest trading price of ADV is (0.031, 0.018)
Moving average of ADV over last 10 days is 0.02
Last gap up date of YOW is 20170330
```

You will need to do much more extensive testing of your program to ensure it works correctly.

The output produced by your program is not being assessed. You could have a set of static test statements in your code, extending what was provided in the **`stock_analysis.py`** file. Or, you would write an interactive test program that asks the user for a stock code and outputs the result of the analyses. The functionality of your program will be determined by the tests driver that will create objects of your classes and send messages to these objects. The provided **`tests.py`** file gives an indication of how this testing will be done. (The final tests will be more thorough than what is provided in **`test.py`**.) If tests.py can successfully execute with your stock_analysis.py code then the full test suite should also execute successfully. (Successful execution is that the test suite can exercise all of your code, not that the tests pass or not.)

Submission

You must submit your completed assignment electronically through Blackboard. You must submit your assignment as a single Python file called **`stock_analysis.py`** (use this name – all lower case). You do **not** need to submit the **`stocks.py`** utility file or the stock market data files. You may submit your assignment multiple times before the deadline – only the last submission will be marked.

Your programs will be tested by an automatic test runner. Consequently, you must implement your classes and methods exactly as specified. You may not change class or method names. You may not change parameters or what is returned from methods. Your code must conform to the interfaces defined in the **Analyser** and **Loader** classes.

Your program should execute automatically when it is loaded by making use of the

```
if __name__ == "__main__" :
```

construct in your **stock_analysis.py** file.

Late submission of the assignment will **not** be accepted. In the event of exceptional personal or medical circumstances that prevent you from handing in the assignment on time, you may submit a request for an extension.

See the course profile for details of how to apply for an extension:

http://www.courses.uq.edu.au/student_section_loader.php?section=5&profileId=85405

Requests for extensions **must** be made no later than 48 hours prior to the submission deadline. The application and supporting documentation (e.g. medical certificate) **must** be submitted to the ITEE Coursework Studies office (78-425) or by email to enquiries@itee.uq.edu.au. If submitted electronically, you **must** retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so.

Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program,
6. apply techniques for testing and debugging

Criteria	Mark
Programming Constructs • Program is well structured and readable, with meaningful identifier names • Algorithmic logic is appropriate, using appropriate constructs	 2 2
Sub-Total	4
Object-Oriented Concepts • Demonstrated correct understanding of objects as instances of classes • Demonstrated correct understanding of classes as units of encapsulation • Demonstrated correct understanding of inheritance • Demonstrated correct understanding of overriding methods • Demonstrated correct understanding of polymorphism	 2 2 1 1 1
Sub-Total	7
Functionality • LoadCSV • LoadTriplet • HighLow • MovingAverage • GapUp	 1 2 1 1 2
Sub-Total	7
Documentation • Entire program is documented clearly and concisely, without excessive or extraneous comments • Program is documented clearly, with all classes and methods having meaningful docstring comments • Some parts of the program have adequate comments	 2 1.5 0.5
Sub-Total	2
Total	**/ 20**

In addition to providing a working solution to the assignment problem, the assessment will involve discussing your code submission with a tutor. This discussion will take place in week 10, in the practical session you have signed up to. You **must** attend that session in order to obtain marks for the assignment.

In preparation for your discussion with a tutor you may wish to consider:

- any parts of the assignment that you found particularly difficult, and how you overcame them to arrive at a solution;
- whether you considered any alternative ways of implementing a given class or method;
- where you have known errors in your code, their cause and possible solutions (if known).

It is important that you can explain both how objects of your classes work together as well as the internal implementation logic of your methods.

Marks will be awarded based on a combination of the correctness of your code and on your understanding of the code that you have written. **A technically correct solution will not achieve a pass mark unless you can demonstrate that you understand its operation.**

A partial solution will be marked. If your partial solution causes problems in the Python interpreter please comment out the code causing the issue and we will mark that. Python 3.6 will be used to test your program. If your program works correctly with an earlier version of Python but does not work correctly with Python 3.6, you will lose **at least** all of the marks for the functionality criteria.

Please read the section in the course profile about plagiarism. Submitted assignments will be electronically checked for potential plagiarism.