CS917 – 2020/21 Coursework 1 - Programming

Deadline: Monday week 6 (8 November 2021) at 12.00 noon. Please read the entire document before starting on Part A.

Background

Through lectures and exercise sheets you have gained some useful experience of the multi-paradigm programming language Python. In this coursework we would like you to use this knowledge to solve a number of real-world problems based on the analysis of cryptocurrency data. The dataset that you will be using is taken from the Bitcoin (BTC) cryptocurrency market, provided by CryptoCompare, a London based global cryptocurrency market data provider.

Data

The data that you need for this coursework is in a single CSV file called **cryptocompare_btc.csv**, which can be found on the module website. Within the CSV file, each row is formatted as follows:

1. time: epoch timestamp in second (in UTC time zone) – indicates the day
2. high: highest BTC price of the day
3. low: lowest BTC price of the day
4. open: first BTC price of the day
5. close: last BTC price of the day
6. volumefrom: total volume (i.e., the total amount of currency exchanged) of the day in BTC
7. volumeto: total (i.e., the total amount of currency exchanged) volume of the day in USD

The data has been collected in daily between 28 April 2015 and 18 October 2020.

In this coursework, you will have to complete 4 exercises (Parts A, B, C, and D). Each part will be provided with a Python skeleton file (named partx.py – where x = {A, B, C,D}).

IMPORTANT: **Please use those skeleton files for submission. That is, write your code into those skeleton files, and follow the instructions regarding the recommended types of the input arguments, as well as the return values**.

Timestamp vs. date time:
During the coursework, you will need to be able to convert the time in epoch timestamp value to human readable date time and vice versa. You can use the following code to do these conversions:

```
#Import time module to handle time conversions
import time
import calendar

#Set a timestamp value. Use any valid value here (smallest is 0)
timestamp = 1545730073
```

```
#Convert timestamp to a time.struct_time object with UTC time
dt = time.gmtime(timestamp)

#You can print out to see how it looks like. Basically it prints out
a tuple of 9 elements,
#e.g., time.struct_time(tm_year=2018, tm_mon=12, tm_mday=25, tm_hour=9,
tm_min=27, tm_sec=53, tm_wday=1, tm_yday=359, tm_isdst=0)
print(dt)

#Print in human readable string
date_string = time.strftime("%d/%m/%Y", dt)
print(date_string)

#For more about formatting, check https://www.programiz.com/python-
programming/time

#Now we convert from time to epoch timestamp (again, choose a valid
date)
date = “03/11/2020”

#We convert t to epoch timestamp in UTC time zone
timestamp = calendar.timegm(time.strptime(date, "%d/%m/%Y"))
print("UTC time:", utc_time)
```

IMPORTANT: Note that the data contains daylight saving time changes, so when you search for dates between start_date and end_date, please do not search for the exact start and end date values in epoch timestamp, as there might be a slight difference due to the convergence differences caused by the different ways daylight saving times are handled. So I suggest you to do the following simple trick: Instead of searching for the exact start_date, search for those dates in the data where **date >= start_date**, and similarly, **date <= end_date**.

Here’s an example for a how to handle it:

```
for d in range(len(data)):
        row = data[d]
        if int(row['time']) >= start_epoch and int(row['time']) <=
end_epoch:
                {...}
```

Part A (25 Marks)

In this first part you are required to define 5 functions.

A1. The first of these functions is

```
highest_price(data, start_date, end_date) -> float
```

which given the data, a start date, and an end date (both are string with “dd/mm/yyyy” format) will return a positive or negative floating point number that is the highest price of the BTC currency in bitcoin within the given period.

A2. The second function

```
lowest_price(data, start_date, end_date) -> float
```

which given the data, a start date, and an end date (both are string with “dd/mm/yyyy” format) will return a positive or negative floating point number (accurate to 2 decimal places) that is the lowest price of the BTC currency in bitcoin within the given period.

A3. The third function

```
max_volume(data, start_date, end_date) -> float
```

which given the data, a start date, and an end date (both are string with “dd/mm/yyyy” format) will return a floating point number that is the **maximal daily amount of exchanged BTC currency of a single day** within the given period.

Update (31/10/2020): I have added the missing word here (max daily amount) to clarify what I would like you to return!

A4. The fourth function

```
best_avg_price(data, start_date, end_date) -> float
```

which given the data, a start date, and an end date (both are string with “dd/mm/yyyy” format) will return a highest daily average price of a single BTC coin in USD within the given period. To calculate the average price of a single BTC coin of a day, we take the ratio between the total volume in USD and the total volume in BTC (the former divided by the latter).

A5. Finally, the fifth function

```
moving_average(data, start_date, end_date) -> float
```

should return the average BTC currency price over the given period of time (accurate to 2 decimal places). The average price of a single day is calculated by the function in A4.

****

The code skeleton below can be found on the module website and you should add your solutions to a copy of this file. If you use this code skeleton then marking should be very easy using our ready-made test harness.

Skeleton files for all parts are provided. Below is an example of the skeleton code provided for part A:

```
"""
    Part A
    Please provide definitions for the following functions
"""

# highest_price(data, start_date, end_date) -> float
# lowest_price(data, start_date, end_date) -> float
# max_volume(data, start_date, end_date) -> float
# best_avg_value(data, start_date, end_date) -> float
# moving_average(data, start_date, end_date) -> float

# Replace the body of this main function for your testing purposes
if __name__ == "__main__":
    # Start the program

    # Example variable initialization
    # data is always the cryptocompare_btc.csv read in using a
DictReader

    data = []
    with open("cryptocompare_btc.csv", "r") as f:
        reader = csv.DictReader(f)
        data = [r for r in reader]

    # access individual rows from data using list indices
    first_row = data[0]
    # to access row values, use relevant column heading in csv
    print(f"timestamp = {first_row['time']}")
    print(f"daily high = {first_row['high']}")
    print(f"volume in BTC = {first_row['volumefrom']}")

    pass
```

Details of the marking scheme that we will use can be found below.

Part B (25 Marks)

In this exercise you are expected to modify the functions from Part A to handle exceptions. In particular, if those functions receive incorrect input arguments (e.g., wrong format, or invalid values), then the functions will not run properly, and the program will terminate with an error (i.e., crash). In order to guarantee smooth functioning (no crash), we need to handle those errors. A possible way to do so is to catch the exception messages the system sends whenever an error occurs.

Your task is to design exception handlers for each of the functions from Part A. First, you need to check what types of errors may occur. Note that all the 5 functions have the same input arguments. Therefore, we have the following cases:

1. Data: the data file might not exist
2. Data: or it does not contain the necessary column types (i.e., the column name is not in the data file)
3. Date (both start and end dates): this can be an invalid number
4. Date: the given dates are out of range (note that the provided **cryptocompare_btc.csv** file only contains data within a certain range of dates)
5. Date: end date is smaller than start date

For each of these errors, please implement an exception handler. In particular, you are required to print the following messages for each of the case:

1. Empty dataset: "Error: dataset not found"
2. Missing column in dataset: "Error: requested column is missing from dataset"
3. Invalid date: "Error: invalid date value"
4. Out-of-range date values: "Error: date value is out of range"
5. End date is smaller than start date: "Error: end date must be larger than start date"

Note that some of these errors are already handled by Python by default (it throws an error), while others are not (the function returns an empty value, which technically is still a correct result, but does not make sense).

In order to know whether an error has already been handled in Python by default, you can manually edit the input arguments of each function (e.g., remove the csv file from current directory, or deliberately give incorrect dates), aiming to provide erroneous inputs, and use the following exception throw and catch framework to see whether that error has already been caught by default (and if yes, what the name of the exception type is).

```
try:
    #Statements

except Exception as ex:

    #Exception type
    ex_type = type(ex).__name__
    print("exception type: " + ex_type)
    #Exception message
    ex_message = ex.args
    print("exception message: " + ex_message)
```

For the cases when exceptions are not raised, you will have to write your own Exception handler. In particular, you will need to define your own Exception class:

```
class MyException(Exception):

    #Exception message set by value
    def __init__(self, value):
        self.parameter = value

    #Exception message to be printed
    def __str__(self):
        return self.parameter
```

Then you can use your Exception file as follows:

```
from myexcept import MyException

try:
    #check error condition
    if <error condition is true>
        raise MyException("My custom error message.")
except MyException as e:
    print("Error: " + str(e))
```

Part C (25 Marks)

In cryptocurrency markets, we should be capable of identifying when it is beneficial to sell or buy the cryptocurrency. One of the most common technical analysis approach is the crossover method. In particular, this method maintains 2 moving averages, one with a shorter time window (e.g., 3 days), one with a longer time window (e.g., 10 days). Let's call the former `moving_avg_short,` and the latter `moving_avg_long.` Now, for any date t, if the value of `moving_avg_short` at that date (i.e., date t) is "crossing" upward the value of `moving_avg_long,` then we should buy. Here, crossing upward means that for the value of `moving_avg_short` at (t-1) is still **smaller or equal** to the value of `moving_avg_long` at (t-1), but at date t, the value of `moving_avg_short` **already larger** than that of `moving_avg_long.` For example, the short moving average at day 100 is 300 and the long moving average is 302. But for day 101, the short moving average jumps to 305, but the long moving average is still 302. In this case, the crossover method will indicate a buy signal.

In the contrary, if the value of `moving_avg_short` at that date (i.e., date t) is "crossing" downward the value of `moving_avg_long,` then we should sell. Here, crossing downward means that for the value of `moving_avg_short` at (t-1) is still **larger or equal** to the value of `moving_avg_long` at (t-1), but at date t, the value of `moving_avg_short` **already smaller** than that of `moving_avg_long.` For example, the short moving average at day 100 is 252 and the long moving average is 250. But for day 101, the short moving average jumps down to 249, but the long moving average is still 250. Here the crossover method will indicate a sell signal.

In this exercise, you will need to implement this crossover method with short window = 3 and long window = 10. The method needs to be able to identify the list of buy and sell dates within a given period of time. The signature of the function is as follows:

```
crossover_method(data, start_date, end_date) -> [buy_list, sell_list]
```

Where the list `buy_list` contains the days where the functions indicates a buy signal. Similarly, list `sell_list` contains the days where the functions indicates a sell signal.

During the implementation, you need to take these into consideration:

- If the date is too close to the first entry of **cryptocompare_btc.csv**, you might not have enough data to calculate the moving averages. In this case, just use the data from the first entry of the csv file.
- Both `buy_list` and `sell_list` should be list type.

You are also expected to implement the following auxiliary functions:

```
1. moving_avg_short(data, start_date, end_date) -> dict
2. moving_avg_long(data, start_date, end_date) -> dict
3. find_buy_list(short_avg_dict, long_avg_dict) -> dict
4. find_sell_list(short_avg_dict, long_avg_dict) -> dict
```

`moving_avg_short(data, start_date, end_date)` takes the dataset with the start and end dates, and it calculate the moving average with time window 3 for all the dates within the given range. The results are stored in a dictionary with key = date, and value = calculated short average.

Update (28/10/20): In more detail, for each date *t* between *start_date* and *end_date*, you will need to use values of dates *(t-2), (t-1)* and *t* to calculate the 3-day moving average of date *t*. Advice: check if *t* is too close to the beginning of the csv file (e.g., t = first 2 lines). In this case, you won't have enough dates prior to t to calculate the 3-day average, so you can use less than 3 dates, and just start from the beginning of the csv file.

`moving_avg_long(data, start_date, end_date)` takes the dataset with the start and end dates, and it calculate the moving average with time window 10 for all the dates within the given range. The results are stored in a dictionary with key = date, and value = calculated long average.

Update (28/10/20): In more detail, for each date *t* between *start_date* and *end_date*, you will need to use values of dates *(t-9), (t-8),...,(t-1)* and *t* to calculate the 10-day moving average of date *t*. Advice: check if *t* is too close to the beginning of the csv file (e.g., t = first 9 lines). In this case, you won't have enough dates prior to t to calculate the 3-day average, so you can use less than 10 dates, and just start from the beginning of the csv file.

`find_buy_list(short_avg_dict, long_avg_dict)` takes the dictionary of short and a dictionary of long averages. It then returns a dictionary with key = date, and value = 1 is there is a buy signal in the corresponding date, and 0 otherwise.

`find_sell_list(short_avg_dict, long_avg_dict)` takes the dictionary of short and a dictionary of long averages. It then returns a dictionary with key = date, and value = 1 is there is a sell signal in the corresponding date, and 0 otherwise.

Part D (25 Marks)

In this exercise you are required to create a new `Investment` class. This class will encapsulate the code you created in Parts A and B (with the exception handlers) so that `Investment` objects can be created. Each `Investment` will have a `data` variable which will store entries from **cryptocompare_btc.csv**. The specification for the new class is as follows:

```
Investment:
#Instance variables
   ● start date
   ● end date
   ● data
#Functions
   • highest_price(data, start_date, end_date) -> float
   • lowest_price(data, start_date, end_date) -> float
   • max_volume(data, start_date, end_date) -> float
   • best_avg_value(data, start_date, end_date) -> float
   • moving_average(data, start_date, end_date) -> float
```

Update (28/10/20): Note that these functions need to be able to accept empty arguments as well. In this case, they will use the instance variables.

If we want to meaningfully participate in the cryptocurrency market, it is not enough to just analyse data, we also need to be able to predict future BTC prices. To do this we will implement a linear regression model to identify the trend and predict the next day's stock prices.

There are various maths libraries in Python that might help you do this, but we would like you to implement this feature by hand. Define a function called `predict_next_average`, which takes an instance of the `Investment` class, calculates the average `price` for the consecutive days of data in **cryptocompare_btc.csv**, given by the start and end dates of the investment instance, and uses this to predict what you think the average `price` will be for the next day of trading.

```
predict_next_average(investment) -> float
```

For those not familiar with linear regression models, the algorithm to generate a simple linear model is available below. In this algorithm, ***m*** is the gradient of a straight line and ***b*** is the *y* intercept. Applying this model to our dataset, for our needs, you will need to assign ***x*** to the day. Here, you can just directly use the epoch timestamp as the value of ***x***. Finally, ***y*** is the average `price` of the corresponding day.

$$m = \frac{\sum_{i=1}^{n} \quad (x_i - \underline{X})(y_i - \underline{Y})}{\sum_{i=1}^{n} \quad (x_i - \underline{X})^2}$$

$$b = \underline{Y} - m\underline{X}$$

The resulting model should result in ***y=mx+b*** which will generate a straight line from which we can extrapolate the `price` of the next in our sequence (note that cryptocurrency markets open at the weekend as well, so we do indeed have real subsequent days).

For many analysis techniques, it is not enough to predict the next bitcoin prices or averages. Instead we will want to classify investments based on how the BTC prices have evolved over the given period (i.e., between the start and end dates of the investment instance). Next implement a function that will return a string classifier that will identify if the BTC price is 'increasing', 'decreasing', 'volatile' or 'other':

```
    classify_trend(investment) -> str
```

To do this, perform a linear regression on the `daily high` and `daily low` for a given investment period and determine whether the highs and lows are increasing or decreasing. You will most likely need to use the linear regression algorithm you implemented for predicting the price, so it may be useful to make the regression its own function.

The classification system works as follows: If the **daily highs are increasing** and the **daily lows are decreasing**, this means that the stock prices have been fluctuating over the time between the start and end date of the investment instance, so **assign 'volatile'** to your result string. If the **daily highs and daily lows are both increasing**, this likely means that the overall prices are increasing so **assign 'increasing'**. Likewise if the **daily highs and lows are both decreasing** then **assign 'decreasing'**. We currently only care about these 3 classifications so if a company shares **do not follow any of the above trends assign 'other'** to your result string.

The marking scheme for Part D can also be found below.

Coursework Submission and Marking

Deadline: Monday week 6 (8 November 2021) at 12.00 noon. Coursework in the department is nearly always submitted through Tabula. The advantage of using this system is that you can be assured that the work has been submitted, a secure record of the work is kept, feedback is easy to distribute and, where appropriate, code can be automatically run and checked for correctness. Instructions on how to register on Tabular and the steps to follow to submit your work will be posted on the module webpage shortly. Please note the university requires that late penalties apply, so if you do submit your work late (by even 5 minutes!) you will be penalised.

You are required to submit four separate files for this coursework: **parta.py**, **partb.py**, **partc.py**, **partd.py**. Each of these files will be run on the data from **cryptocompare_btc.csv** so that it can be checked for correctness. We will also judge each solution on coding style and how well you have made use of the various features of Python that we have covered in the lectures.

The marking scheme for the coursework is as follows:

Part A

The **parta.py** file will have the following functions tested:

5. `highest_price(data, start_date, end_date) -> float`
6. `lowest_price(data, start_date, end_date) -> float`
7. `max_volume(data, start_date, end_date) -> float`
8. `best_avg_value(data, start_date, end_date) -> float`
9. `moving_average(data, start_date, end_date) -> float`

Each function will be tested on four (`start_date`, `end_date`) combinations and checked against our own solutions. Thus, twenty tests will be run in total for Part A and there are 20 possible marks available for these tests. In addition, marks will be available for coding style (2 marks) and how well you have made use of the various language features of Python (3 marks).

In your feedback for this exercise you will be told how many of the twenty tests your code passes, and also how many marks were awarded for coding style and use of language features.

Part B

The **partb.py** file will be tested with 5 different test inputs (to test each exception case) for each function. This will be 25 tests in total for Part B and each is worth 1 mark.

Part C

For Part C, the code in your **partc.py** file will be run with 5 sets of test input, each of which will be testing the 4 auxiliary functions, and the main `crossover_method` function. Each set will consist of 4 tests. Therefore, there will be 20 tests, each worth of 1 mark.

In addition, marks will be available for coding style (2 marks) and how well you have made use of the various language features of Python (3 marks).

Part D

The **partd.py** file will have the following functions tested:

1. `highest_price(data, start_date, end_date) -> float`
2. `lowest_price(data, start_date, end_date) -> float`
3. `max_volume(data, start_date, end_date) -> float`
4. `best_avg_value(data, start_date, end_date) -> float`
5. `moving_average(data, start_date, end_date) -> float`

The implementation of the Investment class is worth 5 marks. Therefore each of these required class functions will be allocated one mark each.

We will then test the two functions:

1. `predict_next_average(investment) -> float`
2. `classify_trend(investment) -> str`

The `predict_next_average` function will be tested on 5 random investment instances generated from the **cryptocompare_btc.csv**. Each correct answer will result in 2 marks each; thus 10 marks in total.

The `classify_trend` function will be also tested on 5 random investment instances generated from the **cryptocompare_btc.csv**, but not necessarily the same as those above, and each test will be worth 2 marks; total of 10 marks.

Please note that no marks will be assigned for solutions that use additional imported libraries not covered in the lectures to solve these questions.

Working Independently and Completion

It is important that you complete this coursework independently. Warwick has quite strict rules about this, and if a submitted solution is deemed to be a copy of another then there are harsh penalties. The Tabula system has built-in checking software that is quite sophisticated: if variable names are changed but the structure of the code is the same, then it will spot this; if you reorder the code, then it will spot this also; if you change the comments, then the checking software will know. So rather than trying to fool the system - which you will not be able to do - it is best just to complete the work by yourself. You do not need to do all the questions to get a decent mark, so just do what you can on your own … and then you will do fine.

This coursework is designed to have stretch goals: Parts A and B to be the easier ones. One the other hand, Parts C and D are more involved and we are not expecting everyone to do this. So we suggest you to start from Part A, then do B, C and D (as they are building on top of each other as well, this seems to be a natural work plan). If you are able to get Parts A-C done well, then you will likely score between 65% and 75%, which is a good mark which you should be pleased with.

Good luck.