CS1010S — Programming Methodology
National University of Singapore

Practical Examination

18 Nov 2017

Time allowed: 2 hours

Instructions (please read carefully):

1. This is **an open-book exam**. You are allowed to bring in any course or reference materials in printed form. No electronic media or storage devices are allowed.
2. This practical exam consists of **three** questions. The time allowed for solving this test is **2 hours**.
3. The maximum score of this test is **30 marks**. Note that the number of marks awarded for each question **IS NOT** correlated with the difficulty of the question.
4. You are advised to attempt all questions. Even if you cannot solve a question correctly, you are likely to get some partial credit for a credible attempt.
5. While you are also provided with the template `practical-template.py` to work with, your answers should be submitted on Coursemology. Note that you can **only run the test cases on Coursemology for a limited number of tries** because they are only for checking that your code is submitted correctly. You are expected to test your own code for correctness using IDLE and not depend only on the provided test cases. Do ensure that you submit your answers correctly by running the test cases at least once.
6. In case there are problems with Coursemology.org and we are not able to upload the answers to Coursemology.org, you will be required to name your file `<mat no>.py` where `<mat no>` is your matriculation number and leave the file on the Desktop. If your file is not named correctly, we will choose any Python file found at random.
7. Please note that it shall be your responsibility to ensure that your solution is submitted correctly to Coursemology and correctly left in the desktop folder at the end of the examination. Failure to do so will render you liable to receiving a grade of **ZERO** for the Practical Exam, or the parts that are not uploaded correctly.
8. Please note that while sample executions are given, it is **not sufficient to write programs that simply satisfy the given examples**. Your programs will be tested on other inputs and they should exhibit the required behaviours as specified by the problems to get full credit. There is no need for you to submit test cases.

GOOD LUCK!

Question 1 : Stop the Train [10 marks]

Temasek Railway Metro System (TRMS) has recently upgraded the signalling system on their subway lines.

In the new system, each rail line is divided into blocks of a unit length. Each station on the line will occupy a particular block. For example, in the figure below, Station A is on block 44 and Station B is on block 89.

A train travelling at speed s covers exactly s blocks in each time unit. Suppose a time unit is the smallest divisible unit. That means if a train travelling at speed s is in block b at time $t = 0$ and maintains its speed, it will be in block $b + s$ at time $t = 1$.

A train could also reduce its speed by half. Since the speed is always an integer value, it will be rounded down to the nearest integer. For example, if a train travelling at speed $s = 5$ is in block b at time $t = 0$ and reduces its speed, it will be in block $b + 2$ at time $t = 1$ and its new speed will be $s = 2$.

Now if it continues to reduce its speed again, it will be in block $b + 3$ at time $t = 2$. Otherwise, it could continue at speed $s = 2$ and be in block $b + 4$ at time $t = 2$.

A. If a train is travelling at a certain speed and requires to stop at a particular block, it needs to start reducing its speed early in order to come to a complete stop at the block. The function `brake_at(dest, speed)` takes as input the block number of a destination station and the speed of the train, and returns the closest block number to the destination at which the train has to start reducing its speed in order to come to a complete stop at the station.

Provide an implementation for the function `brake_at`. Assume that the train is approaching from the direction of a smaller block number.

[5 marks]

Sample Execution:

```
>>> brake_at(89, 4)
86

>>> brake_at(89, 10)
81

>>> brake_at(89, 20)
71
```

Table 1: In order to stop at block 89, the train has to reduce its speed in blocks 74, 84, 88 and finally 89.

Time	Block	Speed
0	44	10
1	54	10
2	64	10
3	**74**	5
4	79	5
5	**84**	2
6	86	2
7	**88**	1
8	**89**	0

B. Now suppose the train is currently at a particular block travelling at a given speed, and has to stop at a destination block. In order to safely arrive in the shortest time, the train has to reduce its speed as late as possible without overshooting the station.

Table 1 shows an example of a train travelling from block 44 to 89 at speed 10.

The function `braking_points(curr, dest, speed)` takes as inputs the current and destination block of the train together with its current speed. It returns a sequence (either list or tuple) of block numbers where the train has to reduce its speed in order to come to a stop at the destination in the least time, i.e., the train will reduce speed at late as possible.

If it is not possible to come to a stop at the destination, an empty sequence is returned.

Provide an implementation for the function `braking_points`. You may assume that $curr < dest$.

[5 marks]

Sample Execution:

```
>>> braking_points(44, 89, 10)
[74, 84, 88, 89]

>>> braking_points(71, 89, 20)
[71, 81, 86, 88, 89]

>>> braking_points(71, 89, 22)
[]
```

Question 2 : Crypto-currency [10 marks]

Important note: You are provided with a data file `crypto.csv` for this question for testing, but your code should work correctly for *any* data file with the same format and it *will* be tested with other data files.

Your friend recently made a lot of money buying Bitcoins, only to lose it all during the recent China ICO panic. Being wary of the returns, you obtained a price history the top few crypto-currencies to perform some analysis.

The first line of the data file is a header which describes each column of data. You should only assume that **the first four columns are fixed** to 'Year', 'Month', 'Day', and 'Currency', and that the rows are unique. The remaining columns are the components, which are not fixed, i.e. different data files can have different components. Your code should take this into account.

Hint: The method `List.index(item)` returns the index of the first matching `item` in the list.

A. The function `monthly_avg` takes as inputs a filename (`str`), currency (`str`), year (`int`) and component (`str`). It returns a dictionary where the keys are months, and the values are the monthly average of the given component of the given currency in the given year, rounded to 4 decimal places. You may assume that all the values in the requested component are floats.

Note, if there are months with no data for the given inputs, then the month is not included in the returned dictionary.

Hint: You can use the function `round(n, d)` to round n to d decimal places.

[5 marks]

Sample execution:

```
>>> monthly_avg('crypto.csv', 'ETH', 2017, 'Close')
{'Nov': 296.4443, 'Oct': 306.2474, 'Sep': 293.0473,
 'Aug': 301.6094, 'Jul': 224.1239, 'Jun': 313.7343,
 'May': 125.7494, 'Apr': 50.3367, 'Mar': 34.7916,
 'Feb': 12.3711, 'Jan': 10.2013}

>>> monthly_avg('crypto.csv', 'BTC', 2013, 'High')
{'Dec': 856.4419, 'Nov': 569.307, 'Oct': 161.9442,
 'Sep': 134.164, 'Aug': 116.0023, 'Jul': 93.869,
 'Jun': 111.3007, 'May': 123.949, 'Apr': 143.4667}
```

B. We are now interested in computing the gain for each component in a month. The gain is calculated by taking the highest value of the component in the month and dividing it by the lowest value in the month. Since the result should be displayed as a percentage gain, it should be subtracted by 1 then multiplied by 100.

For each month in a given year, we want to know which currency had the highest gain for a particular component amongst all the currencies. The function `highest_gain` takes as input a filename (`str`), a year (`int`), and a component (`str`), and returns a dictionary where the keys are the months and the values are a tuple of two elements: the currency and the gain (rounded to 2 decimal places).

You may assume that the values for the components are either integers or floats. Note that it is possible for some values to be missing, in which case it is denoted by a `'-'`. Such rows should be ignored. [5 marks]

Sample Execution:

```
>>> highest_gain('crypto.csv', 2017, "Volume")
{'Nov': ('XPR', 695.17), 'Oct': ('XPR', 3485.19),
 'Sep': ('LTC', 1792.14), 'Aug': ('XPR', 5524.07),
 'Jul': ('LTC', 1354.52), 'Jun': ('XPR', 1071.13),
 'May': ('ETH', 2302.35), 'Apr': ('XPR', 4537.85),
 'Mar': ('XPR', 7003.02), 'Feb': ('ETH', 1049.67),
 'Jan': ('XPR', 1975.93)}

>>> highest_gain('crypto.csv', 2013, "Market Cap")
{'Dec': ('XPR', 272.37), 'Nov': ('LTC', 1862.94),
 'Oct': ('BTC', 88.94), 'Sep': ('XPR', 171.23),
 'Aug': ('XPR', 109.9), 'Jul': ('BTC', 59.08),
 'Jun': ('LTC', 52.22), 'May': ('LTC', 48.27),
 'Apr': ('BTC', 7.15)}
```

C. BONUS QUESTION

Suppose the values of a component of a given data reflects the prices that the crypto-currency was traded for. We can calculate the maximum profit that can be obtained if we can buy and sell the currency **only once**. In other words, we can only buy the currency on any one day, and later sell it all off on another day. The profit would be the difference between the price at which it was bought and which it was sold.

This is commonly known as the *Maximum Single-Sell Profit* problem.

Implement the function `max_single_sell` which takes in a filename (`str`), currency (`str`), and component (`str`), and returns a tuple containing the buying date, selling date and the profit obtained from buying and selling the crypto-currency on the respective dates. See sample execution on the date format.

The profit should be the maximum that can be obtained from the given data set. And obviously, the buying date has to be before the selling date.

Condition: To get the full bonus marks, the time complexity of your code should be faster than $O(n^2)$, i.e., a non brute-force method. [5 marks]

Sample Execution:

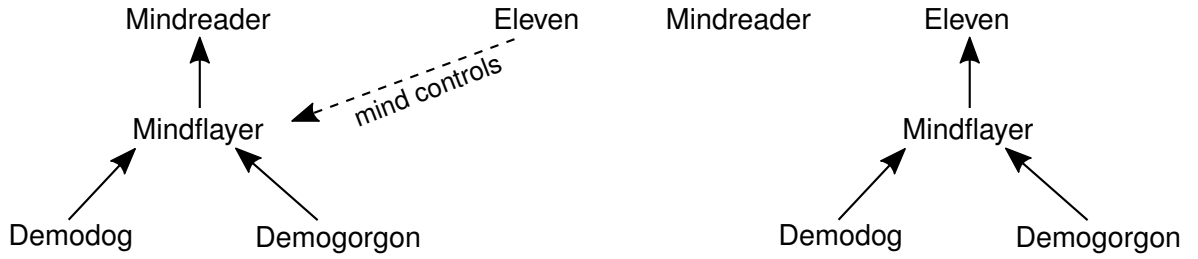
```
>>> max_single_sell('crypto.csv', 'ETH', 'Close')
('2015-Oct-20', '2017-Jun-12', 401.05517100000003)

>>> max_single_sell('crypto.csv', 'BTC', 'Close')
('2013-Jul-5', '2017-Nov-5', 7338.98)

>>> max_single_sell('crypto.csv', 'DASH', 'Close')
('2014-Feb-15', '2017-Aug-26', 399.535135)
```

Question 3 : Stranger Things 2 [10 marks]

Warning: Please read the entire question carefully and plan well before starting to write your code.

In the Upside Down world of Stranger Things, entities exist and roam the world attacking each other. Among the entities, there are some called Overminds, which can mind control other entities, making them slaves.

Overminds, themselves being Entities, can also be mind controlled and become slaves to other Overminds. This it is possible for a slave of an Overmind to also have its own slaves. When an Overmind with slaves gets mind-controlled by another Overmind, its slaves will also come under control of the other Overmind. Figure 1 illustrates an example.



(a) Mindreader controls Mindflayer, who in turn control two slaves. Eleven attempts to mind-control Mindflayer.

(b) Eleven captures control of Mindflayer and all its slaves from Mindreader.

Figure 1: Overminds can control other Overminds, and wrestle control from other Overminds.

There is no limit to the number of slaves an Overmind can control.

An `Entity` is created with two inputs, its name (which is a string) and a damage (which is an integer). `Entity` supports the following methods:

- `get_name()` returns the name of the Entity.
- `get_master()` returns the name of the ultimate controlling Overmind if the Entity is being mind-controlled, i.e. the master Overmind that is not being mind-controlled. Otherwise, if the Entity is not being mind-controlled, `None` is returned.
- `get_damage()` returns the damage value of the Entity.
- `attack(other)` takes as input an Entity and returns a string based on the following conditions:
  - `'<entity name> cannot attack itself'` if `other` is itself.
  - `'<entity name> cannot attack its master'` if `other` is the ultimate controlling Overmind over the entity.
  - `"<entity name> cannot attack its master's slave"` if the entity is mind-controlled and `other` is also a slave of its master.
  - otherwise, `'<entity name> deals <damage> damage to <other name>'` where <damage> is the damage value of the Entity.

`Overmind` is a subclass of `Entity` and it is able to mind-control other Entities as its slaves. **The damage value of an Overmind is thus the sum total of the damage values of all its slaves.** `Overmind` supports the following methods:

- `get_slaves()` returns a tuple of the names of all the slaves that the Overmind is controlling, both directly and vicariously through other Overminds.
- `attack(other)` takes as input an Entity and returns a string based on the following conditions:
  - `'<overmind name> cannot attack its slave'` if `other` is one of its slave.
  - otherwise the same behaviour as `Entity.attack` is returned.
- `mind_control(other)` takes as input an Entity and returns a string based on the following conditions:
  - `'<overmind name> cannot mind-control itself'` if `other` is itself.
  - `'<other name> is already a slave of <entity name>'` if `other` is one of its slave.
  - `'<overmind name> cannot mind-control its master'` if it is a slave if `other`.
  - `'<overmind name> and <other name> have the same master''` if both overmind and other are ultimately mind-controlled by the same master.
  - `'<overmind name> over-mind-controls <other name> from <other overmind name>'` if the above conditions are not met, and other is currently directly mind-controlled by another Overmind. Other then becomes a direct slave of the overmind
  - otherwise, `'<overmind name> mind-controls <other name>'` is returned, and other becomes a direct slave of the overmind.

Provide an implementation for the classes `Entity` and `Overmind`.

For simplicity, you do not have to worry about data abstraction and can access the properties of both classes directly. Take careful note of the characters in the returned strings, especially spaces and punctuation.

Sample Execution:

```
>>> demodog    = Entity('Demodog', 10)
>>> demogorgon = Entity('Demogorgon', 50)
>>> dartagnan  = Entity("D'artagnan", 20)
>>> mindflayer = Overmind('Mindflayer', 25)
>>> mindreader = Overmind('Mindreader', 5)
>>> eleven = Overmind('Eleven', 5)

>>> demodog.attack(demodog)
"Demodog cannot attack itself"

>>> demodog.attack(demogorgon)
"Demodog deals 10 damage to Demogorgon"
```

```
>>> demodog.get_master()
None

>>> mindreader.mind_control(mindreader)
"Mindreader cannot mind-control itself"

>>> mindreader.mind_control(demodog)
"Mindreader mind-controls Demodog"

>>> mindreader.mind_control(demogorgon)
"Mindreader mind-controls Demogorgon"

>>> mindreader.get_slaves()
('Demodog', 'Demogorgon')

>>> demodog.get_master()
"Mindreader"

>>> mindreader.attack(demodog)
"Mindreader cannot attack its slave"

>>> demodog.attack(mindreader)
"Demodog cannot attack its master"

>>> demodog.attack(demogorgon)
"Demodog cannot attack its master's slave"

>>> mindflayer.attack(eleven)
"Mindflayer deals 25 damage to Eleven"

>>> mindflayer.mind_control(mindreader)
"Mindflayer mind-controls Mindreader"

>>> mindflayer.attack(eleven)
"Mindflayer deals 90 damage to Eleven"

>>> mindflayer.mind_control(demodog)
"Demodog is already a slave of Mindflayer"

>>> mindreader.mind_control(mindflayer)
"Mindreader cannot mind-control its master"

>>> mindflayer.mind_control(dartagnan)
"Mindflayer mind-controls D'artagnan"

>>> mindreader.mind_control(dartagnan)
"Mindreader and D'artagnan have the same master"
```

```
>>> mindflayer.get_slaves()
('Demodog', 'Demogorgon', 'Mindreader', "D'artagnan")

>>> eleven.mind_control(mindreader)
"Eleven over-mind-controls Mindreader from Mindflayer"

>>> eleven.get_slaves()
('Demodog', 'Demogorgon', 'Mindreader')

>>> eleven.attack(mindflayer)
"Eleven deals 70 damage to Mindflayer"

>>> mindflayer.attack(eleven)
"Mindflayer deals 45 damage to Eleven"

>>> mindreader.attack(mindflayer)
"Mindreader deals 65 damage to Mindflayer"

>>> eleven.mind_control(dartagnan)
"Eleven over-mind-controls D'artagnan from Mindflayer"

>>> mindflayer.get_slaves()
()

>>> mindreader.mind_control(mindflayer)
"Mindreader mind-controls Mindflayer"

>>> mindreader.get_slaves()
('Demodog', 'Demogorgon', 'Mindflayer')

>>> eleven.get_slaves()
('Demodog', 'Demogorgon', 'Mindflayer', 'Mindreader', "D'artagnan")

>>> mindflayer.mind_control(mindreader)
"Mindflayer cannot mind-control its master"

>>> mindflayer.mind_control(dartagnan)
"Mindflayer and D'artagnan have the same master"
```

You are advised to solve this problem incrementally. Even if you cannot fulfil all the desired behaviours, you can still get partial credit for fulfilling the basic behaviours.

— E N D O F P A P E R —