Practical

Programming Methodology (National University of Singapore)

CS1010S — Programming Methodology
National University of Singapore

Practical Examination

11 Nov 2020

Time allowed: 1 hour 30 mins

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 **1 hour 30 mins**.
3. The maximum score of this test is **20 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 upload the template file to LumiNUS.
7. Please note that it shall be your responsibility to ensure that your solution is submitted correctly to Coursemology (or LumiNUS) 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.
9. While you may use any built-in function provided by Python, you may not import functions from any packages, unless otherwise stated and allowed by the question.

GOOD LUCK!

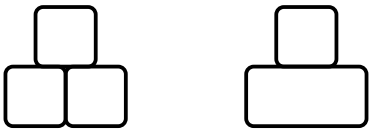
Question 1 : Child's Play [5 marks]

You have been asked to babysit your 5-year-old nephew for the weekend and he likes to play with blocks. The blocks come in two forms: (i) cubes; and (ii) 2-cube cubiods (equivalent to 2 cubes glued together).

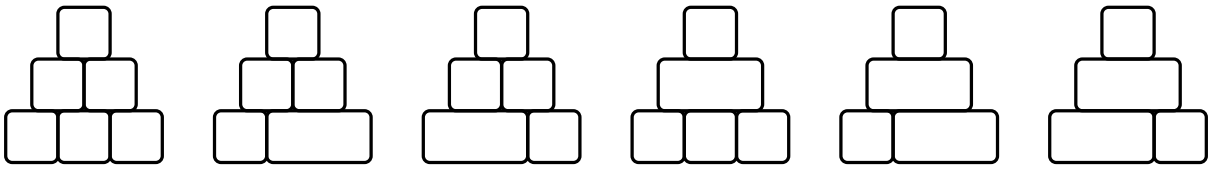
He wants to build pyramids, which has n layers. The bottom layer has length n and each subsequent layer above is 1 cube shorter. At the very top is a cube.

For $n = 1$, there is only one way to build the pyramid:

For $n = 2$, there are two possible ways, using either two cubes, or one cuboid at the base:



For $n = 3$, there are 6 possible ways:



Implement the function `pyramid(n)` that takes in n, the number of layers, and returns the number of possible pyramids. [5 marks]

Sample execution:

```
>>> pyramids(1)
1
>>> pyramids(2)
2
>>> pyramids(3)
6
>>> pyramids(4)
30
```

Question 2 : Suss Out Some Statistics [10 marks]

Important note: You are provided with a data-file `among_us.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.

While your nephew is busy creating all 240 possible combinations of a 5-layered pyramid, you decide to study a usage report of your favourite game, Among Us.

The usage report is in the form of a csv data-file, containing information on the number of players and twitch-viewers of a particular game recorded at 10-minute intervals in chronological order over a period of time. The data has the following column structure:

Date (`yyyy-mm-dd`)	**Timestamp** (`hh:mm`)	**Players**	**Twitch Viewers**

A. You wish to determine the exact timestamps when the total players and twitch-viewers were the greatest for any particular day.

Implement the function `top_k_timestamps(filename, date, k)` which takes as inputs the data filename, a date, and an integer k. Return the list of top-k timestamps for the given date, based on the total number of players and twitch-viewers, with each entry in the form:

```
(timestamp, total_players_and_viewers)
```

Entries with the same total number of players and viewers are tied for the same rank according to the Standard competition ranking scheme[1].

[5 marks]

Sample Execution:

```
>>> top_k_timestamps("among_us.csv", "2020-10-24", 1)
[('23:50', 263111)]

>>> top_k_timestamps("among_us.csv", "2020-10-24", 3)
[('23:50', 263111), ('23:40', 259451), ('23:30', 257646)]

>>> top_k_timestamps("among_us.csv", "2020-11-01", 5)
[('04:00', 440435), ('03:50', 420450), ('05:00', 418154), ('03:40',
   417638), ('05:10', 414781)]
```

[1] **Standard competition ranking ("1224")** Ties are given the same ranking, and a gap is left in the ranking numbers. This gap is one less than the number of tied items.

B. To view player trends, you decide to smoothen the data by taking hourly-averages of the 10-minute interval data. You wish to know, over a date range, which hours the maximum, minimum and median number of players occurred.

Implement the function `player_daily_statistics(filename, start_date, end_date)` which takes as inputs the data filename, a start-date, and an end-date. Return a dictionary containing the daily max and min hourly statistics of each date within the date range (inclusive of both start- and end-dates). Each date should contain information on the maximum hourly average of players, and the minimum hourly average of players in this form: [5 marks]

```
{"max": (hour, hourly_average), "min": (hour, hourly_average)}
```

Note: The player hourly average is defined as the sum of all player counts for timestamps of the same day and hour, divided by the number of those timestamps, rounded-off to **2 decimal places**. You cannot simply divide by 6, since the usage report could start or end recording partway through the hour, and there may be less than 6 entries for that particular hour.

Hint: You may use the `round(value, n)` function to round-off *value* to *n* decimal places.

Sample Execution:

```
>>> player_daily_stats("among_us.csv", "2020-10-24", "2020-10-25")}
{'2020-10-24': {'max': ('23', 148503.83), 'min': ('22', 136173.0)},
 '2020-10-25': {'max': ('05', 279875.67), 'min': ('17', 83682.83)}}

>>> player_daily_stats("among_us.csv", "2020-10-26", "2020-11-01")}
{'2020-10-26': {'max': ('04', 321269.17), 'min': ('16', 68468.83)},
 '2020-10-27': {'max': ('04', 281635.0),  'min': ('16', 66024.17)},
 '2020-10-28': {'max': ('04', 268729.33), 'min': ('16', 64382.33)},
 '2020-10-29': {'max': ('04', 263688.67), 'min': ('16', 61390.17)},
 '2020-10-30': {'max': ('04', 262891.17), 'min': ('16', 59494.0)},
 '2020-10-31': {'max': ('05', 313064.67), 'min': ('17', 65617.5)},
 '2020-11-01': {'max': ('05', 278808.17), 'min': ('17', 70535.83)}}
```

Question 3 : Among Us [5 marks]

*Among Us is an online multiplayer social deduction game. The game takes place in a space-themed setting, in which players each take on one of two roles, most being **Crewmates**, and a predetermined number being **Impostors**.*

The goal of the Crewmates is to identify the Impostors, eliminate them, and complete tasks around the map; the Impostors' goal is to covertly sabotage and kill the Crewmates before they complete all of their tasks. Players suspected to be Impostors may be eliminated via a plurality vote, which any player may initiate at any time. Crewmates win if all Impostors are eliminated or all tasks are completed; Impostors win if there is an equal number of Impostors and Crewmates, or if a critical sabotage goes unresolved.

Source: Wikipedia

In this question, we will be modelling Crewmates performing Tasks at different Places, and Impostors killing Crewmates.

You will be creating three classes: `Place`, `Crewmate`, and `Impostor` which is a subclass of `Crewmate`.

`Place` is initialized with a name, and a list of tasks (which are `str`). It need not have any methods, though you are free to define some for your own use.

Crewmates can move from place to place, performing tasks at each place. Though the names of tasks are not unique, the tasks of each place are independent. E.g. performing task "fix wires" at Cafeteria does not imply anything about the task "fix wires" in Weapons.

A task can be performed by multiple crewmates. But each Crewmate can only perform each task at each place only once.

`Crewmate` is initialized with a name and a starting `Place`. It is supported by the following methods:

- `move` takes a `Place` as input and moves the person to the given place. The method returns the string `'<Crewmate name> moves to <Place name>'`.
- `do_task` takes a task as input and returns a string based on the following conditions:
  - `'No such task <Task name> in <Place name>'` if the current `Place` of the crewmate does not have such a task.
  - `'Task <Task name> in <Place name> already done'` if the crewmate has previously performed the task.
  - Otherwise, the task at the Crewmate's current place is performed, and the string `'<Crewmate name> does <Task name> in <Place name>'` is returned.

`Impostor` is a subclass of `Crewmate`. An Impostor can kill a Crewmate if they are both in the same Place. Crewmates can only be killed once, and after which may continue moving between places and doing tasks as usual.

`Impostor` has the following methods:

- `do_task` overrides its superclass method and always return the string `'Impostors cannot do tasks'`
- `kill` takes no inputs and returns a string based on the following conditions:
  - `'Nobody to kill'` if there are no non-Impostors who are alive at the current place of this Impostor.
  - Otherwise, a Crewmate who is i) not an Impostor, ii) not yet been killed, and iii) at the same place as this Impostor, will be killed. The string `'<Impostor name> kills <Crewmate name> at <Place Name>'` is returned. If there are more than one Crewmate that fulfils the conditions, any one may be arbitrarily chosen.

Consider the following sample execution:

Sample Execution:

```
>>> cafe = Place("Cafeteria",  ["fix wires", "download data"])
>>> weapons = Place("Weapons", ["shoot asteroids", "fix wires"])

>>> ben = Impostor("@evilprof", cafe)
>>> waikay =  Impostor("@waisosus", cafe)
>>> kenghwee = Crewmate("@hweelingdead", cafe)
>>> jonathan = Crewmate("@lordjon", cafe)

>>> kenghwee.do_task("shoot asteroids")
'No such task shoot asteroids in Cafeteria'  # no such task in Cafe

>>> jonathan.do_task("fix wires")
'@lordjon does fix wires in Cafeteria'

>>> jonathan.do_task("fix wires")
'Task fix wires in Cafeteria already done'  # no duplicated task

>>> jonathan.move(weapons)
'@lordjon moves to Weapons'

>>> jonathan.do_task("fix wires")
'@lordjon does fix wires in Weapons'  # same task name at other place

>>> waikay.move(weapons)
'@waisosus moves to Weapons'

>>> waikay.do_task("adjust aiming")
'Impostors cannot do tasks'  # method override

>>> kenghwee.move(weapons)
'@hweelingdead moves to Weapons'
```

```
>>> ben.kill()
'Nobody to kill'  # all alive Crewmates are in Weapons

>>> waikay.kill()
'@waisosus kills @hweelingdead at Weapons'  # either kenghwee or
   jonathan can be killed

>>> ben.move(weapons)
'@evilprof moves to Weapons'

>>> ben.kill()
'@evilprof kills @lordjon at Weapons'  # kills the remaining alive

>>> ben.kill()
'Nobody to kill'  # no more alive crewmates to kill

>>> kenghwee.move(cafe)
'@hweelingdead moves to Cafeteria'  # dead crewmates can still move

>>> kenghwee.do_task("fix wires")
'@hweelingdead does fix wires in Cafeteria'  # and do tasks
```

Provide an implementation for the classes `Place`, `Crewmate` and `Impostor`.

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 —