CS1010S — Programming Methodology
National University of Singapore

Practical Examination

18 April 2015

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 **four** questions, out of which you are expected to answer **three**. The time allowed for solving this quiz is **2 hours**.
3. The maximum score of this quiz is **30 marks** and will consist of the best three scores out of the answers for the four questions (if all four questions are attempted). Note that the number of marks awarded for each question **IS NOT** correlated with the difficulty of the question.
4. The total score for this quiz is capped at **30 marks**. Marks in excess of your practical exam grade (because you attempted more than 3 questions) will be added to the marks for the midterm exams, which is capped at **100 marks**.
5. 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.
6. While you are also provided with the template `practical-template.py` to work with, your answers should be submitted on Coursemology.org. Note that you can run the test cases on Coursemology.org 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.
7. 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 rename your file `practical-exam-<mat no>.py` where `<mat no>` is your matriculation number and upload the file to the Practical Exam folder in IVLE Workbin. You should only upload one file. If you upload multiple files, we will choose one at random for grading.
8. Please note that it shall be your responsibility to ensure that your solution is submitted correctly to Coursemology (or uploaded to IVLE, if there are problems) 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.
9. 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 : Anagrams [10 marks]

"An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example Torchwood can be rearranged into Doctor Who." - Source Wikipedia

A. Write a function `are_anagrams` that takes two strings as inputs and returns `True` if the string are anagrams of each other, otherwise it returns `False`. You may assume there are no spaces in the strings.

Hint: the built-in Python function `sorted()` *can sort a sequence.* [4 marks]

Examples:

```
>>> are_anagrams('dictionary', 'indicatory')
True

>>> are_anagrams('listen', 'silent')
True

>>> are_anagrams('test', 'exam')
False

>>> are_anagrams('melon', 'watermelon')
False
```

B. Write a function `has_anagrams` that takes as input a list of strings, and returns `True` if the list contains at least two strings which are anagrams of each other. You may wish to make use of the function `are_anagrams` from before. [4 marks]

Examples:

```
>>> has_anagrams(['apple', 'banana', 'pear', 'reap'])
True

>>> has_anagrams(['apple', 'banana', 'pear', 'papaya'])
False
```

C. Write a function `find_anagrams` that takes as input a list of strings, and returns a list of anagrams found in the input. [2 marks]

Examples:

```
>>> find_anagrams(['actress', 'grudge', 'recasts', 'rugged',
                   'casters', 'apple', 'pear', 'reap'])
[['actress', 'recasts', 'casters'], ['grudge', 'rugged'], ['pear', 'reap']]

>>> find_anagrams(['these', 'aren\'t', 'the', 'anagrams',
                   'you\'re', 'looking', 'for'])
[]
```

Question 2 : Housing Matters [10 marks]

Important note: You are provided with a data file `hdb-resale-prices.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 real-estate agent friend needs some help to analyse the HDB resale market. He has obtained a history of the average quarterly prices of the flats for each town. The history is in a CSV formatted file with each row recording the average price of the different flat types for a town in a quarter.

Note that some values have a "`*`" or a "`-`" as there is either low or no sales of that flat type during that quarter. You can use the function `string.isdigit()` which returns `True` if the value of `string` is a valid number.

A. Your friend wants to know how much the flat prices have increased for a given town. You are to write a function `find_increase` that takes as input the filename of the CSV file and a given town as a string, and outputs the difference between the lowest and highest price for each flat type in a dictionary according to the format shown in the sample execution. If there are no valid prices for the given flat type, then the price value should be `None`. [5 marks]

Sample execution:

```
>>> find_increase('hdb-resale-prices.csv', 'Ang Mo Kio')
{'3room': 196000, '4room': 235000, '5room': 295500, 'exec': None}

>>> find_increase('hdb-resale-prices.csv', 'Jurong East')
{'3room': 182000, '4room': 218000, '5room': 305500, 'exec': 172000}
```

B. Next, your friend asks to find the cheapest town for each room type in a given year. Write a function `cheapest_town` that takes as input the filename of the CSV file and the year as an integer, and outputs for each room type, the town and average annual price as a tuple.

Note: You should first compute the average price of a flat-type for the year before comparing across the towns. [5 marks]

Sample execution:

```
>>> cheapest_town('hdb-resale-prices.csv', 2011)
{'3room': ('Woodlands', 298125.0), '4room': ('Yishun', 375250.0),
 '5room': ('Woodlands', 436625.0), 'exec': ('Sembawang', 519833.3333333333)}

>>> cheapest_town('hdb-resale-prices.csv', 2013)
{'3room': ('Woodlands', 316875.0), '4room': ('Yishun', 401250.0),
 '5room': ('Woodlands', 468250.0), 'exec': ('Sembawang', 580000.0)}
```

Question 3 : How to Train Your Dragon [10 marks]

The island of Berk, a remote Viking village, was plagued by dragons who periodically steal their livestock.

Toothless the Night Fury Meatlug the Gronckle Stormfly the Nadder

Figure 1: Dragons.

However, the vikings of Berk have since discovered how to train the dragons and now keep them as pets. With sufficient training, they can even ride on the dragons and fly through the sky! How cool is that?

A dragon will only allow a trainer to ride it if it has been successfully trained by the trainer. Other trainers will have to separately train the dragon if they wish to also ride it. Now not all dragons are created equal, and trainers each have different skill levels. Some dragons are harder to train than others, and trainers with higher skill levels take a shorter time to train the dragons.

In this question, you will model the dragons and the human trainers as they attempt to train and ride the dragons.

You are to implement two classes: `Dragon` and `Trainer`. `Dragon` naturally models the different species of dragons and `Trainer` likewise models a human trainer.

A `Trainer` is created with two arguments, the name (which is a String) and a skill level (which is an integer) of the trainer. `Trainer` supports the following **four** methods:

- `train(dragon)` which signifies one attempt at training the dragon. The difference between the skill level of the trainer and the level of the dragon determines how many attempts are needed before the dragon is trained by the trainer.

  For example, if the trainer's skill level is equal or larger than the dragon's level, then the dragon will be immediately trained. If for example, the trainer's skill level is 2 and the dragon's level is 3, then the dragon will be trained only on the 2nd invocation of the train method. *(See sample execution for more examples.)*

  The return value of the method will be a String according to the following conditions:

  - If the dragon has already been trained by the trainer, `"<Dragon name> has already been trained"` will be returned.
  - If the dragon was successfully trained on this attempt, `"<Trainer name> successfully trained <Dragon name>"` will be returned.

  - Otherwise, `"<Trainer name> failed to train <Dragon name>"` will be returned.

- `trained_dragons()` returns a tuple of the **names** of dragons that the trainer has successfully trained.
- `mount(dragon)` allows the trainer to mount the dragon, provided i) the dragon has been successfully trained by the trainer, and ii) both the dragon and the trainer are currently not mounted.

  The return value of the method will be a String according to the following conditions in order:

  - If the trainer is already mounted on a dragon, `"<Trainer name> is currently mounted on <Mounted dragon name>"` will be returned.
  - If another trainer is currently mounted on the dragon, `"<Other trainer name> is currently mounted on <Dragon name>"` will be returned.
  - If the trainer has not yet successfully trained the dragon, `"<Trainer name> has not yet trained <Dragon name>"` will be returned.
  - Otherwise, the trainer will mount the dragon and `"<Trainer name> mounts <Dragon name>"` will be returned.

- `dismount()` will return the string `"<Trainer name> is not mounted"` if the trainer is not currently mounted on any dragon. Otherwise, the trainer will dismount the currently mounted dragon and the string `"<Trainer name> dismounts from <Dragon name>"` will be returned.

A `Dragon` is created with two arguments, the name (which is a String) and the level (which is an integer) of the dragon. `Dragon` supports the following two methods:

- `get_trainers()` will return a tuple of the **names** of the trainers who have successfully trained the dragon.
- `fly()` will return the string `"<Dragon name> flies around with <Trainer name>"` if a trainer is currently mounted. Otherwise, the string `"<Dragon name> does not have a rider"` will be returned.

Sample execution:

```
>>> toothless = Dragon("Toothless", 7)
>>> meatlug = Dragon("Meatlug", 1)
>>> stormfly = Dragon("Stormfly", 5)
>>> hiccup = Trainer("Hiccup", 4)
>>> astrid = Trainer("Astrid", 5)

>>> astrid.train(stormfly)
Astrid successfully trained Stormfly

>>> astrid.mount(stormfly)
Astrid mounts Stormfly

>>> stormfly.fly()
```

```
Stormfly flies around with Astrid


>>> meatlug.fly()
Meatlug does not have a rider

>>> hiccup.mount(meatlug)
Hiccup has not yet trained Meatlug

>>> hiccup.train(meatlug)
Hiccup successfully trained Meatlug

>>> hiccup.train(meatlug)
Meatlug has already been trained

>>> hiccup.mount(meatlug)
Hiccup mounts Meatlug

>>> meatlug.fly()
Meatlug flies around with Hiccup


>>> hiccup.mount(stormfly)
Hiccup is currently mounted on Meatlug

>>> hiccup.dismount()
Hiccup dismounts from Meatlug

>>> hiccup.mount(stormfly)
Astrid is currently mounted on Stormfly

>>> astrid.dismount()
Astrid dismounts from Stormfly

>>> hiccup.mount(stormfly)
Hiccup has not yet trained Stormfly


>>> astrid.trained_dragons()
('Stormfly',)

>>> hiccup.trained_dragons()
('Meatlug',)

>>> stormfly.get_trainers()
('Astrid',)

>>> hiccup.train(stormfly)
```

```
Hiccup failed to train Stormfly

>>> hiccup.train(stormfly)
Hiccup successfully trained Stormfly

>>> stormfly.get_trainers()
('Astrid', 'Hiccup')


>>> hiccup.train(toothless)
Hiccup failed to train Toothless

>>> hiccup.train(toothless)
Hiccup failed to train Toothless

>>> hiccup.train(toothless)
Hiccup failed to train Toothless

>>> hiccup.train(toothless)
Hiccup successfully trained Toothless

>>> hiccup.mount(toothless)
Hiccup mounts Toothless

>>> toothless.fly()
Toothless flies around with Hiccup

>>> hiccup.trained_dragons())
('Meatlug', 'Stormfly', 'Toothless')
```

Question 4 : Bomber Raider [10 marks]

A band of tomb raiders discovered an ancient tomb rigged with bombs set to explode at the slightest touch. Subterranean scans show that the tomb consists of cells arranged in an n by m grid. Each cell either contains a treasure, a bomb or is a solid wall, as depicted by the figure below.

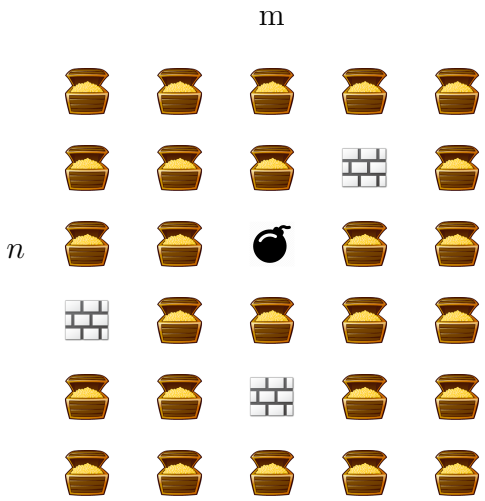


Figure 2: Example of a tomb layout.

When a bomb is triggered, the explosion travels outwards in a straight line along the horizontal and vertical cells, destroying any treasure in the cell it passes until it reaches the end of the tomb, or it hits a wall.

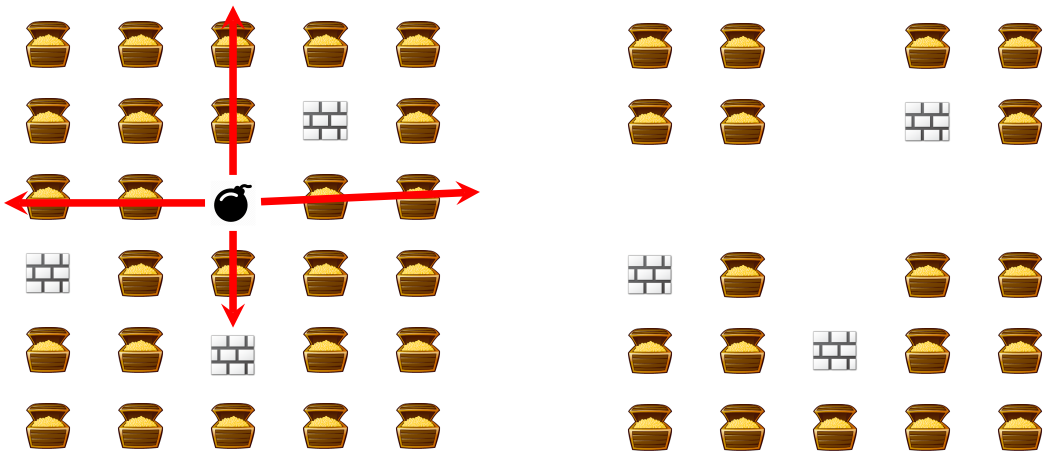
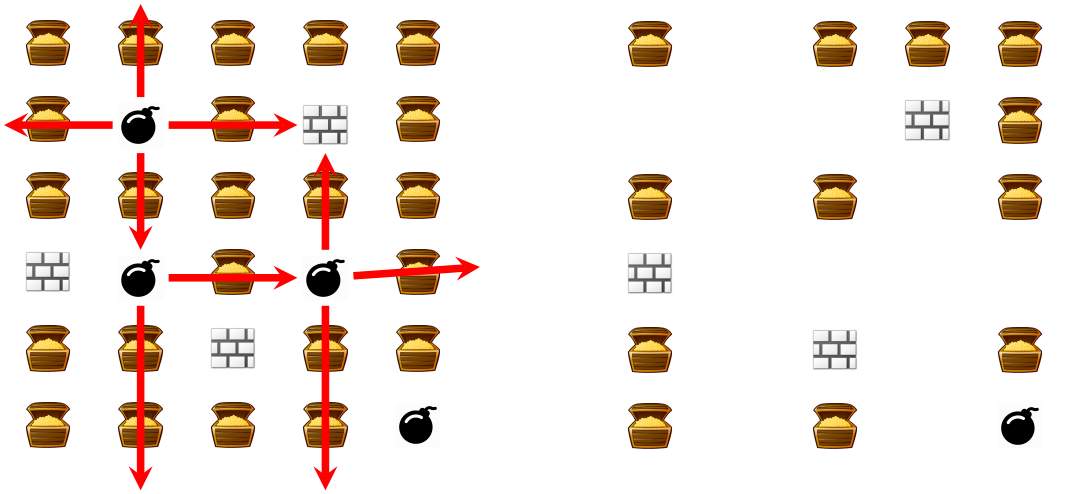


Figure 3: Bomb explode outwards destroying treasures in the path.

If another bomb is caught in the path of the explosion, the bomb is likewise triggered and explodes as well. Thus, it is possible for a chain reaction to occur when one bomb is triggered.



Before After

Figure 4: Bomb explodes causing other bombs to explode

The problem is, these tomb raiders are no Lara Croft, and cannot steal any treasure without triggering off at least one bomb. They have engaged you to use your programming skills to simulate the effects of triggering one bomb and to compute how many cells of treasure are left after the explosion.

Your job is to write a function `bomb_blast(n, m, bombs, walls, trigger)`, which takes in the following arguments:

1. `n`, the number of rows of cells of the tomb.
2. `m`, the number of columns of cells of the tomb.
3. `bombs`, which is a tuple of pairs `(row, col)` which specifies the row and column of each bomb.
4. `walls`, likewise is also a tuple of pairs which specifies the coordinates of the walls.
5. `trigger`, an integer, which is the index of the bomb in `bombs` that is to be triggered.

The output of the function should be the number of cells which contain treasure after the series of explosions. You can assume that the tomb is at least 1 x 1 and there is at least one bomb.

IMPORTANT! The tomb raiders have very limited time and computing resources on hand. Thus, **you cannot use recursion and can only use a single loop** in the function, i.e., either one `for` loop or one `while` loop. However, partial marks will still be awarded for correct answers that violate this condition.

Sample Execution:

Figure 4's example would be:

```
>>> bomb_blast(6, 5,
               ((1, 1), (3, 1), (3, 3), (5, 4)),
               ((3, 0), (4, 2), (1, 3)),
               0)
12

>>> bomb_blast(6, 7,
               ((1, 1), (4, 1), (1, 5), (4, 5)),
               ((2, 5),),
               1)
20
```

— E N D O F P A P E R —