CS1010X — Programming Methodology
National University of Singapore

Practical Examination

3 June 2017

Time allowed: 2 hours

Instructions (please read carefully):

1. This is **an open-book exam**.
2. This practical exam consists of **three** questions. The time allowed for solving this quiz is **2 hours**.
3. The maximum score of this quiz is **30 marks** and you will need to answer all three questions to achieve the maximum score. 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. Your answers should be submitted on Coursemology.org. You will be allowed to run some public tests cases to verify that your submission is correct. Note that you can run the test cases on Coursemology.org **up to 5 times** 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 on Coursemology.org. Do however ensure that you submit your answers correctly by running the test cases at least once.
6. You are also provided with the template `practical-template.py` to work with. If Coursemology.org fails, you will be required to rename your file `practical-exam-<mat no>.py` where `<mat no>` is your matriculation number and submit that file.
7. Please note that it shall be your responsibility to ensure that your solution is submitted correctly to Coursemology 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. You have been warned.
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 : WannaCry Fever [10 marks]

The WannaCry ransomware attack was unleash on the world on 12 May 2017. It was a worldwide cyberattack involving the WannaCry ransomware cryptoworm, which targeted computers running the Microsoft Windows operating system by encrypting data and demanding ransom payments in the Bitcoin cryptocurrency. Within a day it was reported to have infected more than 230,000 computers in over 150 countries. To prepare our country to defend against future cyber attacks, you have been recruited to train in (cyber-)security operations.

As part of your training, you have been assigned to work on a new keylogger. This is a covert program that is installed on machines to record everything they type into the computer. Often, this will allow our operatives to sniff the passwords of foreign adversaries. Because people often make mistakes when they type and use backspace and the arrow keys to correct their typing, you need to write some functions to figure out exactly what they were trying to type.

In this problem, the input can be any valid strings, but we shall assume that `"1"`, `"2"`, and `"3"` are never used and instead, we shall use `"1"`, `"2"`, and `"3"` to represent the left arrow, right arrow and backspace keys respectively.

A. [**Warm-Up**] Write a function `parse_keylog(log)` that takes in a string `log` and returns the string that the user had intended to type. You may assume that `"1"` and `"2"` will not appear in `log` and that `"3"` represents the backspace key. [5 marks]

Sample execution:

```
>>> parse_keylog("coolbeans")
"coolbeans"

>>> parse_keylog("p@sst3wiq33Ore3d")
"p@sswOrd"

>>> parse_keylog("33do3o3o3o3ont")
"dont"
```

B. [**Real Deal**] Write a function `parse_keylog(log)` that takes in a string `log` and returns the string that the user had intended to type. You should handle `"1"`, `"2"`, and `"3"` correctly. [5 marks]

Sample execution:

```
>>> parse_keylog("ac1b")
"abc"

>>> parse_keylog("ac1b2de12f")
"abcdef"

>>> parse_keylog("ac1b2r3de1t32f")
"abcdef"
```

Question 2 : #Twitter Mania [10 marks]

Twitter's #hashtags are a way of tagging (identifying) tweets to be related to a certain event or topic. In this question, you will do an analysis of a hashtag data and derive useful insights.

The hashtag data is provided in the accompanying comma-separated file `htags.csv`, where each entry contains the following information:

- ID : an integer value that uniquely identifies a #hashtag
- Hashtag: the actual hashtag including the # character
- Begin date-time: the date and time at which the hashtag first appeared online
- `Val(i) i.e. Val(1), Val(2), ... , Val(128)`: Number of tweets with the hashtag during the `i`th hour, for 128 hours since the begin date-time.

As an example, consider the following data entry:

```
8361,#Tehran,11/06/2009 21:54,13,11,11,10,9,9,9,11,12,15,17,19,21,...,14
```

Here the ID of the hashtag `#Tehran` is 8361. This hashtag first appeared online at 21:54 on 11/06/2009 (begin date-time). In the first one hour since the begin date-time, there were 13 tweets with this hashtag (including the first tweet). Then in the second hour, there were 11 tweets and so on until 14 tweets in the 128th hour. Such a data is called *volume time-series* data, as it provides the volume (number of tweets) in each hour for 128 consecutive hours.

A. **[Data Parsing]** The first step before analyzing any data is to parse it correctly, convert the values to appropriate `types` and store the records in an easy to operate structure. Here we will use the following nested list **structure** to store each record (brackets denote the datatype):

```
[ID(int),hashtag(str),begin date-time(datetime.datetime),\
 [Val1(int),Val2(int),...,Val128(int)]]
```

Thus, the record for `#Tehran` would look like the following:

```
[8361,'#Tehran',datetime.datetime(2009, 6, 11, 21, 54),[13,11,...,14]]
```

Note that our structure consists of **exactly 4** elements where the ID is stored as an integer, the hashtag is stored as a string, the begin date-time is stored as the Python `datetime.datetime` object and the volume time-series is stored as a list of integers.

Your task: Write a function parse_data, that takes in the name of the file containing the data and returns a **list** of 'records', where each 'record' is the structure described above. To facilitate reading of the csv file, the function read_csv is provided to you in the template file. Also to help construct Python's datetime.datetime object from a string, an additional helper function make_datetime is provided. Please refer to the Appendix for more on using make_datetime.

[2 marks]

Sample execution:

```
>>> parse_data('htags.csv')
[
  [8361, '#Tehran', datetime.datetime(2009, 6, 11, 21, 54), [13, 11,..., 14]],
  [20340, '#fact', datetime.datetime(2009, 6, 12, 7, 25), [4, 3,..., 349]],
  ...
]
```

B. **[Know the event dates!]** Since Twitter reflects all the major happenings in the world with 'trending' hashtags, it is possible to use these hashtags to get an approximate date of an event. Write a function get_approx_date, that takes in two parameters – name of the file containing the hashtag data and a hashtag that identifies an event. The function should return the approximate **date** (NOT time) when the event identified by the hashtag occurred. The event's date is best approximated by the begin date i.e. when the first tweet with that hashtag appeared online. Note that your function should return datetime.date object. Refer to Appendix to know how to return a datetime.date object from a datetime.datetime object. If the hashtag cannot be found, the function should return None. [2 marks]

Sample execution:

```
>>> get_approx_date('htags.csv','#Haiti') # Haiti earthquake
2009-06-11

>>> get_approx_date('htags.csv','#coup') # Hondurus coup d'etat
2009-06-14

>>> get_approx_date('htags.csv','#Jackson') # Tribute to Michael Jackson
2009-06-25
```

C. [Find the peak time!] Every Twitter hashtag in its lifetime reaches a peak time. Write a function `peak_time` that takes two inputs – name of the file containing the hashtag data and a hashtag and returns the peak time. The peak time of a hashtag is defined as the first hour during which maximum number of tweets with that hashtag occurred. The function should return the peak date-time of the hashtag as a Python `datetime.datetime` object. If the hashtag cannot be found, the function should return `None`.

Hint: You will need to use the volume timeseries data of a hashtag to find out the peak time. Refer to the Appendix to know how to add time to a Python `datetime.datetime` object. [3 marks]

Sample execution:

```
>>> peak_time('htags.csv','#GoogleWave')
datetime.datetime(2009, 6, 13, 17, 58)

>>> peak_time('htags.csv','#IranElections')
datetime.datetime(2009, 6, 14, 2, 8)
```

D. [Trending hashtags] *Trending* hashtags are those that are tweeted in large numbers. Write a function `trending_hashtags` that takes in two inputs – name of the file containing the hashtag data and an integer `k`. Your function should return a list of top `k` hashtags in decreasing order of the total number of tweets. If there are any other hashtags with the total number of tweets equal to the `k`th hashtag, include those hashtags as well. Also if there is a tie between two hashtags in terms of the total number of tweets, break the tie using alphabetical ordering i.e. `#Amman` comes before `#Tehran`.

[3 marks]

Sample execution:

```
>>> trending_hashtags('htags.csv',3)
['#fact', '#rememberwhen', '#shoutout']

>>> trending_hashtags('htags.csv',5)
['#fact', '#rememberwhen', '#shoutout', '#Haiti', '#red']
```

Question 3 : Cracking Mastermind [10 marks]

Mastermind is a board game consisting of 2 kinds of pegs. Coloured pegs and indicator pegs. The goal of the game is for a player to guess the configuration of a set of coloured pegs. One player (*"game master"*) sets up the game, but deciding on the configuration, which is hidden from the other player (*"guesser"*) who tries to deduce the hidden configuration.

The game takes place in rounds. In each round, the guessing player tries to guess the configuration of the coloured pegs. If the guessing player guesses correctly, the game ends. If not, the game master provides feedback to the guesser in the form of black and white indicator pegs. Each black indicator peg indicates that one coloured peg is in the right position; each white indicator peg indicates that one coloured peg is in the wrong position. The location of these pegs is not given with the black and white indicator pegs. A sample playthrough is shown in Figure 1. The top row shows the hidden solution. The game progresses from bottom up. When the guess reaches 5 black pegs the guesses has successfully guessed the answer.

In this problem, you will implement a `Mastermind` class. The constructor takes in a tuple of colours (which we represent with strings) and the hidden configuration (*"answer"*). The `Mastermind` class remembers the guesses and supports the following methods:

- `length()` returns the number of coloured pegs in the answer. [1 marks]
- `guesses()` returns the number of valid guesses so far. [1 marks]
- `try_solution(colour_1, colour_2, ..., colour_n)` takes in a number of arguments equal to the length of a guess of the solution and attempts to guess the solution. [4 marks]
- `remaining_possibilities()` returns the number of possible answers given what has been tried. [4 marks]

The return value for `try_solution` depends on the game play. In general, it returns a list of black and white pegs for the guess. However, under the following circumstances, it will have a different return value:

- `"Invalid colour"` – if the guess contains an invalid.
- `"Already solved!"` – if the game had earlier been solved.
- `"Wrong number of pegs"` – if the guess contains a different number of pegs than expected.
- `"Tried before!"` – if the guess had already been tried earlier.
- `"Solution found!"` – if the guess is correct.

Except for `"Solution found!"`, the other error codes will not remembered as valid attempts, i.e. they will be ignored.

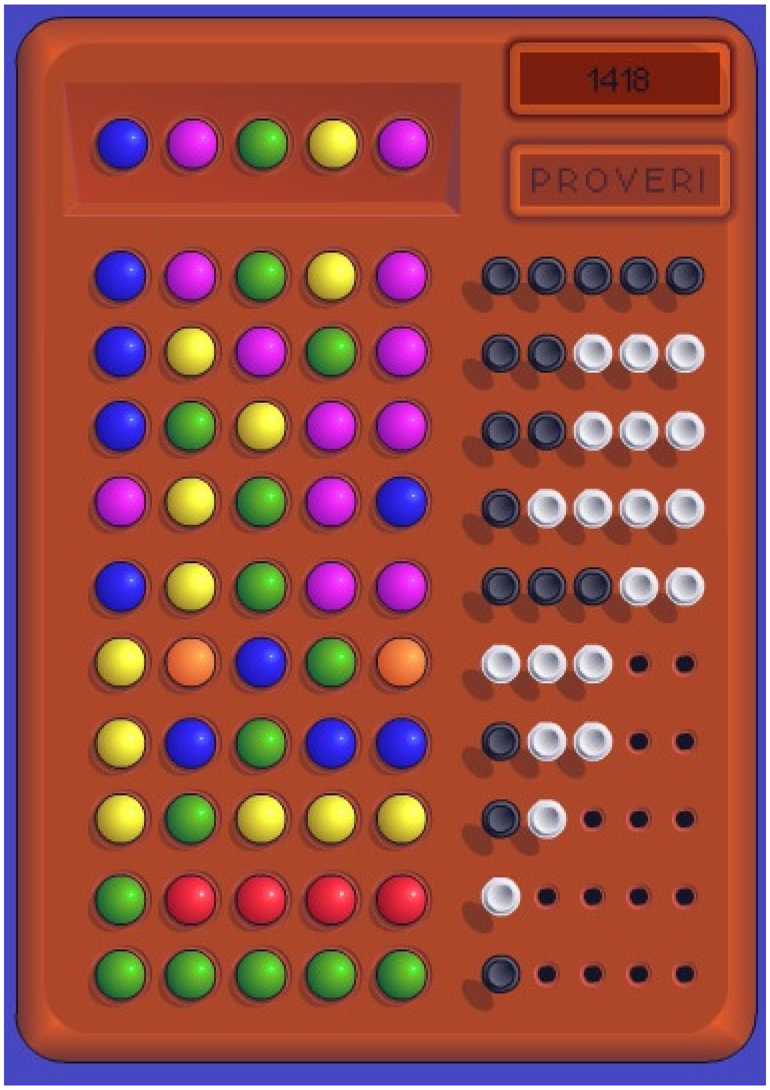

Figure 1: Sample playthrough for mastermind game.

Hint: It would probably be helpful to implement a helper function that when given the solution and a guess, returns the number of black and white pegs.

Sample execution:

```
>>> m = Mastermind(("red","blue"), ("red","red","blue"))
>>> m.remaining_possibilities() # [('red', 'red', 'red'), ('red', 'red', 'blue'),\
('red', 'blue', 'red'), ('red', 'blue', 'blue'), ('blue', 'red', 'red'), \
('blue', 'red', 'blue'), ('blue', 'blue', 'red'), ('blue', 'blue', 'blue')]
8

>>> m.try_solution("red","blue","green")
"Invalid colour"
```

```
>>> m.try_solution("red","blue","blue","red")
"Wrong number of pegs"
>>> m.remaining_possibilities()
8

>>> m.try_solution("red","blue","blue")
[2,0]
>>> m.remaining_possibilities() # [('red', 'red', 'blue'), ('red', 'blue', 'red'),\
 ('blue', 'blue', 'blue')]
3

>>> m.try_solution("red","blue","red")
[1,2]
>>> m.remaining_possibilities() # [('red', 'red', 'blue')]
1

>>> m.try_solution("red","red","blue")
"Solution found!"
>>> m.try_solution("red","red","blue")
"Already solved!"

>>> m = Mastermind(("red","blue","green"), ("red","red","blue"))
>>> m.remaining_possibilities()
27

>>> m.try_solution("red","blue","green")
[1,1]
>>> m.remaining_possibilities()
6

>>> m.try_solution("red","blue","green")
"Tried before!"
>>> m.remaining_possibilities()
6

>>> m.try_solution("green","red","blue")
[2,0]
>>> m.remaining_possibilities()
3

>>> m.try_solution("red","blue","red")
[1,2]
>>> m.remaining_possibilities()
1

>>> m.try_solution("red","red","blue")
"Solution found!"
```

```
>>> m = Mastermind(("red","blue","green","white"), ("red","white","blue"))
>>> m.remaining_possibilities()
64


>>> m.try_solution("red","blue","green")
[1,1]
>>> m.remaining_possibilities()
12

>>> m.try_solution("green","red","green")
[0,1]
>>> m.remaining_possibilities()
3

>>> m.try_solution("white","blue","red")
[0,3]
>>> m.remaining_possibilities()
1

>>> m.try_solution("red","white","blue")
"Solution found!"
```

Appendix

Useful operations with Python's `datetime` library:

```
import datetime

## Converting string to Python datetime.datetime
## Note the string format is the same as the hashtags data file
>>> d = make_datetime('12/06/2009 7:25') ## function provided in the template
>>> d
datetime.datetime(2009, 6, 12, 7, 25)

## extracting datetime.date from datetime.datetime
>>> d.date()
datetime.date(2009, 6, 12)

## adding time to datetime object
## note that the resulting object is datetime.datetime
>>> d + datetime.timedelta(hours=27)
datetime.datetime(2009, 6, 13, 10, 25)
```