UNIVERSITY OF SASKATCHEWAN
Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 2

Dictionaries, and References

Date Due: 24 January 2020, 8pm **Total Marks: 55**

Version History

- **19/01/2020**: Corrected the name of a file in Q4.
- **18/01/2020**: Fixed macro for submission naming from a1qN to a2qN.
- **17/01/2020**: released to students

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions. Plagiarism can include: copying answers from a web page, or from a classmate, or from solutions published in previous semesters. Basically, if you cannot delete your whole assignment and do it again yourself (given adequate time), it's not your work, so don't try to claim credit for it. Your success in this course depends on depends on what you can do, not on what you can hand in.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Read the purpose of each question. Read the Evaluation section of each question.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** If you are not sure, talk to a Lab TA about how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded. **Do not send late assignment submissions to your instructors, lab TAs, or markers.**

Question 1 (10 points):

Purpose: To work with the concept of references a bit more carefully.

Degree of Difficulty: Easy. If you understand references very well, this is not difficult.

In class (and in the readings) we saw a version of Selection sort. As described in the readings, our implementation of selection sort works by repeatedly removing the smallest value from an unsorted list and adding it to end of a sorted list until there are no more values left in the unsorted list (see Chapter 1 of the readings).

```
1 unsorted = [3, 2, 5, 7, 6, 8, 0, 1, 2, 8, 2]
2 sorted = list()
3
4 while len(unsorted) > 0:
5     out = min(unsorted)
6     unsorted.remove(out)
7     sorted.append(out)
8
9 print(sorted)
```

One problem with this implementation is that it modifies the original list (line 6). Unless we know for sure that a list will never be needed in the future, removing all contents is a bit drastic. To address this problem, we can change the code so that a copy of the original list is made first. Since we make the copy, we can say for sure that the copy will never be needed in the future, and so removing all its contents is absolutely fine.

The file `a2q1.py` is available on Moodle, and it contains a function called `selection_sort()` which is very similar to the above code:

```
1  def selection_sort(unsorted):
2      """
3      Returns a list with the same values as unsorted,
4      but reorganized to be in increasing order.
5      :param unsorted: a list of comparable data values
6      :return:  a sorted list of the data values
7      """
8
9      result = list()
10
11     # TODO use one of the copy() functions here
12
13     while len(acopy) > 0:
14         out = min(acopy)
15         acopy.remove(out)
16         result.append(out)
17
18     return result
```

On line 11, there is a TODO item, which is where we will add code to create a copy the original unsorted list.

Also in the file are 5 different functions whose intended behaviour is to copy a list. Your job in this question is to determine which, if any, of these functions does the job right.

Note:

There is a Python list method called `copy()`, which we are not using on purpose. We need to understand references, and we must not side-step the issue.

Task

For each of the 5 `copy()` functions in the file `a2q1.py`, do the following:

- Determine if the function makes a copy of the list. Hint: some do not!
- Determine if the function is suitable for use in `selection_sort()`. Hint: some are not!

Do not change the function `selection_sort()` except to make use of one of the versions of the `copy()` function on line 11. Do not change the code for any of the `copy()` functions.

What to Hand In

Your answers to the above questions in a text file called `a2q1.txt` (PDF, rtf, docx or doc are acceptable). You might use the following format:

```
Question 1
copy1()
   - makes a copy
   - is suitable
copy2()
   - makes a copy
   - is not suitable
```

The above example does not necessarily reflect the right answers!

Be sure to include your name, NSID, student number, section number, and laboratory section at the top of all documents.

Evaluation

Each of the copy functions is worth 2 marks. Your answers to both questions have to be correct to get the marks.

Question 2 (10 points):

Purpose: To work with the concept of references a bit more carefully.

Degree of Difficulty: Moderate. If you understand references very well, this is not difficult.

In the file `ascii-art.py`, you'll find a Python script that prints pictures to the console window, after opening and reading data from a compressed file. The objective of this question is to modify a couple of the functions in this script. You are invited to read the whole script, but you won't have to make changes to the trickiest parts of it.

The script reads a text file containing some data which represents an image. There are several example files given for you to try. The data files are encoded using a technique called *run-length encoding*. Once the data is read from the file, it is decoded, and turned into a 2-dimensional list of single character strings. For example:

```
[['+','+','+'],['+','+','-'],['+','-','-']]
```

Each sub-list represents a row; here we have 3 rows and three columns. We call this list of lists an *image*.

When displayed to the console, each character is displayed on a line, row by row:

```
+++
++-
+--
```

This example is not very artistic, but try out some of the examples!

There are 2 functions in the script we will be studying, and they have the following behaviour

- `flip_updown(image)`
  - Its input `image` is a list-of-lists.
  - Creates a new image containing all the rows of the original input image, but in reverse order; this corresponds to a flip across a horizontal axis.
  - Returns the new image.
  - Does not modify the original image.
- `flip_leftright(image)`
  - Its input `image` is a list-of-lists.
  - Creates a new image containing all the rows of the original input image in the same order, but each row in the new image is reversed; this corresponds to a flip across a vertical axis.
  - Returns the new image.
  - Does not modify the original image.

We will not be concerned with the other functions in the script. You can read the code, but don't be concerned if the details are a bit tricky.

Your task is to rewrite these two functions and change their behaviour to the following:

- `flip_updown(image)`
  - Its input is a list-of-lists.
  - Modifies the image so the rows are in reverse order; this corresponds to a flip across a horizontal axis.
  - Returns `None`
- `flip_leftright()`
  - Its input is a list-of-lists.
  - Modifies the image so the columns are in reverse order; this corresponds to a flip across a vertical axis.
  - Returns `None`

You'll have to adapt the script that calls these two functions (near the bottom of the file). This is part of the exercise. When you are done, your scripts produce the same images as the original script in the same order.

What to hand in:

- Hand in your working program in a file called `a2q2.py`.
- A text-document called `a2q2_demo.txt` that shows that your program working on a few examples. You may copy/paste to a text document from the console.
- If you wrote a test script, hand that in too, calling it `a2q2_testing.txt`

Evaluation

- 4 marks. Your function `flip_updown(image)` correctly modifies the given image.
- 4 marks. Your function `flip_leftright(image)` correctly modifies the given image.
- 2 marks. You modified the functions doc-strings to reflect the changes you made.

Question 3 (20 points):

Purpose: To practice reading and understanding code written by someone else. To experience the difficulty of trying to understand someone's code when it has not documentation at all. To practice documenting code according to the CMPT 145 standards. To practice thinking about references and objects.

Degree of Difficulty: Moderate. Figuring out what the code does, and how the references work may require some time.

References: You may wish to review the following chapters:

- General: CMPT 145 Readings, Chapter 2, 3, 4
- Dictionary Review: CMPT 141 Readings: Chapter 4

Background

On Moodle you will find sevreral Python files for this question.

- `Q3replay.py`: The main script.
- `flavourSW.py`: Some definitions used by the main script.
- `Examples.zip`: Some data files (we will call them replay files) that the main script can read.

The main script has Python code, but no documentation at all. The code makes pretty sophisticated use of dictionaries and references. In later questions on this assignment, you'll make a few adaptations to the code to implement different programs, but there will be a core common functionality.

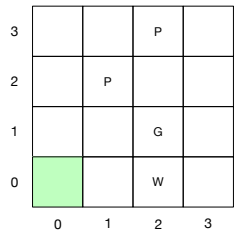
When game companies want to study how games play their games, they often study *game replays*. A game replay simply records all the details of the game, and the players actions. A replay player can read the replay file, and step through the game showing what the user did, and how the game worked out.

The code in `Q3replay.py` is a replay player. It reads a replay file, and simulates the game, showing all the output that the player would have seen.

The game itself is a simple game where a player explores a dungeon. The dungeon is represented by a $n \times n$ grid of rooms. Movement between rooms is limited to the four compass directions North,East,South,West. Some rooms will have one item in them:

- `'G'` This is a prize, and the goal is for the player to find the prize, and leave the dungeon safely. If the player enters the room with a `'G'`, the prize is removed from the room, and the player has taken it. Also, when the player enters the room with the prize, the game displays a message to the console.
- `'P'` This is a trap, and the player has to avoid them. If the player enters a room with a trap, the game is over, and the player loses. Traps do not move.
- `'W'` This is a deadly monster, and the player has to avoid it. If the player enters a room with a monster, the game is over, and the player loses. Monsters do not move.

For example, consider the following 4×4 dungeon, with two traps, and one monster.



In this example, locations are indicated using row-column indices (not x-y coordinates).

The goal of the game is to explore the dungeon starting at room $(0, 0)$ (the green square above), find a prize (G), and exit the dungeon again safely. The main problem for the player is that the player cannot see the items directly.

- The player cannot see the prize G until entering the room where it is located. The player automatically takes the prize.
- The player cannot see the traps P directly. However, if the player enters a room that is adjacent to a trap (that is, one move away), a message will be displayed to the console. This message indicates that the trap is near, but does not tell exactly where the trap is. In the above example, if the player enters room $(1, 1)$, the console will display a message that P is nearby. If the player enters room $(1, 2)$, the game is over, and the player loses.
- The player cannot see the monster W directly. However, if the player enters a room that is adjacent to a monster, a message will be displayed to the console. This message indicates that the monster is near, but does not tell exactly where the monster is. In the above example, if the player enters room $(1, 0)$, the console will display a message that W is nearby. If the player enters room $(0, 2)$, the game is over, and the player loses.

But remember, the replay player `Q3replay.py` does not choose where to move the player; all the player's moves are recorded in the replay file. The program only replays the game that was played earlier.

The replay file indicates the size of the dungeon, the location of the items, and the player's trip through the dungeon as a string made up of letters N,E,S,W:

```
4
G 2 0
P 3 0
P 1 1
W 2 1
NEWENWESWEWS
```

The first line is a number indicating the size. The next four lines show four item locations (in row-column indices). In this assignment there will be one prize, one monster, and two traps. The last line shows what directions the player moved in this game. The player always starts in the $(0, 0)$ room.

Task

1. Read the code very carefully. Give each function in the file a doc-string in the style of CMPT 145 Chapter 4. Specifically you must provide:
   - Purpose: A single sentence, maybe two, that describes what the function does. The purpose will not describe *how* the function works!
   - Pre-conditions: Give the name of each parameter, and describe hat kind of value is expected, and any constraints on the values allowed.
   - Post-conditions: Describe any behaviour of the function that might affect objects outside the function.
   - Return: describe what kind(s) of value are returned, and any constraints on these values.
2. At the top of the file, you usually put your name and student information. You will continue to do that! But for this question, you should add a comment describing the over-all purpose of the script. For example:

```
# CMPT 145 Assignment 2
# Name: A. Student
# ID: 12345678
# NSID: ast123
# Lab: 03
# Lecture: 01
#
# Synopsis:
# YOUR DESCRIPTION OF THE PURPOSE OF THE
# SCRIPT GOES HERE.  A FEW SENTENCES ONLY.
```

   A synopsis can help a reader understand the contents of a script without reading the whole script.
3. Consider the following code snippet, which uses some of the functions, but is not actually part of the replay player:

```
roomA = create_room()
roomB = create_room()
roomC = create_room()
connect(roomA, 'row', roomB)
connect(roomB, 'col', roomC)
```

   Draw a diagram showing the frame for the above snippet, and all the objects involved, as they exist in the heap at the end of the script. To be precise, your diagram does not have to show the frames for the four function calls.
4. On line 96 of the file is the following assignment statement:

```
        loc = loc[s]
```

   Explain what this line does.

What to Hand In

- Your version of the replay program, named `a2q3.py`, documented with doc-strings as per CMPT 145 expectations, and the synopsis.
- Your diagram of the code snippet from part 3 above. You can draw this on paper, and submit a photo or a scan of your diagram. You may use some software to make your diagram, but it might require more time. Name your file `a2q3.JPG`; allowable file formats are JPG, PNG, GIF, PDF.

- Your description of line 96 of the original replay program, in a file named `a2q3.txt`

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents. Put this information on your diagram as well!

Evaluation

- 7 marks: You documented every function using CMPT 145 requirements, and provided a synopsis.
- 7 marks: Your diagram correctly shows the objects and references in the snippet.
- 6 marks: Your explanation indicates the purpose of the statement, and the technical details involving objects and references.

Question 4 (15 points):

Purpose: To reuse and adapt code for another purpose.

Degree of Difficulty: Moderate. You really need to understand the code from Q3 to do this. Only a few functions need to be changed, but you could go far off track.

Textbook: Chapter 2, 3

The replay files were generated at random by a simple Python script. In other words, there was no one playing the game, trying to get the prize (`'G'`) and escape without dying. As you may have noticed, some of the replays end before the whole movement string is processed. Thats' a dead give-away!

In this question, you will adapt and reuse code in the original replay script to create an interactive version of the game, which will allow players to play, rather than to watch a replay.

The game has to start with a dungeon, so we have provided a few examples. The dungeon data files are simply replay files with the movement string deleted. For example:

```
4
G 2 0
P 3 0
P 1 1
W 2 1
```

Task

Implement an interactive version of the script, as follows:

1. Make a copy of your well-documented version of previous question, and modify it so that it reads a dungeon file, and interacts with the user to play the game. A dungeon file is simply a replay file without any movement information. Your program will keep track of where the player is, provide descriptions of the rooms, as appropriate, and ask the user for a direction to move (in contrast, the replay script used the moves recorded in the replay file).

   You may have to modify the doc-strings of any functions you modified, and provide doc-strings for any functions you added. Use the CMPT 145 standards!

2. So maybe you're not a fan of Star Wars. Make a copy of the ~~`starwars.py`~~ `flavourSW.py` script (which contains all dialogue used in the game), and modify the strings to make your version of the game.

3. Change the score returned at the end of the game, as follows:
   - If the player dies, the score is -1000.
   - If the player collects the prize and returns to the start, the player's score is $1000 - n$ where n is the number of steps taken in total. In other words, a short path to the prize and back is best!
   - If the player returns to the start without dying, but without collecting the prize, the player's score is $(T^2 - n)n$, where n is the number of steps taken in total, and T is the size of the dungeon. This formula rewards longish paths, but penalizes very short paths (encouraging the player to explore) and penalizes very long paths (discouraging the player from getting a high score moving back and forth). For example, in a 4×4 dungeon ($T = 4$) a player returns to the start after 10 moves ($n = 10$), the score will be $10 \times (4^2 - 10) = 60$.

What to Hand In

Your implementation of the interactive game:

- The main script `a2q4.py`

- Your version of the dialogue file named `a2q4_dialog.py`. Your program should import this.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your code is clearly adapted from the given code.
- 5 marks: Your code is correct.
- 5 marks: Your code is well-documented, according to the CMPT 145 standard.