CE151 ASSIGNMENT 2 2014

Set by: Mike Sanderson

Credit: 20% of total module mark

Deadline: 11.59.59, Monday 8 December

Submission of this assignment will be via the online submission system; your programs may be tested during your lab on 11.

It is expected that marks and feedback will be returned by the beginning of the spring term.

You should refer to pages 20-21 and 54-57 of the Undergraduate Students' Handbook for details of the departmental policy regarding late submission and university regulations regarding plagiarism; the work handed in must be entirely your own.

This assignment comprises two exercises, to be written as separate programs in files called **`ex1.py`** and **`ex2.py`**. The first will contribute 40% of the mark and the second 50%. The remaining 10% of the total mark for the assignment will be awarded for programming style and documentation. Sample data files to be used for testing the programs will be supplied.

Exercise 1

The attributes of an student are name, year, registration number and degree scheme.

In the file **`ex1.py`** write a function which takes a string argument, and creates and returns a tuple containing details of the student specified by the string. The string should be assumed to have the format

```
12345 2 G400 Bartholomew Homer Simpson
```

The first three items in the line will be the registration number, year and degree scheme and the rest of the line will be the name. There will be no spaces in the degree scheme but there will be spaces in the name. The number of words in the name may vary. The tuple should contain exactly 5 items: the registration number, year, degree scheme, surname and other names. (The other names should be stored as a single string; the surname will be the last name in the input line so in the example above the surname is `Simpson` and the other names are `Bartholomew Homer`.)

Next, write a function that will print the details of a student on a single output line. It should take a tuple as its argument and print the details in fixed-width fields using a layout such as

```
Simpson, Bartholomew Homer       12345  G400   Year 2
```

The name should be displayed using the format shown above so you will need to create a string containing the name in this format and print it in a fixed-width field. You may assume that no name will contain more than 32 characters when displayed in this format, no degree scheme will contain more than 6 characters, registration numbers will contain at most 7 digits and all the year will be a single-digit number.

In the main body of the program, write code which prompts the user for a filename and attempts to open the file whose name is supplied. If the file cannot be opened an error message should be output and the program should terminate; otherwise the program should read each line from the file and supply it to the student-creation function, storing the tuples returned by this function in a list. You may assume that the file contains lines that conform to the format described for the argument to tuple-creation function.

After the data has been input the program should display details of all students in the list in a neat table (using the function already written) then enter a loop in which the user should be given the option of requesting the display of full details of all of the students with a particular degree scheme or with a year in a particular range (e.g. 1 to 2) or the name of the student with a particular registration number, or quitting the program. The user should then be asked to supply the degree scheme, registration number or upper and lower bounds of the year range (as appropriate), the list should then be searched and the appropriate output. For the degree scheme and year-range searching the output should be performed using the function written earlier, but for the registration-number search you should output just the name using the format `Bartholomew Homer Simpson`. The degree-scheme search output should be sorted alphabetically by last name (also using the first name if the last names of two students are the same).

Appropriate messages should be displayed if a search produces no results.

Exercise 2

Hangman is a traditional pencil-and-paper word-guessing game (see for example http://en.wikipedia.org/wiki/Hangman_(game)).

The aim of this exercise is to write a Python version of the game in which the user is the guesser. Instead of displaying the hangman diagram the program will instead simply tell the user how many "lives" (permitted wrong guesses) remain. A typical interactive session might look like

```
WORD: ******; you have 5 lives left

Guess a letter: E
The letter E does not occur in the word

WORD: ******; you have 4 lives left

Guess a letter: O
The letter O occurs once in the word
WORD: ****O*; you have 4 lives left

Guess a letter: T
The letter T occurs once in the word

WORD: **T*O*; you have 4 lives left
```

eventually leading to

```
WORD: PYTHO*; you have 2 lives left

Guess a letter: N
The letter N occurs once in the word

WORD: PYTHON
Well done - you have guessed the word correctly
```

or

```
WORD: **T*ON; you have only 1 life left

Guess a letter: S
The letter S does not occur in the word
You have no lives left - you have been hung!
The word was PYTHON
```

2.1 The Game-State Data Structure

The state of a game of hangman must be stored in a single item of structured data so that it can be passed as an argument to the functions that have to be written. Since that state of the game will change frequently and tuples are immutable the use of a tuple is not appropriate. Hence the game state must be stored a dictionary object whose keys are the names of the attribute (optionally abbreviated) and whose values are the values of the attributes. The attributes should be the secret word that the player is trying to guess (stored as a string), the current guess (e.g `**T*ON` in the example above) and the number of lives remaining. The current guess must be stored as a list of single character strings (e.g. `["*","*","T","*","O","N"]`) since it will need to be modified after a correct guess and strings are immutable.

For the part of the exercise six functions must be written. The first should take two arguments: a string representing the secret word and an integer representing the total number of lives available. It should generate and return a dictionary object of the form described above with the current guessed word being a list of `"*"` whose length is the same as the string argument.

The second function should take as an argument a dictionary object representing the state of the game and return a Boolean result indicating whether the word has been successfully guessed.

The third and fourth functions should take as an argument a dictionary object and return the value of its lives-left attribute or its secret-word attribute.

The fifth function should take as an argument a dictionary object and generate and return a string of the form

```
WORD: PYTHO*; you have 2 lives left
```

The final function will be used to update the game state after a guess has been made. This should have two arguments: a dictionary object and a single-character string (which will always contain an upper-case letter). If the letter occurs in the word the current guessed word should be updated, otherwise the number of lives remaining should be decremented. The method should return the number of occurrences of the letter in the word.

The functions written for this section must be the only functions that make use of knowledge of how the structured data has been stored – you must not use the names of the keys in any other parts of the program

2.2 The `playGame` Function

A `playGame` function must be provided; this should take two arguments, a word and a number of lives, supply these arguments to the first function described above, storing the result returned in a variable (which here I have assumed is called `hm`) using these arguments, and then enter a loop of the form

```
WHILE livesLeft(hm)>0 and not guessed(hm):
   ...........
```

Inside this loop the current incomplete word and life count should be output (calling the function written in section 2.1 to generate the string to be output) then the user should be asked to guess a letter. A check should be made that the user has supplied a single upper-case letter and this should be passed as an argument to the update-game-state function. A message indicating the number of occurrences of the letter (as returned by the function) should then be output.

Beneath the loop you should add code that outputs an appropriate success or failure message and also informs the user what the word was if he or she has no lives left.

[It is recommended that you test this function interactively by calling it from the IDLE shell before proceeding to section 2.3.]

2.3 The Main Body

The main body of the program should contain code to ask the user to supply a filename, attempt to open the file, and read the lines of the file into a list of strings. Each line in the file will contain a single upper-case word. If the file cannot be opened the program should terminate immediately.

A loop should then be entered; its body should start by asking the user which level he or she wants – easy, intermediate or hard. The level should be used to determine the number of lives; the hard level should allow the user 5 lives and the easy level about 10 lives. A random word should then be selected from the list and supplied along with the number of lives as arguments to a call to the `playGame` function. After returning from this function the user should be asked if he or she wants to play again.

Documentation and Commenting

All functions in each of the programs should have documentation strings stating precisely what they do (but ***not*** how they do it), what their arguments are and what they return.

Within function bodies occasional brief comments, such as or "update current guessed word" should be used to indicate what is being done by blocks of code – comments stating what each individual line does should not be provided.

Marking Scheme

Exercise 1: 15 marks will be awarded for the successful input of the data into tuples and the output of the list of students; the remaining 25 marks will be available for successful searching.

Exercise 2: 25 marks will available for section 2.1, 15 marks for section 2.2 and 10 marks for section 2.3.

Submission

If you have attempted both exercises place your two files **`ex1.py`** and **`ex2.py`** in a single **`.zip`** or **`.7z`** file and submit this to FASER. The file *must* be in one of these two formats; work submitted in any other format will not earn any marks.

If you have attempted just one of the two exercises it is acceptable to submit a single **`.py`** file.