Programming Principles (CSP1150)

Assignment 2:	Individual programming project (Quiz Program)
Assignment Marks:	Marked out of 30, (30% of unit)
Due Date:	2 November 2020, 9:00AM

Background Information

This assignment tests your understanding of and ability to apply the programming concepts we have covered throughout the unit. The concepts covered in the second half of the unit build upon the fundamentals covered in the first half of the unit.

Assignment Overview

You are required to design and implement two related programs:

- "**admin.py**", a CLI program that allows the user to manage a list of quiz questions which are stored in a text file. This program is referred to as the "Main Program" in the marking rubric. Develop this program before "quiz.py".
- "**quiz.py**", a GUI program that uses the data in the text file to implement a quiz game for the user to play. This program is referred to as the "GUI Program" in the marking rubric. Develop this program after "admin.py".

The following pages describe the requirements of both programs in detail.

Starter files for both programs are provided along with this assignment brief to help you get started and to facilitate an appropriate program structure. *Please use the starter files.*

Please read the entire brief carefully, and refer back to it frequently as you work on the assignment.
If you do not understand any part of the assignment requirements, **contact your tutor**.
Be sure to visit the **Blackboard discussion boards** regularly for extra information, tips and examples.

Pseudocode

As emphasised by the case study of Module 5, it is important to take the time to properly *design* a solution before starting to write code. Hence, this assignment requires you to *write and submit pseudocode of your program design for "admin.py"*, but not "quiz.py" (pseudocode is not very well suited to illustrating the design of an event-driven GUI program). Furthermore, while your tutors are happy to provide help and feedback on your work throughout the semester, they will expect you to be able to show your pseudocode and explain the design of your code.

You will gain a lot more benefit from pseudocode if you actually attempt it *before* trying to code your program – even if you just start with a rough draft to establish the overall program structure, and then revise and refine it as you work on the code. This back and forth cycle of designing and coding is completely normal and expected, particularly when you are new to programming. The requirements detailed on the following pages should give you a good idea of the structure of the program, allowing you to make a start on designing your solution in pseudocode.

See Reading 3.3 and the discussion board for further advice and tips regarding writing pseudocode.

Write a *separate section of pseudocode for each function* you define in your program so that the pseudocode for the main part of your program is not cluttered with function definitions. Ensure that the pseudocode for each of your functions clearly describes the parameters that the function receives and what the function returns back to the program. Pseudocode for functions should be presented *after* the pseudocode for the main part of your program.

It may help to think of the pseudocode of your program as the content of a book, and the pseudocode of functions as its appendices: It should be possible to read and understand a book without necessarily reading the appendices, however they are there for further reference if needed.

The functions are required in "admin.py" are detailed later in the assignment brief.

The following pages describe the requirements of both programs in detail.

Overview of "admin.py"

"admin.py" is a program with a Command-Line Interface (CLI) like that of the programs we have created throughout the majority of the unit. The program can be implemented in under 180 lines of code (although implementing optional additions may result in a longer program). *This number is not a limit or a goal – it is simply provided to prompt you to ask your tutor for advice if your program significantly exceeds it.* Everything you need to know in order to develop this program is covered in the first 7 modules of the unit. This program should be developed before "quiz.py".

This program allows the user to manage a collection of quiz questions that are to be stored in a text file named "data.txt". Use the "`json`" module to write data to the text file in JSON format and to read the JSON data from the file back into Python. See Reading 7.1 for details regarding this.

To illustrate the structure of the data, below is an example of the file content in JSON format:

```json
JSON
[
    {
        "question": "In which year was Halley's Comet last visible
                     from Earth?",
        "answers": ["1986"],
        "difficulty": 3
    },
    {
        "question": "Where were the 2016 Summer Olympics held?",
        "answers": ["rio de janeiro", "rio", "brazil"],
        "difficulty": 2
    }
]
```

This example contains the details of two quiz questions, stored in a *list*. The details of each question are stored in a *dictionary* consisting of three items that have keys of:

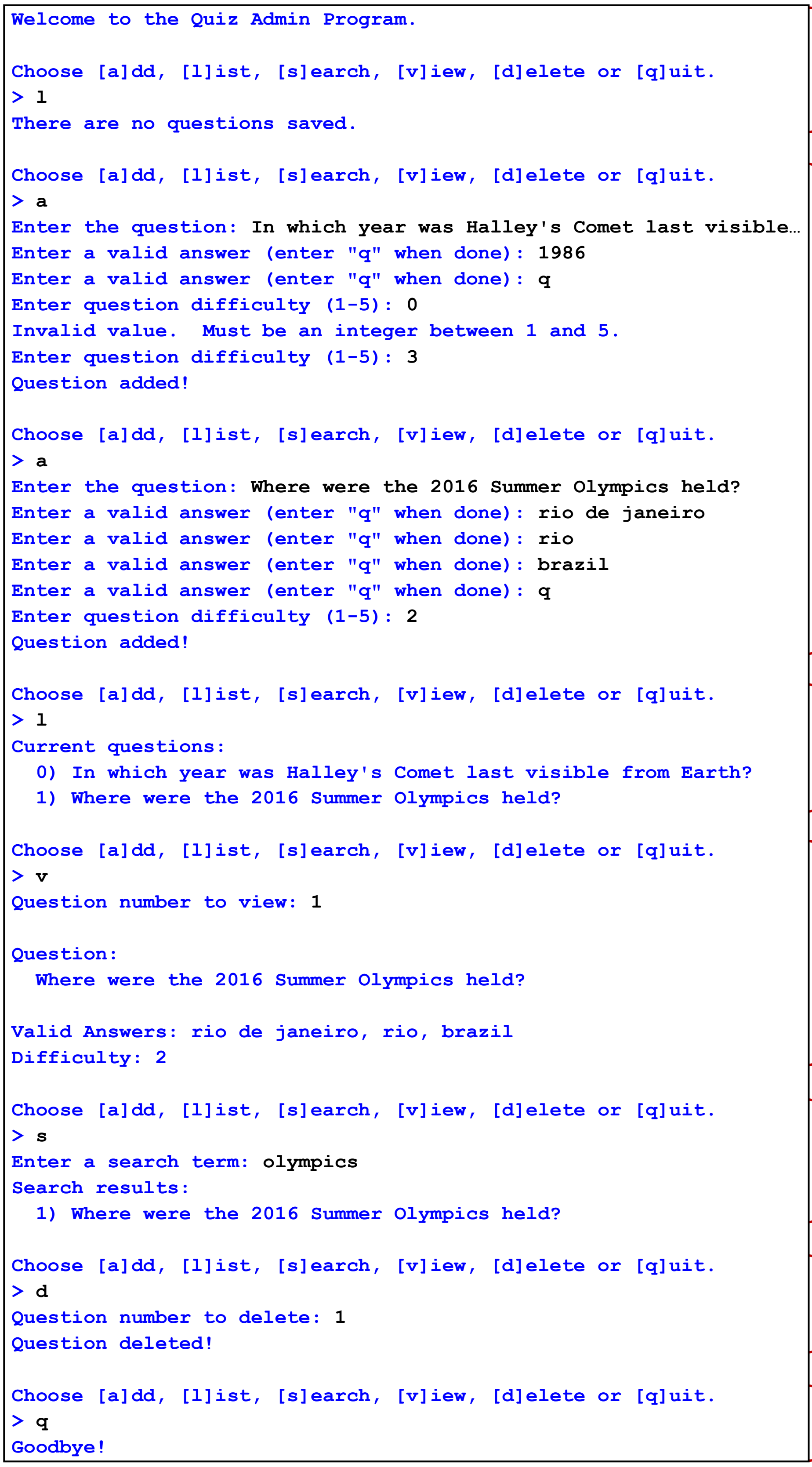
- **"question"** (a string of the quiz question)
- **"answers"** (a list of strings representing all of the answers that are considered correct)
    - *Since the user will be typing their answer in the GUI program, a list of strings allows the program to account for different variations of how a correct answer may be typed. It also allows for questions with multiple correct answers.*
    - *They have been stored in lowercase to make it easier to match against the answer entered by the user.*
- **"difficulty"** (an integer between 1 and 5 representing how difficult the question is)

If this file was to be read into a Python variable named `data`, then "`data[0]`" would refer to the entire dictionary containing the question about Halley's Comet, and "`data[0]['difficulty']`" would refer to the integer of 3. "`data[1]['answers']`" would refer to the list of `["rio de janeiro", "rio", "brazil"]`.

Understanding the structure of this data and how you can use it is *very important* in many aspects of this assignment – in particular, you will need to understand how to loop through the items of a list and how to refer to items in a dictionary. Revise Module 3 and Module 7 if you are unsure about how to interact with lists and dictionaries, and see the **Blackboard discussion board** for further help.

Output Example of "admin.py"

To help you visualise the program, here is a screenshot of it being used:



```
Welcome to the Quiz Admin Program.

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> l
There are no questions saved.

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> a
Enter the question: In which year was Halley's Comet last visible…
Enter a valid answer (enter "q" when done): 1986
Enter a valid answer (enter "q" when done): q
Enter question difficulty (1-5): 0
Invalid value.  Must be an integer between 1 and 5.
Enter question difficulty (1-5): 3
Question added!

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> a
Enter the question: Where were the 2016 Summer Olympics held?
Enter a valid answer (enter "q" when done): rio de janeiro
Enter a valid answer (enter "q" when done): rio
Enter a valid answer (enter "q" when done): brazil
Enter a valid answer (enter "q" when done): q
Enter question difficulty (1-5): 2
Question added!

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> l
Current questions:
  0) In which year was Halley's Comet last visible from Earth?
  1) Where were the 2016 Summer Olympics held?

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> v
Question number to view: 1

Question:
  Where were the 2016 Summer Olympics held?

Valid Answers: rio de janeiro, rio, brazil
Difficulty: 2

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> s
Enter a search term: olympics
Search results:
  1) Where were the 2016 Summer Olympics held?

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> d
Question number to delete: 1
Question deleted!

Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> q
Goodbye!
```

The program welcomes the user, then shows a list of options to choose from.

The user chooses "l" to list the current questions but there are none saved yet.

The user chooses "a" and is prompted to enter the details of a question (which did not quite fit into this screenshot). The user entered one valid answer before entering "q" to move on, and they entered an invalid difficulty before being re-prompted and entering "3".

They then choose "a" again to enter the details of another question. This one has three valid answers that the user enters.

Our list of questions now contains two items.

***Note:** When entering questions and answers, the colour of what you type may change if you use words or symbols that Python recognises. This won't impact the input or the program.*

The user chooses "l" and is shown a list of the questions that have been added.

The user chooses "v" and then enters "1" when prompted for an index number.

They are shown the details of the selected question. The details are presented nicely, including all of the answers and the difficulty.

The user chooses "s" and then enters "olympics" when prompted for a search term.

They are shown a list of questions containing the search term in the question text.

The user chooses "d" and then enters "1" when prompted for an index number.

The selected question is deleted from the list.

Finally, the user chooses "q" to quit.

Requirements of "admin.py"

In the following information, numbered points describe a *requirement* of the program, and bullet points (in italics) are *additional details, notes and hints* regarding the requirement. Ask your tutor if you do not understand the requirements or would like further information. The requirements are:

1. The first thing the program should do is try to open a file named "data.txt" in read mode, then load the data from the file into a variable named `data` and then close the file.
   - *The data in the file should be in JSON format, so you will need to use the "`load()`" function from the "`json`" module to read the data into your program. See the earlier page for details of the structure.*
   - *If any exceptions occur (e.g. due to the file not existing, or it not containing valid JSON data), then simply set the `data` variable to be an empty list. This will occur the first time the program is run, since the data file will not exist yet. This ensures that you are always left with a list named `data`.*
   - *This is the first and only time that the program should need to read anything from the file. After this point, the program uses the `data` variable, which is written to the file whenever a change is made.*

2. The program should then print a welcome message and enter an endless loop which starts by printing a list of options: "Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit." and then prompts the user to enter their choice. Once a choice has been entered, use an "`if/elif`" statement to handle each of the different choices (detailed in the following requirements).
   - *This requirement has been completed for you in the starter file.*

3. If the user enters "`a`" (add), prompt them to enter a question, then prompt them to enter as many answers as desired, and finally prompt them to enter the difficulty of the question, which must be an integer between 1 and 5. Place the details into a new dictionary with the structure shown on the previous page, and append the dictionary to the `data` list. Finally, write the entire data list to the text file in JSON format to save the data.
   - *Use your "`input_something()`" function (detailed below) when prompting for question and answers, to ensure that they are re-prompted until they enter something other than whitespace.*
   - *How you handle prompting the user to enter multiple answers is up to you, as long as you ensure that they enter at least one answer. You may want to store the answers in lowercase, to make it easier to match the user's answer to them in the GUI program.*
   - *Use your "`input_int()`" function (detailed below) when prompting for an index number, to ensure that the user is re-prompted until they enter an integer. Ensure that the number is between 1 and 5.*
   - *Once the dictionary for the new question has been appended to the `data` list, call your "`save_data()`" function (detailed below) to write the data to the text file in JSON format.*

4. If the user enters "`l`" (list), print a list of all the questions (just the question text, not the answers or difficulty) in the `data` list, preceded by their index number.
   - *If the `data` list is empty, show a "No questions saved" message instead.*
   - *Use a "`for`" loop to iterate through the items in the `data` list. Remember: each item is a dictionary.*
   - *You can use the "`enumerate()`" function to make sure you have a variable containing the index number of each question as you loop through them (see Lecture 3).*

5. If the user enters "`s`" (search), prompt them for a search term and then list the questions that contain the search term. Include the index number of the question next to each result.

- *If the `data` list is empty, show a "No questions saved" message instead of prompting for a search term.*
- *Use your "`input_something()`" function (detailed below) when prompting for a search term, to ensure that the user is re-prompted until they enter something other than whitespace.*
- *The code to search will be similar to the code used to list, but this time the loop body needs an "`if`" statement to only print questions that contain the search term (use the "`in`" operator – see Lecture 3).*
- *Convert the search term and question text to lowercase to find matches regardless of case.*

6. If the user enters "`v`" (view), prompt them for an index number and then print details of the corresponding question in full. This should include the question text, answers, and difficulty.

- *If the `data` list is empty, show a "No questions saved" message instead of prompting for index number.*
- *Use your "`input_int()`" function (detailed below) when prompting for an index number, to ensure that the user is re-prompted until they enter an integer.*
- *Print an "Invalid index number" message if the index number entered doesn't exist in the `data` list.*
- *Print the list of answers in a pleasing way (without "[]"), rather than simply printing the entire list. e.g.*

```
Valid Answers: rio de janeiro, rio, brazil
```

7. If the user enters "`d`" (delete), prompt them for an index number and then delete the corresponding question's dictionary from the `data` list, then print a "Question deleted" message.

- *If the `data` list is empty, show a "No questions saved" message instead of prompting for index number.*
- *Use your "`input_int()`" function (detailed below) when prompting for an index number, to ensure that the user is re-prompted until they enter an integer.*
- *Print an "Invalid index number" message if the index number entered doesn't exist in the `data` list.*
- *Once the question has been deleted from the `data` list, call your "`save_data()`" function (detailed below) to write the data to the text file in JSON format..*

8. If the user enters "`q`" (quit), print "Goodbye!" and `break` out of the loop to end the program.

9. If the user enters anything else, print an "Invalid choice" message (the user will then be re-prompted for a choice on the next iteration of the loop).

This concludes the core requirements of "admin.py". The following pages detail the functions mentioned above and optional additions and enhancements that can be added to the program. Remember that you are required to submit pseudocode for your design of "admin.py". Use the starter file and requirements above as a guide to structure the design of your program.

Functions in “admin.py”

The requirements above mentioned 3 functions - “`input_int()`”, “`input_something()`”, and “`save_data()`”. As part of “admin.py”, you must define and use these functions.

1. The “`input_int()`” function takes 1 parameter named `prompt`. The function should repeatedly re-prompt the user (using the `prompt` parameter) for input until they enter an integer of 0 or more (i.e. minimum of 0). It should then return the value as an integer.

- *See Workshop 4 for a task involving the creation of a very similar function.*

2. The “`input_something()`” function takes 1 parameter named `prompt`. The function should repeatedly re-prompt the user (using the `prompt` parameter) for input until they enter a value which consists of at least 1 non-whitespace character (i.e. the input cannot be nothing or consist entirely of spaces, tabs, etc.). It should then return the value as a string.

- *Use the “`strip()`” string method on a string to remove whitespace from the start and end. If a string consists entirely of whitespace, it will have nothing left once you strip the whitespace away.*
- *Note that exception handling is not needed in this function.*

3. The “`save_data()`” function takes 1 parameter named `data_list` (the `data` list from the main program). The function should open “data.txt” in write mode, then write the `data_list` parameter to the file in JSON format and close the file. This function does not return anything.

- *This is the only part of the program that should be writing to the file, and it always overwrites the entire content of the file with the entirety of the current data.*
- *See Reading 7.1 for an example of using the “`json`” module. You can specify an additional `indent` parameter in the “`dump()`” function to format the JSON data nicely in the text file.*

The definitions of these functions should be at the start of the program (as they are in the starter file provided), and they should be called where needed in the program. Revise Module 4 if you are uncertain about defining and using functions. Ensure that the functions behave exactly as specified; It is important to adhere to the stated function specifications when working on a programming project.

In particular, remember that the “prompt” parameter of the input functions is for *the text that you want to show as a prompt*. Here is an example of the function being called and its output:

```
age = input_int('Enter your age: ') ➡ Enter your age: |
```

You are welcome to define and use additional functions if you feel they improve your program, but be sure to consider the characteristics and ideals of functions as outlined in Lecture 4.

Optional Additions and Enhancements for "admin.py"

Below are some suggestions for minor additions and enhancements that you can make to the program to further test and demonstrate your programming ability. They are *not required* and you can earn full marks in the assignment without implementing them.

- Add 1 to the index number of questions whenever the list or search results are shown, so that the numbers begin at 1 instead of 0. Remember to subtract 1 from the user's input when viewing or deleting questions so that you reference the appropriate index of the `data` list.
  - *If you implement this, you can also change the "`input_int()`" function to accept a minimum value of 1, rather than 0.*

- If the text of a question is longer than 50 characters, truncate it so that it fits into 50 characters with "..." added to the end whenever you list questions or show search results.
  - *The "`shorten()`" function from the "`textwrap`" module can help with this.*
  - *The question should be stored in its entirety, and only truncated when listing/searching.*

```
Current questions:
  1) In which year was Halley's Comet last visible...
  2) Where were the 2016 Summer Olympics held?
```

- When searching for a question, show a "No results found" message if the search term is not found in the text of any of the questions.

- Enhance the search feature so that your code searches for the search term within the list of answers for each question, as well as within the text of the question.

- When viewing the details of a question, make sure that the wording is correct when printing the question's answer or answers, i.e. print "Answer:" if there is only one valid answer, or "Answers:" if there are multiple valid answers.

- Add a new menu option of "[b]reakdown" which shows the number of questions of each difficulty (1 to 5) currently in the `data` list.

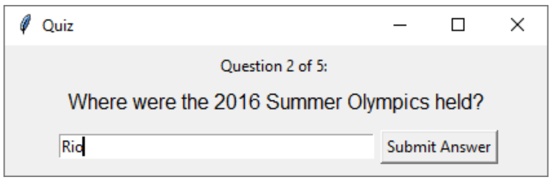
- Allow users to use the search, view and delete options more efficiently by allowing input of "s <search term>", "v <index number>" and "d <index number>". For example, instead of needing to type "s" and then "hobbit", the user could type "s year", and instead of needing to type "v" then "2", the user could type "v 2".
  - *This feature takes a reasonable amount of extra thought and effort to implement efficiently.*

Overview of "quiz.py"

"quiz.py" is a program with a Graphical User Interface (GUI), as covered in Module 9. It should be coded in an Object Oriented style, as covered in Module 8. Everything you need to know in order to develop this program is covered in the first 9 modules of the unit. This program should be developed after "admin.py".

The entirety of this program can be implemented in under 125 lines of code (although implementing optional additions may result in a longer program). *This number is not a limit or a goal – it is simply provided to prompt you to ask your tutor for advice if your program significantly exceeds it*. You must use the "`tkinter`" module and the "`tkinter.messagebox`" module to create the GUI.

This program uses the data from the "data.txt" file. Similar to the admin program, this program should load the data from the file *once only* - when the program begins. The program implements a simple quiz by displaying a series of randomly selected questions for the user to answer one at a time.

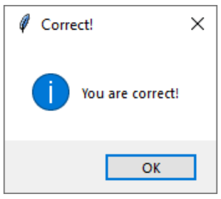

Quiz

Question 2 of 5:

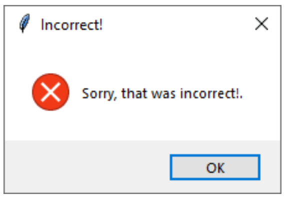
Where were the 2016 Summer Olympics held?

Rio

Once the user types their answer and presses the Submit Answer button, the program checks whether what they typed matches one of the answers in the list of correct answers for the question and displays an appropriate messagebox.

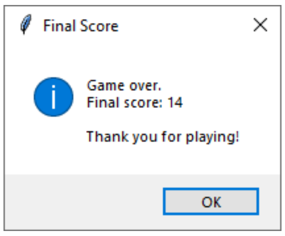

Correct!

You are correct!


Incorrect!

Sorry, that was incorrect!.

A different question is then displayed. After five questions have been answered, the program shows a final score and ends.


Final Score

Game over.
Final score: 14

Thank you for playing!

While running, the program must…

- Keep track of the current question number and display it, as seen in the above screenshot.
- Display a label containing "This is a hard one - good luck!" in blue if the difficulty of the question is 4 or 5. Do not display this for questions of lower difficulties.
- Keep track of the user's score. Points are awarded for correct answers at a rate of two times the question difficulty, e.g. correctly answering a difficulty 3 question is worth 6 points.
  - *The score is only shown at the end of the game (after answering five questions).*

The following pages detail how to implement the program.

Constructor of the GUI Class of "quiz.py"

The **constructor** (the "`__init__()`" method) of your GUI class must implement the following:

1. Create the main window of the program and give it a title of "Quiz".
   - *You are welcome to set other main window settings to make the program look/behave as desired.*

✓

2. Try to open the "data.txt" file in read mode and load the JSON data from the file into an attribute named "`self.data`", and then close the file.
   - *If any exceptions occur (due to the file not existing, or it not containing valid JSON data), show an error messagebox with a "Missing/Invalid file" message and call the "`.destroy()`" method on the main window to end the program. Include a "`return`" statement in the exception handler after destroying the main window to halt the constructor so that the program ends cleanly.*

✓

3. If the `self.data` list contains fewer than five questions, show an error messagebox with an "Insufficient number of questions" message, and end the program.
   - *See Requirement 2 for tips regarding how to end the program.*

✓

4. Create attributes named "`self.current_question`" and "`self.score`" attribute to keep track of which question the user is up to and their score. Set them both to 0.
   - *You can start `self.current_question` at 1 if you prefer – you will most likely use it to refer to an index number of a list, print the current question number, and check if the user has reached the final question, so regardless of whether you start at 0 or 1 there is likely to be a "+ 1" or "- 1" in your code.*

✓

5. Use `Label`, `Entry`, `Button` and `Frame` widgets from the "`tkinter`" module to implement the GUI depicted on the previous page. The layout is up to you as long as it functions as intended.
   - *You will save time if you design the GUI and determine exactly which widgets you will need and how to lay them out before you start writing the code.*
   - *See Reading 9.1 for information regarding various settings that can be applied to widgets to make them appear with the desired padding, colour, size, etc.*
   - ***Do not set the text for the labels that will contain the current question number and question text at this point.** They will be set in the "`show_question()`" method. See the Discussion Board for help.*

✓

6. Lastly, the constructor should end by calling the "`show_question()`" method to display the first question in the GUI, and then call "`tkinter.mainloop()`" to start the main loop.
   - *To call a method of the class, include "`self.`" at the start, e.g. "`self.show_question()`".*

That is all that the constructor requires. The following pages detail the methods mentioned above, and some optional additions. You are *not* required to submit pseudocode for "quiz.py".

Methods in the GUI class of "quiz.py"

This program requires you to define 2 methods to implement the functionality specified - "`show_question()`" and "`check_answer()`". These should be defined as part of the GUI class.

1. The "`show_question()`" method is responsible for putting the text of the current question in the GUI, as well as the question number and hard question messages. It is called at the end of the constructor and by the "`check_answer()`" method.
   - *To make the program more user-friendly, the method should clear the `Entry` box of any previous text, and set it to be the focus of the window so that the user can immediately type their answer.*
   - *The "`configure()`" method (see Reading 9.1) can be used on a widget to change the text it displays. Alternatively, you can use a `StringVar` to control the text of a widget (see Lecture 9).*
   - *There are ways to ensure that each question displayed is different. Consider either using "`random.sample()`" to select five random questions from `self.data` during the constructor and working your way through them, or deleting questions from `self.data` after you ask them.*
   - *You can use the "`.pack()`" and "`.pack_forget()`" methods to display or hide the hard question message depending upon the difficulty of the current question. When packing a widget, you can specify that it should be packed before or after another widget, e.g.*

     ```
     self.hard_label.pack(before=self.question_label)
     ```

2. The "`check_answer()`" method is called when the user clicks the "Submit Answer" button. It is responsible for checking whether the answer that the user typed matches one of the correct answers for the question, and showing an appropriate messagebox. It also uses the `self.current_question` attribute to determine whether to show the "Game Over" messagebox and final score, or to call `self.show_question()` to display the next question.
   - *The code of this method should involve adding 1 to `self.current_question` at some point.*
   - *The "`.get()`" method of an Entry widget will return a string of what has been typed into it.*
   - *An "`in`" comparison can be used to determine whether the text in the `Entry` widget matches any of the strings in the list of valid answers of the current question's dictionary.*
   - *Remember to add two times the question's difficulty to `self.score` if the user answered correctly.*
   - *Ensure that the comparison between the user's answer and the correct answers is not case-sensitive.*
   - *After the fifth question has been answered, show a "Game Over" messagebox that includes the user's final score and call the "`.destroy()`" method on the main window to end the program.*

These methods are all that are required to implement the functionality of the program, but you may write/use additional methods if you feel they improve your program.

GUIs and OOP are massive topics that we do not explore in depth in the unit. As such, this program is likely to involve more research, experimentation and combining/adapting of examples from the than the main program. This is expected, and much of what this program tests is your ability to rapidly produce a working product despite having limited familiarity or experience.

Do not be alarmed/concerned if you find this program confusing or difficult for those reasons, and be sure to check the discussion board regularly for tips and examples. The GUI program is worth fewer marks than the main program of this assignment.

Optional Additions and Enhancements for "quiz.py"

Below are some suggestions for minor additions and enhancements that you can make to the program to further test and demonstrate your programming ability. They are *not required* and you can earn full marks in the assignment without implementing them.

- Display the user's current score somewhere in the GUI, updating it after every question. You could even track how many questions they have answered correctly and what the maximum possible score is, and display these pieces of information in the final messagebox.

- Prevent the user from submitting an answer without having typed anything. This can be achieved by checking if there is nothing in the `Entry` widget at the start of the `check_answer()` method, or disabling the "Submit Answer" button when the `Entry` widget is empty.

- Make it so that the user can press Enter to submit their answer, instead of requiring them to click the Submit Answer button. Pressing Enter should call `check_answer()` as normal.

- Implement a countdown timer of 5 seconds that is shown on the GUI – if the user does not answer within 5 seconds, show a messagebox saying that they ran out of time and continue to the next question by calling "`show_question()`". Timers are covered in Reading 9.2.
  - *Remember to add 1 to `self.current_question` if the user runs out of time.*
  - *For an added challenge, make the timer value appear in red when the timer is two or less.*

- Keep track of how often a question is answered correctly and incorrectly, and record this information in the text file. This involves additions to both programs:
  - *When adding a question in the admin program, include two extra key-value pairs in the question's dictionary – "correct_count" and "incorrect_count", both set to 0.*
  - *When viewing a question in the admin program, show these values alongside the other details.*
  - *In the GUI program, add code to the `check_answer()` method that adds 1 to the appropriate key of the current question's dictionary and then writes the entire `self.data` list to the text file in JSON format, just like the `save_data()` function of the admin program.*

Submission of Deliverables

It is recommended that you send your work to your tutor for feedback at least once prior to submitting it. Once your assignment is complete, submit both the **pseudocode for "admin.py"** (".pdf" file) and the **source code for "admin.py" and "quiz.py" code** (".py" files) to the appropriate location in the Assessments area of Blackboard. *Zipping the files is not required.* An assignment cover sheet is not required, but **include your name and student number at the top of all files (not just in the filenames)**.

Academic Integrity and Misconduct

The **entirety of your assignment must be your own work** (unless otherwise referenced) and produced for the current instance of the unit. Any use of unreferenced content you did not create constitutes plagiarism, and is deemed an act of academic misconduct. All assignments will be submitted to plagiarism checking software which includes previous copies of the assignment, and the work submitted by all other students in the unit.

Remember that this is an **individual** assignment. *Never give anyone any part of your assignment – even after the due date or after results have been released. Do not work together with other students on individual assignments – you can help someone by explaining a concept or directing them to the relevant resources, but doing any part of the assignment for them or alongside them, or showing them your work is inappropriate.* An unacceptable level of cooperation between students on an assignment is collusion, and is deemed an act of academic misconduct. If you are uncertain about plagiarism, collusion or referencing, simply contact your tutor, lecturer or unit coordinator.

You may be asked to **explain and demonstrate your understanding** of the work you have submitted. Your submission should accurately reflect your understanding and ability to apply the unit content.

Marking Key

Criteria	Marks
Pseudocode These marks are awarded for submitting pseudocode which suitably represents the design of your admin program. Pseudocode will be assessed on the basis of whether it clearly describes the steps of the program in English, and whether the program is well structured.	**5**
Functionality These marks are awarded for submitting source code that implements the requirements specified in this brief, in Python 3. Code which is not functional or contains syntax errors will lose marks, as will failing to implement requirements as specified.	**15** *Main Program: 10* *GUI Program: 5*
Code Quality These marks are awarded for submitting well-written source code that is efficient, well-formatted and demonstrates a solid understanding of the concepts involved. This includes appropriate use of commenting and adhering to best practise.	**10** *Main Program: 6* *GUI Program: 4*

Total:	**30**

See the "Rubric and Marking Criteria" document provided with this brief for further details!