Programming Principles (CSI6208)

Assignment 2: Individual programming project (“Fast-Food Quiz” Program)
Assignment Marks: Marked out of 30, worth 30% of unit
Due Date: 1 November 2021, 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 add, list, search, view and delete nutritional information about fast food menu items, and stores the data in a text file. Develop this program before “foodquiz.py”.
- “**foodquiz.py**”, a GUI program that uses the data in the text file to quiz the user about the nutritional information of the fast-food menu items. 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 "foodquiz.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 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 230 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 to develop this program is covered in the first 7 modules of the unit. This program should be developed before "foodquiz.py".

This program allows the user to manage a collection of fast-food item nutritional information that is 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 of this.

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

```json
JSON
[
    {
        "name": "Big Mac",
        "energy": 2360,
        "fat": 31,
        "protein": 27,
        "carbohydrates": 42,
        "sugar": 7,
        "sodium": 1020
    },
    {
        "name": "Whopper",
        "energy": 2700,
        "fat": 39,
        "protein": 26,
        "carbohydrates": 48,
        "sugar": 8,
        "sodium": 844
    }
]
```

This example data contains the details of two fast-food items in a *list*. Each of those items is a *dictionary* consisting of 7 items which have keys of "name", "energy", "fat", "protein", "carbohydrates", "sugar" and "sodium".

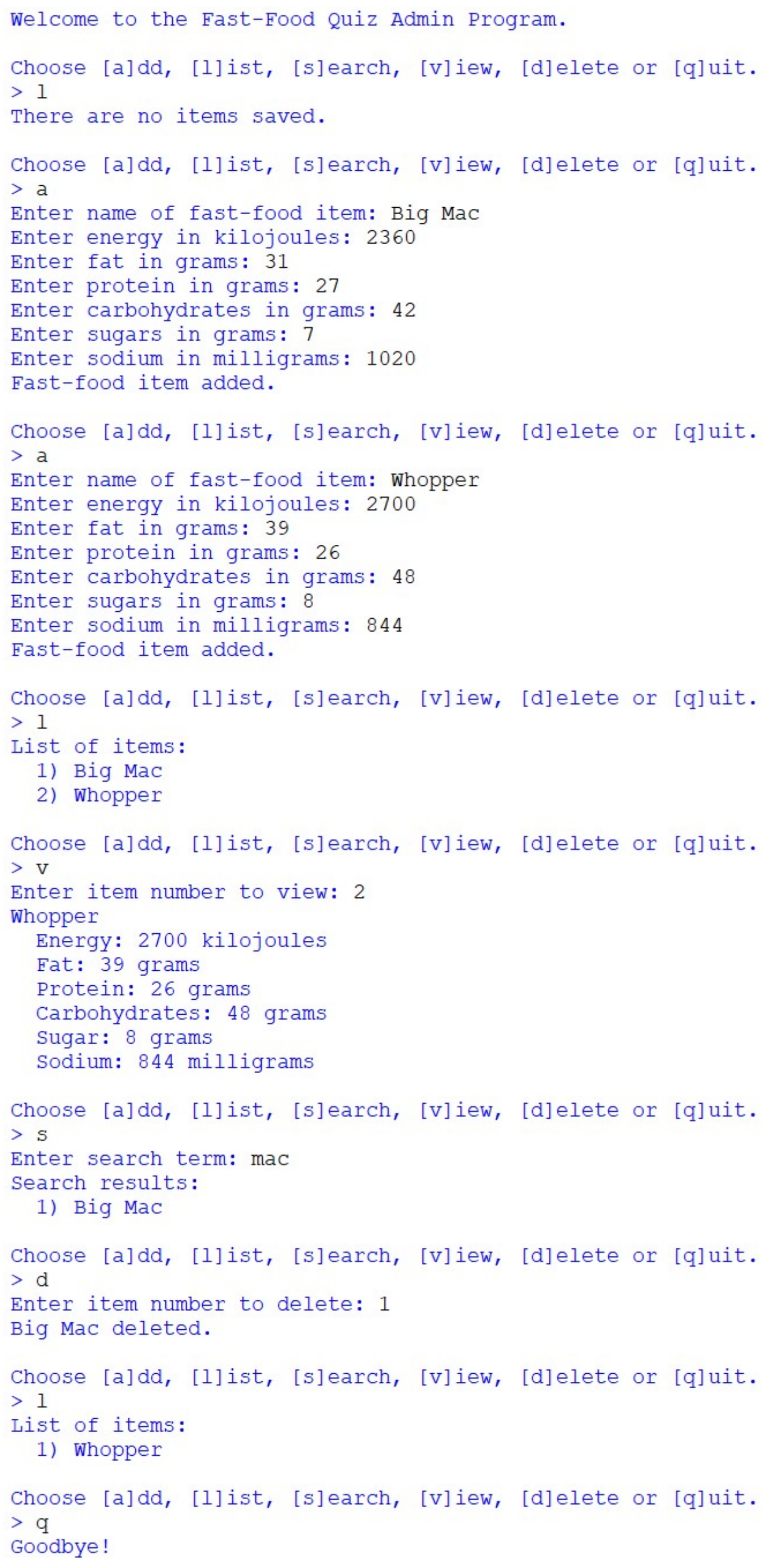
If this file was to be read into a Python variable named `data`, then "`data[0]`" would refer to the dictionary containing the first menu item (Big Mac), and "`data[0]['fat']`" would refer to the integer of 31.

Understanding the structure of this data and how to interact with 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 an example screenshot of the program being run:



```
Welcome to the Fast-Food Quiz Admin Program.

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

In this example there was no previously saved data to load, so we start without any items.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> a
Enter name of fast-food item: Big Mac
Enter energy in kilojoules: 2360
Enter fat in grams: 31
Enter protein in grams: 27
Enter carbohydrates in grams: 42
Enter sugars in grams: 7
Enter sodium in milligrams: 1020
Fast-food item added.
```

We enter “a” to add a new item, entering its name and nutritional details when prompted.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> a
Enter name of fast-food item: Whopper
Enter energy in kilojoules: 2700
Enter fat in grams: 39
Enter protein in grams: 26
Enter carbohydrates in grams: 48
Enter sugars in grams: 8
Enter sodium in milligrams: 844
Fast-food item added.
```

We enter “a” again to add a second item.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> l
List of items:
  1) Big Mac
  2) Whopper
```

Now when we enter “l” to list, we can see the two items. They have also been saved to the text file in JSON format.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> v
Enter item number to view: 2
Whopper
  Energy: 2700 kilojoules
  Fat: 39 grams
  Protein: 26 grams
  Carbohydrates: 48 grams
  Sugar: 8 grams
  Sodium: 844 milligrams
```

We enter “v” to view an item, specifying the number of the item we want to view.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> s
Enter search term: mac
Search results:
  1) Big Mac
```

Entering “s” allows us to search the items, only listing the items that contain the search term.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> d
Enter item number to delete: 1
Big Mac deleted.
```

We enter “d” to delete an item, specifying the number of the item we want to delete.

```
Choose [a]dd, [l]ist, [s]earch, [v]iew, [d]elete or [q]uit.
> l
List of items:
  1) Whopper
```

Listing the items confirms that it has been deleted.

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

Entering “q” ends the program by breaking out of the menu loop.

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 all 7 details of a fast-food item, beginning with the name. Place the details into a new dictionary with the structure shown earlier, 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 the user for the name of the item, to ensure that they are re-prompted until they enter something other than whitespace.*
   - *Use your "`input_int()`" function (detailed below) when prompting for the nutritional details of the item, to ensure that the user is re-prompted until they enter an integer.*
   - *Your prompts for input should specify that energy is measured in kilojoules, while fat, protein carbohydrates and sugars are measured in grams, and sodium is measured in milligrams.*
   - *Once the dictionary for the new item 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 the names of all items in the `data` list, preceded by their index number plus 1 (so that the list starts from 1 rather than 0).
   - *If the `data` list is empty, show a "No items 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 ensure that you have access to variables containing the index number and dictionary of each item 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 items that contain the search term in their name. Include the index number plus 1 of the item next to each result. Display search results in the same way that you display items when listing them.
   - *If the `data` list is empty, show a "No items saved" message instead of prompting for a search term.*
   - *Use your "`input_something()`" function (detailed below) when prompting the user for a search term, to ensure that they are 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 list items that contain the search term (use the "`in`" operator – see Lecture 3).*
   - *Ensure that the searching is not case-sensitive, e.g. "mac" should find an item named "Big Mac".*
   - *If the search term is not found in the name of any items, show a "No results found" message.*

6. If the user enters "`v`" (view), prompt them for an index number and then print the corresponding item's name and all of its nutritional information, including the units of measurement.
   - *If the `data` list is empty, show a "No items 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.*
   - *Remember: Index numbers shown to/entered by users start from 1, but the `data` list starts from 0.*
   - *Print an "Invalid index number" message if the index number entered does not exist in the `data` list.*
   - *When viewing an item, also show the number of calories (rounded up to the nearest whole number) in parentheses after the energy in kilojoules. 1 kilojoule is 0.239 calories.*

7. If the user enters "`d`" (delete), prompt them for an index number and then delete the corresponding item's dictionary from the `data` list, then print a confirmation message.
   - *If the `data` list is empty, show a "No items 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.*
   - *Remember: Index numbers shown to/entered by users start from 1, but the `data` list starts from 0.*
   - *Print an "Invalid index number" message if the index number entered does not exist in the `data` list.*
   - *Include the name of the item in the confirmation message, e.g. "Big Mac deleted."*
   - *Once the item 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` – a string containing the message to display before waiting for input. The function should repeatedly re-prompt the user (using the `prompt` parameter) for input until they enter an integer that is greater than or equal to 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` – a string containing the message to display before waiting for input. 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.

- When adding an item, check if there is already an item with the same name (case-insensitive) in the `data` list after the user enters the name of the new item. If you find a match, inform the user and tell them its index number (plus 1) before returning them to the main menu.

- When adding an item, prompt for the name of the fast-food chain (e.g. "McDonald's") as well as the name of the item, storing it with a key of "chain" in the dictionary. Be sure to show the name of the chain when listing, searching and viewing, e.g. "Big Mac (McDonald's)" .
  - *Also include the name of the chain when displaying item names in "foodquiz.py"*
  - *If implementing this feature as well as the previous one, the program should check whether two items from the same fast-food chain have the same name.*

- When viewing an item, also show the percentage of average daily intake that each nutritional component represents, based on an 8700 kilojoule diet.
  - *According to the Australia New Zealand Food Standards Code, the daily intake values are 70g of fat, 50g of protein, 310g of carbohydrates, 90g of sugar and 2300mg of sodium.*

- Add a new menu option of "[b]reakdown" which shows the number of items, and the average value for each of the nutritional components, rounded to 2 decimal places.
  - *You could expand upon this by also including the name (and amount) of the item(s) with the highest value in each of the nutritional components, or other interesting statistics.*

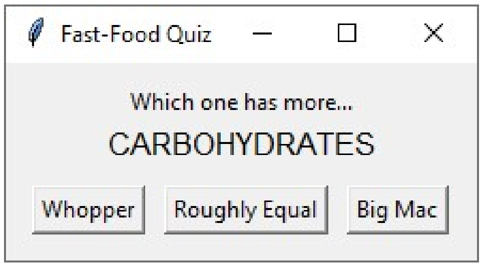
- Allow users to use the search, view and delete options more efficiently by allowing input of "s <search term>", "v <index number>" and "d <index number>". For example, instead of needing to type "s" and then "mac", the user could type "s mac", and instead of needing to type "v" then "2", the user could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

Overview of "foodquiz.py"

"foodquiz.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 170 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 randomly selecting two different fast-food items from the data and one nutritional component (energy, fat, protein, carbohydrates, sugar or sodium), then asking the user which of the two items contains more of that nutritional component.

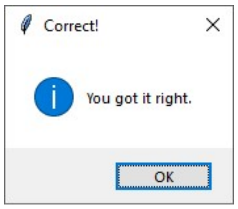


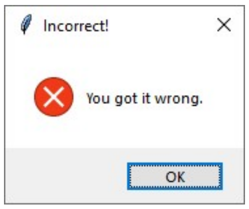
Question is asked using the currently selected nutritional component, chosen at random.

Left and right buttons contain the currently selected item names, middle button is always the same.

*Your program does not need to look the same as this.
Adjust the layout and formatting as desired.*

When the user clicks one of the buttons, the program should determine whether their answer is correct and show an appropriate messagebox:





The program continues like this until the user closes it – selecting two random menu items and a random nutritional component each time. It does *not* need to keep track of a "score" of any kind.

The following pages detail how to implement the program.

Constructor of the GUI Class of "foodquiz.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 "Fast-Food 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.*
   - *Once you've loaded the data, check that `self.data` contains at least 2 items (the minimum needed for the quiz). If it contains fewer than 2 items, end the program in the same way as described above.*

3. Create an attribute named "`self.components`" and set it to a list containing strings of "energy", "fat", "protein", "carbohydrates", "sugar" and "sodium".
   - *This will be used in the "`show_question()`" method to randomly select a nutritional component.*
   - *Make sure that you spell the words exactly the same as the keys of the dictionaries in `self.data`.*

4. Use `Label`, `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 specified.
   - *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 buttons that will contain item names at this point.** They will be set in the "`show_question()`" method. See the Discussion Board for help.*
   - *All three of the buttons call the "`check_answer()`" method when clicked, but each need to pass it a different parameter value – a string of "left", "middle" or "right" (to know which button was clicked). This involves setting the "`command`" of buttons in a special way. Here is some sample code:*

```
self.left_button = tkinter.Button(..., command=lambda: self.check_answer('left'))
self.middle_button = tkinter.Button(..., command=lambda: self.check_answer('middle'))
self.right_button = tkinter.Button(..., command=lambda: self.check_answer('right'))
```

5. 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 other methods required, and some optional additions. You are *not* required to submit pseudocode for "foodquiz.py".

Methods in the GUI class of "foodquiz.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 randomly selecting two fast-food items and a nutritional component and showing them in the GUI. Simply select two random items from `self.data` and one random nutritional component from `self.components`, then use them to set the text for appropriate label and buttons in your GUI.
   - *This method is called at the end of the constructor to select and show the first question, and at the end of the "`check_answer()`" method to select and show the next question.*
   - *Use "`random.sample()`" to ensure the two items you select from `self.data` are different. "`random.choice()`" can be used to select a random nutritional component.*
   - *Store the selected items and the selected nutritional component as attributes (i.e. include "`self.`" at the start of the variable name) so that you can refer to them in the "`check_answer()`" method.*

2. The "`check_answer()`" method is called when the user clicks one of the buttons. It is responsible for determining whether the user answered correctly and showing an appropriate messagebox, before calling the "`show_question()`" method to proceed to the next question.
   - *This method receives a parameter named "`choice`" that contains a string of "left", "middle" or "right" depending on which button was clicked.*
   - *If `choice` is "left" or "right", simply compare the selected nutritional component of the selected items to determine whether the user was correct.*
   - *If `choice` is "middle", it is correct if the smaller value is at least 90% of the higher value, i.e. if it is within 10% of it. For example, if the two nutritional values being compared had values of 90 and 100, then the middle button would be correct, but if they had values of 89 and 100 then it would be incorrect.*

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.

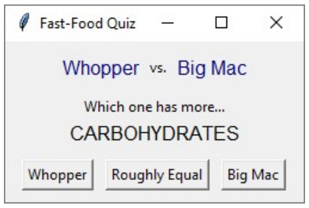
GUIs and OOP are massive topics that we do not explore in depth in the unit. As such, this program is likely to involve more research, experimentation and combining/adapting of examples from the unit content than the main program. This is expected, and much of what this program tests is your ability 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 "foodquiz.py"

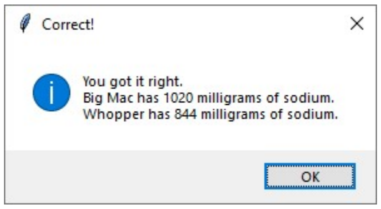
Below are some suggestions for minor additions and enhancements that you can make to the program to further test and demonstrate your programming ability. They are *not required* and you can earn full marks in the assignment without implementing them.

- Prominently display the names of the fast-food items being compared in the GUI, using a differently sized or coloured text to make it stand out.
  - *Consider using "`StringVar`" objects to control the text of widgets in a more streamlined way.*



- Use attributes to keep track of the number of questions that have been asked and how many questions have been answered correctly, and use it to show the users "score" in the GUI.

- Make the messagebox shown after answering a question more informative by including the relevant nutritional information of both items.



You got it right.
Big Mac has 1020 milligrams of sodium.
Whopper has 844 milligrams of sodium.

- Implement a countdown timer of 6 seconds that is shown on the GUI. If the user does not answer within 6 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.

Submission of Deliverables

It is recommended that you send your work to your tutor for feedback at least once while working on it. Once your assignment is complete, submit both the **pseudocode for "admin.py"** (".pdf" file) and the **source code for "admin.py" and "foodquiz.py"** (".py" files) to the appropriate location in the Assessments area of Blackboard. *Please do <u>not</u> zip the files.* 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: 5* *GUI Program: 5*

Total:	**30**

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