UNIVERSITY OF CANBERRA

INTRODUCTION TO INFORMATION TECHNOLOGY (4478/8936)

ASSIGNMENT 2: Programming with Python

Instructions

This assignment will test your knowledge and skills in writing an application software for a task, understanding the business rules of a particular problem, coding these in a computer program, developing a graphical user interface, and reading data from a text file from a disk. As such, the assignment requires you to integrate and synthesise what you have learnt so far in this unit, in order to design and create a proper working solution.

The context of this programming assignment is free for you to choose. Think about any problem you may be facing that is suitable to be tackled from an IT-programming perspective. I have chosen a case study to show you how would you approach the problem from the very beginning, the compulsory elements need to be included in your solution, and the final (and hopefully nice and useful) product.

This assessment has three stages:

- **Stage 1**: A simple Python program using an interactive text-based menu (no GUI)
- **Stage 2**: The same as stage 1, but wrapped in a graphical user interface (GUI)
- **Stage 3**: The input comes from a text file – read only once, then information stored in a suitable data structure.

The following case study shows the integration of these stages. Please bear in mind that this is an example only. Your solution does not need to be exactly the same as this one. However, be sure that you include every single element described at the end of each proposed stage. Allocated marks are shown in **[bold red]**, whereas instructions/comments are shown in **[bold blue]**.

Planetary Exploration App

Credits: NASA Solar System Exploration @solarsystem.nasa.gov

I Background [MARKS: 3 marks for a suitable background about the context of your problem]

Our astronauts are about to leave for a trip to outer space. They may be visiting the Moon, Mercury, Venus, Mars or one of the larger moons of Jupiter (IO, Europa, Ganymede or Callisto). Each astronaut has a mass allowance in kg for items that they can take along the trip, including tools and personal items. The amount they are allowed depends on their job description as follows:

Flight Crew are allowed a total of 100kg, and Mission Specialists are allowed a total of 150kg.

Each of the solar system bodies (celestial bodies) has less gravity than the earth. For example, if a mass of 100kg on earth weights 100kg, on the Moon will weigh the equivalent of 16.6kg.

II Business Rules

a) The Problem at Hand [MARKS: 3 marks for a clear and specific description of the problem to be solved]

We need to calculate how much mass each astronaut has left over for personal items, the total available mass and the average available personal mass allowance across all six astronauts. We need to also calculate the weight of the average available personal mass allowance on the celestial body the astronauts are travelling to. We assume there are three crew astronauts and three mission specialists. **[Be sure to clearly state your assumptions!!!]**

b) Describing Inputs and Outputs [MARKS: 3 marks for a clear and specific description of inputs and outputs]

We need a list that contains the names of the celestial bodies that our astronauts may travel to and their Mass multipliers. This list must be stored using an appropriate data structure. Do not hard-code this table. **[This list is part of our inputs]**.

Celestial body	Mass Multiplier
Mercury	0.378
Venus	0.907
Moon	0.166
Mars	0.377
Io	0.1835

Europa	0.1335
Ganymede	0.1448
Callisto	0.1264

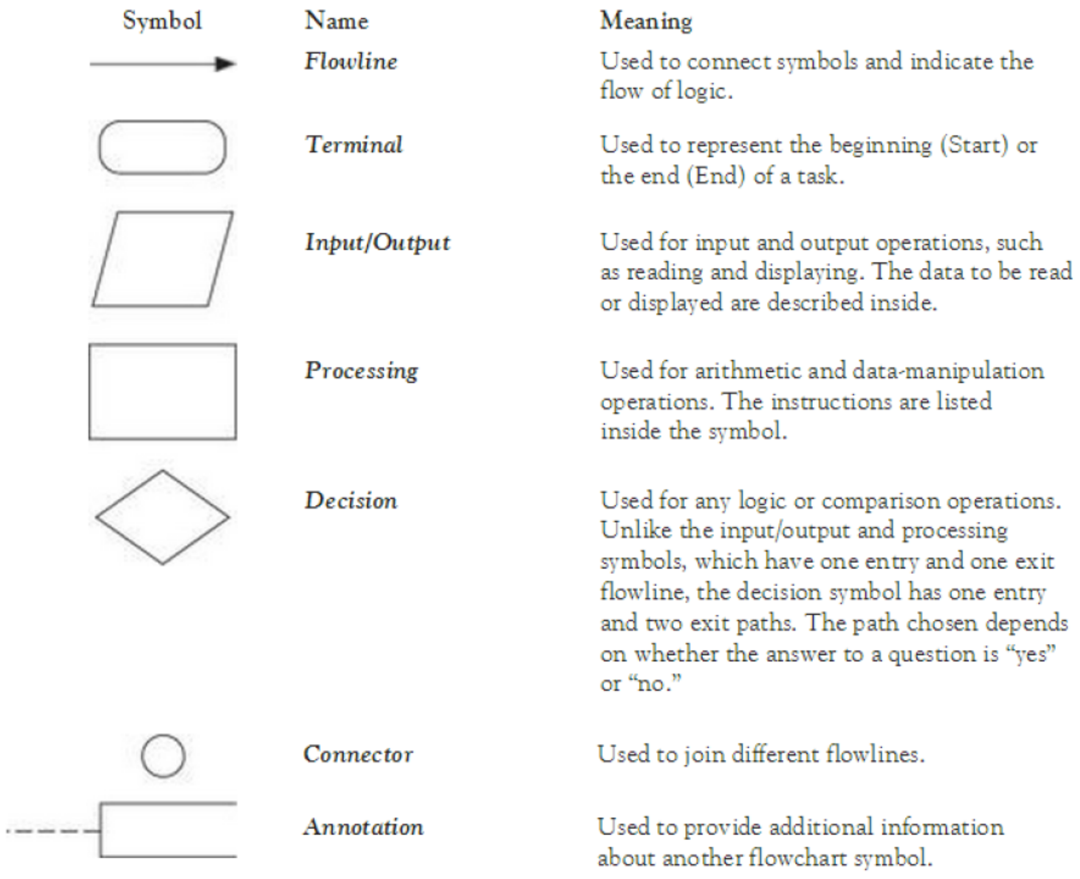
Also, the user needs to decide which celestial body the astronauts are travelling to, alongside their mass **[Second set of inputs]**.

The code will display a list of celestial bodies that are possible destinations along with their mass multipliers. Our code needs to also display the weight allowances for the two different types of astronaut **[All of these are our outputs]**.

III Develop a "Hand" Calculation [MARKS: 3 marks for an example of manual calculations, 5 marks for the flowchart, and 3 marks for the pseudocode.]

We will create (pure creation, that it is!!!) a *flowchart* to start with.

A flowchart consists of special symbols connected by arrows. Within each symbol is a phrase presenting the activity at that step. The shape of the symbol indicates the type of operation that is to occur. The arrows connecting the symbols, called *flowlines*, show the progression in which the steps take place. Below is the ANSI standard for flowchart symbols (American National Standards Institute)[1]:



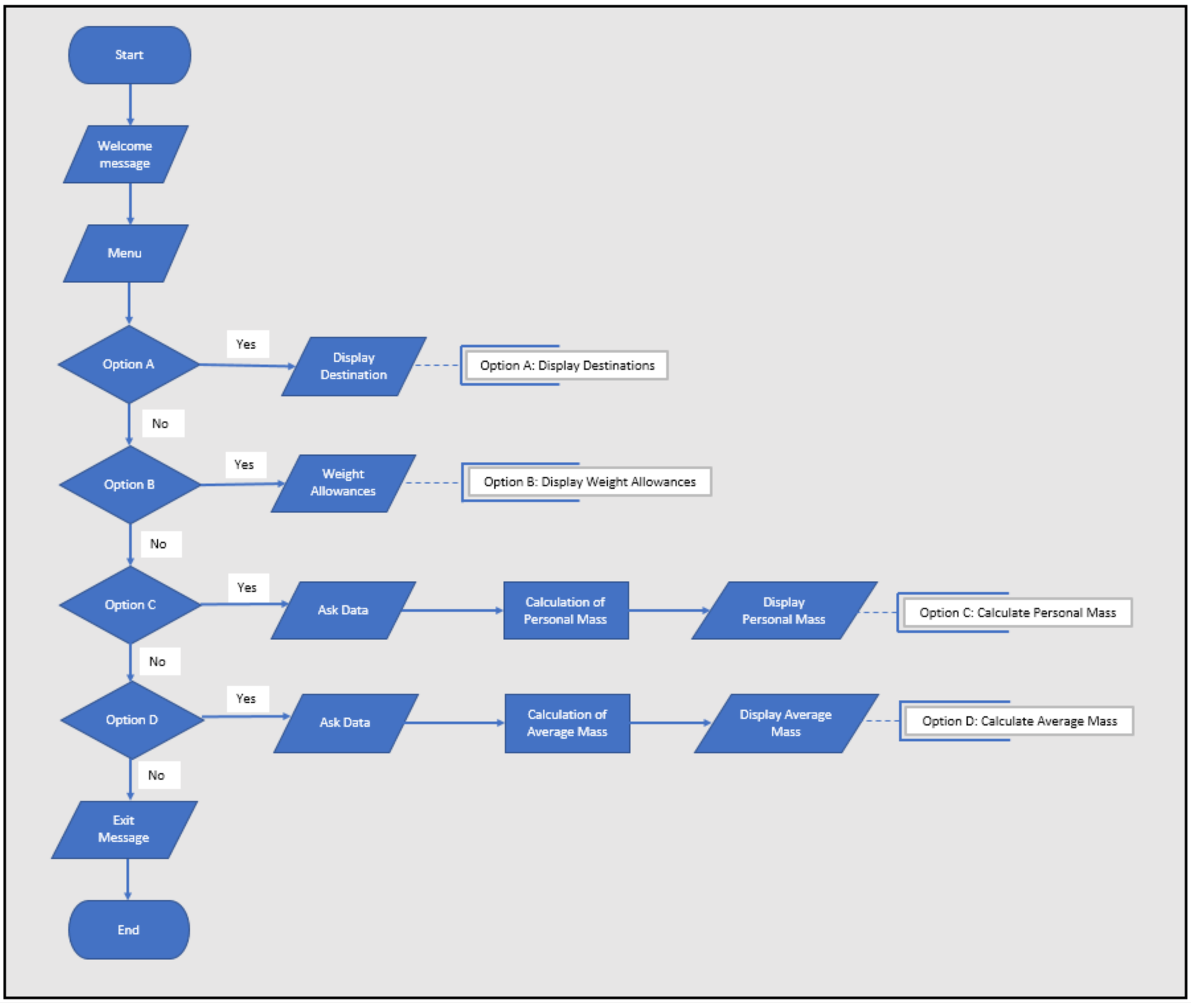
Symbol	Name	Meaning
→	*Flowline*	Used to connect symbols and indicate the flow of logic.
▢	*Terminal*	Used to represent the beginning (Start) or the end (End) of a task.
▱	*Input/Output*	Used for input and output operations, such as reading and displaying. The data to be read or displayed are described inside.
□	*Processing*	Used for arithmetic and data-manipulation operations. The instructions are listed inside the symbol.
◇	*Decision*	Used for any logic or comparison operations. Unlike the input/output and processing symbols, which have one entry and one exit flowline, the decision symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is "yes" or "no."
○	*Connector*	Used to join different flowlines.
- - - [	*Annotation*	Used to provide additional information about another flowchart symbol.

Once finishing with our flowchart, we'll develop a *"pseudocode"*, which is an abbreviated version of actual computer code (hence, pseudocode). The geometric symbols used in flowcharts are replaced

[1] "An Introduction to Programming Using Python", Schneider, D.I. Pearson, 2016

by English-like statements that outline the process. As a result, pseudocode looks more like computer code than does a flowchart. Pseudocode allows the programmer to focus on the steps required to solve the problem rather than on how to use the computer language.

Ok then; let's start with the flowchart according to the problem at hand and input/output (I/O) description:

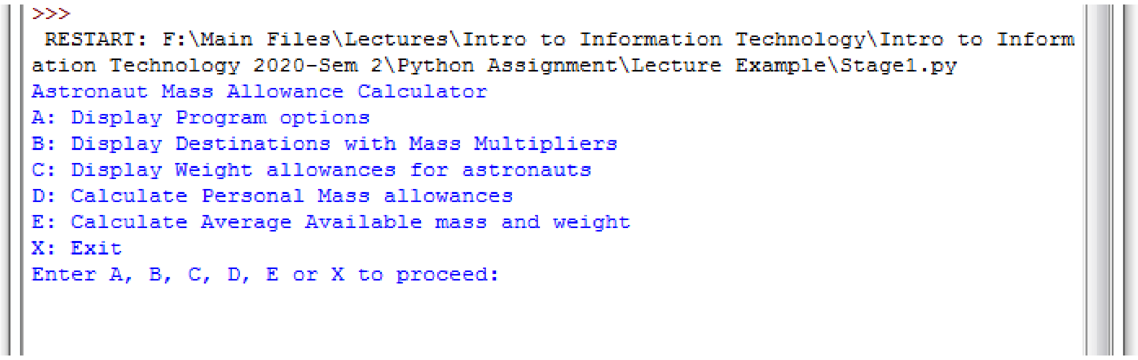

Start
Welcome message
Menu
Option A
Yes
Display Destination
Option A: Display Destinations
No
Option B
Yes
Weight Allowances
Option B: Display Weight Allowances
No
Option C
Yes
Ask Data
Calculation of Personal Mass
Display Personal Mass
Option C: Calculate Personal Mass
No
Option D
Yes
Ask Data
Calculation of Average Mass
Display Average Mass
Option D: Calculate Average Mass
No
Exit Message
End

The above flowchart clearly shows the sequence of tasks that our program will be undertaking. We now translate this flowchart into a pseudocode:

```
Program: Calculate Astronauts' Mass Allowance
start
Print welcome message on the screen
Print (display) the menu on the screen
If [option A] then
        print destination on the screen.
        Else [option B]
                print weight allowances.
        Else [option C]
                Input destination
                input mass and personal items each crew member.
                Calculate personal allowable mass for each crew member as:
                Formula here [use functions for your calculations]
                Print personal mass for each crew member.
        Else [option D]
                print weight allowances.
                Input destination
                input mass and personal items each crew member.
                Calculate average allowable mass for crew members as:
                Formula here [use functions for your calculations]
                Print average mass for crew members.
Print exit message
end
```

A running example will be shown during the lectures.

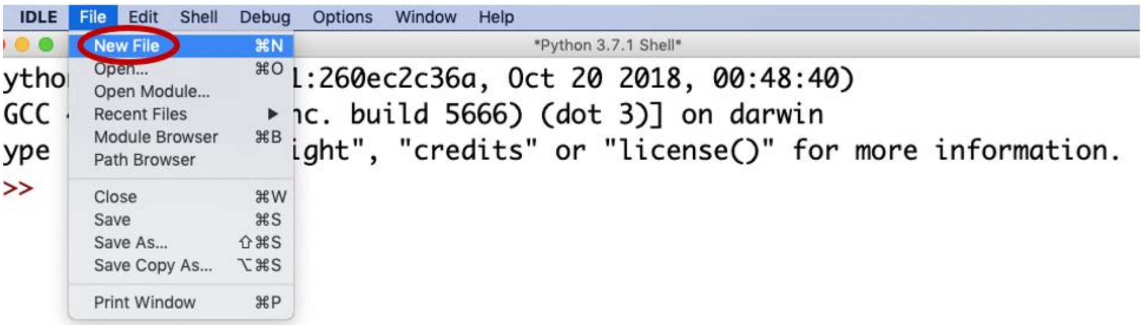
Stage 1. Developing a Command-Line menu.



```
>>>
 RESTART: F:\Main Files\Lectures\Intro to Information Technology\Intro to Inform
ation Technology 2020-Sem 2\Python Assignment\Lecture Example\Stage1.py
Astronaut Mass Allowance Calculator
A: Display Program options
B: Display Destinations with Mass Multipliers
C: Display Weight allowances for astronauts
D: Calculate Personal Mass allowances
E: Calculate Average Available mass and weight
X: Exit
Enter A, B, C, D, E or X to proceed:
```

Step-by-Step Guide for Stage 1 (using the case study as an example)

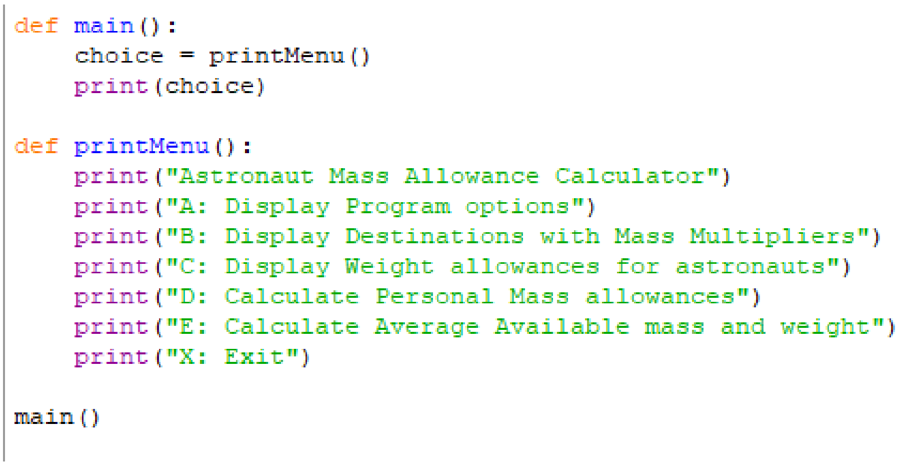
1. Think about your strategy on how you will display the options for user to choose from according to your flowchart. How you will calculate the available personal item mass for each astronaut and how to calculate the average available mass and average available weight. Think about writing simple functions for the repetitive calculations.
2. Create a new Python file, you may call it ***ProgAsgStage1***



```
ytho  l:260ec2c36a, Oct 20 2018, 00:48:40)
GCC   nc. build 5666) (dot 3)] on darwin
ype   ight", "credits" or "license()" for more information.
>>
```

3. In the Stage1 class (file `ProgAsgStage1.py`), you may put all code for the user interaction and the calculation into the main method. (You might still need to define global variables and constants outside the main method at the top of the editor window). You might like to use nested lists to hold the name and mass multiplier of the celestial bodies.

4. You will need to write a function to print a menu to the user. One possible example is:



```
def main():
    choice = printMenu()
    print(choice)

def printMenu():
    print("Astronaut Mass Allowance Calculator")
    print("A: Display Program options")
    print("B: Display Destinations with Mass Multipliers")
    print("C: Display Weight allowances for astronauts")
    print("D: Calculate Personal Mass allowances")
    print("E: Calculate Average Available mass and weight")
    print("X: Exit")

main()
```

5. Now add the code that implements your strategy to display the different destinations with their mass multipliers, **TEST**.
6. Add the code to display the weight allowances for the two different types of astronaut, **TEST.**
7. Add the code to calculate the personal mass allowances of each of the six astronauts along with the total available mass, **TEST**.
8. Add the code to calculate the average available mass allowance and weight on destination, **TEST.**
9. Finally, add print statements to print the output to the user.
10. Test your implementation with the test cases mentioned below (and additionally your own).

For steps 7 & 8 work out your tests ***before*** *you do any coding*

Examples of Test cases:

Test 1

Destination Selected	The Moon
Entered tool weights for crew	100, 100, 100
Entered tool weights for mission specialists	150, 150, 150
Available mass for astronauts:	0,0,0,0,0,0
Total Available Mass	0 kg
Average available mass:	0.0 kg
Average available weight on destination	0.0 kg

Test 2

Destination Selected	Mars
Entered tool weights for crew	90, 90, 90
Entered tool weights for mission specialists	140, 140, 140
Available mass for astronauts:	10,10,10,10,10,10
Total Available Mass	60 kg
Average available weight	10 kg
Average available weight on destination	3.77 kg

Test 3

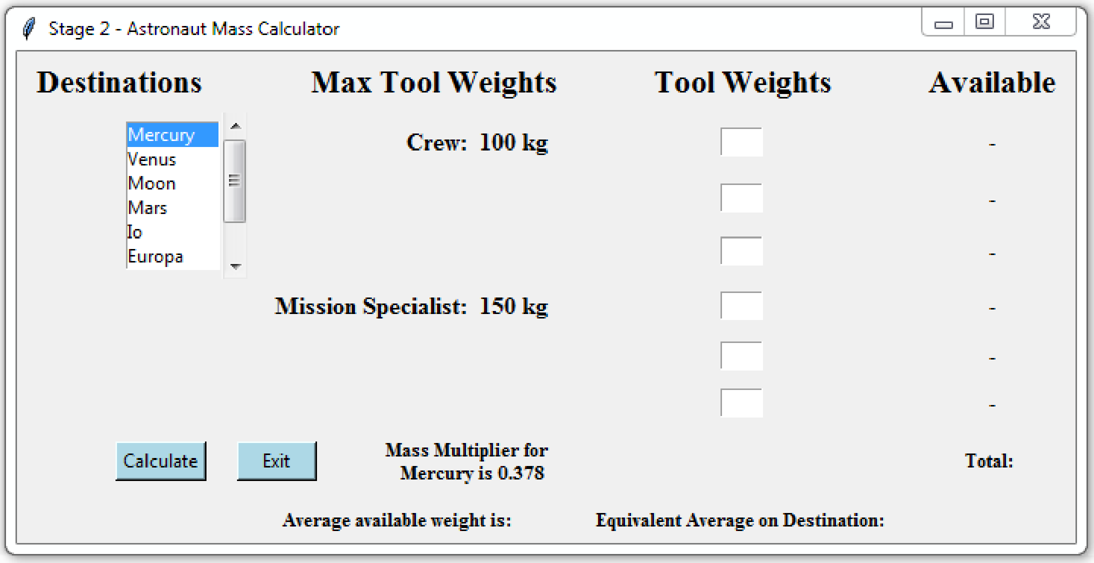
Destination Selected	Venus
Entered tool weights for crew	99, 98, 101
Entered tool weights for mission specialists	150, 149, 151
Available mass for astronauts:	1,2,-1,0,1,-1
Total Available Mass	2 kg
Average available mass:	0.333333 kg
Average available weight on destination	0.302333 kg

How the marks for Stage 1 are allocated: [Total Marks: 40]

Marks are allocated as follows:

- **Constants vs literals**. Using constants is important for the ease of maintenance. Not using constants will result in lower marks. For example, in this case study we consider constants for the mass multipliers for each celestial body and the upper limit for each type of astronaut. **[3 marks]**
- **Good names for your variables and constants. Check style against the Python style guide attached below. [3 marks]**
- The program must have **at least one mathematical calculation**. **[3 marks]**
- You **must use functions for each task in your code**, for example a *printMenu* function to print a menu and *calculateAverageMass* to calculate the average available mass. **[3 marks]**
- Your program **must have a *main()* function** to appropriately direct the program. **[2 marks]**
- Program **code layout**. Separate blocks of code by a blank line. **[2 marks]**
- Program is written using **well-defined functions including a *main()* function** to direct the program. **[4 marks]**
- **Use of comments.** A comment is not an essay. Comments are important for the maintenance of a program and should contain enough details but keep them concise. Do not comment every single line. **[2marks]**
- The program must have a **short introduction**. **Check style against the Python style guide attached below. [3 marks]**
- The program **must work correctly**. That is:
    - A text-based menu is displayed when first running your program. **[5 marks]**
    - The menu works properly. **[3 marks]**
    - Calculations are correctly performed. **[3 marks]**
    - The output on the shell is properly formatted.
- **Business rules are met**. That is, your code solves the problem you are proposing (or at least contributes to its solution). **[4 marks]**

Stage 2. Developing a Graphical User Interface (GUI)


Stage 2 - Astronaut Mass Calculator
Destinations
Max Tool Weights
Tool Weights
Available
Mercury
Venus
Moon
Mars
Io
Europa
Crew: 100 kg
Mission Specialist: 150 kg
-
-
-
-
-
-
Calculate
Exit
Mass Multiplier for Mercury is 0.378
Total:
Average available weight is:
Equivalent Average on Destination:

In *stage1*, the user input and output is not very satisfactory from the human computer interaction and usability point of view, your task in this stage is to design and implement a Graphical User interface (GUI) using buttons and labels that provides an easy to use interface.

Use the same code for the calculations from Stage 1. Reusing code is an important part of software development.

You have a great degree of freedom in what GUI elements you choose and how you would like to design the layout of your GUI. What matters is the functionality of the design and that the user can input the required data in a sensible fashion. However, some elements are compulsory, as we need to assess your understanding of performing a suitable GUI.

How the marks for Stage 2 are allocated: [Total Marks: 30]

Your GUI application must allow a user to input various data elements. To this end, your GUI must:

- Use the *Grid* element **[2 marks]**
- Add *Label* elements throughout your GUI. **[2 marks]**
- Use the *Background Colour* property. **[2 marks]**
- Allow a user to select choices from a *Listbox* element. **[2 marks]**
- Allow a user to input the data using an *entry widget*. **[2 marks]**
- Output data using an entry widget. **[2 marks]**
- Allow a user to click on a *Calculate* button that triggers any calculation. **[2 marks]**
- Allow a user to press an *Exit* or *Quit* button to properly close the program. **[2 marks]**

Marks are also allocated as follows:

- **GUI Design**. Is it simple to use and easy to understand? **[2 marks]**

- Program is written u**sing well-defined functions** including a *main()* function to direct the program. **[2 marks]**

- Program **code layout**. Separate blocks of code by a blank line. **[2 marks]**

- **Separate GUI functionality from calculations**. You should use a separate method for the calculations. It is good practice and follows good programming style to separate GUI code from other code. **[2 marks]**
- Use **comments** **[2 marks]**
- The program must have an **introduction**. **Check style against the Python style guide attached below.** **[2 marks]**
- **Business rules are met**. That is, your code solves the problem you are proposing (or at least contributes to its solution). **[2 marks]**

Stage 3. Reading from an external file

Your task will be to add code to your Stage 2 program that reads an external .txt file (in our case study, the astronaut tool weight data), put these into appropriate storage in memory (I suggest you use a nested list) and display the details to user when requested. Please note that the text file needs to be written by yourself according to your proposed problem. You can use any text editor such as Notepad (do not use Word!!!) You should also use the advanced Python GUI components to create a graph to show some of your calculations (compare the mass multipliers of the different destinations in our case study)

During the lecture, I'll go through the file *astronaut.txt* that contains the astronaut tool allowance weights along with their destination. Each line consists of destination as the first element, followed by a comma, that destination's mass multiplier followed by comma and the weight of each astronaut's tools in kilograms (kgs).

```
Moon,0.166,92,93,90,135,140,143
```

Also, for our case study:

- Celestial body (destination) will be displayed to the user in ascending order of mass multiplier.
- The text file will be read once.
- We will write a function called *readFile* to read the data from the text file.

How the marks for Stage 3 are allocated: [Total Marks: 10]

Marks are allocated as follows:

- The test file is properly written according to the problem proposed **[2.5 marks]**
- The text file should only be read once. You probably want to do that at the start of the program. Consider how you would do that. **[2.5 marks]**
- Correct display of mass multipliers based on the details in the text file. **[2 marks]**
- A simple graph is used to present the data gathered from the text file and the business rules proposed at the early stages of this assignment. **[3 marks]**

Python Style Guide

General

Your programs should be

- Simple
- Easy to read and understand
- Well structured
- Easy to maintain

Simple programs are just that. Avoid convoluted logic, nested if-statements and loops, duplicating execution (such as reading files multiple times), repeated code, being “clever”.

Programs can be made easier to read by following the “Layout” and “Comments” guidelines below.

Well-structured code uses functions to help tame complexity.

If programs are simple, easy to understand and well-structured they will be easy to maintain. Easily maintained programs will also use constants rather than literals and use built-in functions and types.

Layout

The IDLE text editor does a good job of automatically laying out your program and it also helps with the indentation of the code inside an/a if-statements, loops and functions. It is very important in Python that you indent your code correctly.

White space

Use white space freely:

- A blank line after declarations
- 2 blank lines before each function declaration

Names

Well-chosen names are one of the most important ways of making programs readable. The name chosen should describe the purpose of the identifier. It should be neither too short nor too long. As a guide, you should rarely need to concatenate more than 3 words.

Letter case

Constants use ALL_CAPITALS separated by underscore if needed
e.g. `PI, MAX_STARS`

Variables use firstWordLowerCaseWithInternalWordsCapitalised
e.g. `cost, numStars`

It is a good idea to use the ***Hungarian notation*** where the first letter (or two) indicate what type the variable has:

- i for int, e.g. `iNum`
- f for float, e.g. `fFloatingPointNum`
- s for String, e.g. `sName`

- bo for Boolean, e.g. `boCar`

Functions use firstWordLowerCaseWithInternalWordsCapitalised
e.g. `cost(), numStars()`

Types of names

1. Names of *GUI widgets* should have a prefix indicating the type of control (Python TK Interface). The prefix should be lower case with the remainder of the name starting with upper case.

Control	prefix	example
Entry Widget	`ent`	`entQuantity`
Label Widget	`lbl`	`lblResult`
List box Widget	`lst`	`lstStudents`
Scrollbar Widget	`scroll`	`yscroll for vertical scroll bar`
Button Widget	`btn`	`btnQuit`
Radiobutton Widget	`rb`	`rbColour`
Checkbutton Widget	`chkb`	`chkbSugar`

2. Names of *functions that does not return a value* should be **nouns** or **noun-phrases**

Example:

```
newSum = sum(alpha, beta, gamma)
```

or

```
newSum = sumOfStudents(alpha, beta, gamma)
```

rather than

```
newSum = computeSum(alpha, beta, gamma)
```

3. Names of *functions that returns a value* should be **imperative verbs.**

Example:

```
calculateCost(quantity, price)
```

rather than

~~results(newSum, alpha, beta, gamma)~~

4. Names of *Boolean functions* should be **adjectives** or **adjectival phrases**

Example:

```
if (empty(list))
```

or

```
if (isEmpty(list))
```

rather than

~~if (checkIfEmpty(list))~~

Comments

Comments should explain as clearly as possible what is happening. They should summarise what the code does rather than translate line by line. Do not over-comment.

In Python, use `##Comment` for a one-line comment.

Comments are used in three different ways in a program:

- Heading comments
- Block comments
- End-of-line comments.

Heading comments: These are generally used as a prologue at the beginning of each program and function. They act as an introduction, also describe any assumptions and limitations. They should show the names of any files, give a short description of the purpose of the program, and must include information about the author and dates written/modified.

Block comments: In general, you should not need to comment blocks of code. When necessary, these are used to describe a small section of following code. They should be indented with the program structure to which they refer. In this way the program structure will not be lost.

End-of-line comments: These need to be quite short. They are generally used with parameters to functions (to explain the purpose and mode of the parameter), and/or with variable declarations (to explain the purpose of the variable) and are not meant to be used for code.

Comment data, not code: Comments about the data – what it is and how it is structured - are much more valuable than comments about the code.

Prologue

Each of your programs should have a prologue. This is a set of "header comments" at the top of editor window. It should include

who wrote it	`## Author: Julio Romero`
when it was written	`## Date created: 04 Oct 2020`
when it was last changed	`## Date last changed: 04 Oct 2020`
what it does	`## This program does...`
what files it reads / writes	`## Input: sample.txt, Output: none`

Named Constants

Program sometimes employs a special constant used several times in program. By convention Python programmers create a global variable and name is written in uppercase letters with words separated by underscore. For instance, SPEEDING_FINE, INTEREST_RATE

The special naming convention reminds the programmer that no reassignments to the variable should be made during the execution of the program. Since Python allows reassignments to any variable, hence the programmer is responsible for not changing the value of the variable.

This makes your program easier to maintain. If the speeding fine or interest rate changes, the change only has to be made in the one place.

Functions

Using functions is one of the simplest and most effective ways of dealing with complexity in a program.

> When you write your own function to perform a calculation, it should not refer to any GUI controls.
> Do not mix IO and calculations in a function.

Function that does not return a value

- encapsulates a task
- name should be a npun. Example**: sum**()
- should only do 1 task.

Function that returns a value

- encapsulates a query
- return type is **int, boolean, double**…;
- if return type is **boolean**, name should be an adjective, otherwise name should be a verb.
    - Examples: **isEmpty(list), SquareRoot(number)**
- should answer only 1 query.

Comments for each function should include (to be placed on the line before the function starts)

name	above rules apply
purpose	what it evaluates or what it does
assumptions	any assumptions made, particularly on the arguments

Duplicated Code

View any duplicated code with suspicion. Look for a way of factoring the duplicated code into a function.

Built-in functions

Unless you have a good reason, use built-in functions rather than writing (or not writing) your own. They will be correct, flexible and familiar to a reader.

Types

Using types is an important way of making your intentions clear. Python does not allow you to be sloppy with your types. In IIT / IIT G, you are expected to choose appropriate types. You are also expected to use type conversions such as `int (23.26)`

Simplicity

Simplicity has long been recognised as one of the most important characteristics of programming. There are many ways of simplifying programs. These include avoiding nested IF statements, nested loops and complex conditions.