Introduction to Programming

Lab Worksheet

Week 5

Prior to attempting this lab tutorial ensure you have read the related lecture notes and/or viewed the lecture videos on MyBeckett. Once you have completed this lab you can attempt the associated exercises.

You can download this file in Word format if you want to make notes in it.

You can complete this work using the Python interpreter in interactive mode. This could be inside an IDE, or just a command prompt.

Topics covered:

- Writing code in files
- Executing program files
- Accessing command line arguments
- Writing and importing modules
- The modules search path

For more information about the module delivery, assessment and feedback please refer to the module within the MyBeckett portal.

Writing code in files

The Python interpreter allows code to be entered interactively in Read-Evaluate-Print and Loop (REPL mode) or executed directly from a text file. REPL mode is mainly used to write very small pieces of code for learning and experimentation purposes.

Larger more complex programs need to be stored in one or more text files, then executed later. Such Python programs are often called *scripts*. Placing code in text files allows the code to grow and evolve over time. It also provides an easy mechanism of re-running programs over and over again.

The files are simple plain text files and there is nothing special about them other than the name which must include the `.py` suffix. The program code within the files is exactly the same as the code that can be typed in REPL mode.

However, there is one important difference between executing statements in interactive-mode compared to from a file. When in REPL mode the result of expressions is automatically printed to the screen (this is the 'P' within REPL). Such output is not done automatically when executing scripts, and an explicit call to the `print()` function is required to generate output.

Text files will often contain many thousands of lines of code. We can also define *functions* and *classes* within our files. These may be reused across several different programs. i.e. we can import the same content (to share the same functionality) between different programs.

Choosing an Editor

The contents of the scripts can be produced in many ways, given that they are simply plain text files. Any generic editor such as 'notepad' on Windows, or 'vi' on Linux can be used.

Many professional programmers tend to use Integrated Development Environments (IDEs). These include text editors that provide additional features such as syntax colouring, auto-completion, and automatic fixes to syntax errors. They also provide a way to directly execute and debug a program from within the same environment. Popular IDEs that support Python development include PyCharm, IDLE, Eclipse and IntelliJ.

These lessons focus on the Python language rather than the chosen development environment, so everything discussed can be done using either a basic text editor, or via a fully blown professional IDE. That being said, it is useful to know how to develop and execute code *manually* without the use of an IDE, so a better understanding of how the whole process actually works can be gained.

Executing a Script

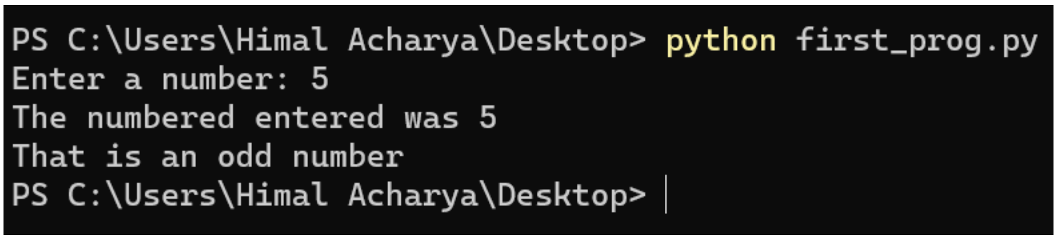
The Python interpreter will attempt to load and execute a *script* from any filename passed when the program is started, e.g. typing the following at the operating system's command line would cause the Python interpreter to load then execute the contents of the `my_program.py` file.

```
python my_program.py
```

For this to work of course, the script file would have to exist (in the current directory) and contain some meaningful Python statements.

Note: when working at the command line it is often better to change into the directory where your programs are stored, allowing easy access for editing and execution.

TASK: Use an editor to input the Python program shown below then save it to a file called `first_prog.py`. Once that has been done, execute the program from the command line.



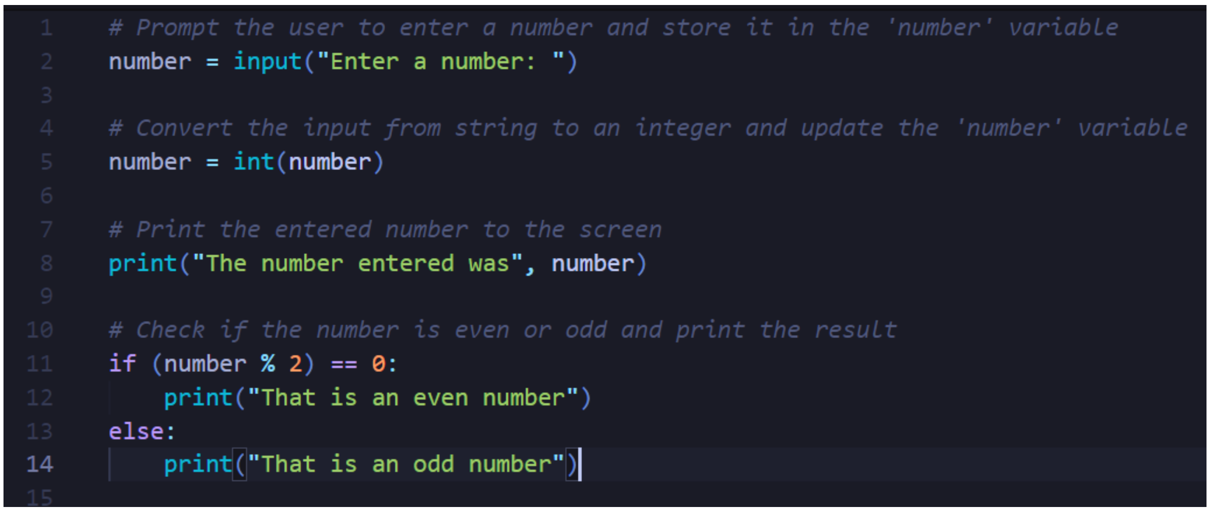
```
PS C:\Users\Himal Acharya\Desktop> python first_prog.py
Enter a number: 5
The numbered entered was 5
That is an odd number
PS C:\Users\Himal Acharya\Desktop>
```

```
number = input("Enter a number: ")

number = int(number)

print("The numbered entered was", number)

if (number % 2) == 0:
      print("That is an even number")
else:
      print("That is an odd number")
```

Execute the program multiple times inputting several different values for testing purposes, and ensure no errors are reported.

Commenting code that is stored within files is good practice, since code evolves over time and becomes very large and complex, having comments that explain what the program does is very useful.

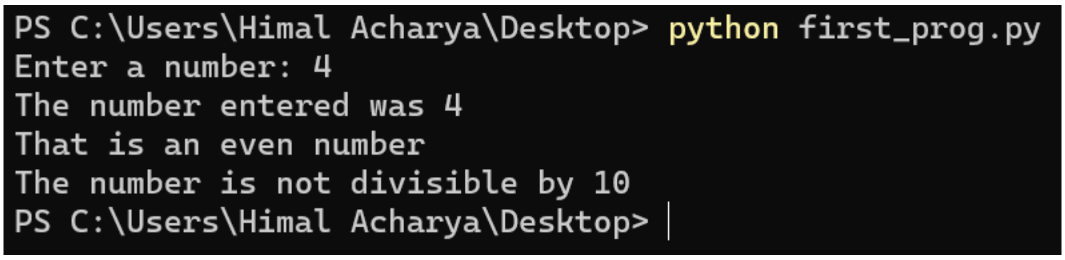
TASK: Open the `first_prog.py` and add comments above each statement within the file describing what that statement does (using a ‘`#`’ at the beginning of the line) . Save the file and execute it again for testing purposes.



```python
# Prompt the user to enter a number and store it in the 'number' variable
number = input("Enter a number: ")

# Convert the input from string to an integer and update the 'number' variable
number = int(number)

# Print the entered number to the screen
print("The number entered was", number)

# Check if the number is even or odd and print the result
if (number % 2) == 0:
    print("That is an even number")
else:
    print("That is an odd number")
```

We now start to see the major benefit of storing programs in files i.e. we can edit and execute the code as many times as we want without having to re-type the whole program.

TASK: Open the `first_prog.py` and add some extra code that identifies and prints a message stating whether the entered number is divisible by `10`. You should be able to base the new code on the `if` statement already provided. Once completed, save the file and execute it again for testing.



```
PS C:\Users\Himal Acharya\Desktop> python first_prog.py
Enter a number: 4
The number entered was 4
That is an even number
The number is not divisible by 10
PS C:\Users\Himal Acharya\Desktop>
```

Command Line Arguments

All programs take *input*, do some *processing*, then generate *output*. The input can come from many sources, including the user. User input itself can have multiple forms, e.g. if a graphical user interface (GUI) is being used then input may be from the user pressing a button, or selecting a menu option. When running a command line program the user can be prompted to enter values using the `input()` function, as we have seen above.

When working from the command line a common mechanism of providing input to a program is via *command line arguments*. Rather than the user being explicitly asked to provide input, values are provided at the point at which the program is executed. This is done by typing values after the name of the file to be executed, e.g. the following asks the Python interpreter to execute the program `total.py`, but also specifies three values that should be *passed* to the program as arguments.

```
python total.py 25 90 15
```

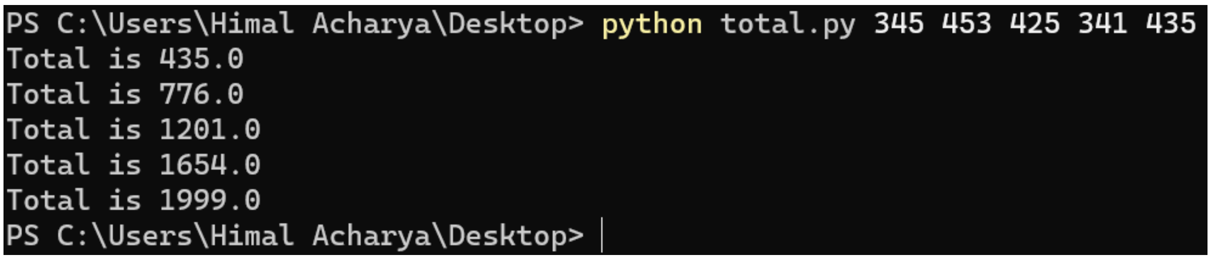
Arguments passed in this way must be separated by spaces (rather than a comma when calling a function within the Python language).

The argument values can then be accessed from within the program, and treated as user input. This alternative form of user input is very popular with ‘utility’ type programs, or programs which are designed to be executed automatically as part of a larger process.

The Python interpreter stores any passed arguments in a *List* type variable called `sys.argv` making them extremely easy to access from within the program itself. The first element of this list is always the name of the program itself.

Note: the arguments are always stored as a list of *string* type values, so need explicitly converting if they are to be used as some other type of value. This behaviour is in line with how the `input()` function works.

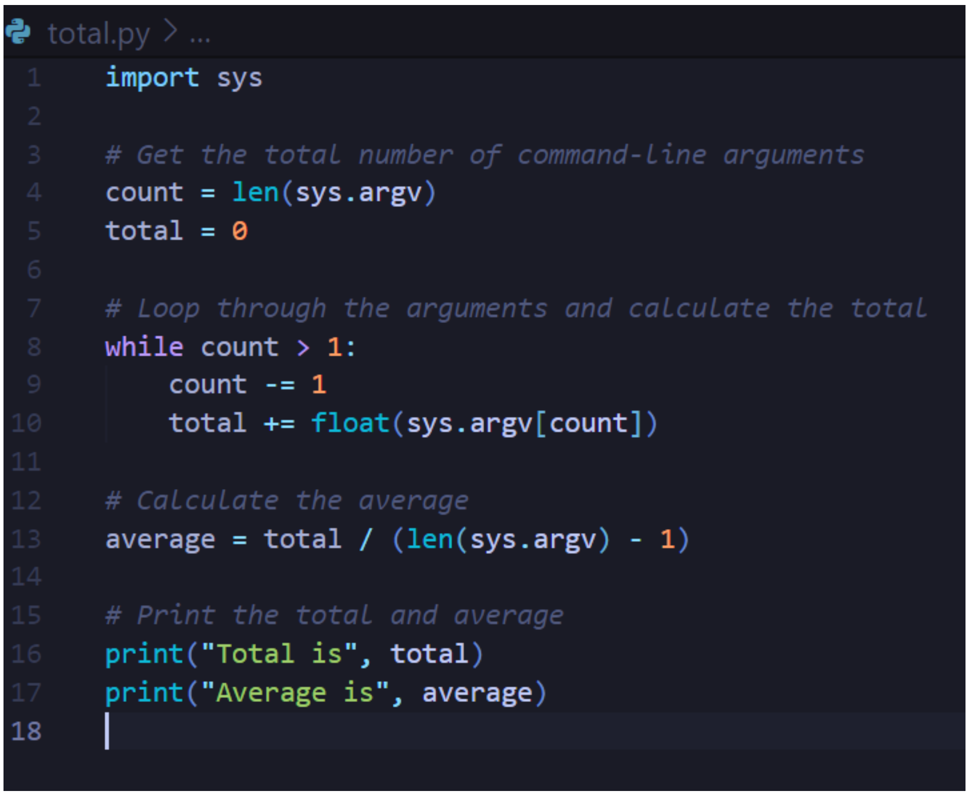
TASK: Use an editor to input the Python program shown below then save it to a file called `total.py`. Once that has been done, execute the program from the command line, passing several numeric values for testing.



```
PS C:\Users\Himal Acharya\Desktop> python total.py 345 453 425 341 435
Total is 435.0
Total is 776.0
Total is 1201.0
Total is 1654.0
Total is 1999.0
PS C:\Users\Himal Acharya\Desktop>
```

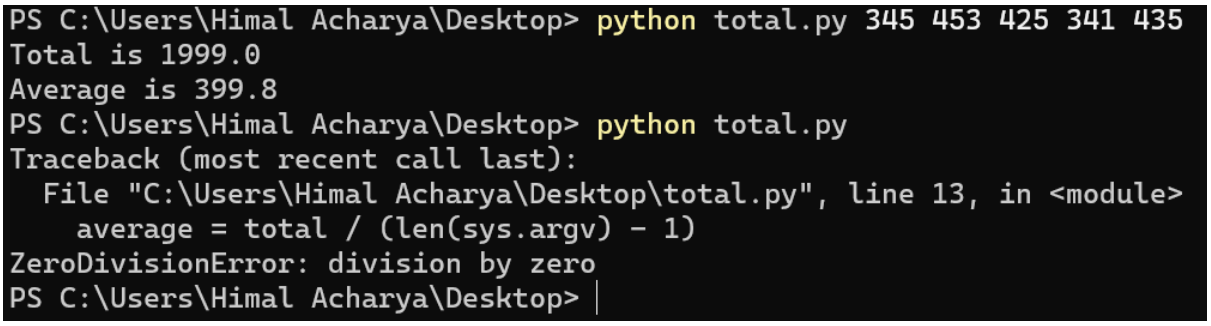
```
import sys

count = len(sys.argv)
total = 0
while count > 1:
      count -= 1
      total += float(sys.argv[count])

print("Total is", total)
```

TASK: Improve the previous program by adding additional code that not only prints the total of any passed arguments, but also calculates and prints the average. Execute the program several times for testing. What happens if no arguments are passed?



total.py > ...

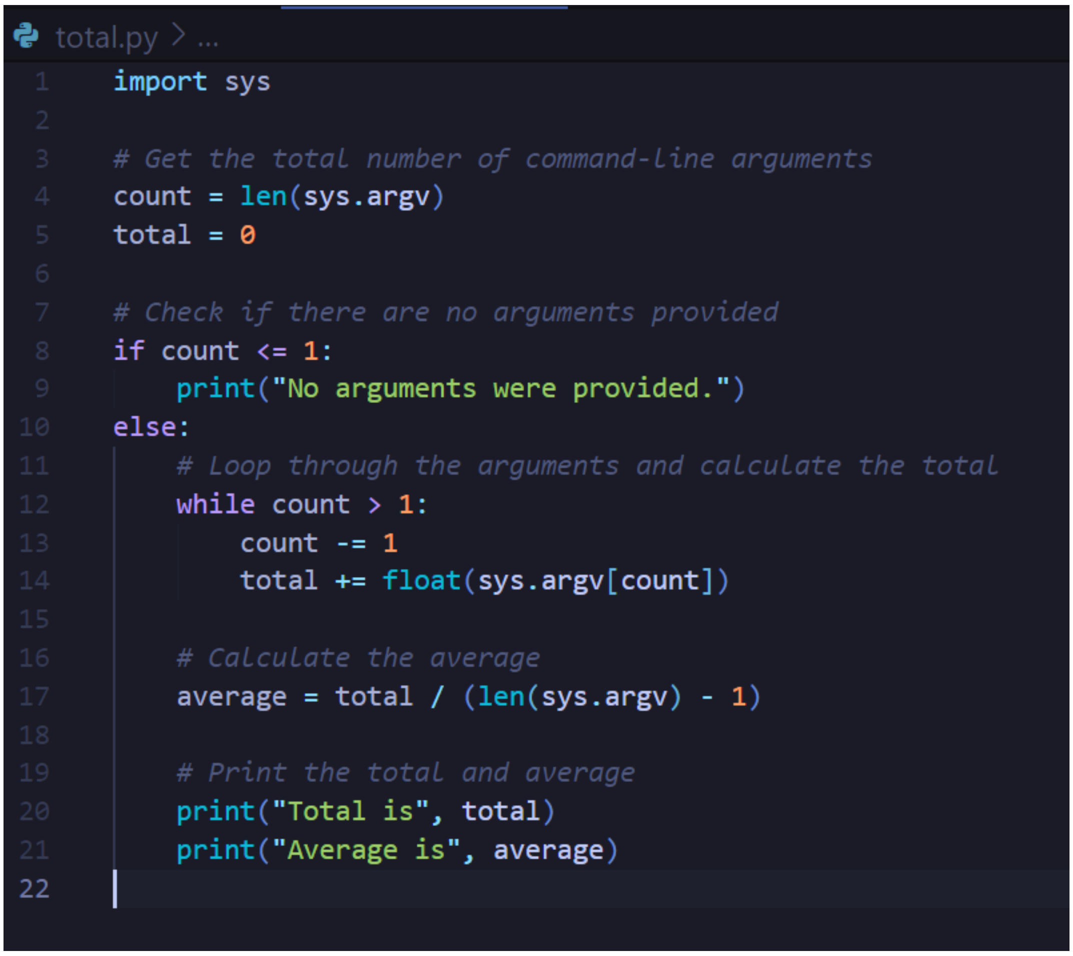
```python
import sys

# Get the total number of command-line arguments
count = len(sys.argv)
total = 0

# Loop through the arguments and calculate the total
while count > 1:
    count -= 1
    total += float(sys.argv[count])

# Calculate the average
average = total / (len(sys.argv) - 1)

# Print the total and average
print("Total is", total)
print("Average is", average)

```



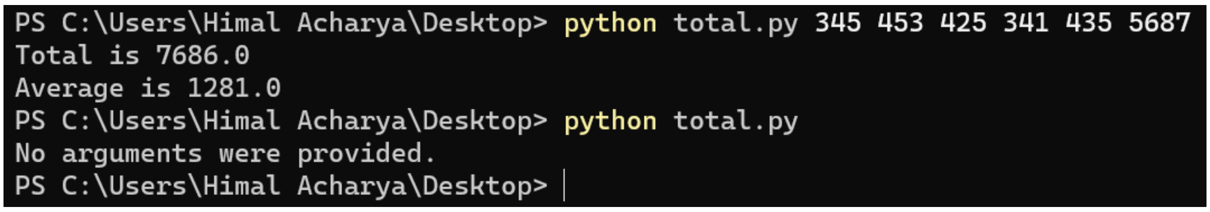
```
PS C:\Users\Himal Acharya\Desktop> python total.py 345 453 425 341 435
Total is 1999.0
Average is 399.8
PS C:\Users\Himal Acharya\Desktop> python total.py
Traceback (most recent call last):
  File "C:\Users\Himal Acharya\Desktop\total.py", line 13, in <module>
    average = total / (len(sys.argv) - 1)
ZeroDivisionError: division by zero
PS C:\Users\Himal Acharya\Desktop>
```

Remember, the first element of the `sys.argv` list is the program name itself, so it is not uncommon to ignore or skip processing of that first element.

TASK: Improve the program once more by adding a check to see whether no arguments have been passed, if so print a message saying "`no arguments were provided`". Also add comments to the program. Execute the program several times for testing.



```python
import sys

# Get the total number of command-line arguments
count = len(sys.argv)
total = 0

# Check if there are no arguments provided
if count <= 1:
    print("No arguments were provided.")
else:
    # Loop through the arguments and calculate the total
    while count > 1:
        count -= 1
        total += float(sys.argv[count])

    # Calculate the average
    average = total / (len(sys.argv) - 1)

    # Print the total and average
    print("Total is", total)
    print("Average is", average)

```



```
PS C:\Users\Himal Acharya\Desktop> python total.py 345 453 425 341 435 5687
Total is 7686.0
Average is 1281.0
PS C:\Users\Himal Acharya\Desktop> python total.py
No arguments were provided.
PS C:\Users\Himal Acharya\Desktop>
```

Writing Modules

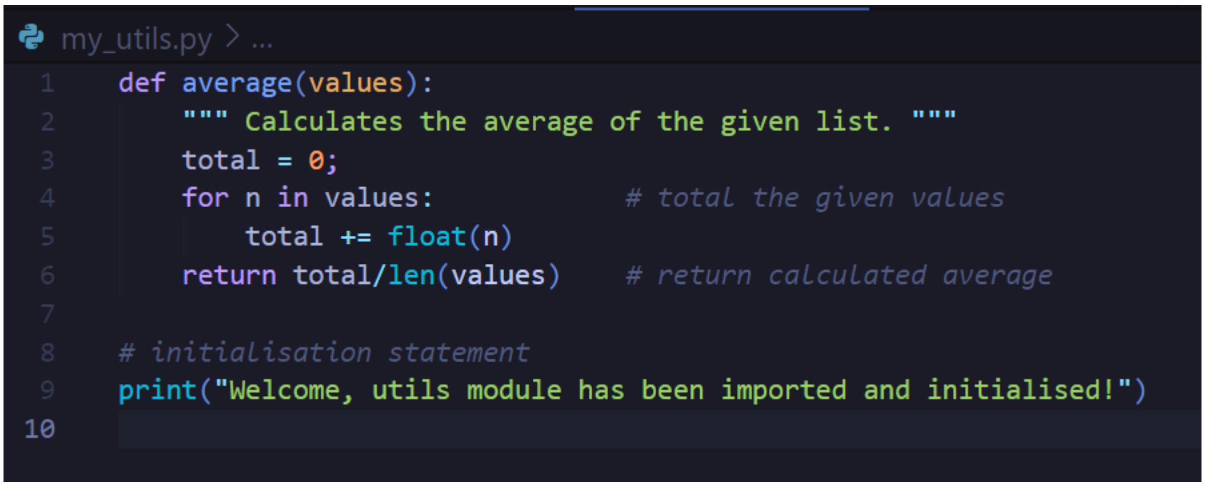
The code that we write in files can not only be executed as a *script* from the command line, but can be *imported* by other Python programs. Any `.py` file containing definitions and statements is called a *module*. A set of built-in *modules* are provided, e.g. we have just seen the `sys` module being imported. A collection of modules is often called a *library*.

It is possible to write your own *modules* that are designed to be imported into other programs. A module is like any other Python program but often contains specific content such as a set of related *function* definitions. As a module is being imported, any Python statements are executed, these are usually used to initialise the module e.g. setup variable values etc.

Code that is designed to be used as a module is stored in a text file as normal, i.e. the file name ends with a `.py` suffix. When imported however, the `.py` is omitted from the name.

TASK: Use an editor to input the Python program shown below then save it to a file called `my_utils.py`.

```
def average(values):
    """ Calculates the average of the given list. """
    total = 0;
    for n in values:                # total the given values
        total += float(n)
    return total/len(values)        # return calculated average

# initialisation statement
print("Welcome, utils module has been imported and initialised!")
```



my_utils.py > ...

```
def average(values):
    """ Calculates the average of the given list. """
    total = 0;
    for n in values:            # total the given values
        total += float(n)
    return total/len(values)    # return calculated average

# initialisation statement
print("Welcome, utils module has been imported and initialised!")
```

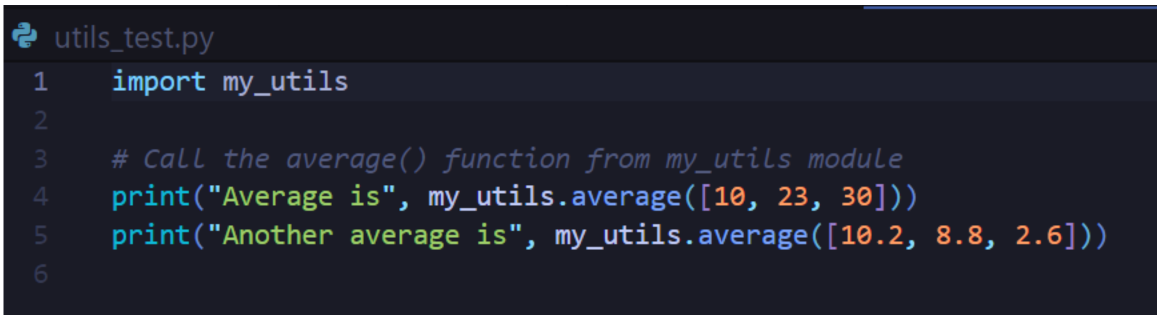
Notice how a comment has been included directly below the function name using a triple quoted string. You may recall this is called a *docstring* and should be provided with function definitions that appear within modules.

Once that file has been saved, execute it from the command line using the Python interpreter. If this is working correctly the 'welcome' message should be displayed.

This particular file however is not really designed to be executed in a stand-alone fashion. It is designed to be imported and used as a *module*.

TASK: Use an editor to input another Python program `utils_test.py`. This program should `import my_utils` then call the `average()` function several times, passing a list of values as a parameter, e.g.

```
print("Average is", my_utils.average([10, 23, 30]))
print("Another average is", my_utils.average([10.2, 8.8, 2.6]))
```



utils_test.py

```python
import my_utils

# Call the average() function from my_utils module
print("Average is", my_utils.average([10, 23, 30]))
print("Another average is", my_utils.average([10.2, 8.8, 2.6]))

```

Methods of Importing

Throughout these exercises we have used `import` on several occasions. The need to import from a module is required in just about all non-trivial programs, since it provides access to modules present within the *Python Standard Library* and many other external libraries.

Just to recap:

- Python has a number of functions and types built directly into the language. These do not need to be imported prior to use. e.g. `print()`, `input()`, `len()`.

- The *Python Standard Library* provides a number of very commonly used modules, these do need to be imported prior to use, and are guaranteed to be available to all Python installations. e.g. `math`, `sys`.

- There are many thousands of third party libraries available. These need to be imported, but also may need to be explicitly installed on a specific Python installation prior to use.

- We can use our own modules, which also need to be imported and must be distributed along with any of our programs that rely on them being present.

There are several different approaches to importing module content into a program. Which of these mechanisms is used affects how access to the imported content is performed.

In order to understand why different approaches are available it is necessary to understand the concept of a *symbol table*. This is simply a location where all the names we use in our programs are stored. For example, the names of any variables and functions we define within our programs get stored in the *symbol table*.

Each module has its own *local symbol table*, thus allowing variable and function names not to clash. This means module developers can use variable and function names without the worry that they may be the same as names within programs into which they are imported.

The way in which a module is imported determines how the content is merged into the *symbol-table* of our own program.

We have seen to most basic form of `import` many times already:

```
import math

print("The square root of 20954 is", math.sqrt(20954))
```

This imports the entire module content (all variables, functions etc.), but does not add the names of these into the importing program's *symbol table*. Hence, all access to the content has to be prefixed with the module's name.

We can also explicitly import specific content from a module. In this case however the names are imported into the program's *symbol table*. Hence access no longer needs to be prefixed with the module's name, e.g.

```
from math import sqrt

print("The square root of 20954 is", sqrt(20954))
```

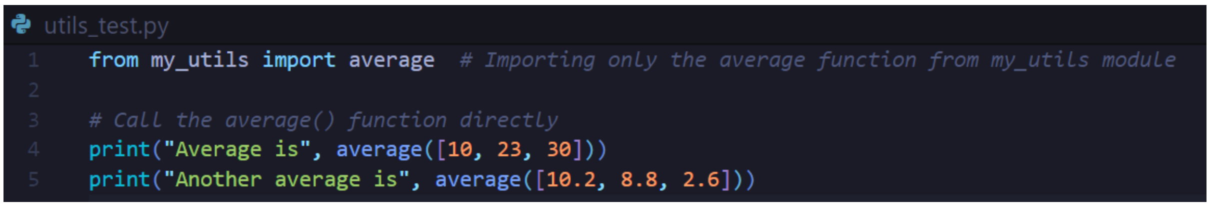
This form is usually safe to use, since we can easily check whether the name of the imported function clashes with any of the names we use within our own program.

The third approach to importing allows all content to be directly imported into the program's own *symbol table*, e.g.

```
from math import *

print("The square root of 20954 is", sqrt(20954))
print("The sine 0.653 is", sin(0.653))
print("The cosine 0.623 is", cos(0.623))
```

This form is not recommended, since unless you are aware of all names defined in the imported module, clashes become very likely to happen. Hence this form tends to be used to save time when programming in *interactive-mode* but should otherwise be avoided.

TASK: Update the previous program `utils_test.py`, so that the `import` statement explicitly imports the `average()` function directly into the program's *symbol table*, allowing the prefix to be removed from the later function calls.



```python
from my_utils import average  # Importing only the average function from my_utils module

# Call the average() function directly
print("Average is", average([10, 23, 30]))
print("Another average is", average([10.2, 8.8, 2.6]))
```

Using Import Aliases

The above three approaches to importing have been covered within earlier lessons. However there is another mechanism available during importing which helps avoid name clashes, or can be used to save time while typing.

The `import` statement can be extended to allow an alternative name, or *alias*, to be applied to the imported element. This means that the names of either modules or specific elements, such as functions, can be explicitly changed within the current program. This makes avoiding clashes easier, e.g. the following example explicitly changes the name of the imported `sqrt()` function to be `root()` instead.

```
from math import sqrt as root

print("The square root of 20954 is", root(20954))
```

Using an alias is also convenient when a general `import` is being used, since the name of the whole module can be *aliased* into something shorter, therefore avoiding the need to import all content into the program's symbol-table. So:

```
import math as m

print("The square root of 20954 is", m.sqrt(20954))
print("The sine 0.653 is", m.sin(0.653))
print("The cosine 0.623 is", m.cos(0.623))
```

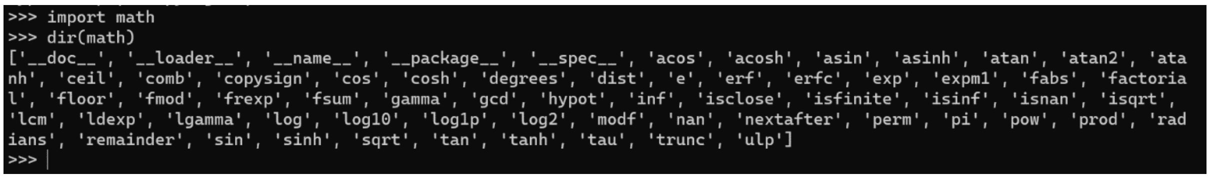
Listing Symbol Table Names

When working with the Python interpreter in *interactive mode* it is sometimes useful to be able to quickly list the names defined within the current *symbol table*, or those within an imported module. This can be done easily using the built-in `dir()` function.

If called with no parameters this lists the names defined within the program's *symbol-table*. If an object name is passed, such as that of an imported module, it will list the names defined in the module's *symbol-table*.

TASK: Start Python in interactive mode and input the following statements.

```
>>> import math

>>> dir(math)
```

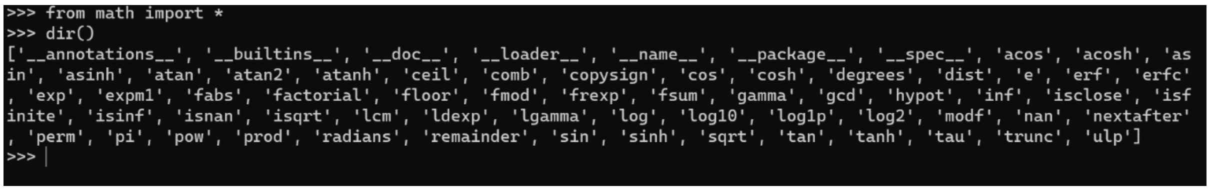


```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'ata
nh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factoria
l', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt',
'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod', 'rad
ians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
>>> |
```

Look at the list of names shown, these should generally look familiar by now. These are all of the names defined within the `math` module.

TASK: Now enter the following statements:

```
>>> from math import *

>>> dir()
```



```
>>> from math import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'as
in', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc'
, 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isf
inite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter'
, 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc', 'ulp']
>>> 
```

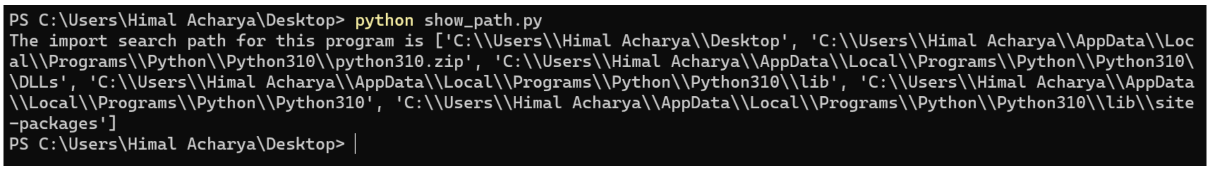
Since an asterisk '`*`' was used to `import` the entire contents of the `math` module into the program's own *symbol table*, it now contains everything that was defined within the `math` module.

The Module Search Path

The `import` statement looks at specific locations within the host system's directory structure for files to be imported. When executed, the import statement first looks for the given name within the modules provided as part of the standard distribution, i.e. those within the Python Standard Library. If the given name is not found in those modules, then a list of directories stored in the `sys.path` variable is searched. The `sys.path` is a list of string values that is initialised when the program starts, and includes the directory from which the input script was loaded, or the current directory if no script was specified.

TASK: Use an editor to input a Python program `show_path.py`. Execute the script and examine the results. Notice the first entry will be the directory in which the program file itself is stored.

```
import sys

print("The import search path for this program is", sys.path)
```



```
PS C:\Users\Himal Acharya\Desktop> python show_path.py
The import search path for this program is ['C:\\Users\\Himal Acharya\\Desktop', 'C:\\Users\\Himal Acharya\\AppData\\Loc
al\\Programs\\Python\\Python310\\python310.zip', 'C:\\Users\\Himal Acharya\\AppData\\Local\\Programs\\Python\\Python310\
\DLLs', 'C:\\Users\\Himal Acharya\\AppData\\Local\\Programs\\Python\\Python310\\lib', 'C:\\Users\\Himal Acharya\\AppData
\\Local\\Programs\\Python\\Python310', 'C:\\Users\\Himal Acharya\\AppData\\Local\\Programs\\Python\\Python310\\lib\\site
-packages']
PS C:\Users\Himal Acharya\Desktop> 
```

The `sys.path` value can be programmatically changed, allowing the program itself to include specific search locations when an `import` statement is being performed. However, hard coding system specific directory paths into code is bad practice, as it reduces program portability. A better option is to use the environment variable called `PYTHONPATH.` This can be setup within the host environment (OS). Any paths included in this variable are also included within the `sys.path` list. How this environment variable is set is Operating System specific, within a Unix type system it could be set a follows -

```
export PYTHONPATH='/my_modules:/util_modules'
```

This example will add two additional paths to the `sys.path` list the next time the python interpreter is used. Setting the search path in this way makes the code more portable between different systems, since only the environment variable needs changing rather than the program code itself.

Writing a ‘Script’ and ‘Module’

Placing code into files allows large programs to be developed, improved and debugged over a long period of time. It is also an extremely useful way of developing reusable modules that may be imported into many other programs.

Python is flexible enough to be able to allow a single file to be used as either a standalone program, or a reusable module. i.e. a single file can either be executed as a *script* from the command-line, or imported by other programs as a reusable *module*. The code within a file has to be deliberately written in such a way to achieve this however.

A program can detect whether it is executing directly as a *script* or is being imported as a *module* by accessing a special variable called __name__. The value of this variable is set as follows -

- If a program is being executed directly as a *script*, it is set to ‘`__main__`’.
- If a program is being imported as a *module*, it is set to the name of the `.py` file.

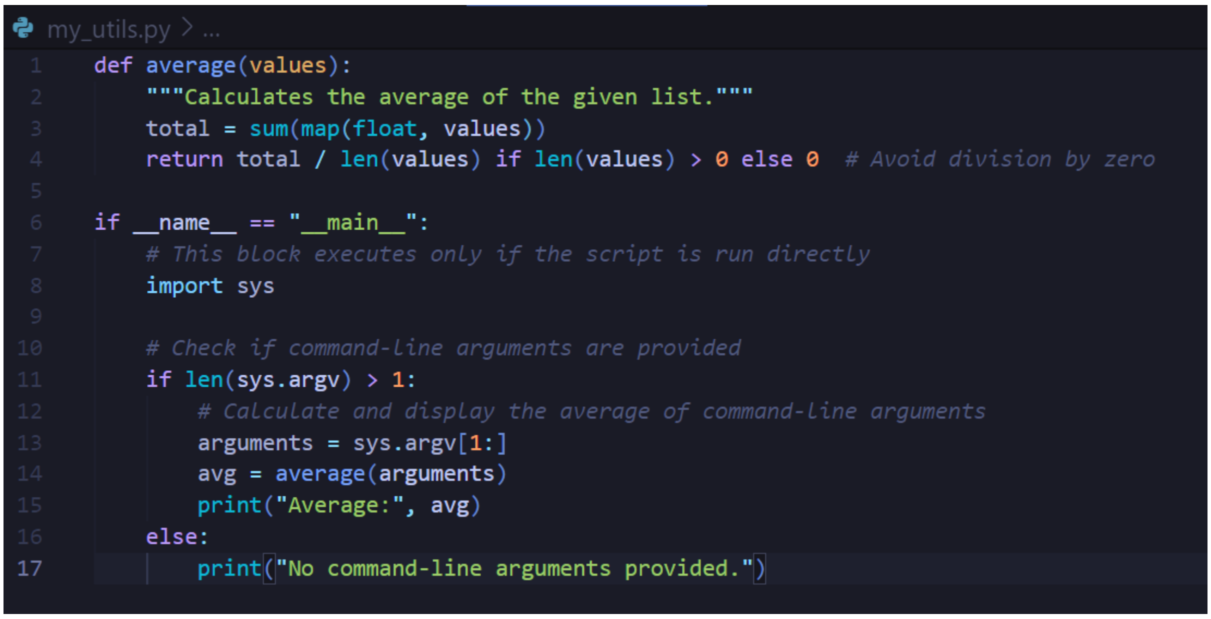
Therefore a program can examine this variable to determine whether it is executing as a script, or an imported module.

For example, the following code can be used either as a stand-alone program, or be imported:

```
def display_heading(text):
      """ Prints a heading between two bars. """
      print("===================================")
      print(text)
      print("===================================")



if __name__ == "__main__":
      import sys
      if len(sys.argv) > 1:
            display_heading(sys.argv[1])
```

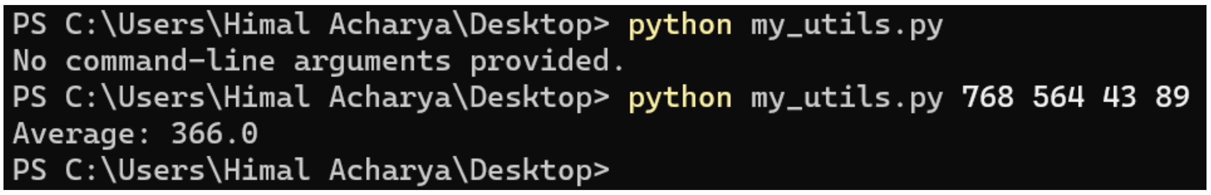
When this program executes, the `if` statement detects whether it is executing as a script. If so then it calls the `display_heading()` function passing a *command-line argument*. If it is being imported as a module however this call is not made and the program that imported the module is responsible for calling the `display_heading()` function instead.

TASK: Update the earlier program `my_utils.py`, so that when executed directly from the command line it displays the average of any provided *command-line arguments*. However when imported by another program, nothing is displayed until the `average()` function is explicitly called.



my_utils.py > ...

```python
def average(values):
    """Calculates the average of the given list."""
    total = sum(map(float, values))
    return total / len(values) if len(values) > 0 else 0  # Avoid division by zero

if __name__ == "__main__":
    # This block executes only if the script is run directly
    import sys

    # Check if command-line arguments are provided
    if len(sys.argv) > 1:
        # Calculate and display the average of command-line arguments
        arguments = sys.argv[1:]
        avg = average(arguments)
        print("Average:", avg)
    else:
        print("No command-line arguments provided.")
```



```
PS C:\Users\Himal Acharya\Desktop> python my_utils.py
No command-line arguments provided.
PS C:\Users\Himal Acharya\Desktop> python my_utils.py 768 564 43 89
Average:  366.0
PS C:\Users\Himal Acharya\Desktop>
```

____

Key Terminology

Understanding key terminology is important not only when programming but for understanding computer science in general. Certain terms, technologies and phrases will often re-occur when reading or learning about many computer science topics. Having an awareness of these is important, as it makes learning about the various subjects and communication with others in the field easier.

TASK: Look at each of the phrases below and ensure you understand what each of these means. For any that you do not understand, do a little research to find a definition of each term. This research may involve looking back over these notes, or the associated lecture notes. It may also involve searching for these terms on the Internet.

- IDE
- Module
- Command Line Arguments
- Symbol-table
- Search path

IDE: Software tool that provides a comprehensive environment for software development.

Module: A file containing Python definitions and statements.

Command Line Arguments: Values provided to a program when it's executed from the command line.

Symbol-table: Data structure tracking symbols (variables, functions) in a program.

Search Path: List of directories where the interpreter looks for modules during import.

Practical Exercises

You have now completed this tutorial. Now attempt to complete the associated exercises.