Introduction to Programming

Lab Worksheet

Week 4

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:

- Importing and Using Functions
- Defining and Documenting Functions
- Default and keyword arguments
- Arbitrary Length Argument Lists
- Lambda Expressions

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

Importing and Using Functions

Functions provide us with a mechanism of reusing existing code. We can call functions to achieve many common tasks. The Python language has a number of built-in functions, some of the most common include `print()`, `input()` and `range()`.

The functions that are built into the language can be accessed directly from code. However many thousands of other functions also exist. These functions are contained in special files called '*modules*' which act as a container for various elements including functions. In order to use functions that are not directly built-in to the language we need to *import* them first.

The most commonly used functions are present within the **Python Standard Library**. This is provided with all Python distributions so you can be sure the modules will always be available. However, many third-party modules also exist, and cover just about everything you could ever need. Using these however often requires that the libraries are explicitly installed within the Python distribution. So for now we shall be sticking with the Python Standard Library modules only.

In our Python programs, we can import functions from modules in several different ways. First of all we can import *all* functions that are defined within a module as follows -

```
import math
```

This would import all of the functions (and other elements) defined within the `math` module.

Once imported, we can call the functions as usual. However, when the whole *module* has been imported (as above) we need to prefix the function name with the module name, e.g.

```
result = math.sin(6.2)
```

TASK: Write some code that imports the `math` *module*, then calculates and prints the square root of the number 2401. Use the `sqrt()` function provided by the math module.

```
import math


number = 2401

square_root = math.sqrt(number)

print(f"The square root of {number} is: {square_root}")
```

Individual functions can also be imported from a module. For example, we could import the `sqrt()` function only using the following code:

```
from math import sqrt
```

This approach to importing functions has the advantage of allowing the function to be called without the need to use the module prefix. For example:

```
root = sqrt(49)
```

TASK: Write some code that imports only the `log2()` function from the `math` *module*, then call this function to calculate the log base 2 value of 1024. Print the result to the screen.

from math import log2

number = 1024

log_base_2 = log2(number)

print(f"The log base 2 of {number} is: {log_base_2}")

Finally, we can directly import all module content using a wildcard (`*`). In this case all the functions can be called without the need for a prefix. This approach is not recommended since it pollutes the *namespace*, however it is convenient when using the interpreter in interactive-mode, like so:

```
from math import *
root = sqrt(49)
val = cos(radians(45))
```

It is also worth noting that we can import other elements from modules as well as functions. For example, we can also import constant values:

```
from math import pi

print("The value of pi is", pi)
```

Defining Functions

As well as using pre-existing functions, Python allows us to define our own. This allows us to wrap commonly used code into a reusable package. A single program may call the same function several times over, hence it clearly makes sense to write the code only once rather than many times. Over time we can even start to create our own *modules* containing commonly used functions. This allows code to be reused over several different projects.

Within Python we define our own functions using the `def` keyword, followed by the function name, the names of the expected arguments (formal parameters), and the block of code to be executed when the function is called.

For example, we could define a function that prints the given message out twice to the screen as follows -

```
def displayTwice(msg):
      print(msg)
      print(msg)
```

In this case the function's name is `displayTwice` and it requires a single argument when called.

TASK: Input the above function definition. Once that is done make several calls to the function passing different argument values for testing.

```
def displayTwice(msg):
    print(msg)
    print(msg)
displayTwice("Hello")
displayTwice("Testing")
displayTwice(123)
displayTwice([1, 2, 3])
```

Docstrings

It is standard practice to document a function using a triple quoted string. These are known as *docstrings* and appear as the first line of the function. The docstring should contain a concise description of the function's purpose. The text should begin with a capital letter and end with a period (`.`). If multiple lines exist, the second line should be blank, to separate the heading from the rest of the description.

Note: *docstrings* must be indented in the same way as any other statements within the block.

The Python help system is able to extract and display such docstrings if `help()` about that function is requested.

TASK: Re-Input the above function definition, but this time add a docstring that includes a description of the function's purpose. Once that is done enter a command such as `help(displayTwice)` and see what it displays.

```
>>> def displayTwice(msg):
...     """
...     Display the given message twice.
...     """
...     print(msg)
...     print(msg)
...
>>> # Use the help() function to display the docstring
>>> help(displayTwice)
Help on function displayTwice in module __main__:

displayTwice(msg)
    Display the given message twice.

>>>
```

Formal and Actual Parameters

When a function is being defined, the arguments we specify are usually referred to as the *formal parameters* of the function. In the above example the *formal parameter* is called `msg`. When a function is being called, the argument values provided are called the *actual parameters*.

The formal parameter names act like local variables within the function, and can only be accessed within that specific function block. Any variables declared within the function block can also only be accessed from within that block, and cease to exist once the function ends. This idea of *local-scope* means functions do not have to worry about using variable names that may already exist elsewhere in the program or within other functions.

Returning a value

As we have seen when calling built-in functions, such as `input()`, a function can return a value to the caller. When a function is being defined, a value can be returned by using the `return` statement within the code block. Once that statement is executed the function finishes and returns to the caller. If such a statement is not present, the function will return the special value `None` once the final statement within the block is executed.

The following is an example of a function that returns a value.

```
def findMax(a,b):
     """Finds the maximum of two values."""
     if ( a > b ):
          max = a
     else:
          max = b
     return max
```

TASK: Input the above function definition. Once that is done make several calls to the function passing different argument values and displaying the returned value.

```
def findMax(a, b):
    """
 Finds the maximum of two values.
 """
    if a > b:
        max_value = a
    else:
        max_value = b
    return max_value


# Test the function with different argument values
result1 = findMax(10, 5)
result2 = findMax(3.14, 2.71)
result3 = findMax(-8, 0)


# Display the returned values
```

print("Result 1:", result1)

print("Result 2:", result2)

print("Result 3:", result3)

Default Arguments

As we have seen earlier, when a function is being defined the formal parameters are specified as a list of names between parentheses. It is possible to specify *default argument* values, for situations where the function call is made without an actual parameter value being provided. This allows functions to be more flexible in nature, allowing common parameter values to be omitted from the call when the default is appropriate.

Default arguments are specified by assigning a value to the formal parameter within the function definition. For example, given the following function definition:

```
def displayMessage(msg, suffix = "?"):
     """Prints a message with a suffix."""
     print(msg, suffix)
```

Calls to the function could be made as follows:

```
>>> displayMessage("Save file")
Save file ?

>>> displayMessage("An error occurred", "!")
An error occurred !
```

Notice how the first call only passes one argument, therefore the function uses the default `suffix` value. On the second call however both arguments are passed, so the provided suffix is used instead of the default.

Note: default arguments can only be specified to the right of parameters that do not have defaults provided.

TASK: Define a function that takes two numeric values, multiplies them together then returns the result. If the function is called with only a single argument however, then the value should be multiplied by itself. Once the function is defined, call it several times and display the returned values for testing purposes.

def multiply_values(a, b=None):

    """

```
    Multiply two values together. If called with only one argument, multiply it by itself.

    Parameters:

    - a (int or float): First value.

    - b (int or float, optional): Second value. Default is None.

    Returns:

    - int or float: The result of the multiplication.

    """

    if b is None:

        result = a * a

    else:

        result = a * b

    return result


result1 = multiply_values(5)

result2 = multiply_values(3, 4)

result3 = multiply_values(-2.5, 2.5)

print("Result 1:", result1)

print("Result 2:", result2)

print("Result 3:", result3)
```

Keyword Arguments

Up until now all the function calls made have assumed that the passed *formal parameter* list appears in the same order as the *actual parameters* (arguments) specified within the function definition. So given the following function:

```
def someFunc(x, y, z):
     print("x is", x, "\ny is", y, "\nz is", z)
```

The following call would result in:

```
someFunc(1,2,3)

x is 1
y is 2
z is 3
```

However, as an alternative we can call a function with argument values that are explicitly named. This means that the order that we pass the *actual parameters* does not need to match the order in which they were formally defined. For example, the previous function could have been called using *keyword arguments*, where the parameter names are explicitly provided in the call:

```
someFunc(y=2, z=3, x=1)

x is 1
y is 2
z is 3
```

Notice how the result is the same, but the order in which the *actual parameters* (arguments) were passed is irrelevant since the name of each was explicitly provided.

TASK: Enter the example function shown above, then try calling it using the *keyword arguments* in several different orders.

```
>>> def someFunc(y, z, x):
...     """
...     Display the values of three parameters.
...     """
...     print(f"x is {x}")
...     print(f"y is {y}")
...     print(f"z is {z}")
...
>>>
>>> someFunc(y=2, z=3, x=1)
x is 1
y is 2
z is 3
```

```
>>> print("---")
---
>>> someFunc(x=1, y=2, z=3)
x is 1
y is 2
z is 3
>>> print("---")
---
>>> someFunc(z=3, x=1, y=2)
x is 1
y is 2
z is 3
>>>
>>>
```

Any parameters that do not have *default argument* values defined must appear in the call, therefore the previous function could NOT be called as follows -

```
someFunc(z=99, y=4)
```

This would result in an error, since no value for `x` has been provided.

However the following call would be valid:

```
someFunc(1, z=99, y=4)

x is 1
y is 4
z is 99
```

In this case the value for `x` has been provided as a *positional* parameter. Hence it is clearly possible to mix non-keyword and keyword arguments within the same call. However, be aware that *keyword arguments* can only appear to the right of any *positional* arguments within a call.

For example, given the following function definition:

```
def showMsg(title, body="", prefix="INFO", suffix="."):
     print(prefix,title,body,suffix)
```

This function could be legally called in all the following ways:

```
showMsg("File opened")
showMsg("File not opened", prefix="ERROR" )
showMsg("File missing", body="Insert Disk", suffix="Press return" )
```

In all cases the parameter `title` must be provided (since it has no default), and the *keyword arguments* must all appear after that value within the call. Any *keyword arguments* not provided use the default specified within the function definition.

TASK: The built-in `print()` function supports a keyword argument called `sep`. This is used to decide what character to display between each of the provided positional parameters. Write some code that makes several calls to the `print()` function while setting the `sep` argument to values other than a space (which is the default).

```
>>> print("One", "Two", "Three", sep="-")

One-Two-Three

>>> print("Apple", "Banana", "Orange", sep=", ")

Apple, Banana, Orange

>>> print("Red", "Green", "Blue", sep=" | ")

Red | Green | Blue

>>>
```

Arbitrary Length Argument Lists

It is possible to define a function that takes an arbitrary number of arguments, i.e. the exact number of arguments is not fixed. The built-in `print()` function does exactly that, and can take a variable number of argument values when called.

We can specify the same type of behaviour when defining our own functions, by using *Tuples*. Tuples are covered in a future lesson, but in many ways are similar to Lists.

We define arbitrary length arguments by prefixing the formal parameter name with an asterisk (`*`) character, which indicates the use of a Tuple.

For example we could define a function that calculates the average of the given parameters as follows:

```
def calcAve(*numbers):
      total = 0
      for num in numbers:
            total += num
      return total/len(numbers)
```

TASK: Enter the example function shown above, then try calling it several times, passing a different number of numeric arguments each time.

```
def calcAve(*numbers):


    total = 0

    for num in numbers:

        total += num

    return total / len(numbers)


# Call the function with different numbers of arguments

result1 = calcAve(5, 10, 15)

result2 = calcAve(2, 8, 4, 6)

result3 = calcAve(7, 3)

print("Result 1:", result1)

print("Result 2:", result2)

print("Result 3:", result3)
```

Note: *variadic* arguments are normally defined last in the formal parameter list (and can only be followed by keyword type parameters).

Lambda Expressions

A Lambda expression is a method of defining a small simple function. Unlike regular functions, lambda expressions are anonymous and can be dealt with as a value. This means they can be stored in variables or passed to functions as arguments.

Lambda expressions can only consist of a single expression, rather than a regular code block. This means they tend to be very small and very concise in nature. The most common use of Lambda expressions is to allow a simple function to be passed as an argument to another function. This allows functions to be very flexible in how they work and allows common functions to be more general purpose.

The Python language has a `lambda` keyword that is used to define a Lambda expression. As with regular function definitions, a Lambda expression includes formal parameters, which are then accessible as local variables within the associated expression. The following example uses a `lambda` expression to define a simple anonymous function (this assumes the `math` module has already been imported).

```
hypot = lambda a,b : math.sqrt(a * a + b * b)
```

Since this expression was assigned to the `hypot` variable it can now be called using that identifier, in the same way as a regular function:

```
>>> hypot(3,4)
5.0
```

TASK: Enter the example lambda expression shown above, then find out the *data type* of the `hypot` variable using a call to the `type()` function. Notice the result.

```
import math

hypot = lambda a, b: math.sqrt(a * a + b * b)

result = hypot(3, 4)

print("Result:", result)

print("Data type of hypot:", type(hypot))
```

The formal parameters within a Lambda expression can include *default* and *variable length* arguments in exactly the same way as a regular function. When a lambda expression is called, *keyword arguments* can also be specified in the same way as when calling a regular function.

TASK: Write a lambda expression that takes two formal parameters, `hours` and `minutes`. The expression should calculate and return the total number of equivalent seconds. Assign the expression to a variable called `to_seconds`, then call the function several times for testing.

```
# Lambda expression to calculate total seconds
to_seconds = lambda hours, minutes: hours * 3600 + minutes * 60

result1 = to_seconds(1, 30)  # 1 hour and 30 minutes
result2 = to_seconds(2, 45)  # 2 hours and 45 minutes
result3 = to_seconds(0, 15)  # 15 minutes

print("Result 1:", result1, "seconds")
print("Result 2:", result2, "seconds")
print("Result 3:", result3, "seconds")
```

Given the sample input shown below, your solution should display the same results -

```
>>> to_seconds(0,2)
120

>>> to_seconds(2,0)
7200

>>> to_seconds(1,30)
5400
```

TASK: Improve your previous lambda expression so that if only one argument is passed within a call, then the number of `minutes` defaults to `0`, as detailed below:

```
to_seconds = lambda hours, minutes=0: hours * 3600 + minutes * 60

result1 = to_seconds(1, 30)  # 1 hour and 30 minutes
result2 = to_seconds(2, 45)  # 2 hours and 45 minutes
result3 = to_seconds(0, 15)  # 15 minutes
result4 = to_seconds(1)      # 1 hour
result5 = to_seconds(2)      # 2 hours
```

```
print("Result 1:", result1, "seconds")

print("Result 2:", result2, "seconds")

print("Result 3:", result3, "seconds")

print("Result 4:", result4, "seconds")

print("Result 5:", result5, "seconds")
```

```
>>> to_seconds(1)
3600

>>> to_seconds(2)
7200
```

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.

- Module
- The Python Standard Library
- Formal Parameters
- Actual Parameters (argument values)
- Default and Keyword Arguments
- Lambda Expression

Module: A file containing Python code for reuse in other programs.

The Python Standard Library: A collection of built-in modules and packages in Python.

Formal Parameters: Parameters in a function's definition.

Actual Parameters (argument values): Values passed to a function during a call.

Default and Keyword Arguments: Default values for parameters and values assigned using parameter names.

Lambda Expression: A concise way to define anonymous functions in Python.

Practical Exercises

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