Introduction to Programming

Lab Worksheet

Week 1

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. There is a video on MyBeckett that shows the various options.

Topics covered:

- Types of Programming Languages
- Using the Python Interpreter
- Entering basic expressions
- Understanding operators and precedence
- Dealing with Errors
- Key programming terminology

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

Types of Programming Language

Over the years programming languages have developed in terms of sophistication and usability. Third generation languages have been the most widely used for many years now, and provide a good balance between usability, portability and efficiency. Fourth generation languages, such as the Structured Query Language (SQL) or R are more abstract and allow developers to specify ‘what’ they want, rather than exactly ‘how’ the program needs to work. Although 4GLs lead to rapid product development, they tend to be more application specific in nature e.g. SQL is used exclusively to define and manipulate data within a database management system. Whereas 3GLs tend to be much more general purpose and applicable to just about any type of application.

Within 3GL languages different programming styles or *paradigms* also exist. These determine the approach to be taken when designing and implementing a solution. Initially all languages supported a ‘procedural’ approach to development, later languages began to support an alternative ‘object-oriented’ approach. Other paradigms include ‘functional’ and ‘aspect-oriented’. Some modern languages, such as Python, are often multi-paradigm by design. This means they provide multiple styles of programming depending on the approach the programmer wishes to take.

Programmers write ‘source code’, which is usually written in a language that makes sense to humans, but not to computers. Hence the ‘source code’ needs to be converted into a form that can be run or ‘executed’ by the computer. Python is a third generation *interpreted* language. This means the tool that we use to execute Python programs is called an *interpreter*, in contrast to either an Assembler (for translation of Assembly language into machine code) or a compiler (for compiling 3GLs such as C or C++ into an executable program).

Interpreted languages are typically translated into an executable form at the time the program is loaded e.g. from a file. Although this incurs a slight performance penalty it means the language can be very *dynamic* in nature, usually leading to a more flexible style of programming and even the ability to change code at run-time.

Although these lessons are using the Python language many of the concepts are applicable to all 3GL type languages. Hence, the skills gained by learning Python make it fairly easy to transfer over to one of the alternative languages should the need arise (which it will). Unlike in the past, it is now common for single applications to be built using multiple languages and technologies. This is especially true when considering web-based, client-server type solutions. The ‘client’ side is often built using JavaScript and HTML/CSS, whereas the ‘server’ side is often built using Java, PHP, C#, Ruby, etc. Therefore the ability to develop in multiple languages is a common requirement in today’s software engineering organisations.

We will be using the CPython interpreter, which is the most commonly used Python interpreter. The program itself is written in C and Python and is freely available to download. We shall be using version 3.x of Python (rather than the older version 2.6).

How to use the Python Interpreter

The majority of this module will be taught using the standard Python Interpreter. Initially we will be using this in *interactive mode*, which allows for rapid input and feedback of small *snippets* of code. You must become familiar with how to use this environment so you are able to complete the various lessons.

When started in *interactive-mode* the Python interpreter supports the **REPL** interaction model. This means as we type commands they are '**R**ead' then '**E**valuated', the result is '**P**rinted' and then this process '**L**oops' again.

Starting the Python Interpreter

The Python interpreter can behave as an interactive-shell. This basically means we work at the command line within a 'shell' that understands the Python language. To start Python we simply type the following at the command line:

```
python3
```

Once started this will produce output similar to the following:

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)

[GCC 8.3.0] on linux

Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

The '>>>' is called the input prompt, and indicates that we are able to type our next command to be executed. Once a command is input it can be executed simply by pressing the **<enter>** key.

TASK: Try inputting and executing the code below:

```
print("the program has executed")
```

If you run the above code correctly, the words **`"the program has executed"`** should have been displayed. Make sure you are typing at the Python Interpreter (check the prompt).

TASK: Try using a similar command to output at least three alternative messages.

Using command history

The Python interpreter remembers the commands that have been previously typed. This is called the ‘command history’. When first learning to program it is very easy to make minor mistakes. The command history means you do not have to re-type already entered text, you can simply recall previous commands using the ‘up’ (↑) and ‘down’ (↓) arrow keys. Once a previous command has been recalled it can be edited (usually to correct any errors) before it is executed again by pressing the **<enter>** key.

TASK: Use the up and down arrow keys to recall the commands you have already input, select one of these then re-execute the command by pressing the **<enter>** key.

Getting help

The Python interpreter has an in-built help system. This provides a convenient way to lookup available commands to check for correct syntax etc. The help system can be used in interactive mode, or in single command mode to provide information about a specific *object*.

To enter interactive help mode type the following command:

```
help()
```

This will produce output similar to the following:

```
Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Notice that when in help mode the input prompt has changed to ‘`help>`’.

By reading the displayed help information you should be able to use the help system at a basic level.

TASK: Use the interactive help system to display a list of available language “keywords”, then quit the interactive help to return back to the Python interpreter.

Hint: always look for the presence of the input prompt ‘`>>>`’ to determine whether the Python interpreter is ready to accept commands.

Rather than use full interactive mode you may often want to lookup information about a specific object or command. This is achieved by providing a name between the brackets when typing the ‘help’ command, e.g. to lookup help on the ‘print’ object enter the following:

```
help(print)
```

This will produce output similar to the following:

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

(END)
```

When used in this way the help system does not enter full interactive help mode, but returns back to the Python interpreter. Note: you may have to press '`q`' to return back to the interpreter.

TASK: Enter a single 'help' command to look up information about the 'input' object then return back to the interpreter.

Quitting the Python Interpreter

Once you have finished using the Python interpreter the program can be closed by typing the following command:

```
quit()
```

Note: You can also use the short-cut **CTRL + D** to quit the interpreter (**CTRL + Z** in Windows).

TASK: Try quitting and then restarting the interpreter several times.

Entering Basic Expressions

Now that you understand how to use the Python interpreter, it is time to input and execute some more commands.

TASK: Try inputting and executing the code below:

```
45 + 20
```

The command you entered is known as an *expression*. Once the **<enter>** key was pressed It was Read, Evaluated and Printed by the interpreter applying the **REPL** interaction model producing the following output:

```
65
```

Expressions are often used when writing larger programs, in order to

1. perform calculations
2. help make decisions about which statements are to be executed (selection)
3. help make decisions about how many times a group of statements are to be executed (iteration)

Expressions consist of ***operands*** and ***operators***. In the above example the *operands* are the numbers 45 and 20 and the *operator* is the addition (+)

Other common *operators* used within expressions include -

`**`	(exponentiation), i.e. raise to the power
`*`	(multiplication)
`/`	(division)
`//`	(floor division)
`%`	(modulus, remainder)
`-`	(subtraction)

TASK: Input then execute the following expressions (note: you will have to re-enter each expression separately). Ensure you understand each *operator* and the result produced.

```
10 + 20 - 15
```

```
10 * 5
```

```
100 / 33
```

```
100 // 33
```

```
10 ** 2
```

```
10 % 3
```

Operator Precedence

Expression evaluation is not always performed left to right. There is an operator *precedence* at work, i.e. operators with higher precedence are evaluated prior to those will lower precedence. You may have met this idea is BODMAS (or BEDMAS) in mathematics.

TASK: To see precedence at work input then execute the following expressions.

```
10 + 5 * 2
```

```
10 - 5 * 10 + 5
```

```
5 * 10 ** 2
```

From the above results, it should be clear that multiplication (`*`) and division (`/`) have a higher precedence than addition (`+`) and subtraction (`-`).

Also Exponentiation (`**`) has a higher priority than both multiplication (`*`) and division (`/`).

Parentheses (curved brackets) can be used to enforce or alter precedence.

TASK: Input and execute the following expressions, then compare the results to those of the previous task.

```
(10 + 5) * 2
```

```
10 - 5 * (10 + 5)
```

```
(5 * 10) ** 2
```

The results should indicate that the evaluation order has changed due to the use of the parentheses.

Incorrect precedence within complex expressions is the source of many *logical errors*, so it is often wise to include parentheses even when not strictly required. This makes code clearer to read and less prone to programming errors. The precedence of the operators we have seen so far is shown below. Those shown higher in the list have higher precedence. Those at the same level will evaluate left to right.

`()`	parentheses
`**`	exponentiation
`*, /, //, %`	multiplication, division, floor division, remainder
`+, -`	addition, subtraction

Also note, that parentheses can be nested (appear within other expressions that include parentheses). In this case the contents of the innermost parentheses are executed first.

TASK: Input and execute the following expressions. Notice the different results.

```
12 + (5 * 2 + 3)
```

```
12 + (5 * (2 + 3))
```

For the first expression, the evaluation order is as follows -

5 * 2 = 10
10 + 3 = 13
12 + 13 = 25

For the second expression (since the inner parentheses are evaluated first) , the evaluation order is as follows -

2 + 3 = 5
5 * 5 = 25
12 + 25 = 37

Errors

When writing computer programs errors are inevitable. Some errors are obvious and simple to fix, others are very difficult to identify and fix. The ability to find and fix errors is a key skill required by any programmer. Before developing this ability you will need to have an understanding of the different types of errors that can occur.

Errors are generally categorized as **syntax errors** or **logical errors**.

Syntax errors are usually the most common, but also the most easy to fix. These sort of errors occur when the programming language syntax has not been used correctly. Most development environments provide some sort of assistance with identifying such errors. Depending on the programming language, it is often not possible to start running a program until all syntax errors are fixed (although this is not the case with some scripting type languages). As a new programmer you will probably experience many syntax errors until you learn the language *grammar rules*. Even experienced programmers regularly make syntax errors - much like having a typo or spelling error in a document.

Logical errors occur during run-time, i.e. as the program is executing and being used (rather than during development). Logical errors occur when a program is syntactically correct, but the underlying algorithm was incorrectly designed or poorly implemented. Logical errors can be very difficult to find, since the development environment typically cannot identify these types of errors. These types of errors are commonly referred to as **"bugs"**. Poorly designed programs containing logical errors do not behave as originally intended. Sometimes logical errors can actually cause program execution to cease, resulting in what is known as a **run-time** error.

Both syntax errors and run-time errors are usually reported by the execution environment (The Python interpreter in our case). Such reporting allows programmers to find these issues during development or testing, and fix the program syntax or logic to avoid the same thing happening in the future.

To see a typical example of a *syntax* error being reported, input and execute the following code:

```
10 +
```

This should report a '*syntax error*' since the right hand *operand* of the expression was missing.

To see a typical example of a run-time error being reported, input and execute the following code:

```
print("Ten divided by zero is", 10/0 )
```

This should report a 'Traceback' message, which contains information to help us identify and fix the problem. In this case the error was caused because we attempted a *division by zero*, which is mathematically impossible.

Run-time errors often occur because of input received from an external source, such as user input from the keyboard. A well written program should always *validate* the input to ensure it is in a form expected by the program. If not, then the data should NOT be used and the program should report the issue in some graceful way. Failure to deal with invalid input is a very common type of logical error. If an error does occur a program should always handle it gracefully, reporting the problem to the user and/or logging the error.

Remember: many logical errors do not result in obvious system failure, and are often NOT reported by the execution environment as a run-time error.

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.

- Source code
- Machine code
- Interpreter
- Compiler
- 2GL, 3GL, 4GL
- Executable
- Expressions
- Operators and Operands
- Syntax Errors
- Logical Errors

Practical Exercises

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

ASSIGNMENTS

TASK: Try inputting and executing the code below:

```
print("the program has executed")
```

```
>>> print("the program has executed")
the program has executed
```

TASK: Try using a similar command to output at least three alternative messages.

```
>>> print("Hello, world!")
Hello, world!
```

```
>>> print("Welcome to python")
Welcome to python
```

```
>>> print("Welcome to programming")
Welcome to programming
```

TASK: Use the interactive help system to display a list of available language “keywords”, then quit the interactive help to return back to the Python interpreter.

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
True                def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
break               for                 not

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
>>>
```

TASK: Try inputting and executing the code below:

```
45 + 20
```

```
>>> 45+20
65
```

TASK: Input then execute the following expressions (note: you will have to re-enter each expression separately). Ensure you understand each *operator* and the result produced.

```
10 + 20 - 15
```

```
>>> 10+20-15
15
```

```
10 * 5
```

```
>>> 10*5
50
```

```
100 / 33
```

```
>>> 100/33
3.0303030303030303
```

```
100 // 33
```

```
>>> 100//33
3
```

```
10 ** 2
```

```
>>> 10**2
100
```

```
10 % 3
```

```
>>> 10%3
1
```

TASK: To see precedence at work input then execute the following expressions.

```
10 + 5 * 2
```

```
>>> 10+5*2
20
```

```
10 - 5 * 10 + 5
```

```
>>> 10-5*10+5
-35
```

```
5 * 10 ** 2
```

```
>>> 5*10**2
500
```

TASK: Input and execute the following expressions, then compare the results to those of the previous task.

```
(10 + 5) * 2
```

```
>>> (10+5)*2
30
```

```
10 - 5 * (10 + 5)
```

```
>>> 10-5*(10+5)
-65
```

```
(5 * 10) ** 2
```

```
>>> (5*10)**2
2500
```

TASK: Input and execute the following expressions. Notice the different results.

```
12 + (5 * 2 + 3)
```

```
>>> 12+(5*2+3)
25
```

```
12 + (5 * (2 + 3))
```

```
>>> 12+(5*(2+3))
37
```

TASK: Identify various errors

```
10 +
```

```
>>> 10+
  File "<stdin>", line 1
    10+
      ^
SyntaxError: invalid syntax
```

```
print("Ten divided by zero is", 10/0 )
```

```
>>> print("Ten divided by zero is", 10/0 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```