Introduction to Programming

Lab Worksheet

Week 2

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:

- Working with variables
- Data-types
- Calling built-in functions
- Using single, double and triple quotes
- Indexing and Slicing
- Introduction to Lists

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

Working with Variables

In the previous lesson we covered evaluation of expressions, involving inputting various expressions and then viewing the result. When writing programs however we need a mechanism of storing values, such as the results of expressions, for later use.

We use the concept of *variables* to store values for later access. We refer to these using *identifiers* which are basically textual names. The programmer decides upon each identifier (or variable name), and it is usually something that indicates what the stored value represents. Hence variable names are usually “nouns”.

Python has specific syntax rules regarding variable names, which restricts the characters that can be used. An *identifier* is a case-sensitive name that consists of letters, digits and underscores (_) but may not begin with a digit.

Examples of good variable names are -

```
total
customer_name
score
result
```

Examples of poor (but legal) variable names are -

```
calcResult
a
fffff
john
```

Since variable names are case sensitive, the following three examples refer to different (and unrelated) variables -

```
age
Age
AGE
```

Variable assignment

A variable is created and initialised using an *assignment* operator, which is the ‘=’ symbol. Do not get this confused with a test for equality; this operator assigns the right hand value to the left. It does not test whether the right and left operands are equal.

Once a variable has been assigned a value, that value can then be accessed using the given name. A variable cannot be used prior to a value being assigned since the variable does not exist until that happens.

TASK: Try inputting the following code and examine the results.

```
total = 100

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

```
>>> total = 100
>>> print("The total is", total)
The total is 100
```

The right hand side of the operand does not have to be a literal value, as in the above example. It can be an expression that can itself refer to existing variables.

TASK: Try inputting the following code and examine the results.

```
total = total + 99
print("The total is now", total)
```

```
>>> total = total + 99
>>> print("The total is now", total)
The total is now 199
```

Note: The assignment operator '=' has a low precedence, which means any expression on the right hand side is evaluated prior to the assignment taking place. In the above example the `total` is accessed, increased by 99, then assigned back to the variable.

Since it is common to refer to the variable itself within an assignment expression a short-cut syntax exists. This is called *augmented assignment*, and has been inherited from the 'C' programming language.

For example, the following code is equivalent to that in the above task -

```
total += 99
print("The total is now", total)
```

Augmented assignment can be performed with most operators. The syntax of these involves moving the operator to before the assignment ('=') symbol and removing the variable name that would normally appear within the expression.

For example, the following assignment -

```
total = total * 100
```

Becomes as follows, when written as an augmented expression -

```
total *= 100
```

Note: Augmented assignment syntax only makes sense when the right-hand expression refers to the variable itself, e.g. the following could NOT be written using an augmented expression -

```
total = value + 99
```

Since the right hand side of the assignment does not refer to '`total`', it refers to '`value`'.

TASK: Try inputting the following code, but replace each of the assignment expressions with the equivalent augmented assignment.

```
total = total - 1
print("The total is", total)

total = total * 4
print("The total is", total)

total = total / 2
print("The total is", total)
```

```
>>> total -= 1
>>> print("The total is", total)
The total is 193.4
>>> total /= 2
>>> print("The total is", total)
The total is 96.7
>>> total *= 1
>>> total *= 4
>>> print("The total is", total)
The total is 386.8
```

TASK: Try extending the code below so that it creates a new variable called '`average`', that is set to equal the average calculated from the two other variables.

```
total = 98.2
count = 5

# add your extra code here
```

```
>>> total = 98.2
>>> count = 5
>>> average = total / count
>>> print(average)
19.64
```

Data-Types

All values used within a programming language such as Python are based on a *data-type*. A data-type indicates the nature of the value, i.e. does the value represent a whole number? A decimal number? A piece of text? etc.

The reason why the data-type is important is to ensure that the operations applied to the values are performed correctly, e.g. the addition operator (‘+’) applied to a number should clearly add the two numbers together, but what should it do when applied to two text-type values?

Also some operations make sense for some types of value, but not others. The computer needs to be aware of the data-type of a value in order to perform operations appropriately, or prevent them from happening at all.

Python has a number of *primitive* data-types, the most commonly used of which are -

`int` - used to store whole numbers, e.g, `100`
`float` - used to store numbers which include a fractional part e.g. `20.23`
`boolean` - used to store `True` or `False` values
`string` - used to store textual data, e.g. `"Dead Parrot"`

A Variable’s data-type

Since variables store values they also can be thought of as having a *data-type*. The current data-type of a variable depends on the last value assigned to the variable. Hence the current value stored by the variable determines the variable’s data-type, and therefore determines how operations are applied to that variable.

The nature of the Python language means that the programmer does not have to explicitly specify the data-type. The interpreter works out the data-type by examining the provided values, for example -

```
total = 10      # total’s data-type is now an integer

total = 10.5    # total’s data-type is now a float

total = True    # total’s data-type is now a boolean

total = "10.5"  # total’s data-type is now a string
```

This approach is known as *dynamic typing*, which means the data-type of a variable is dynamically determined and can therefore change throughout a program. Many other languages use *static typing*, which ensures each variable can only ever store one specific type of value. The fact that variables can change data-type over their life-time can lead to unexpected run-time errors, so care must be taken when programming.

As can be seen in the above example, the syntax of a value determines the data-type. String type values are identified by the presence of quotes (`"`), whereas floats are identified by the presence of a number that include decimal point (`.`) etc.

The Python interpreter provides a way of determining the type of a value in the form of a `type()` *function*, that displays the type of a given object e.g.

```
>>> type(10.5)
<class 'float'>

>>> type(10)
<class 'int'>

>>> total = 10
>>> type(total)
<class 'int'>

>>> type(total * 2)
<class 'int'>
```

Notice how the provided object can be a literal value, variable, or even an expression.

TASK: Use the `type()` *function* to determine the type of each of the following values.

```
False
1000
100.111
"Hello"
True
100 / 5
100 // 5
```

```
>>> type(False)
<class 'bool'>
>>> type(1000)
<class 'int'>
>>> type(100.111)
<class 'float'>
>>> type("Hello")
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(100 / 5)
<class 'float'>
>>> type(100 // 5)
<class 'int'>
```

Note: The division (‘`/`’) operator always returns a float-type value, even when the operands are integer-type values.

As an example of how the data-type is used during evaluation, input in the following code.

```
10 + 10
```

The result of `20` should be no surprise. However, now input the following similar code -

```
"10" + "10"
```

You should see a very different result. That is because the `+` operator, when applied to string type operand values, actually performs a concatenation operation (sticks the strings together) rather than an addition operation. Hence the data-type of values is very important when it comes to expression evaluation.

TASK: Input the following code and examine the result. What is the `*` operator doing to the given string operand?

```
"ABC" * 10
```

```
>>> "ABC" * 10
'ABCABCABCABCABCABCABCABCABCABC'
```

****Causes 'ABC' to be repeated 10 times as concatenation**

Calling Built-in functions

When writing code we often want to do common tasks, such as displaying information to the screen or reading input from the keyboard. Rather than always having to write this code ourselves we can use pre-written code. Within the Python language such pre-written code is represented by *functions*. Functions save us a lot of time as a programmer, since they allow us to execute code that has already been written by someone else. Python provides many such pre-defined functions, as part of *libraries*.

We can *call a function*, by referring to it by name in a similar way that we refer to a variable. However, unlike a variable we include a set of parentheses `()` after the name, which can pass values to be used by the function. These values are often called *arguments* or *parameters*. The number of arguments passed within the parentheses `()` depends on the function being called, and can even vary for the same function.

In the previous section we used the `type()` function to determine the data-type of an object, which was passed as a single *argument* within the parentheses. In other examples we have also seen the `print()` function, which is used to display information to the screen and can take multiple *arguments*.

e.g. the call to the `print()` function below passes two arguments.

```
print("The average was", total/count)
```

Some functions *return* a value, hence it is not uncommon to see them called as part of variable assignments or expressions.

For example, the code below calls the `len()` function to calculate the length of a string value, then stores the returned value in the '`name_length`' variable -

```
name_length = len("Eric Idle")
```

The arguments passed to functions can be literal values, variable names, expressions or even calls to other functions (since they return a value).

For example, within the code below the second argument passed to the `print()` function is the result of calling the `len()` function -

```
print("The length of you name is", len(name))
```

Any expressions or function calls that appear as arguments are executed prior to the outer function being called. So in the above example the `len()` function is called before the `print()` function, allowing the result to be passed as an argument.

TASK: Write some code that calls the `print()` function several times, displaying your name, address and contact details. Add additional calls to the `print()` function which includes an argument that calculates and prints the length of your name, by calling the `len()` function.

Getting input from the user

We often need to read input from the user. We can use the built-in function `input()` to do this. The `input()` function takes an *argument* which is displayed as a prompt to the user. It then returns whatever value the user types as a string value. For example,

```
name = input("What's your name? ")
print("Hello there", name)
```

The fact that the `input()` function only ever returns a string can sometimes cause an issue. What if you want to ask the user to input a numeric type value?

TASK: Input the following code, when asked to type your age input a numeric value such as `20`. Does this program work? If not, why?

```
age = input("Enter your age ")
print("in one year your age will be", age + 1)
```

```
>>> age = input("Enter your age")
Enter your age 20
>>> print("in one year your age will be", age + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

****It gives an error as the input function converts all the values into string as default.**

The problem is that an attempt is being made to apply an addition (`+`) operator to a string type (‘`age`’) and integer type operand, which cannot be done. Although we asked for a number the `input()` function returned a string type value that just happened to contain digits.

We can solve this problem using yet another built-in *function*. In this case we can use the `int()` function, which given a string type argument returns the integer type equivalent. Therefore the above can be re-written as follows -

```
age = input("Enter your age ")
age = int(age)
print("in one year your will be", age + 1)
```

TASK: Write a program that prompts the user to input two numeric values. Once the values have been input display the product of these values, using the multiply (`*`) operator.

```
>>> num1 = input("Enter first number ")
Enter first number 4
>>> num2 = input("Enter second number ")
Enter second number 6
>>> num1 = int(num1)
>>> num2 = int(num2)
>>> print("The product is", num1 * num2)
The product is 24
```

Single, Double and Triple Quotes

It has already been mentioned that string type values are identified by the presence of quotes (`"`). However, within the python language those do not always have to be double quotes.

A string value can actually be delimited by either single (`'`) or double (`"`) quotes. For example, both the following assignments are equivalent -

```
name = "John"
```

```
name = 'John'
```

The main reason this is allowed, is to provide a mechanism for including quotes within strings themselves, e.g.

```
comment = 'I would have "thought" you knew better!'
```

TASK: Try writing the above assignment statement but only use double quotes instead of single quotes as the string delimiter. What is the result?

```
>>> comment = 'I would have "thought" you knew better!'
>>> comment = "I would have "thought" you knew better!"
  File "<stdin>", line 1
    comment = "I would have "thought" you knew better!"
                            ^^^^^^^
SyntaxError: invalid syntax
```

****It gives a syntax error.**

Escape Sequences

Strings can also contain *escape sequences* which are a mechanism for including special characters within a string, i.e. characters that can't normally be typed at the keyboard can be included.

Escape sequences are included by the use of a back-slash (`\`) character followed by a special value. When the string is displayed the escape sequence is replaced by whatever special character was specified, e.g. a newline character can be included in a string using a `\n` escape sequence, e.g.

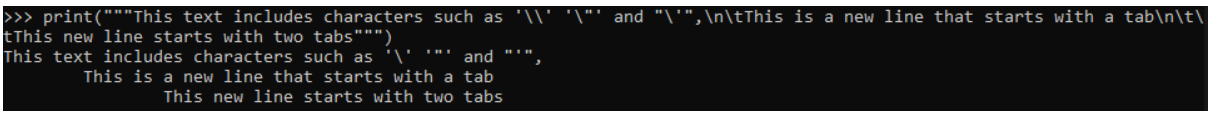
```
>>> print("Hello there,\nWhat is your name?")
Hello there,
What is your name?
```

The escape sequence supported are as follows -

`\n`	Insert a newline character in a string.
`\t`	Insert a horizontal tab.
`\\`	Insert a backslash character in a string.
`\"`	Insert a double quote character in a string.
`\'`	Insert a single quote character in a string.

TASK: Write some code that calls a `print()` function, which takes a single string argument that results in the following text being displayed (exactly as shown).

```
This text includes characters such as '\' '"' and "'",
     This is a new line that starts with a tab
          This new line starts with two tabs
```



TASK: Write some code that calls a `print()` function, which takes a single string argument that results in the following text being displayed (exactly as shown).

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
Hello there!
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

Using Triple Quotes

Within Python a string value can also be delimited by using triple quotes (actually it is triple - double quotes), e.g.

```
message = """This is a triple quoted string"""
```

Triple quoted strings can include single and double quotes, as well as new-lines without using *escape sequences*. For example, the following is a valid single statement -

```
print("""Welcome, "John", it's 'nice' to
See you again
After such a long time""")
```

Triple quoted strings are often used as a convenient way to include multi-line strings within code, and are used to document our own functions (see *docstrings* in a later lesson).

TASK: Write some code that calls a `print()` function, which takes a single string argument that results in the following text being displayed (exactly as shown). Do this without the use of any *escape sequences*.

```
This text spans three lines,
and includes both single (') ,
and double quotes (").
```

```
>>> print("""This text spans three lines,
... and includes both single (') ,
... and double quotes (").
... """)
This text spans three lines,
and includes both single (') ,
and double quotes (").
```

Indexing and Slicing

As we have already seen, strings are a very commonly used data-type. They are of course very different to the other primitive types such as integers and floats, and have their own set of useful operations.

Two very common operations that can be performed on strings are called *indexing* and *slicing*. Both of these operations are achieved by appending square brackets `[ ]` to string type values.

Indexing allows access to single characters within a string. A zero-based character position can be specified within the square brackets that identifies the character to be accessed. For example, to access the first character of a string, an index of `0` can be used as follows -

```
surname = "Palin"
initial = surname[0]
```

TASK: Rewrite the above example, so that the third letter of the '`surname`' is accessed rather than the first, then print this letter to the screen.

```
>>> surname = "Palin"
>>> initial = surname[2]
>>> print(initial)
l
```

If the given index value is out of range i.e. larger than the length of the string, then an error occurs. An index needs to be in the range `0..len(strVal)-1`

TASK: Rewrite the above example, so that the tenth letter of the '`surname`' is accessed, and note the result.

```
>>> initial = surname[9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

The Python language supports the concept of *negative index* values. If the given index is less than zero then it is used as an offset from the right-side of the string, rather than the left-side. For example, to access the last character of a string, an index of `-1` can be used as follows -

```
surname = "Palin"
last = surname[-1]
```

TASK: Rewrite the above example, so that the second from last letter of the '`surname`' is accessed rather than the last, then print this letter to the screen.

```
>>> surname = "Palin"
>>> second_last = surname[-2]
>>> print(second_last)
i
```

As with positive indices, attempting to use a negative index that is out of range will result in an error, e.g. the following would cause an error -

```
surname = "Palin"
val = surname[-6]
```

Slicing

Slicing operations are similar to *indexing* type operations but are more powerful, and able to access multiple characters rather than single characters only.

A *slice* specifies a range of characters within a string, as a *start index* and an *end index*. Since two index values are required a colon (`:`) is used within the square brackets to separate the start and end index values. For example, the following slice would create a sub-string containing the word `"ali"`

```
surname = "Palin"
middle = surname[1:4]
```

Notice how the start indexed value is included, and the end indexed value is excluded.

TASK: Rewrite the above example, so that all of the characters of the '`surname`' except the *first* character are sliced and then displayed on the screen.

```
>>> surname = "Palin"
>>> initial = surname[1:]
>>> print(initial)
alin
```

Both the start and index values may be omitted. If the start index is not present then it defaults to `0` (the first character), and if the end index is not present then it defaults to the length of the string. Therefore a slice that contains all the characters from a string could be accessed using the following -

```
copy = surname[:]
```

However, in most cases at least one or both index values are provided. For example, accessing all characters of a string (except the first) could be achieved as shown below.

```
tail = surname[1:]
```

This works since the end-index is not provided, and therefore defaults to the length of the string.

TASK: Write code that accesses and prints all characters of the '`surname`' except the *last* character.

```
>>> tail = surname[:-1]
>>> print(tail)
Pali
```

As with regular indexing it is possible to use negative indices when *slicing*. These work in the same way as with regular indexing, and identify a position relative to the right side of the string rather than the left. For example, to access the last three characters of a string then following code could be used -

```
last_three = surname[-3:]
```

Or to access all characters except the last three then following code could be used -

```
last_three = surname[:-3]
```

One final point about slicing, is that out of range start or end index values do not cause errors to be reported. Instead anything outside the boundary of the string is simply ignored. For example, the following would NOT cause an error although the length of the string has been exceeded -

```
tail = surname[1:10000]
```

Introducing Lists

A string is essentially a *sequence* of character values that taken together usually have some special meaning. The Python language includes a similar type called a *list*. The values within a list however can be of any data-type, rather than just characters. The values within a list are ordered, hence like a string a list is a *sequence* of values. Therefore similar operations, such as *indexing* and *slicing* can be applied to a list in the same way as a string. The concatenation (`+`) and multiplication (`*`) operators can also be applied to lists in the same way as strings.

A list is written as a comma separated list of values, appearing between square brackets, for example -

```
names = ["Terry", "John", "Michael", "Eric", "Terry", "Graham"]
```

In this case all the values happen to be a string data-type, i.e. it is a list of strings, but that is not a necessity. Notice that within a list the same value can appear more than once, `"Terry"` in this case.

As another example, consider a list containing integer type values -

```
primes = [2, 3, 5, 7, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

It is also possible to have a mixture of multiple data-types within the same list, although in practice this is rarely done since there is another compound type (called *tuples*) designed to support mixed types.

TASK: Write code that uses *slicing* to access then print the first four prime numbers defined within the '`primes`' list given above. Note: you will have to input that list first for testing purposes.

```
>>> primes = [2, 3, 5, 7, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> first_four = primes[0:4]
>>> print(first_four)
[2, 3, 5, 7]
```

Mutable and Immutable types

Although lists and strings are very similar in many ways there is one really important difference, which is that lists are *'Mutable'* whereas strings are *'Immutable'*.

A *mutable* value can be changed after it has been created. So in the case of a list, values can be inserted, existing values can be replaced, and old values can be removed. In contrast to this an *immutable* value can never be changed once it has been created. So in the case of a string, characters can never be inserted, replaced or removed following initialisation.

The concept of *mutable* and *immutable* types is extremely important in all programming languages, and it is essential that the fundamental difference is clearly understood.

Since lists are *mutable* they are slightly more flexible than strings. For example the following code could be used to replace an element within the '`names`' list.

```
names[0] = "Terry, G."
```

This example uses indexing to update a value, rather than simply access a value. Attempting to perform the same type of operation on a string would result in an error, since strings are *immutable*.

The contents of lists can also be mutated (changed) by assigning values to slices. The provided values replace the specified slice. The interesting thing about this approach is that the number of new values does not need to match the number of values within the replaced slice. This allows multiple values to be inserted or removed at any position within the list.

For example, three new names could be inserted at the beginning of the '`names`' list using the following code -

```
names[0:0] = ["Tim", "Bill", "Graeme"]
```

This works by replacing a slice (of zero length) with three new values.

TASK: Write code that uses *slicing* to insert two new names just before the final name within the '`names`' list.

```
>>> names = ["Tim", "Bill", "Graeme"]
>>> names[len(names):] = ["Mark", "Brian"]
>>> print(names)
['Tim', 'Bill', 'Graeme', 'Mark', 'Brian']
```

Values can also be added to the end of a list using the `append()` *method* (note: a *method* is like a function that works on a specific type of object). For example -

```
names.append("Brian")
```

As with strings, the `+` operator can also be applied to lists in order to concatenate them together, e.g.

```
names = names + ["Mark"]
```

Note: This type of concatenation does NOT mutate the existing list but simply creates a brand new list. It is therefore less efficient to add values to a list in this way, compared to using *slicing* or the `append()` method. Also, the following *augmented assignment* DOES mutate the existing list -

```
names += ["Jacob"]
```

Notice how both of the previous examples created a list of a single element, rather than attempting to concatenate a single string type value. Try concatenating a string directly to a list and see what happens.

Lists also support the `*` operator in the same way as strings. This allows the contents of a list to be duplicated a number of times, e.g.

```
nums = [1,2,3] * 5
```

TASK: Work out in your head what the contents of the '`nums`' list would be, then check this using the Python interpreter.

```
>>> nums = [1,2,3] * 5
>>> print(nums)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

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.

- Assignment
- Data-type
- Argument
- Indexing
- Slicing
- Mutable
- Immutable

Practical Exercises

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