Introduction to Programming

Lab Worksheet

Week 3

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:

- Boolean Expressions
- Decision making using ‘if’
- Membership testing
- Logical operators
- Ternary operators
- Iteration using ‘while’ and ‘for’
- Breaking and continuing loops

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

Boolean Expressions

Boolean expressions are often used in conjunction with control statements, such as `if` and `while` to help decide what action is to be taken. Boolean expressions always result in a value with a Boolean data-type (that is, either `True` or `False`)

Relational operators often appear within boolean expressions in order to support *comparison* type operations. These operators are shown below.

Algebraic operator	Python operator	Sample condition	Meaning
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y

TASK: Start the Python Interpreter and input the following expressions, noting each result.

```
10 < 100
```

```
100 != 100
```

```
50 >= 50
```

```
>>> 10 < 100
True
>>>
>>> 100 != 100
False
>>>
>>> 50 >= 50
True
>>>
```

Although these produce Boolean type results, it would be very unusual to see such expressions within a real program, simply because we know the result without letting the

computer calculate this for us. Hence, when used along with control statements, Boolean expressions usually involve at least one *variable* value.

TASK: Input a program that defines a variable called '`age`' that is initialised to your own age. Then type several boolean expressions that compare the variable to see whether it is less than '18', '21' then '31'.

```
>>> age = 20

>>>

>>> # Boolean expressions to compare 'age'

>>> is_under_18 = age < 18

>>> is_under_21 = age < 21

>>> is_under_31 = age < 31

>>>

>>>

>>> print(f"Is age under 18? {is_under_18}")

Is age under 18? False

>>> print(f"Is age under 21? {is_under_21}")

Is age under 21? True

>>> print(f"Is age under 31? {is_under_31}")

Is age under 31? True

>>>
```

Boolean expressions do not have to compare just numeric type values, they can also be used to compare other types.

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

```
"a" < "b"
```

```
"b" < "a"
```

```
"John" < "Terry"
```

```
>>> "a" < "b"

True

>>>

>>> "b" < "a"

False

>>>

>>> "John" < "Terry"

True

>>>
```

In this case the comparisons are done *alphabetically* since the data-type of the values are strings. When comparing string values you need to be aware that all capital letters evaluate to be *less-than* all lower-case letters.

TASK: Try inputting the following code and note the result. Try to work out why the answer is different from the previous expression (look carefully, it *is* different).

```
"john" < "Terry"

>>> "john" < "Terry"
False
>>>

because lowercase "j" comes after uppercase "T" in lexicographic
order.
```

It is even possible to apply relational operators to lists (but less commonly seen), e.g.

```
>>> [1,2,3] < [1,2,3]
False

>>> [1,2,2] < [1,2,3]
True
```

The operands provided to the relational operators must be either based on the same data-type, or be convertible to the same data-type. For example it is possible to compare an integer to a floating point value, since an integer is convertible (and therefore compatible) to a floating point value. However, if the operands that are provided cannot be sensibly compared then an error will occur.

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

```
5 < 10.2
```

```
5 < "Monty"
```

```
5 < "5"
```

```
>>> 5 < 10.2
True
>>>
>>> 5 < "Monty"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
>>>
>>> 5 < "5"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
>>>
```

Notice how the first expression executes as expected even though the operands are of a different data-type. Whereas the second and third expressions result in an error since the data-types are incompatible.

The equality operators are slightly more forgiving however, and rather than report an error due to type mismatch, can return unexpected results. For example, note how the result of the following expression is `False`, rather than an error.

```
>>> 5 == "5"
False
```

This is because Python allows mixed operand types within equality (`==`) and inequality (`!=`) operators. So although no error is reported on the above example the result is '`False`' because an integer and a string are fundamentally different values (even if they look the same in the code)

This is often a cause of unexpected logical errors within Python programs. Although in this case it may be obvious that one value is an integer, and the other is a string, this distinction is not always easy to notice when using variables (which is the more usual case).

For example, the expression in the following program would never return `True` since the `input()` function always returns a string.

```
age = input("Enter your age: ")
18 == age
```

Hint: Always ensure you understand exactly what the data-types of the operands are when writing expressions that contain relational operators. Mismatching types will either result in a run-time error being reported, or even worse a logical error that is not reported and will be fiendishly difficult to find..

Logical Operators within Expressions

If a Boolean expression needs to evaluate more than one condition then logical operators can be used. These allow multiple comparisons to be combined. Python provides three logical operators, called '`and`', '`or`' and '`not`'.

- `and` returns `True` if both operands evaluate to `True`
- `or` returns `True` if either operand evaluates to `True`
- `not` returns `True` if the operand evaluates to `False` (and vice-versa)

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

```
age = 30

age >=18 and age <=65
age <18 or age >65
not age > 18

>>> age = 30
>>>
>>> age >=18 and age <=65
True
>>> age <18 or age >65
False
>>> not age > 18
False
>>>
```

Multiple logical operators can also be used within a single expression. In these situations it is often wise to include parentheses to ensure the evaluation precedence is clear, e.g.

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

```
age = 30

(age >=18 and age <=65) and (not age==30)

>>> age = 30
>>>
>>> (age >=18 and age <=65) and (not age==30)
False
>>>
```

The above expression would have produced the same result even if the parentheses had been omitted, since the `not` operator has a higher precedence than the `and` operator. However, adding the parentheses makes this explicit to the reader.

Chaining relational operators

Python allows relational operators to be chained. Chaining relational operators is equivalent to using the logical `and` operator, so these two expressions are equivalent:

```
18 < age <= 65

18 < age and age <= 65
```

TASK: Try inputting two expressions that use operator chaining and are equivalent to the two expressions shown below. (note: you may first want to first assign values to the '`weight`' and '`height`' variables for testing purposes)

```
100 < weight and weight < 200

height > 131 and height < 160

>>> weight = 150
>>> height = 140
>>>
>>>
>>> expression1 = 100 < weight < 200
>>> expression2 = 131 < height < 160
>>>
>>>
>>> print(expression1)
True
>>> print(expression2)
True
>>>
>>>
```

Membership Testing

Python provides a very simple mechanism for testing whether a specific value appears within a compound type, such as a *string* or *list*.

Such membership tests are often used as an alternative to, or in combination with, relational operator expressions. The `in` and the `not in` operators perform membership testing.

For example, the `in` operator can be used to test if a value is within a specific list.

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

The `not in` operator can test whether a value is NOT within the list, for example -

```
>>> "Mark" not in names
True
```

When the right-hand operand value is a string, then membership tests will return `True` if a substring is present, for example -

```
>>> message = "Hello there, my name is John"
>>> "nam" in message
True
```

TASK: Input the examples above but with alternative operand values, that result in both `True` and `False` results.

```
>>> message = "Hello there, my name is John"

>>>

>>> # Alternative values

>>> substring1 = "nam"  # True case

>>> substring2 = "xyz"  # False case

>>>

>>> # Check if substrings are in the message

>>> result1 = substring1 in message

>>> result2 = substring2 in message

>>>

>>> print(result1)

True
```

```
>>> print(result2)

False

>>>
```

Expressions: Points to note

Expressions are used extensively throughout most programs, thus a thorough understanding of how to write these is of fundamental importance.

Arithmetic expressions are used to calculate values, whereas boolean expressions are used to help support decision making within control statements such as `if` and `while`.

Expressions are a very common source of logical errors, so always ensure you understand the operator precedence that is being applied, and add parentheses to make this explicit. Also always ensure you understand the current data-type of each variable being used within an expression.

Be aware that dynamically-typed languages such as Python allow a variable's data-type to change at run-time (depending on the most recently assigned value), thus an expression may work correctly in some circumstances, but not others. This is a common argument for using more statically-typed languages such as Java, C, C++ and C#.

The 'if' statement

The `if` statement provides a mechanism to implement the concept of *selection*. It relies on the result of a condition to decide whether code should be executed or not. The condition is often written as a Boolean expression. If the condition evaluates to `True` then the code provided within the associated block is executed, otherwise it is not executed (skipped).

A code block is associated with `if` statements using *indentation*. All statements that are indented by the same amount are seen as being part of that particular block.

The general syntax of an `if` statement is as follows:

```
if <condition> :
      <statement 1>
      <statement 2>
      ...
      <statement n>
```

For example an `if` statement may look like the following:

```
if age > 100:
     print("you are very old,")
     print("well done!")
```

In this example both the indented print statements are part of the code block, and only executed when the condition within the `if` statement evaluates to `True`.

Note: the Python interpreter recognises indented code blocks, and will wait for the whole statement to be input before executing the code.

TASK: Try writing an `if` statement that checks if someone is between the ages of 18 and 30 inclusive. If they are, then print a message saying `"you are still young!"`

```
>>> age = 20

>>>

>>>

>>> if 18 <= age <= 30:

...     print("You are still young!")

...

You are still young!

>>>
```

Using the ‘else’ clause

An `if` statement can optionally include an `else` clause. This allows a code block to be provided that should execute if the condition does NOT evaluate to `True`. Therefore when an `if` statement includes an `else` clause then one or the other code block executes, but never both.

For example an `if` statement with the `else` clause may look like the following -

```
if age > 100:
     print("you are very old,")
     print("well done!")
else:
     print("you are not very old - yet")
```

Using the ‘elif’ clause

If there are multiple conditions that need to be checked then `if` statements can be chained using one or more `elif` clauses. These must follow the initial `if` statement. If the optional `else` clause is provided it must always appear last.

Chaining `if` statements in this way only makes sense if only one code block out of many needs to be executed.

An example of using an `elif` statements is shown below:

```
if age > 100:
     print("you are very old,")
     print("well done!")
elif age > 80:
     print("you are fairly old")
     print("pretty good!")
elif age > 40:
     print("you are middle aged")
     print("never mind")
else:
     print("you are not very old - yet")
```

TASK: Try writing an `if` statement similar to the last example that includes an extra `elif` clause to check ages between 30-40. Print a suitable message in the associated code block.

>>> age = 35

>>>

>>>

>>> if age > 100:

... print("You are very old, well done!")

... elif age > 80:

... print("You are fairly old.")

... print("Pretty good!")

... elif age > 40:

... print("You are middle-aged.")

... print("Never mind.")

... elif 30 <= age <= 40:

... print("You are in your thirties.")

```
...     print("This is a unique time!")

... else:

...     print("You are not very old - yet.")

...

You are in your thirties.

This is a unique time!

>>>
```

Non-boolean conditions

The condition within an `if` statement does not strictly need to be a Boolean expression (unlike languages such as Java). If the expression results in a numeric value then a zero equates to `False` and a non-zero equates to `True`, for example:

```
if total:
     print("Total is non-zero")
else:
     print("Total is zero")
```

Conditions that result in *string* or *list* type values can also be used. In this case an empty sequence equates to `False` and a non-empty sequence equates to `True`, for example:

```
name = input("Enter your name: ")
if name:
     print("Your name is", name)
else:
     print("Name not entered")
```

Hint: It is often better to write a condition as a Boolean expression, since this leads to clearer code.

TASK: Rewrite the above code that inputs a name then prints a message, but change the condition so it explicitly uses a Boolean expression. Use the example below to help.

```
name = input("Enter your name: ")

if bool(name):

    print("Your name is", name)

else:

    print("Name not entered")
```

```
if total != 0:
     print("Total is non-zero")
else:
     print("Total is zero")
```

The Ternary Operator

A ternary operator is similar to an `if` statement but is written on a single line. Rather than execute blocks of code a ternary operator evaluates and returns one of two possible values. The first value is returned if a given expression evaluates to `True`, and the second value is returned if it evaluates to `False`, e.g.

```
highest = a if a > b else b
```

Notice how the first value `a` appears before the `if` keyword, and the second value `b` appears after the `else` keyword. The condition in this case is the expression `a > b`.

The above example is equivalent to the following `if` control statement:

```
if a > b:
    highest = a
else:
    highest = b
```

Unlike an `if` control statement a ternary operator must always include the `else` keyword and usually appears on a single line. They are very concise and particularly useful when being used as part of a larger statement or expression, e.g.

```
print("a if the highest" if a > b else "b is the highest")
```

This example prints the first string when `a > b` evaluates to `True` or prints the second string when it evaluates to `False`.

TASK: Rewrite the code shown below as a single line Ternary expression.

```
if age >= 18:
     print("you are an adult")
else:
     print("you are not an adult, yet!")

print("You are an adult") if age >= 18 else print("You are not an
adult, yet!")
```

Using ‘while’ and ‘for’ loops

The Python language provides various mechanisms to support *iteration*, which are commonly known as *loops*.

The `while` statement allows a code block to execute repeatedly while a condition remains `True`. As with the `if` statement, the condition is often written as a Boolean expression.

The general syntax of a `while` statement is as follows:

```
while <condition> :
      <statement 1>
      <statement 2>
      ...
      <statement n>
```

For example a `while` loop may look like the following:

```
count = 10
while count > 0:
      print("Countdown is ", count)
      count -= 1
```

In this example both the indented statements are part of the code block, and only executed *while* the condition evaluates to `True`. Once the condition becomes `False` the code block no longer executes.

Since the condition is checked *before* the first execution of the code block, it is possible for such a loop to iterate zero times, i.e. the code does not even execute once.

As well as using the `while` statement, iteration can be achieved within Python by using the `for` statement, which iterates over a sequence of values, such as the elements within a *string* or a *list*.

For example a `for` loop may look like the following:

```
for n in [2, 4, 6, 8, 10]:
      print(n, "multiplied by", n, "equals", n * n)
```

TASK: Input and execute a `for` loop that iterates over a list of four names, printing each of them to the screen.

>>> names = ["Ram", "Shyam", "Sita", "Gita"]

>>>

>>> # Iterate over the list and print each name

>>> for name in names:

```
...     print(name)

...

Ram

Shyam

Sita

Gita

>>>
```

The range() function

When using `for` to iterate over a set of numeric values it is often more convenient to use the `range()` function, which allows a range of values to be specified. The range function generates an *arithmetic progression* between two boundaries with an optional 'step' value.

If only a single argument is provided then the range starts at `0` and progresses to the given value `-1`, so to iterate over the values between `0` to `4` we could use the following:

```
for n in range(5):
     print(n, "doubled is", n * 2)
```

A lower bound can also be specified, so to iterate over the values between `10` to `19` we could use the following:

```
for n in range(10, 20):
     print(n, "doubled is", n * 2)
```

Finally, a 'step' value can be included. This allows the generated values to either increase (or decrease) by values other than 1. So to iterate over the values from `20` down to `1` we could use the following:

```
for n in range(20, 0, -1):
     print(n, "doubled is", n * 2)
```

TASK: Input and execute a `for` loop that uses a `range()` function to generate the following output:

```
2 to the power of 2 = 4
4 to the power of 4 = 256
6 to the power of 6 = 46656
```

```
8 to the power of 8 = 16777216
10 to the power of 10 = 10000000000

>>> for i in range(2, 11, 2):
...     result = i ** i
...     print(f"{i} to the power of {i} = {result}")
...
2 to the power of 2 = 4
4 to the power of 4 = 256
6 to the power of 6 = 46656
8 to the power of 8 = 16777216
10 to the power of 10 = 10000000000
>>>
```

Using 'break' within a loop

Loops can be exited early by using the `break` statement. When encountered within a loop, the `break` statement causes the loop to terminate immediately, without the need to check the looping condition. This is known as *breaking out* of a loop and is typically done within a nested `if` statement's code block.

Being able to break out of a loop early is convenient when some condition occurs indicating that the remaining iterations are not required. For example, the following code will break out of the loop early if the condition indicates that `value` is found not to be a prime number, hence there is no reason to finish the remaining iterations.

```
value = int(input("enter a number: "))
for n in range(2, value//2):
     if value % n == 0:
          print(value,"is not a prime number")
          break
```

An optional `else` clause can be associated with loops that contains a code block which is only executed when the loop terminates normally, i.e. not due to a '`break`' statement. For example, the code below shows a slight improvement to the previous example that ensures an appropriate message is shown if the loop terminated as normal.

```
value = int(input("enter a number: "))
for n in range(2, value//2):
     if value % n == 0:
          print(value,"is not a prime number")
          break
else:
     print(value, "is a prime number");
```

Using 'continue' within a loop

A `continue` statement affects a loop in a slightly different way to a `break` statement. Rather than cause a termination, a `continue` causes the next iteration to begin immediately, without executing any subsequent statements within the code block.

For example, the following code only executes all statements within the loop if a grade is a pass, otherwise it skips the remaining statements and continues onto processing the next grade.

```
grades = [20, 50, 43, 33, 90, 15]
pass_mark = 40
passes = 0
total = 0
for grade in grades:
      if grade < pass_mark:
            continue
      passes += 1
      total += grade

print("average pass mark was", total/passes)
```

TASK: Input code containing a `for` loop that iterates over a list of numbers, printing a running total during each iteration. You may wish to first define the `numbers` list as follows:

```
>>> numbers = [10, 20, 30, 90, 200, 30, 22, 11]
>>> running_total = 0
>>> for num in numbers:
...     running_total += num
...     print(f"Running total: {running_total}")
...
Running total: 10
Running total: 30
Running total: 60
Running total: 150
Running total: 350
Running total: 380
Running total: 402
Running total: 413
```

```
>>>
```

```
>>> numbers = [10, 20 , 30, 90, 200, 30, 22, 11]
```

TASK: Amend your previous solution so that if any value within the list is found to be over 100 then the loop should `break` immediately.

```
>>> numbers = [10, 20, 30, 90, 200, 30, 22, 11]
>>>
>>> running_total = 0
>>> for num in numbers:
...     running_total += num
...     print(f"Running total: {running_total}")
...     if num > 100:
...         print("Breaking loop - number over 100")
...         break
...
Running total: 10
Running total: 30
Running total: 60
Running total: 150
Running total: 350
Breaking loop - number over 100
>>>
```

TASK: Amend your previous solution once again, so that the message "`all numbers processed`" is printed when the loop completes, but only if all values were less or equal to 100 (i.e. the loop did not break early)

```
numbers = [10, 20, 30, 90, 200, 30, 22, 11
running_total = 0
number_over_100 = False
```

```
for num in numbers:
    running_total += num
    print(f"Running total: {running_total}")

    if num > 100:
        print("Breaking loop - number over 100")
        number_over_100 = True
        break

# Check if all numbers were less or equal to 100
if not number_over_100:
    print("All numbers processed")
```

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.

- Boolean Expression
- Relational Operator
- Logical Operator
- Operator Chaining
- Ternary Operator
- Iteration
- Nested Loop

Boolean Expression: An expression that evaluates to either true or false, often involving the use of relational and logical operators.

Relational Operator: An operator used to establish a relationship between two values, such as equality (==), inequality (!=), greater than (>), etc.

Logical Operator: An operator used to perform logical operations on boolean values, including AND (&&), OR (||), and NOT (!).

Operator Chaining: Using multiple operators consecutively in an expression, where the result of one operation becomes an operand for the next.

Ternary Operator: A concise way to write an if-else statement in a single line, using the syntax condition? expression_if_true : expression_if_false.

Iteration: The repetition of a set of instructions or statements in a program, typically facilitated by loops, allowing the execution of code multiple times.

Nested Loop: A loop that is placed inside another loop. The inner loop will run multiple times for each iteration of the outer loop, enabling more complex iteration patterns.

Practical Exercises

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