Introduction to programming

Making decisions and repetition

if statements

Last week you learnt about using operators such as less than, greater than or equals to check two values. The result of this **expression** or **condition** is either **True** or **False**. Within most types of programming you can make a program do different things depending on the result of the condition you check. For example, if a user enters a correct password, log them in.

```
password = raw_input('Enter your password:')
if password == 'letmein':
    print "Password correct"
    # Do more login actions...
# Program continues…
```

To achieve this in Python, you can use an **if** statement. If you run the above code you will see that nothing is printed unless you enter **letmein** as the password when prompted.

The format of **if** statements are:

```
if <condition>:

<tab> Perform these actions.
```

So in the previous example, **password == ‘letmein’** is the Boolean condition where Python checks if the variable **password**, which holds the input from the user, is equal to the string **‘letmein’**. If this is true then the lines of code which are indented with a tab underneath are executed. If the condition is false, the program continues and skips the indented block of code.

Indentation of code

Indenting code after using an **if** statement for example in Python is mandatory. Other languages do not have this format and instead use curly braces { }. For example, the same login if statement as above written in Java would like this:

```
if(password.equals("letmein"))
{
System.out.println("Password correct");
}
```

Java will know that if the Boolean condition is true then the lines of code within the curly braces will run.

if, elif, else

Most of the time you will want to perform an action if a Boolean condition is true and perform another action if it is false. For example, if a user enters a correct password log them in, otherwise alert them to a failed log in. To handle false results from an if statement the **else** clause can be used:

```
password = raw_input("Enter your password:")
if password == "letmein":
    print "Password correct"
    # Do more login actions...
else:
    print "Login failed"
# Program continues...
```

If you try running the above program and enter a response other than **letmein** you will see the **Login Failed** message is displayed. This is because the value of password will not be equal to **letmein**. The result of the Boolean condition will be **false** so the statements after the **else** keyword will be used and the other lines of code will be skipped.

We can also add additional Boolean conditions within the overall if statement. In the below example there is an **elif** keyword which checks the user_type variable again to see if is equal to the string **power**.

```
user_type = "power"
if user_type == "admin":
    print "You are logged in as an admin user."
elif user_type == "power":
    print "You are logged in as a power user."
else:
    print "You are logged in as a normal user."
```

Try running the above code and you will see the second message being displayed. This is because the first Boolean condition is false as the string does not equal **admin** so Python checks the variable again to see if it is equal to **power**. This time it is true so the message is printed to the screen. Notice how the other messages including the one in the **else** statement are not displayed.

Using this if… elif… else… format you can create complex statements that perform different actions based on a variable or some other input.

Ternary operator

Most programming languages allow you to compact if statements into one line of code. This is called a **ternary operator**. Python's approach is a little unconventional but the above if… else… code for the password login program would look like this using a ternary operator:

```
password = raw_input("Enter your password:")
print "Logged in." if password == "letmein" else "Login Failed."
```

In other languages the format would be similar to the below line using **?** to perform an action when the condition is true and **:** when the condition is false.

```
print password == "letmein" ? "Logged in." : "Login Failed."
```

Lists

Having a variable allows you to store a value in the computer's memory and then access the value by using the variable's name. When storing multiple values it can become messy to use a separate variable name for each value. For example if you were storing a student's school subjects:

```
subject1 = "Chemistry"
subject2 = "Maths"
subject3 = "Physics"
subject4 = "Drama"
```

This is fine for a few values but imagine if there were 10's or 100's of values to store. To make life easier you can store these values in a **list.** To create a list you need to specify a variable name and then provide the values in square brackets **[]**, separated by commas.

```
subjects = ["Chemistry", "Maths", "Physics", "Drama"]
```

This makes much less work for the programmer and also makes it easier to read for other programmers. To access one of the values in the list you specify the index of the item you want. For example to print out **Chemistry**:

```
print subjects[0]
```

You need to use zero as the index to access the **Chemistry** value as the list index in most programming languages, including Python, starts a **0** for the first value. So to access **Physics** use 2 for the index:

```
print subjects[2]
```

Create your own list and try printing out the different values to get used to working with lists.

The term **lists** is something fairly specific to Python. Most other programming languages refer to lists as **Arrays**. The two terms are usually fairly synonymous and they do the same thing. This is how an array would be created and accessed in Java:

```
String[] subjects = {"Chemistry", "Maths", "Physics", "Drama"};
System.out.println(subjects[0]);
```

Range

You can put any values you like into lists. For example strings, objects and numbers would all be created and accessed in the same way as the examples above. Python has a built in function called **range** which can be used to quickly create lists of numbers. So if you wanted to create a list of numbers from 0 to 10, you could do it either way:

```
numbers = range(10)

numbers = [0,2,3,4,5,6,7,8,9]
```

If you provide two values to the range function, you will generate a list of numbers between the values. The code:

```
fivetoten = range(5,10)
```

Would give the values:

```
[5,6,7,8,9]
```

Notice how the last number is not included. The number 10 is not included above so you would need to put the second value to 11.

Repetition of code (looping)

When programming, quite regularly you will want to repeat a certain statement or group of statements a number of times. This can be done by copying the code and adjusting it slightly each time if necessary but this is not a very DRY way of coding and can create messy programs.

To reduce writing the same or similar lines of code more than once you can use a loop which will repeat a set of statements several times. To write a loop, you can use the **for** keyword.

```
subjects = ["Chemistry", "Maths", "Physics", "Drama"]
for sub in subjects:
    print "-" * 10
    print sub
```

In the above example, there is a subjects list containing some strings and the **for** statement can be read as:

*" For every value in the **subjects** list, run the below code, setting the variable **sub** equal to the current value from the list, then move on to the next value."*

Trying running the above program and you will see that all the subjects are printed out along with line of dashes.

Another example, using **for** to loop over a list of numbers:

```
for num in range(1,11):
    print num, "x 7 = ", num * 7
```

Try running the above for loop and you will see that you will produce the seven times table.

As long as you provide the **for** loop with a list of some values, it does not matter what they are. Likewise, the name you give the variable that is assigned the current value (**num** in the above example) does not matter either although it is helpful to give it a fairly descriptive name.

While loops

As an alternative to for loops, which will loop a finite amount of times through a list provided, you can use a while loop which will continue to repeat code until a certain Boolean condition has been met. A while loop will execute the lines of code underneath it whilst the condition provided is **True**.

```
counter = 0
while counter < 10:
    counter +=1
    print counter
```

The above program is an alternative way of printing out the numbers 1 to 10. The while statement checks if the Boolean condition is **True** i.e. is the value of counter less than 10 which initially, it is. Therefore the block of code below that adds one to the counter variable and prints it out will run. The program will then loop back to the top of the while loop and check the Boolean condition again. Eventually the counter variable will reach 10 and the condition will no longer be true and hence the loop stops.

It is worth noting that if the condition does not start of as being **True** the code underneath the while statement will never run. For example if counter was set to 10 or higher initially:

```
counter = 20
while counter < 10:
    counter +=1
    print counter
```

There would not be any output from the while loop.

The above example could be neater coded as a for loop as there is a finite number of times the loop will run (assuming the condition is initially true). While loops are really useful when the outcome of the Boolean condition is not known. For example when taking user input:

```
while raw_input("What number am I thinking of?") != '7':
    print "Not quite! Try again."
```

The above example repeatedly asks for user input and checks whether the value the user enters does not equal **7**. If the value is not equal, the condition is **True** so the code below the while statement is executed, in this case a simple print statement. When the user enters **7** the condition is false (7 != 7 is **false**) so the while loop finishes.

Infinite loops

As a while loop repeats when a condition is true it is possible that it can repeat a large or even infinite number of times. With the number guessing program above it is likely at some point that the user will guess the number but under extreme situations or if there are logical errors in the program, an infinite loop may be created.

```
counter = 0
while counter < 10:
    counter -= 1
    print counter
```

With the above program, the while loop will continue until the counter variable 10 or greater. However, in the code that gets repeated, the counter variable is **decremented** by 1 each loop so the condition will never become false. Try running the above program and you will see that Python will happily carry on subtracting 1 from counter and displaying the result until you force it to stop (**CTRL-C** in case you were wondering).

This would likely be an error in a programmer’s logic i.e. they made a mistake in telling Python what to do with the counter variable. Identifying infinite loops can be tricky depending on what Boolean condition is being checked but if there is part of your program that gets repeatedly stuck when it is executed, you would need to check the logic in your while loops.