www.thegordon.edu.au

Python Programming

Section 5 – Repetition, WHILE and FOR loops

UEE60411 Advanced Diploma of Computer Systems Engineering

UEENEED103A Evaluate and modify Object Oriented code programs

For further information:
Steve Gale
ICT – Science and Technology
Ph: (03) 5225 0940
Email: sgale@gordontafe.edu.au

1 – Repetition

As we have already looked at SEQUENCE and SELECTION, it is time to look at the last broad programming structure: REPETITION, or "looping".

There are two types of looping structure in Python; the **while** loop, and the **for** loop. They achieve similar things, but have some differences in the way they are executed, so we'll be looking at them individually.

The WHILE loop.

The basic structure of a **while** loop is that it will test a condition at the start, and whilever that condition is **true**, it will continue to execute the code within it. Remember that all the code you want to execute inside the **while** loop needs to be **indented** inside the loop.

The syntax is as follows:

```
while condition = true:
      execute instruction
      execute instruction
      execute instruction
      …
      update condition
```

Note that last line: "update condition". This is extremely important in a **while** loop. Ask yourself this: if the `condition = true` at the start, what will happen if you don't update the condition as part of the loop?

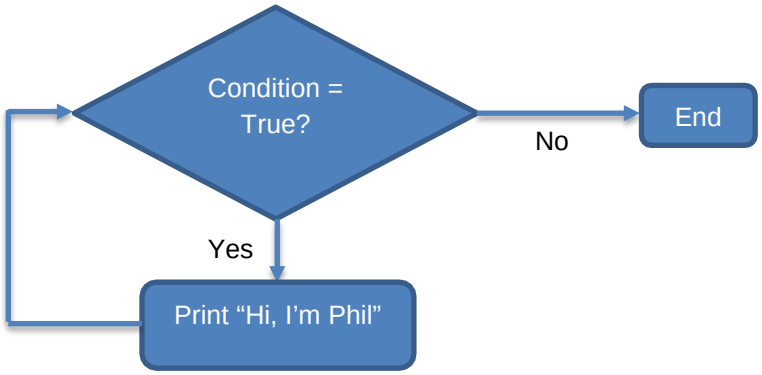
Enter the following eample code (as **"while_loop_demo1.py"** in your editor and execute it to see:

```
3  while True:
4      print("Hi, I'm Phil.")
```

What happens?

It may be easier to visualise this with a simple flowchart:

The "Yes" branch will always execute because the condition is always true – there is nothing that will change the condition to "False".



Here is a slightly more useful example of a **while** loop. In it, we ask the user to input a number, add that to a total, and then compare the total to a pre-coded maximum. If the total is below that maximum, we iterate through (repeat) the loop again until the total exceeds the maximum, at which time the program ends.

One of the reasons we use a **while** loop here is that there is no way of knowing in advance how many iterations of the loop will be necessary. The user might enter multiple "1" or "2" values, or they might enter "100" the first time.

Create a program called **"while_loop_demo2.py"** and code it as follows, then test it out.

```
total = 0
num = 0

num = (int(input("Enter a whole number between 1 and 100: ")))

total += num

while total < 100:
    num = (int(input("Enter a whole number between 1 and 100: ")))
    total += num

print("Total is {0}".format(total))
```

Note how we actually get a value *before* the loop begins? This is called a **priming read**, and needs to be repeated again within the loop.

The following are a series of exercises that will get you used to writing some **while loops**.

WHILE loop exercises

Write programs to perform the following tasks using a **while loop.**

1. **while01.py** – display the whole numbers between 1 and 20, on a single line, separated by commas.

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
```

2. **while02.py** – display the odd numbers between 0 and 20, one line per number.

```
1
3
5
7
9
11
13
15
17
19
```

3. **while03.py** – calculate and display the first 10 multiples of 5

```
5
10
15
20
25
30
35
40
45
50
```

4. **while04.py** – calculate and display the sum of the even numbers from 2 to 20, inclusive.

```
Sum of even numbers 2 to 20 is: 110
```

5. **while05.py** – input a positive integer. Display the odd integers between 1 and the number entered, inclusive.

```
Enter a positive integer: 28
Odd integers between 1 and 28
1
3
5
7
9
11
13
15
17
19
21
23
25
27
```

6. **while06.py** – calculate and display the average of 5 student test scores entered from the keyboard.

```
Enter test score: 68
Enter test score: 55
Enter test score: 87
Enter test score: 75
Enter test score: 59
Average result: 68.8
```

The FOR loop.

A **for loop** achieves a similar goal to a **while loop** – it repeats a series of instructions. The biggest difference is that a **for loop** is used when the number of iterations is known *before* the loop executes. In other words, you don't use a True/False condition like you do with a **while loop**.

The syntax is as follows:

```
for num in range(100):
      execute instruction
      execute instruction
      execute instruction
      …
```

Note we don't have to include a condition update in a **for loop**, as the loop will iterate a set number of times.

As an example, similar to the one we used earlier, create a program called **for_loop_demo1.py** and enter the following code, then execute it.

```
for num in range(10):
    print("Num's value is currently {0}".format(num))
```

You should get the following output:

```
Num's value is currently 0
Num's value is currently 1
Num's value is currently 2
Num's value is currently 3
Num's value is currently 4
Num's value is currently 5
Num's value is currently 6
Num's value is currently 7
Num's value is currently 8
Num's value is currently 9
```

The `range()` function we use here is an incredibly useful tool available in Python. We can easily test a variety of ranges with a simple **for loop**. In this instance, by putting a single argument in the range function (10, in this case) we will test that number of values, starting at 0. So in this case, `range(10)` means 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

We can add two aruments in there to test ranges from start to (one more than) finish. Modify the above code to the following, and you will see the result:

```
for num in range(2, 11):
    print("Num's value is currently {0}".format(num))
```

```
Num's value is currently 2
Num's value is currently 3
Num's value is currently 4
Num's value is currently 5
Num's value is currently 6
Num's value is currently 7
Num's value is currently 8
Num's value is currently 9
Num's value is currently 10
```

Note that we still finish 1 **less than** the "top end" of the range.

You can also force the `range()` function to increment by a value other than 1 by adding a *third* argument, which is the number of "steps" to increment the range by. Let's create a program that will count up by 3s from the value 0 to 30.

```
for num in range(0, 31, 3):
    print("Num's value is currently {0}".format(num))
```

Which should give this result:

```
Num's value is currently 0
Num's value is currently 3
Num's value is currently 6
Num's value is currently 9
Num's value is currently 12
Num's value is currently 15
Num's value is currently 18
Num's value is currently 21
Num's value is currently 24
Num's value is currently 27
Num's value is currently 30
```

Note that I've used 31 as the top of the range values. What would happen if I used 30? Try it and see.

FOR loop exercises

Write programs to perform the following tasks using a **FOR loop.**

1. **for01.py** – display the whole numbers between 1 and 20, on a single line, separated by commas.

```
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
```

2. **for02.py** – display the odd numbers between 0 and 20, one line per number.

```
1
3
5
7
9
11
13
15
17
19
```

3. **for03.py** – calculate and display the first 10 multiples of 5

```
5
10
15
20
25
30
35
40
45
50
```

4. **for04.py** – calculate and display the sum of the even numbers from 2 to 20, inclusive.

```
Sum of even numbers 2 to 20 is: 110
```

5. **for05.py** – input a positive integer. Display the odd integers between 1 and the number entered, inclusive.

```
Enter a positive integer: 28
Odd integers between 1 and 28
1
3
5
7
9
11
13
15
17
19
21
23
25
27
```

6. **while06.py** – calculate and display the average of 5 student test scores entered from the keyboard.

```
Enter test score: 68
Enter test score: 55
Enter test score: 87
Enter test score: 75
Enter test score: 59
Average result: 68.8
```