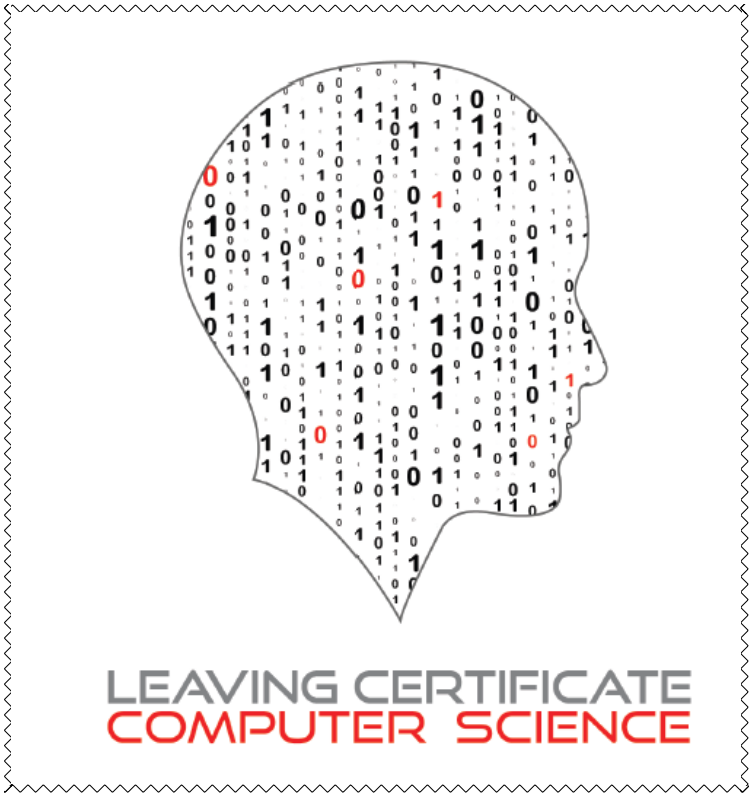
LC CS Python

Student Manual



Section 5

Programming Logic 2

While and *For* loops

Name: _____

Iteration (`for` and `while` loops)

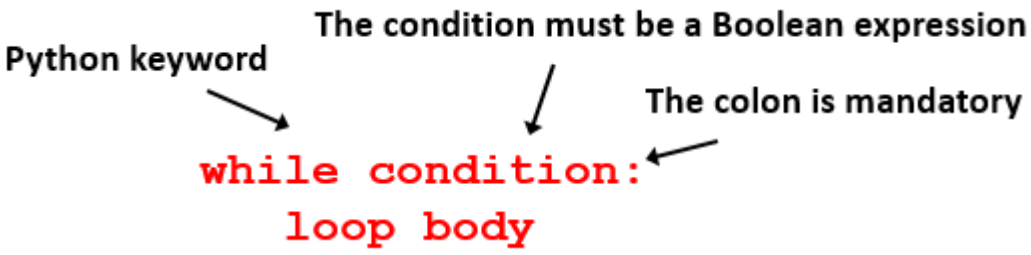
Iteration is a programming technique that allows programs to execute statements multiple times. Python provides built in supports for two different kinds of iteration statements:

- the `while` loop
- the `for` loop

We now consider these in turn.

The `while` loop

This is Python's most general (and therefore) flexible loop construct.
The syntax and semantics of Python's `while` loop are illustrated and described below.

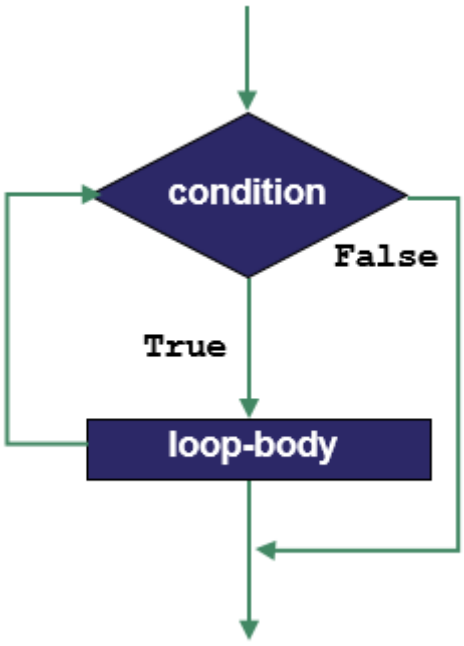


In Python, while loops are introduced with the keyword `while`. This is followed by some condition which has to be made up by the programmer (this is the 'hard' part!).

If the result of the condition is `True`, the statement(s) that make up the *loop body* are executed. These statements *must* be indented.

When Python reaches the last line of the loop body the flow of control loops back to the condition which is evaluated again. (Python will know the last line of the loop body from the levels of indentation.)

The above process continues until the result of the condition is found to be `False`.



Flow chart illustration of `while` loop

When (and if) the condition is `False` Python skips the loop body, and the flow of control resumes at the next statement following the loop body.

It should be noted that it is the programmer's responsibility to ensure that the loop body contains a line of code that will cause the loop condition to eventually become `False`. Otherwise, the loop will never terminate. Such loops are called *infinite loops*.

It is also worth noting out that the loop body might not ever be executed. This situation would arise when the condition evaluates to `False` before the first iteration. If this happens, the loop body is skipped and the flow of control continues from the first statement after the loop body.

Because the condition 'guards' entry into the loop, it is referred to as the *loop guard*.

KEY POINT: The **loop body** is executed each time the **loop guard** is evaluated to `True`.

STUDENT TIP
It is useful to think of a loop guard as a sentinel who operates a green-red signal system. A green signal means enter the loop and red means skip the loop.

Example

The short program serves to demonstrate main features of a `while` loop.

```
1.  # Simple while loop
2.
3.  # Initialise a loop counter
4.  counter = 1
5.
6.  # Loop 10 times
7.  while counter <= 10:
8.      print("Hello World")  # Display a message
9.      counter = counter + 1 # Increment the counter
10.
11. # This line is only executed once
12. print("Goodbye")
```

Simple `while` loop demo.

The program displays the message *Hello World* ten times. The string Goodbye is displayed once before the program exits.

- The loop is introduced by the `while` keyword on line 7. Note the use of colon (:) at the end of this line.
- The condition **`counter <= 10`** is central how the loop operates. The loop will be executed as long as this condition remains `True`. The condition is initially `True` because the variable `counter` was initialised to 1 on line 4
- Lines 8 and 9 make up the loop body.
  - line 8 tells Python to display the string, *Hello World*
  - line 9 tells Python to increase the value of `counter` by 1 (recall running totals)
- The next line to be executed after line 9 is *always* line 7. (This is the iteration)
- Each time line 7 is executed, the value of `counter` will have increased by one since the previous iteration. Eventually, `counter` will have reached a value of 11 and the condition will be found to be `False`. At this point the flow of control jumps beyond the loop body and line 12 is the next, and final, line of the program to be executed.

- In handout, do **Task 1** (code shown above).

It is well worth investing some time in this example to make sure you understand exactly how 'while loops' are executed at runtime.
If we define an iteration to be the number of times a loop body has been executed, we can use the technique of tracing to keep track of the loop's progress.
Initially, (before any iterations), `counter` is set to 1, there is no program output displayed, and the condition `counter <= 10` is `True`. After one iteration of the loop, the string *Hello World* is displayed, `counter` has increased from 1 to 2 and so the condition (`counter <= 10`) remains `True`.

- In handout, do **Task 2** (shown below).

Example (Guess game v4)

Let's say we wanted to enhance our guessing game to give the user *three chances*.
We start off in the knowledge that version 3 of the program works properly for one chance. Our enhancement can be achieved simply by 'wrapping' the code from version 3 inside a while loop that runs three times. A solution is presented below.

```
1.  # Guess Game - 3 guesses
2.  import random
3.
4.  number = random.randint(1, 10)
5.  print(number) # have a sneak peek!
6.
7.  # Initialise a loop counter
8.  counter = 0
9.
10. # Loop 3 times
11. while counter < 3:
12.
13.     guess = int(input("Enter a number between 1 and 10: "))
14.     if guess == number:
15.         print("Correct")
16.         break # exit the loop immediately!
17.     elif guess < number:
18.         print("Too low")
19.     else:
20.         print("Too high")
21.
22.     counter = counter + 1
23.
24. print("Goodbye")
```

Guessing Game v4

- In handout, do **Task 3** (code shown above).

The technique of wrapping code inside a loop in very important in the development of computer programs and systems.

STUDENT TIP
When starting to learn loops for the first time students should be encouraged to work with code they are already familiar with and wrap it inside a loop structure.

Code wrapping is based on a notion that, if a piece of working code can be written using sequence/selection control structures only, then it should be relatively straightforward to put that code inside a loop.

- In handout, do **Task 4**.

Example (Guess game v5)

In this version of the game we will introduce a Boolean variable to enable the program to continue until the user makes the correct guess (no matter how many guesses this may take!).

```
1.  # Guess Game v5 - while not found
2.  import random
3.
4.  number = random.randint(1, 10)
5.  print(number) # have a sneak peek!
6.
7.  correct = False # initialise the loop guard variable
8.
9.  # Loop until the variable correct becomes True
10. while not correct:
11.
12.     guess = int(input("Enter a number between 1 and 10: "))
13.     if guess == number:
14.         print("Correct")
15.         correct = True # this will cause the loop to exit
16.     elif guess < number:
17.         print("Too low")
18.     else:
19.         print("Too high")
20.
21. print("Goodbye")
```

Guessing Game v5

- In handout, do **Task 5** (code shown above).

A Boolean variable is a variable used to store a Boolean value. In Python the only two Boolean values are `True` and `False`.

In this example, the name of the Boolean variable is `correct`. It is initialised to `False` on line 7. The use of the variable `correct` (lines 7, 10 and 15) is *the* central feature of this program.

The loop keeps going as long as the correct is `False`. Logically, this is the same as saying that the loop continues as long as `not correct` is `True`. The only place `correct` is set to `True` is on line 15 which gets executed if and only if the value of `guess` is the same as the value of `number`.

- In handout, do **Task 6**.

Example (Guess game v6)

In this version, we add one final piece of functionality. This time when the user guesses correctly, instead of terminating the loop (and program), this program will ask the user if they want to play another game. If the user responds with anything other than N (for no), the program generates a new random number and continues.

```
1.  # Guess Game v6 - while not correct - ask, go again?
2.  import random
3.
4.  number = random.randint(1, 10)
5.  print(number) # have a sneak peek!
6.
7.  # Initialise the loop guard variable
8.  keepGoing  = True
9.
10. # Loop as long as keepGoing is True
11. while keepGoing:
12.
13.     guess = int(input("Enter a number between 1 and 10: "))
14.
15.     if guess == number:
16.         print("Correct")
17.         goAgain = input("Play again? (Y/N): ")
18.         if goAgain == "N":
19.             keepGoing = False
20.         else:
21.             # Generate a new number
22.             number = random.randint(1, 10)
23.             print(number) # why not?
24.
25.     elif guess < number:
26.         print("Too low")
27.
28.     else:
29.         print("Too high")
30.
31. print("Goodbye")
```

Guessing Game v6

This 'play again' logic is incorporated by using another Boolean variable, `keepGoing` which is initially set to `True` (line 8).

The loop will continue as long as the condition on line 11 evaluates to `True`. But the condition here is simply `keepGoing` so as long as this variable remains `True` the loop will continue.

The only circumstances where `keepGoing` is set to `False` is on line 19. Can you figure out from the code what these circumstances are?

- In handout, do **Task 7**.

Example (Guess game v7)

In this last version of the program there is no new functionality added. Rather, the program demonstrates a standard technique used to validate data entered by the user.

If you run any version of the guess game before this and enter a non-numeric value as the guess you will notice that the program crashes (runtime error). The reason for this is that all earlier versions make the (incorrect) assumption that a user will always enter the correct type of data, which is not very realistic for any production system.

```
1.  # Guess Game v7 - while - go again? - data validation
2.  import random
3.
4.  number = random.randint(1, 10)
5.
6.  # Initialise the loop guard variable
7.  keepGoing  = True
8.
9.  # Loop as long as keepGoing is True
10. while keepGoing:
11.
12.     guess = input("Enter a number between 1 and 10: ")
13.     # Validate. Make sure the value is a number
14.     while not guess.isdigit():
15.        guess = input("Enter a number between 1 and 10: ")
16.
17.     # Conver the string to an integer
18.     guess = int(guess)
19.
20.     if guess == number:
21.         print("Correct")
22.         goAgain = input("Play again? (Y/N): ")
23.         if goAgain.upper() == "N":
24.             keepGoing = False
25.         else:
26.             # Generate a new number
27.             number = random.randint(1, 10)
28.
29.     elif guess < number:
30.         print("Too low")
31.
32.     else:
33.         print("Too high")
34.
35. print("Goodbye")
```

Guessing Game v7

Take some time to study the validation technique used here (lines 12 – 15) and see if you can figure out how it works. The condition on line 14 is key. Also notice that lines 12 and 15 are identical but appear at different levels of indentation. A good starting point would be to isolate the four lines of code into a separate program and experiment.

- In handout, do **Task 8**.

The following pseudo-code outlines a general pattern used to ensure some value entered by the end-user is valid.

```
read value from end-user
loop for as long as the value is invalid:
   [display error message]
   read value from end-user
process value
```

STUDENT TIP
A common student misconception is to think that a while loop's condition is being constantly evaluated and the loop exits the instant it becomes `False` inside the loop body.

- In handout, do **Task 9**.

Experiment!
Try making the following changes to see what happens.

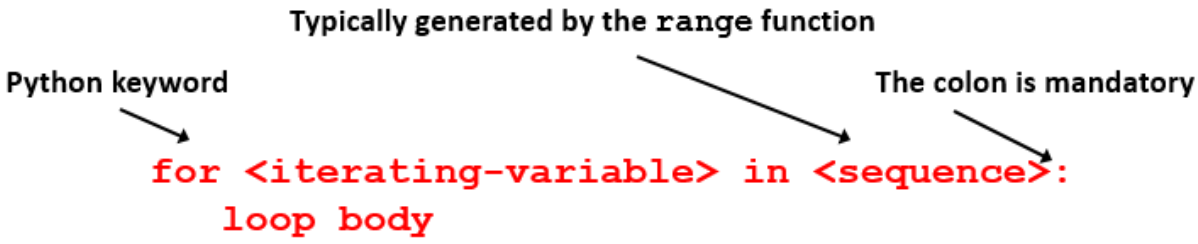
- Version 4 change the initial value of `counter` to 10
- Version 5 change the initial value of `correct` to be `True`
- Version 6 change the initial value of `keepGoing` to be `False`
- Version 7 remove (comment out line line 18

The `for` loop

The `for` loop is a more specific iteration construct than its `while` counterpart in the sense that it is designed specifically for stepping through the items in a sequence.

`for` loops are the preferred looping mechanism when the number of required iterations is known in advance (of runtime). Because of their nature they are also commonly used to traverse strings and lists (which are both sequence types).

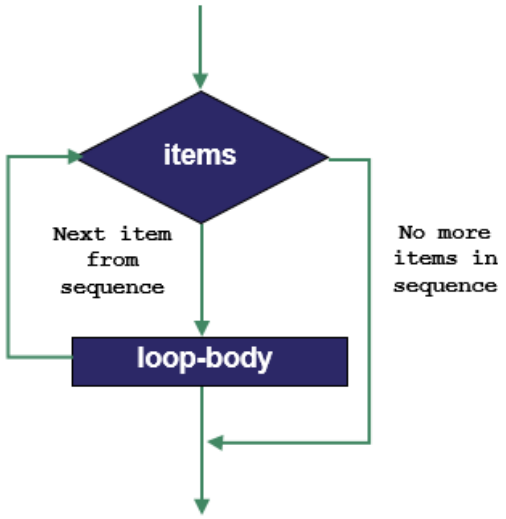
The ***syntax*** and ***semantics*** of a `for` loop are described and illustrated as follows:



The loop starts by assigning the first item in the *sequence* to the loop variable referred to as *iterating_variable*. Next, the statement(s) that make up the loop body are executed.

The loop continues the cycle of assigning the next item in the sequence to the iterating variable and then processing it in the loop body until the entire sequence is used up.

Observe the use of colon (:) and also that the statements which make up the loop body are indented.



`for` loop flowchart

KEY POINT:

It is essential to be able to recognise situations where a loop is required in a program. The choice of loop construct does not matter greatly. Most loops that can be programmed with a `while` construct can also be constructed using a `for` construct and vice versa.

Example

The short program serves to demonstrate main features of a `for` loop.

```
# Simple for loop

for counter in range(10):
    print("Hello World", counter)

print("Goodbye")
```

Simple `for` loop demo.

```
Hello World 0
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
Hello World 7
Hello World 8
Hello World 9
Goodbye
```

Program Output

The program output shown to the right above displays the message *Hello World* ten times. The value of counter is also displayed alongside the string. The string Goodbye is displayed once before the program exits.

The first thing to observe is how much shorter this program is compared to the simple while loop program we say earlier.

In order to understand the example, it is helpful to understand the range built-in function. (see https://docs.python.org/3/library/stdtypes.html#typesseq-range for a complete description.)

Range is a built-in function which returns a list of values. In the above example the call `range(10)` returns a list of all the integers from 0 to 9

In this example, the `for` loop works by iterating over each value in the sequence (i.e. 0 through to 9). At the start of each iteration the value of the next item in the sequence is assigned to the loop variable `counter`.

Notice how at the start of each iteration the iterating variable is assigned the next value in the sequence. The loop ends when the last value in the sequence has been used.

Once the loop terminates, execution continues at the next line after the loop body.

Example - u*sing a for loop to draw the square*

We now consider how to use a loop to draw a square using the turtle graphics library. Recall from earlier

```
pen.forward(100)
pen.right(90)
pen.forward(100)
pen.right(90)
pen.forward(100)
pen.right(90)
pen.forward(100)
```

As can be seen the two lines shown below are repeated 4 times – this is an indication that it should be possible to use a loop.

```
pen.forward(100)
pen.left(90
```

The same functionality can be achieved by wrapping these lines in a for loop as follows.

```
# Draw a square
for count in range(4):
    pen.forward(100)
    pen.left(90
```

We are now ready to attempt our first `for` loop programming exercises.