Chapter 7
Iteration/Looping

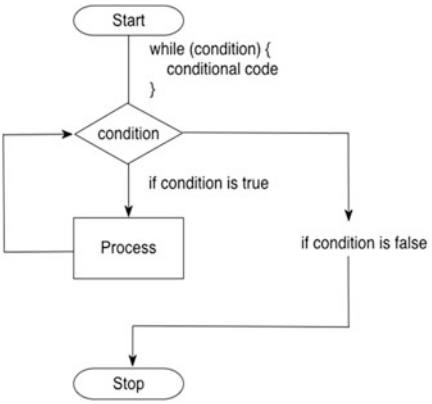
7.1 Introduction

In this section we will look at the `while` loop and the `for` loop available in Python. These loops are used to control the repeated execution of selected statements.

7.2 While Loop

The `while` loop exists in almost all programming languages and is used to iterative (or repeat) one or more code statements as long as the test condition (expression) is `True`. This iteration construct is usually used when the number of times we need to repeat the block of code to execute is not known. For example, it may need to repeat until some solution is found or the user enters a particular value.

The behaviour of the while loop is illustrated in below.



J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_7

The Python while loop has the basic form:

```
while <test-condition-is-true>:
    statement or statements
```

As shown both in the diagram and can be inferred from the code; *while* the test condition/expression is `True` then the statement or block of statements in the `while` loop will be executed.

Note that the test is performed *before* each iteration, including the first iteration; therefore, if the condition fails the first time around the loop the statement or block of statements may never be executed at all.

As with the `if` statement, indentation is key here as the statements included in the while statement are determined by indentation. Each statement that is indented to the same level after the while condition is part of the `while` loop.

However, as soon as a statement is no longer following the `while` block; then it is no longer part of the `while` loop and its execution is no longer under the control of the test-condition.

The following illustrates an example `while` loop in Python:

```
count = 0
print('Starting')
while count < 10:
    print(count, ' ', end='') # part of the while loop
    count += 1                # also part of the while loop
print() # not part of the while loop
print('Done')
```

In this example while some variable `count` is less than the value 10 the `while` loop will continue to iterate (will be repeated). The `while` loop itself contains two statements; one prints out the value of count variable while the other increments `count` (remember `count +=1` is equivalent to `count = count + 1`).

We have used the version of `print()` that does not print a carriage return when it prints out a value (this is indicated by the `end=''` option passed to the `print()` function).

The result of running this example is:

```
Starting
0 1 2 3 4 5 6 7 8 9
Done
```

As you can see the statements printing the starting and done messages are only run once. However, the statement printing out the `count` variable is run 10 times (printing out the values 0–9).

Once the value of `count` is equal to 10 then the loop finishes (or terminates).

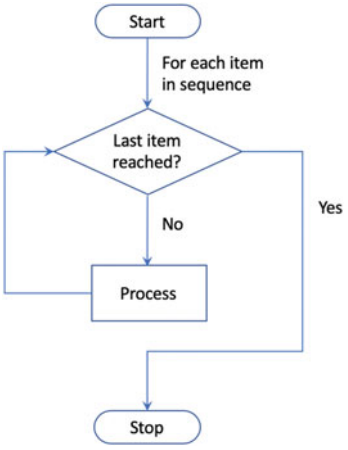
Note that we needed to initialise the `count` variable before the loop. This is because it needs to have a value for the first iteration of the `while` loop. That is

before the `while` loop does anything the program needs to already know the first value of count so that it can perform that very first test. This is a feature of the while loops behaviour.

7.3 For Loop

In many cases we know how many times we want to iterative over one or more statements (as indeed we did in the previous section). Although the `while` loop can be used for such situations, the `for` loop is a far more concise way to do this. It typically also clearer to another programmer that the loop must iterate for a specific number of iterations.

The `for` loop is used to step a *variable* through a series of values until a given test is met. The behaviour of the `for` loop is illustrated in below.



This flow chart shows that some sequence of values (for example all integer values between 0 and 9) will be used to iterate over a block of code to process. When the last item in the sequence has been reached, the loop will terminate.

Many languages have a `for` loop of the form:

```
for i = from 0 to 10
    statement or statements
```

In this case a variable `'i'` would take the values 0, 1, 2, 3 etc. up to 10.

In Python the approach is slightly different as the values from 0 up to 10 are represented by a *range*. This is actually a function that will generate the *range* of values to be used as the sequence in the `for` loop. This is because the Python for loop is very flexible and can loop over not only a range of integer values but also a set of values held in data structures such as a list of integers or strings. We will

return to this feature of the `for` loop when we look at collections/containers of data in a later chapter.

The format of the Python `for` loop when using a range of values is

```
for <variable-name> in range(...):
    statement
    statement
```

An example is shown below which equates to the `while` loop we looked at earlier:

```
# Loop over a set of values in a range
print('Print out values in a range')
for i in range(0, 10):
    print(i, ' ', end='')
print()
print('Done')
```

When we run this code, the output is:

```
Print out values in a range
0  1  2  3  4  5  6  7  8  9
Done
```

As can be seen from the above; the end result is that we have generated a `for` loop that produces the same set of values as the earlier `while` loop. However,

- the code is more concise,
- it is clear we are processing a range of values from 0 to 9 (note that it is up to but not including the final value) and
- we did not need to define the loop variable first.

For these reasons *for* loops are more common in programs in general than *while* loops.

One thing you might notice though is that in the `while` loop we are not constrained to incrementing the `count` variable by one (we just happened to do this). For example, we could have decided to increment `count` by 2 each time round the loop (a very common idea). In fact, `range` allows us to do exactly this; a third argument that can be provided to the `range` function is the value to increment the loop variable by each time round the loop, for example:

```
# Now use values in a range but increment by 2
print('Print out values in a range with an increment of 2')
for i in range(0, 10, 2):
    print(i, ' ', end='')
print()
print('Done')
```

When we run this code, the output is

```
Print out values in a range with an increment of 2
0  2  4  6  8
Done
```

Thus the value of the loop variable has jumped by 2 starting at 0. Once its value was 10 or more then the loop terminates. Of course, it is not only the value 2 we could use; we could increment by any meaningful integer, 3, 4 or 5 etc.

One interesting variation on the `for` loop is the use of a wild card (a '_') instead of a looping variable; this can be useful if you are only interested in looping a certain number of times and not in the value of the loop counter itself, for example:

```
# Now use an 'anonymous' loop variable
for _ in range(0,10):
    print('.', end='')
print()
```

In this case we are not interested in the values generated by the range per se only in looping 10 times thus there is no benefit in recording the loop variable. The loop variable is represented by the underbar character ('_').

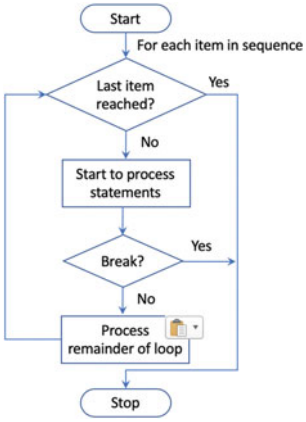
Note that in actual fact this is a valid variable and can be referenced within the loop; however by convention it is treated as being anonymous.

7.4 Break Loop Statement

Python allows programmers to decide whether they want to *break* out of a loop early or not (whether we are using a `for` loop or a `while` loop). This is done using the `break` statement.

The `break` statement allows a developer to alter the normal cycle of the loop based on some criteria which may not be predictable in advance (for example it may be based on some user input).

The `break` statement, when executed, will terminate the current loop and jump the program to the first line after the loop. The following diagram shows how this works for a for loop:



Typically, a guard statement (`if` statement) is placed on the `break` so that the `break` statement is conditionally applied when appropriate.

This is shown below for a simple application where the user is asked to enter a number which may or may not be in the range defined by the `for` loop. If the number is in the range, then we will loop until this value is reached and then break the loop (that is terminate early without processing the rest of the values in the loop):

```
print('Only print code if all iterations completed')
num = int(input('Enter a number to check for: '))
for i in range(0, 6):
    if i == num:
        break
    print(i, ' ', end='')
print('Done')
```

If we run this and enter the value 7 (which is outside of the range) then all the values in the loop should be printed out:

```
Enter a number to check for: 7
0 1  2  3  4  5 Done
```

However, if we enter the value 3 then only the values 0, 1 and 2 will be printed out before the loop breaks (terminates) early:

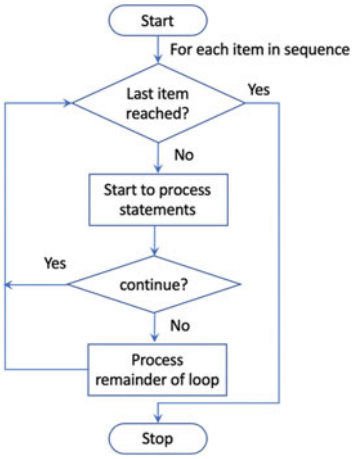
```
Enter a number to check for: 3
0 1  2  Done
```

Note that the string 'Done' is printed out in both cases as it is the first line after the `for` loop that is not indented and is thus not part of the `for` loop.

Note that the `break` statement can come anywhere within the block of code associated with the loop construct (whether that is a `for` loop or a `while` loop). This means that there can be statements before it and after it.

7.5 Continue Loop Statement

The `continue` statement also affects the flow of control within the looping constructs `for` and `while`. However, it does not terminate the whole loop; rather it only terminates the current iteration round the loop. This allows you to skip over part of a loop's iteration for a particular value, but then to continue with the remaining values in the sequence.



A guard (`if` statement) can be used to determine when the `continue` statement should be executed.

As with the `break` statement, the `continue` statement can come anywhere within the body of the looping construct. This means that you can have some statements that will be executed for every value in the sequence and some that are only executed when the continue statement is not run.

This is shown below. In this program the `continue` statement is executed only for odd numbers and thus the two `print()` statements are only run if the value of `i` is even:

```
for i in range(0, 10):
    print(i, ' ', end='')
    if i % 2 == 1:
        continue
    print('hey its an even number')
    print('we love even numbers')
print('Done')
```

When we run this code we get

```
0  hey its an even number
we love even numbers
1  2  hey its an even number
we love even numbers
3  4  hey its an even number
we love even numbers
5  6  hey its an even number
we love even numbers
7  8  hey its an even number
we love even numbers
9  Done
```

As you can see, we only print out the messages about a number being even when the values are 0, 2, 4, 6 and 8.

7.6 For Loop with Else

A `for` loop can have an *optional* `else` block at the end of the loop. The `else` part is executed if and only if all items in the sequence are processed. The `for` loop may fail to process all elements in the loop if for some reason an error occurs in your program (for example if you have a syntax error) or if you `break` the loop.

Here is an example of a `for` loop with an `else` part:

```
# Only print code if all iterations completed over a list
print('Only print code if all iterations completed')
num = int(input('Enter a number to check for: '))
for i in range(0, 6):
    if i == num:
        break
    print(i, ' ', end='')
else:
    print()
    print('All iterations successful')
```

If we run this code and enter the integer value 7 as the number to check for; then the `else` block executes as the `if` test within the `for` statement is never `True` and thus the loop is never broken, and all values are processed:

```
Only print code if all iterations completed
Enter a number to check for: 7
0  1  2  3  4  5
All iterations successful
```

However, if we enter the value 3 as the number to check for; then the `if` statement will be `True` when the loop variable 'i' has the value 3; thus only the values 0, 1 and 2 will be processed by the loop. In this situation the `else` part will *not* be executed because not all the values in the sequence were processed:

```
Only print code if all iterations completed
Enter a number to check for: 3
0  1  2
```

7.7 A Note on Loop Variable Naming

Earlier in the book we said that variable names should be meaningful and that names such as 'a' and 'b' were not in general a good idea. The one exception to this rule relates to loop variable names used with for loops over ranges. It is very common to find that these loop variables are called 'i', 'j' etc.

It is such a common convention that if a variable is called 'i' or 'j' people expect it to be a loop variable. As such

- you should consider using these variable names in looping constructs,
- and avoid using them elsewhere.

But this does raise the question why 'i' and 'j'; the answer is that it all goes back to a programming language called Fortran which was first developed in the 1950s. In this programming language loop variables had to be called 'i' and 'j' etc. Fortran was so ubiquitous for mathematical and scientific programming, where loops are almost *di rigour*, that this has become the convention in other languages which do not have this restriction.

7.8 Dice Roll Game

The following short program illustrates how a `while` loop can be used to control the execution of the main body of code. In this game we will continue to roll a pair of dice until the user indicates that they do not want to roll again (we use the `random` module for this which is discussed in the next chapter). When this occurs the `while` loop will terminate:

```
import random

MIN = 1
MAX = 6

roll_again = 'y'

while roll_again == 'y':
    print('Rolling the dices...')
    print('The values are....')
    dice1 = random.randint(MIN, MAX)
    print(dice1)
    dice2 = random.randint(MIN, MAX)
    print(dice2)
    roll_again = input('Roll the dices again? (y / n): ')
```

When we run this program the results of rolling two dice are shown. The program will keep looping and printing out the two dice values until the user indicates that they no longer want to roll the dice:

```
Rolling the dices...
The values are....
2
6
Roll the dices again? (y / n): y
Rolling the dices...
The values are....
4
1
Roll the dices again? (y / n): y
Rolling the dices...
The values are....
3
6
Roll the dices again? (y / n): n
```

7.9 Online Resources

See the Python Standard Library documentation for:

- https://docs.python.org/3/tutorial/controlflow.html the online Python flow of control tutorial.

7.10 Exercises

There are two exercises for this chapter. The first exercise will require a simple `for` loop while the second is more complicated requiring nested `for` loops and a `break` statement.

7.10.1 Calculate the Factorial of a Number

Write a program that can find the factorial of any given number. For example, find the factorial of the number 5 (often written as 5!) which is `1 * 2 * 3 * 4 * 5` and equals 120.

The factorial is not defined for negative numbers and the factorial of Zero is 1; that is 0! = 1.

Your program should take as input an integer from the user (you can reuse your logic from the last chapter to verify that they have entered a positive integer value using `isnumeric()`).

You should

1. If the number is less than Zero return with an error message.
2. Check to see if the number is Zero—if it is then the answer is 1—print this out.
3. Otherwise use a loop to generate the result and print it out.

7.10.2 Print All the Prime Numbers in a Range

A Prime Number is a positive whole number, greater than 1, that has no other divisors except the number 1 and the number itself.

That is, it can only be divided by itself and the number 1, for example the numbers 2, 3, 5 and 7 are prime numbers as they cannot be divided by any other whole number. However, the numbers 4 and 6 are not because they can both be divided by the number 2 in addition the number 6 can also be divided by the number 3.

You should write a program to calculate prime number starting from 1 up to the value input by the user.

If the user inputs a number below 2, print an error message.

For any number greater than 2 loop for each integer from 2 to that number and determine if it can be divided by another number (you will probably need two for loops for this; one nested inside the other).

For each number that cannot be divided by any other number (that is its a prime number) print it out.