7.3 The while loop

In Python, while loops are constructed like so:

```
while [a condition is True]:
    [do something]
```

The something that is being done will continue to be executed until the condition that is being assessed is no longer true.

Let's create a small program that executes a while loop. In this program, we'll ask for the user to input a password. While going through this loop, there are two possible outcomes:

- If the password *is* correct, the while loop will exit.
- If the password is *not* correct, the while loop will continue to execute.

We'll create a file called password.py in our text editor of choice, and begin by initializing the variable password as an empty string:

password.py

```
password = ''
```

The empty string will be used to take in input from the user within the while loop.

Now, we'll construct the while statement along with its condition:

password.py

```
password = ''

while password != 'password':
```

Here, the while is followed by the variable password. We are looking to see if the variable password is set to the string password (based on the user input later), but you can choose whichever string you'd like.

This means that if the user inputs the string password, then the loop will stop and the program will continue to execute any code outside of the loop. However, if the string that the user inputs is not equal to the string password, the loop will continue.

Next, we'll add the block of code that does something within the while loop:

password.py

```
password = ''

while password != 'password':

    print('What is the password?')

    password = input()
```

Inside of the while loop, the program runs a print statement that prompts for the password. Then the variable password is set to the user's input with the input() function.

The program will check to see if the variable `password` is assigned to the string `password`, and if it is, the `while` loop will end. Let's give the program another line of code for when that happens:

password.py

```
password = ''
while password != 'password':
    print('What is the password?')
    password = input()
print('Yes, the password is ' + password + '. You may enter.')
```

The last `print()` statement is outside of the `while` loop, so when the user enters `password` as the password, they will see the final print statement outside of the loop.

However, if the user never enters the word `password`, they will never get to the last `print()` statement and will be stuck in an infinite loop.

An **infinite loop** occurs when a program keeps executing within one loop, never leaving it. To exit out of infinite loops on the command line, press `CTRL + C`.

Save the program and run it. You'll be prompted for a password, and then may test it with various possible inputs. Here is sample output from the program:

Output

```
What is the password?
hello
What is the password?
sammy
What is the password?
PASSWORD
What is the password?
password
Yes, the password is password. You may enter.
```

Keep in mind that strings are case sensitive unless you also use a string function to convert the string to all lower-case (for example) before checking.

Example Program with While Loop

Now that we understand the general premise of a `while` loop, let's create a command-line guessing game that uses a `while` loop effectively.

First, we'll create a file called `guess.py` in our text editor of choice. We want the computer to come up with random numbers for the user to guess, so we'll import the `random` module with an `import` statement.

guess.py

```
import random
```

Next, we'll assign a random integer to the variable `number`, and keep it in the range of 1 through 25 (inclusive), in the hope that it does not make the game too difficult.

guess.py

```
import random
number = random.randint(1, 25)
```

At this point, we can get into our `while` loop, first initializing a variable and then creating the loop.

guess.py

```
import random
number = random.randint(1, 25)
number_of_guesses = 0
while number_of_guesses < 5:
   print('Guess a number between 1 and 25:')
   guess = input()
   guess = int(guess)
   number_of_guesses = number_of_guesses + 1
   if guess == number:
       break
```

We've initialized the variable `number_of_guesses` at 0, so that we increase it with each iteration of our loop so that we don't have an infinite loop. Then we added the `while` statement so that the `number_of_guesses` is limited to 5 total. After the fifth guess, the user will return to the command line, and for now, if the user enters something other than an integer, they'll receive an error.

Within the loop, we added a `print()` statement to prompt the user to enter a number, which we took in with the `input()` function and set to the `guess` variable. Then, we converted `guess` from a string to an integer.

Before the loop is over, we also want to increase the `number_of_guesses` variable by 1 so that we can iterate through the loop 5 times.

Finally, we write a conditional `if` statement to see if the `guess` that the user made is equivalent to the `number` that the computer generated, and if so we use a `break` statement to come out of the loop.

The program is fully functioning, and we can run it. Though it works, right now the user never knows if their guess is correct and they can guess the full 5 times without ever knowing if they got it right. Sample output of the current program looks like this:

Output

```
Guess a number between 1 and 25:
11
Guess a number between 1 and 25:
19
Guess a number between 1 and 25:
```

```
22
Guess a number between 1 and 25:
3
Guess a number between 1 and 25:
8
```

Let’s add some conditional statements outside of the loop so that the user is given feedback as to whether they correctly guess the number or not. These will go at the end of our current file.

guess.py

```
import random
number = random.randint(1, 25)
number_of_guesses = 0
while number_of_guesses < 5:
    print('Guess a number between 1 and 25:')
    guess = input()
    guess = int(guess)
    number_of_guesses = number_of_guesses + 1
    if guess == number:
        break
if guess == number:
    print('You guessed the number in ' + str(number_of_guesses) + ' tries!')
else:
    print('You did not guess the number. The number was ' + str(number))
```

At this point, the program will tell the user if they got the number right or wrong, which may not happen until the end of the loop when the user is out of guesses.

To give the user a little help along the way, let’s add a few more conditional statements into the `while`loop. These can tell the user whether their number was too low or too high, so that they can be more likely to guess the correct number. We’ll add these before our `if guess == number` line

guess.py

```
import random
number = random.randint(1, 25)
number_of_guesses = 0
while number_of_guesses < 5:
    print('Guess a number between 1 and 25:')
    guess = input()
    guess = int(guess)
    number_of_guesses = number_of_guesses + 1
    if guess < number:
        print('Your guess is too low')

    if guess > number:
        print('Your guess is too high')

    if guess == number:
        break

if guess == number:
    print('You guessed the number in ' + str(number_of_guesses) + ' tries!')

else:
    print('You did not guess the number. The number was ' + str(number))
```

When we run the program again with `python guess.py`, we see that the user gets more guided assistance in their guessing. So, if the randomly-generated number is `12` and the user guesses `18`, they will be told that their guess is too high, and they can adjust their next guess accordingly.

Example Sum Program with While Loop

```
# Sum from 1 to the given upperbound
n = int(input('Enter the upperbound: '))
i = 1
sum = 0
while (i <= n):
    sum += i
    i += 1
print(sum)
```

7.4 The `for-in` loop

In Python, `for` loops are constructed like so:

```
for [iterating variable] in [sequence]:
    [do something]
```

The something that is being done will be executed until the sequence is over.

Let’s look at a `for` loop that iterates through a range of values:

```
for i in range(0,5):
   print(i)
```

When we run this program, the output looks like this:

Output

```
0
1
2
3
4
```

This `for` loop sets up `i` as its iterating variable, and the sequence exists in the range of 0 to 5.

Then within the loop we print out one integer per loop iteration. Keep in mind that in programming we tend to begin at index 0, so that is why although 5 numbers are printed out, they range from 0-4.

You'll commonly see and use `for` loops when a program needs to repeat a block of code a number of times.

For Loops using range()

One of Python's built-in immutable sequence types is `range()`. In loops, `range()` is used to control how many times the loop will be repeated.

When working with `range()`, you can pass between 1 and 3 integer arguments to it:

- `start` states the integer value at which the sequence begins, if this is not included then `start`begins at 0
- `stop` is always required and is the integer that is counted up to but not included
- `step` sets how much to increase (or decrease in the case of negative numbers) the next iteration, if this is omitted then `step` defaults to 1

We'll look at some examples of passing different arguments to `range()`.

First, let's only pass the `stop` argument, so that our sequence set up is `range(stop)`:

```
for i in range(6):
   print(i)
```

In the program above, the `stop` argument is 6, so the code will iterate from 0-6 (exclusive of 6):

Output

```
0
1
2
3
4
5
```

Next, we'll look at `range(start, stop)`, with values passed for when the iteration should start and for when it should stop:

```
for i in range(20,25):
   print(i)
```

Here, the range goes from 20 (inclusive) to 25 (exclusive), so the output looks like this:

Output

```
20
21
22
23
24
```

The `step` argument of `range()` is similar to specifying stride while slicing strings in that it can be used to skip values within the sequence.

With all three arguments, `step` comes in the final position: `range(start, stop, step)`. First, let’s use a `step` with a positive value:

```
for i in range(0,15,3):
   print(i)
```

In this case, the `for` loop is set up so that the numbers from 0 to 15 print out, but at a `step` of 3, so that only every third number is printed, like so:

Output

```
0
3
6
9
12
```

We can also use a negative value for our `step` argument to iterate backwards, but we’ll have to adjust our `start` and `stop` arguments accordingly:

```
for i in range(100,0,-10):
   print(i)
```

Here, 100 is the `start` value, 0 is the `stop` value, and `-10` is the range, so the loop begins at 100 and ends at 0, decreasing by 10 with each iteration. We can see this occur in the output:

Output

```
100
90
80
70
60
50
40
30
20
10
```

When programming in Python, `for` loops often make use of the `range()` sequence type as its parameters for iteration.

Iterating through a Sequence (String, List, Tuple, Dictionary, Set)

The `for-in` loop is primarily used to iterate the same process through all the items of a sequence, for example,

```
# String: iterating through each character
>>> for char in 'hello': print(char)
h
e
l
l
o

# List: iterating through each item
>>> for item in [123, 4.5, 'hello']: print(item)
123
4.5
hello

# Tuple: iterating through each item
>>> for item in (123, 4.5, 'hello'): print(item)
123
4.5
hello

# Dictionary: iterating through each key
>>> dct = {'a': 1, 2: 'b', 'c': 'cc'}
>>> for key in dct: print(key, ':', dct[key])
a : 1
c : cc
2 : b

# Set: iterating through each item
>>> for item in {'apple', 1, 2, 'apple'}: print(item)
1
2
apple
```

The `iter()` and `next()` Built-in Functions

The built-in function `iter(iterable)` takes a `iterable` (such as sequence) and returns an `iterator` object. You can then use `next(iterator)` to iterate through the items. For example,

```
>>> i = iter([11, 22, 33])
>>> next(i)
11
>>> next(i)
22
>>> next(i)
33
>>> next(i)  # Raise StopIteration exception if no more item
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
StopIteration

>>> type(i)
<class 'list_iterator'>
```

The `range()` Examples

```
# Sum from 1 to the given upperbound
upperbound = int(input('Enter the upperbound: '))
sum = 0
for number in range(1, upperbound+1):  # list of 1 to n
    sum += number
print("The sum is:" , sum)

# Sum a given list
lst = [9, 8, 4, 5]
sum = 0
for item in lst:  # Each item of lst
    sum += item
print(sum)

# Use built-in function
del sum   # Need to remove the sum variable before using builtin function sum
print(sum(lst))
```

The `reversed()` Built-in Function

To iterate a sequence in the reverse order, apply the `reversed()` function which reverses the iterator over values of the sequence. For example,

```
>>> lst = [11, 22, 33]
>>> for item in reversed(lst): print(item, end=' ')
33 22 11

>>> str = "hello"
>>> for c in reversed(str): print(c, end='')
olleh
```

Multiple Sequences and the `zip()` Built-in Function

To loop over two or more sequences concurrently, you can pair the entries with the `zip()` built-in function. For examples,

```
>>> lst1 = ['a', 'b', 'c']
>>> lst2 = [11, 22, 33]
>>> for i1, i2 in zip(lst1, lst2): print(i1, i2)
a 11
b 22
```

```
c 33
>>> zip(lst1, lst2)   # Return a list of tuples
[('a', 11), ('b', 22), ('c', 33)]
```

Nested For Loops

Loops can be nested in Python, as they can with other programming languages. A nested loop is a loop that occurs within another loop, structurally similar to nested `if` statements. These are constructed like so:

```
for [first iterating variable] in [outer loop]: # Outer loop
    [do something]  # Optional
    for [second iterating variable] in [nested loop]:   # Nested loop
        [do something]
```

The program first encounters the outer loop, executing its first iteration. This first iteration triggers the inner, nested loop, which then runs to completion. Then the program returns back to the top of the outer loop, completing the second iteration and again triggering the nested loop. Again, the nested loop runs to completion, and the program returns back to the top of the outer loop until the sequence is complete or a break or other statement disrupts the process.

Let's implement a nested `for` loop so we can take a closer look. In this example, the outer loop will iterate through a list of integers called `num_list`, and the inner loop will iterate through a list of strings called `alpha_list`.

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']

for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
```

When we run this program, we'll receive the following output:

Output

```
1
a
b
c
2
a
b
c
3
a
b
c
```

The output illustrates that the program completes the first iteration of the outer loop by printing `1`, which then triggers completion of the inner loop, printing `a`, `b`, `c` consecutively. Once the inner loop has completed, the program returns to the top of the outer loop, prints `2`, then again prints the inner loop in its entirety (`a`, `b`, `c`), etc.

Nested `for` loops can be useful for iterating through items within lists composed of lists. In a list composed of lists, if we employ just one `for` loop, the program will output each internal list as an item:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0, 1, 2],[9.9, 8.8, 7.7]]

for list in list_of_lists:
    print(list)
```

Output

```
['hammerhead', 'great white', 'dogfish']
[0, 1, 2]
[9.9, 8.8, 7.7]
```

In order to access each individual item of the internal lists, we'll implement a nested `for` loop:

```
list_of_lists = [['hammerhead', 'great white', 'dogfish'],[0, 1, 2],[9.9, 8.8, 7.7]]

for list in list_of_lists:
    for item in list:
        print(item)
```

Output

```
hammerhead
great white
dogfish
0
1
2
9.9
8.8
7.7
```

When we utilize a nested `for` loop we are able to iterate over the individual items contained in the lists.

7.5 break, continue and pass

Like C/C++/Java, the `break` statement breaks out from the innermost loop; the `continue` statement skips the remaining statements of the loop and continues the next iteration.

The `pass` statement does nothing. It serves as a placeholder for an empty statement or empty block.

Break Statement

In Python, the `break` statement provides you with the opportunity to exit out of a loop when an external condition is triggered. You'll put the `break` statement within the block of code under your loop statement, usually after a conditional `if` statement.

Let's look at an example that uses the `break` statement in a `for` loop:

```
number = 0

for number in range(10):
   number = number + 1

   if number == 5:
      break    # break here

   print('Number is ' + str(number))

print('Out of loop')
```

In this small program, the variable `number` is initialized at 0. Then a `for` statement constructs the loop as long as the variable `number` is less than 10.

Within the `for` loop, the number increases incrementally by 1 with each pass because of the line `number = number + 1`.

Then, there is an `if` statement that presents the condition that *if* the variable `number` is equivalent to the integer 5, *then* the loop will break.

Within the loop is also a `print()` statement that will execute with each iteration of the `for` loop until the loop breaks, since it is after the `break` statement.

To see when we are out of the loop, we have included a final `print()` statement outside of the `for` loop. When we run this code, our output will be the following:

Output

```
Number is 1
Number is 2
Number is 3
Number is 4
Out of loop
```

This shows that once the integer `number` is evaluated as equivalent to 5, the loop breaks, as the program is told to do so with the `break` statement.

The `break` statement causes a program to break out of a loop.

Continue Statement

The `continue` statement gives you the option to skip over the part of a loop where an external condition is triggered, but to go on to complete the rest of the loop. That is, the current iteration of the loop will be disrupted, but the program will return to the top of the loop.

The continue statement will be within the block of code under the loop statement, usually after a conditional if statement.

Using the same for loop program as in the Break Statement section above, we'll use a continue statement rather than a break statement:

```
number = 0

for number in range(10):
   number = number + 1

   if number == 5:
      continue    # continue here

   print('Number is ' + str(number))

print('Out of loop')
```

The difference in using the continue statement rather than a break statement is that our code will continue despite the disruption when the variable number is evaluated as equivalent to 5. Let's look at our output:

Output

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 6
Number is 7
Number is 8
Number is 9
Number is 10
Out of loop
```

Here we see that the line Number is 5 never occurs in the output, but the loop continues after that point to print lines for the numbers 6-10 before leaving the loop.

You can use the continue statement to avoid deeply nested conditional code, or to optimize a loop by eliminating frequently occurring cases that you would like to reject.

The continue statement causes a program to skip certain factors that come up within a loop, but then continue through the rest of the loop.

Pass Statement

When an external condition is triggered, the pass statement allows you to handle the condition without the loop being impacted in any way; all of the code will continue to be read unless a break or other statement occurs.

As with the other statements, the pass statement will be within the block of code under the loop statement, typically after a conditional if statement.

Using the same code block as above, let's replace the break or continue statement with a pass statement:

```
number = 0
for number in range(10):
   number = number + 1

   if number == 5:
      pass    # pass here

   print('Number is ' + str(number))

print('Out of loop')
```

The `pass` statement occurring after the `if` conditional statement is telling the program to continue to run the loop and ignore the fact that the variable `number` evaluates as equivalent to 5 during one of its iterations.

We'll run the program and take a look at the output:

Output

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9
Number is 10
Out of loop
```

By using the `pass` statement in this program, we notice that the program runs exactly as it would if there were no conditional statement in the program. The `pass` statement tells the program to disregard that condition and continue to run the program as usual.

The `pass` statement act as a placeholder when working on new code and thinking on an algorithmic level before hammering out details.