While Loops

Section 22.5

Completed

< back to module

Skip straight to editor/terminal

In the previous section, we saw the *for loop*, which takes a collection of items and loops through them one by one, executing the same block of code on each of those items, until it runs out of items.

But there's another kind of useful loop we can use: the *while loop*. Instead of giving the while loop a predetermined set of things to work through, we give it a condition: as long as that condition is true, the while loop will keep running.

The simplest kind of while loop usually contains some kind of counter, so let's have a look at this first.

```
counter = 1

while counter <= 3:
    print(counter)
    counter += 1

print('Loop complete!')
```

We start by creating the variable `counter` and assigning in the integer value of 1. Then we declare our while loop, and give it the condition `while counter <= 3`, which means the loop will run as long as the value of the variable `counter` is less than or equal to 3. Inside our while loop, we print the `counter` value for our reference, and then just before we leave our loop, we ask it to increment our counter variable by 1, which if you remember is the same as saying `counter = counter + 1`.

After this, we've reached the end of our while loop, so we start again by checking to see if the condition `counter <= 3` is still true. Our `counter` value is now 2, but since 2 is less than or equal to 3 our condition is still true so our while loop runs the code block again.

Once again, at the end of the block we increment our `counter` again, so now it's 3, and we start again by checking if our condition is still true. It is, because 3 = 3, so the code block within the loop runs a third time, incrementing our `counter` to 4 at the end.

Now when we check our condition for the forth time, because we've incremented our `counter` variable to 4, the condition `counter <= 3` evaluates to `False`. Our while loop will only continue to run if the condition is true, so now that it's false, our while loop closes and allows the computer to move on to the next block of code outside the while loop, which is our printed statement "Loop complete!".

So our final output would be:

```
1
2
3
Loop complete!
```

Using a flag to stop a while loop

The previous example is a simple one where we just want to run the while loop a certain number of times. But programs can be complicated, with lots of different branching logic. There might be many different conditions where we'd want our while loop to stop, depending on many different factors and considerations. In these situations, it's often helpful to use a *flag* to control your while loop.

Here's a slightly contrived example, but it gives you a reasonable idea how a flag can work.

```
active = True
countdown = 3

while active == True:
    if countdown == 0:
        countdown = 'Go!'
        active = False
    else:
        print(countdown)
        countdown -= 1

print(countdown)
```

Let's walk through this code.

First, we set and assign 2 variables: `active = True`, which we'll use as our flag to control the while loop, and `countdown = 3` which we'll use elsewhere in our code.

Next we create our while loop using `while active == True` which will force our while loop to keep running over and over until the `active` variable is set to something other than `True`.

Inside our while loop, we have a conditional if statement. Here's where our `countdown` variable comes into play. If our `countdown` variable is equal to 0, then we'll change its value to the string 'Go!' and set the `active` flag to `False`. Otherwise we follow the `else` pathway, where we'll print the current value of our `countdown` variable first before we reassign its value to itself - 1. (This works like the += we saw in the previous example, where we incremented the value of a variable by 1. This time we're decreasing the value of a variable by 1 instead.)

If the `countdown` variable is equal to anything except 0, the while loop will keep running over and over, printing the value of `countdown` then decreasing its value by 1 each time until it reaches 0. When it reaches 0, then we follow the other branch of our if statement, which sets our `countdown` variable to the string 'Go!' and – most importantly – sets the `active` variable to `False`. This means, when the while loop checks again… `active` does not equal `True`. Our while loop has finished running, and we move outside the loop to the last bit of code that needs to execute outside the loop, which is `print(countdown)`, outputting the final end value of our `countdown` variable.

The output of this program is below.

```
3
2
1
Go!
```

Using break to exit a loop

The above example can be written slightly more efficiently using the *break* keyword. This keyword allows us to immediately break out of a loop without executing any more code inside it, or requiring us to check the condition. It's kind of like slamming down on the brakes in a car: stop this while loop *immediately* instead of when this particular loop is complete.

Here's our revised code using the *break* keyword.

```
countdown = 3

while True:
    if countdown == 0:
        countdown = 'Go!'
        break

    print(countdown)
    countdown -= 1

print(countdown)
```

Notice we don't use a flag in this code: we simply write `while True` as our while loop condition. `True` is always true, so this loop is set to run forever – a potential infinite loop, which we'll talk about in greater detail below. But in this loop, we use the `break` keyword in our `if countdown == 0` conditional. As soon as this if statement is true – as soon as `countdown == 0`, we set the `countdown` variable to 'Go!' and then *immediately* break out of the loop *without* executing the other code inside the loop or checking to see if the `while True` condition is still true (which it is). This allows us to escape our loop.

Infinite loops

What happens if we create a loop like the one below?

```
counter = 1

while True:
    print(counter)
    counter += 1
```

This is an infinite loop. Since there's no way to get out of this loop – no condition that can ever be anything except true, and no *break* to get us out of the loop – this while loop will run… forever! (Or until your computer crashes.) I tried running this on my machine, and before I cancelled it just a few seconds later my printed counter had counted all the way up to 1080308!

Every programmer writes an occasional accidental infinite loop once in a while. It's always good to know how to cancel your program's execution manually in case you find yourself trapped in an infinite loop. If you're executing Python via the command line, like in the editors we provide below, you can use control-C to cancel the execution. If you're using a local code editor to run your Python, you should know what your specific editor or tool uses to cancel code execution.

Manipulating lists with while loops

After you've been programming for a while you'll find that while loops are useful in lots of different ways – it would take us a long time to definitively go through all the many different ways we can use them.

But here are a couple of quick examples of ways we can use while loops to manipulate lists.

```
invited = ['Agent Q', 'Agent M', 'Agent J', 'Agent S', 'Agent M']

while 'Agent M' in invited:
    invited.remove('Agent M')

invited.append('Agent M')
```

Here we have a list of people invited to attend an important meeting, but we see Agent M has been accidentally added twice.

This while loop first checks to see if the string `Agent M` can be found inside the list `invited`. As soon as it finds one, it removes it, then starts the while loop again. This repeats twice – because Agent M is listed twice in this list – and then the third time the while loop runs the check, Agent M is no longer anywhere in the list, so the loop is finished.

Finally, since we do still want Agent M to come to our meeting, we add her back to the list, knowing now she'll only be on there once.

Here's an example of how we can move our agents from one list to another after the meeting has happened, taking them from the invited list and moving them automatically onto the attended list.

```
invited = ['Agent Q', 'Agent M', 'Agent J', 'Agent S']
attended = []

while invited:
    current_agent = invited.pop()
    print(current_agent + ' attended the meeting.')
    attended.append(current_agent)

print('Attended list: ' + str(attended))
```

Here, the `while invited` is checking to see if the `invited` list has anything in it. As long as it has at least 1 item in it, the while loop is true, so it will execute the code inside it. Once we've moved everyone from the invited list to the attended list using the `pop()` and `append()` methods, the `invited` list is empty and so our while loop check is false. This allows us to escape our loop and print out our final attendee list.

Time Remaining: Loading… **Language:** Python 2 | Python 3 | C **Theme:** Dark | Light **Status:** ● Editor ● Terminal ● Server **Server:** Reset **Log:** Download

Steps

● **Step 1**
Create a list of the top 3 places you want to travel to. Then create a while loop to print out that list neatly.

○ **Step 2**
Modify your list so it's a dictionary, and for each place you want to visit add 3 reasons why. Then modify your program so that you use both a while loop and a for loop to print out the list of places, and for each place, a list of the reasons why you want to visit that place.

○ **Step 3**
Using a while loop, create a program with a counter that starts at 1, then takes that number and doubles it every time the loop runs. This while loop should keep going until the counter is greater than or equal to 100,000,000.

Editor – file: ~/output.py

```
counter = 1
moves = 0

while counter <= 100000000:
  counter *= 2
  moves += 1

print(counter, moves)
```

Save

Terminal – user: Plcgs40QPX

Completed

< back to module

Copyright 2020 SANS. Version 1.8.1