Loops Using while

by Sophia

WHAT'S COVERED

In this lesson, you will learn about some of the patterns that generate while loops when coding an algorithm. Specifically, this lesson covers:

1. Getting Started With while
2. break and continue Statements
3. else Statement

1. Getting Started With while

We looked at the `while` loop in the past lesson. As a recap, the structure of the `while` loop looks like the following:

EXAMPLE

```
while <expression>:
  <statement(s)>
```

Note: In the example above, the <expression> and <statement(s)> terms with outside arrows are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of the parts of a `while` loop.

The <statement(s)> represents the block of code that should be executed repeatedly; this is also called the body of the loop. The <expression> is typically based on one or more variables that are initialized outside of the loop and then modified within the body of the loop. The `while` loop continually evaluates the <expression> (the condition), looking for a True or False value. It keeps going (looping) as long as the evaluated condition is True. Once it is False, it exits the loop.

We also previously discussed the iteration variable. The body of the loop should change the value of one or more variables (in the <expression>) so that eventually the condition becomes false and the loop terminates. The iteration variable is what changes each time the loop executes and controls when the loop finishes. Without an iteration variable to change the condition, the loop would never finish.

Here is a simple program that counts down from five and then says “Blastoff!”

EXAMPLE

```
myNumber = 6
while myNumber > 0:
    myNumber = myNumber - 1
    print(myNumber)
print('Blastoff!')
```

Note that we initialize the variable `myNumber` with a numeric value of 6 and not 5. That is because in the body of the loop we immediately subtract 1 from `myNumber`. `myNumber` is the iteration variable. This variable will change with each loop (iteration) to eventually make the expression condition False and as such end the loop.

And if we run this program:

```
5
4
3
2
1
0
Blastoff!
```

We see the expected results.

You can almost read the `while` loop as if it were English. It means, “while `myNumber` is greater than 0, reduce the value of `myNumber` by 1 and then display the value of `myNumber`. When you get to 0, exit the `while` loop and display the word Blastoff!”

More formally, here is the flow of execution for a `while` loop:

1. Evaluate the expression’s condition, yielding True or False.
2. If the condition is False, exit the `while` loop and continue execution at the next statement or line of code.
3. If the condition is True, execute the body of the loop and then go and check the condition again.

Each time the body of the loop is executed is called an iteration. For the above `while` loop, we would say, “It had five iterations,” which means that the body of the loop was executed five times.
Let’s look at another example where we have a text entry that’s required to exit the loop.

EXAMPLE

```
textEntered = ""
stringBuilder = ""
while textEntered != "quit":
  textEntered = input("Enter in a string, enter quit to exit the loop:")
```

```
  if textEntered != "quit":
    stringBuilder += textEntered + " "
print(stringBuilder)
```

Let's break down this program.

```
textEntered = ""
stringBuilder = ""
```

Here, we have two string variables that are declared. The first `textEntered` will be for storing the user input. The second `stringBuilder` will be used to store the string being built.

```
while textEntered != "quit":
  textEntered = input("Enter in a string, enter quit to exit the loop:")
  if textEntered != "quit":
    stringBuilder += textEntered + " "
print(stringBuilder)
```

Next, we have a `while` loop that will loop while the `textEntered` string is not set to the word "quit". We'll first prompt the user for a string and then store it in `textEntered`. We'll check if the `textEntered` is not equal to "quit". If it isn't, then the `stringBuilder` adds the `textEntered` with a space. Once the user enters the word "quit", the loop ends and the `stringBuilder` string is output to the screen.

```
Enter in a string, enter quit to exit the loop:this
Enter in a string, enter quit to exit the loop:is
Enter in a string, enter quit to exit the loop:a
Enter in a string, enter quit to exit the loop:test
Enter in a string, enter quit to exit the loop:quit
this is a test
```

In this case, if the word "quit" is not entered in, the `while` loop can keep on going forever. Remember that this is a key part of a `while` loop: it will continue to run as long as the condition is True.

TRY IT

Directions: Go ahead and enter the program into the IDE and test the `while` loop.

```
textEntered = ""
stringBuilder = ""
while textEntered != "quit":
  textEntered = input("Enter in a string, enter quit to exit the loop:")
  if textEntered != "quit":
    stringBuilder += textEntered + " "
print(stringBuilder)
```

2. break and continue Statements

With the `while` loops that we've seen so far, we had the entire body of the loop executed on each iteration. Python has two reserved keywords that allow a loop to end execution early. The **break statement** can immediately terminate a loop entirely and disregard the execution of the loop. When this occurs, the program goes to the first statement/line of code after the loop. There is also the **continue statement**, which ends the current loop iteration, meaning the execution jumps to the top of the loop. The expression is then evaluated to determine if the loop will execute again or end there.

Let's see how the `break` statement would work.

EXAMPLE

```
myNumber = 6
while myNumber > 0:
    myNumber = myNumber - 1
    if myNumber == 2:
        break
    print(myNumber)
print('Blastoff!')
```

We set a variable called `myNumber` to 6. Then, we have a `while` loop that checks if `myNumber` is greater than 0. If it is not, it subtracts 1 from `myNumber`. We also have an if statement inside the `while` loop that is checking if myNumber is equal to 2. If it is, the `break` statement is called, the `while` loop is terminated completely, and "Blastoff!" is printed to the screen.

TRY IT

Directions: Go ahead and enter the program above into the IDE and let's run and test this `while` loop with a `break` statement.

```
5
4
3
Blastoff!
```

Did you get the same output? In the `while` loop, `myNumber` is first subtracted by 1 so the first print function outputs 5. Then 4, then 3. But now on the fourth iteration, `myNumber` is equal to 2 so that if statement inside of the `while` loop is triggered, the `break` statement cancels the `while` loop, and "Blastoff!" is output to the screen.

Let's try the same program, only this time we will switch to using the `continue` statement instead.

EXAMPLE

```
myNumber = 6
while myNumber > 0:
    myNumber = myNumber - 1
    if myNumber == 2:
        continue
    print(myNumber)
print('Blastoff!')
```

TRY IT

Directions: Go ahead and change your program in the IDE to use a `continue` statement rather than the `break` statement.

This time when `myNumber` is equal to 2, the `continue` statement is run, so the `print(myNumber)` function isn't executed. However, the processing goes back to the `while` loop evaluation and runs again as usual. The loop resumes and terminates when `myNumber` becomes 0 and the expression condition becomes False.

TRY IT

Directions: Go ahead and run the program. You should see the following output with the number 2 omitted since that iteration was cut short by the `continue` statement.

```
5
4
3
1
0
Blastoff!
```

TERMS TO KNOW

break
This is a reserved keyword that creates a break statement for loops. The `break` statement can immediately terminate a loop's execution. When this occurs, the program goes to the first statement/line of code after the loop.

continue
This is a reserved keyword that creates a continue statement for loops. The `continue` statement will end the current loop iteration, meaning that the execution jumps back to the top of the loop. The expression is then evaluated to determine if the loop will execute again or end there.

3. else Statement

Thinking back, we used the reserved keyword “else” as an else statement when using conditional if statements. If you recall, the else statement was referred to as a catchall case. Does this program ring a bell?

```
grade = 15
if grade > 90:
    print("You got an A")
elif grade > 80:
    print("You got a B")
elif grade > 70:
    print("You got a C")
elif grade > 60:
    print("You got a D")
else:
    print("You got a F")
```

Well, Python can also use this keyword as an optional else statement (also known as else clause for loops) at the end of a `while` loop. This is a unique feature of Python.

EXAMPLE

```
while <expression>:
  <statement(s)>
else:
  <additional statements>
```

Note: Remember in the example above, the terms <expression>, <statement(s)>, and <additional statements> and outside arrows are just for information purposes; these are not keywords or actual code. These are just to explain what goes into each of these parts of a `while` loop.

The <additional statements> that we have in the else statement will execute when the `while` loop terminates as long as it finishes when the expression’s condition becomes False. This is important, as any statement that is defined after the `while` loop like the following would always execute as follows:

```
while <expression>:
  <statement(s)>

<additonal statement>
```

Remember our example above:

```
myNumber = 6
while myNumber > 0:
    myNumber = myNumber - 1
    if myNumber == 2:
```

```
        continue
    print(myNumber)
print('Blastoff!')
```

The printing of "Blastoff!" is always done after the loop is completed regardless of what happens in the loop.

However, with the else statement, if the loop exits using a `break` statement, the else statement won't be executed. This is a unique part of the language. Let's take a look at two examples with and without using the `break` statement.

TRY IT

Directions: Go ahead and try this following program that uses both the `break` and else statements.

```
myNumber = 6
while myNumber > 0:
   myNumber = myNumber - 1
   if myNumber == 2:
       break
   print(myNumber)
else:
 print('Blastoff!')
```

The output of this program:

```
5
4
3
```

Did you notice that after the `break` is called, "Blastoff!" is not output to the screen. That is because the `break` statement will omit the else statement when used like this.

However, if the loop ends normally (without any `break` statement), the else statement will be called:

TRY IT

Directions: Now test this program that does not use a `break` statement, only the else statement.

```
myNumber = 6
while myNumber > 0:
   myNumber = myNumber - 1
   print(myNumber)
else:
 print('Blastoff!')
```

The output of this program:

```
5
4
3
2
1
0
Blastoff!
```

In this case, there was no `break` statement so the else statement is called after the loop exits.

SUMMARY

In this lesson, we learned about the **while loop** in more detail. We also learned about the **break** statement, which allows us to exit out of a loop, and the **continue** statement, which allows us to jump past the end of the loop (that current iteration) and continue back at the loop's conditional check. We also learned about the **else statement**, which is rarely used, but executes if the expression's condition of the loop becomes False.

Best of luck in your learning!

TERMS TO KNOW

break

This is a reserved keyword that creates a break statement for loops. The break statement can immediately terminate a loop's execution. When this occurs, the program goes to the first statement/line of code after the loop.

continue

This is a reserved keyword that creates a continue statement for loops. The continue statement will end the current loop iteration, meaning that the execution jumps back to the top of the loop. The expression is then evaluated to determine if the loop will execute again or end there.