# Loops

In this module you'll explore the intricacies of loops in Python! You'll learn how to use while loops to continuously execute code, as well as how to identify infinite loop errors and how to fix them. You'll also learn to use for loops to iterate over data, and how to use the range() function with for loops. You'll also explore common errors when using for loops and how to fix them.

**Learning Objectives**

* Implement while loops to continuously execute code while a condition is true
* Identify and fix infinite loops when using while loops
* Utilize for loops to iterate over a block of code
* Use the range() function to control for loops
* Use nested while and for loops with if statements
* Identify and correct common errors when using loops

## While Loops

### Introduction to Loops

So far we've seen how to organize our code and functions. We've also made our code branch in two different paths depending on certain conditions.

In this module we'll learn how to get computers to do repetitive tasks, which is another cornerstone of programming. computers are great at repeating the same task over and over. They never get bored or make a mistake. You could ask a computer to do the same calculation a thousand times and the first result would be just as accurate as the last. The ability to accurately perform repetitive tasks and never get tired is why computers are so great for automation. The automated task could be anything like copying files to a bunch of computers on a network, sending personalized emails to a list of users, or verifying that a process is still running. It doesn't matter how complex the task is, your computer will do it as many times as you tell it to, which leaves you time for more interesting things like planning future hardware needs, or managing software rollouts. In the next few videos we'll explore three techniques for automating repetitive tasks. These are **while loops**, **for loops**, and **recursion**.

Each of these techniques are used to tell the computer to repeat a task, but each takes a slightly different approach. We're going to learn how to write the code for each and how to know when to use one technique instead of the others. So, are you ready? Let's get started.

### Review: What is a while loop?

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

x = 0

while x < 5:

print("Not there yet, x=" + str(x))

x = x + 1

print("x=" + str(x))

### What is a while loop?

First off, we're going to talk about while loops.

While loops instruct your computer to continuously execute your code based on the value of a condition.

This works in a similar way to branching if statements. The difference here is that the body of the block can be executed multiple times instead of just once.

Check out this program. Can you guess what it does?

x = 0

while x < 5:

print("Not there yet, x=" + str(x))

x = x + 1

print("x=" + str(x))

Before we execute it to find out, let's go through it together line by line.

In the first line we're assigning the value of 0 to the variable x. We call this action initializing, to give an initial value to a variable.

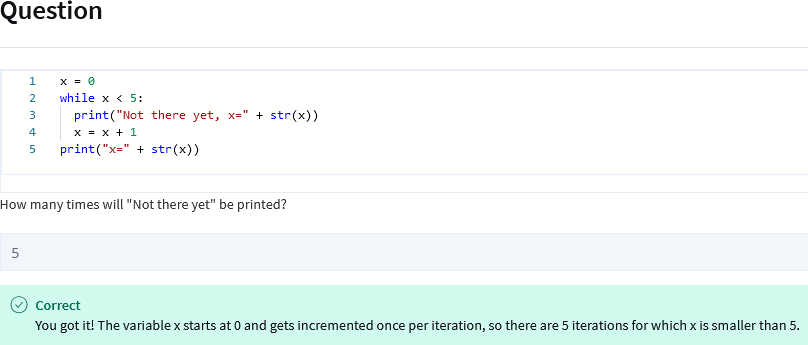
In the line after that, we're starting the while loop. We're setting a condition for this loop that x needs to be smaller than 5.

Right now we know that x is 0 since we've just initialized it, so this condition is currently true.

On the next two lines, we have a block that's indented to the right. Here, we can use what we learned about functions and conditionals to identify that this is the while loop's body. There are two lines in the body of the loop. In the first line, we print a message followed by the current value of x. In the second line, we increment the value of x. We do this by adding 1 to its current value and assigning it back to x.

So after the first execution of the body of the loop, x will be 1 instead of 0. Because this is a loop, the computer doesn't just continue executing with the next line in the script. Instead, it loops back around to re-evaluate the condition for the while loop. And because 1 here is still smaller than 5, it executes the body of the loop again. It then prints the message and once more increments x by 1. So the x is now 2. The computer will keep doing this until the condition isn't true anymore. In this example, the condition will be false when x is no longer smaller than 5.

Once the condition is false, the loop finishes, and the next line is executed. And finally, the last line of our code prints the last value of x. So now that this code makes a bit more sense, what do you think will happen when we execute it? Ready to find out? Let's execute the code and see what happens.



So we had five lines with the message, Not there yet, and then at the end of the script the value of x was 5. This was a simple example of how a while loop behaves. As we've said before, we're learning the building blocks of programming. Once you know those building blocks, you can combine them to create more complex expressions. As an IT specialist, while loops can be super helpful. You can use them to keep asking for a username if the one provided isn't valid, or maybe try an operation until it succeeds. Knowing how to construct these expressions can help you get your computer to do a whole lot with only a little bit of code. It's pretty powerful stuff we're learning here. Now that you've got an idea of how a while loop works, let's spice it up with another example.

### Anatomy of a While Loop

A *while* loop will continuously execute code depending on the value of a condition. It begins with the keyword *while,* followed by a comparison to be evaluated, then a colon. On the next line is the code block to be executed, indented to the right. Similar to an *if* statement, the code in the body will only be executed if the comparison is evaluated to be true. What sets a *while* loop apart, however, is that this code block will keep executing as long as the evaluation statement is true. Once the statement is no longer true, the loop exits and the next line of code will be executed.

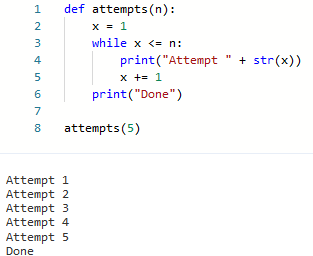
### Review: More while loop examples

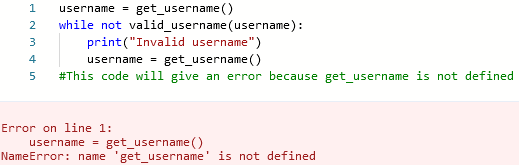
This reading contains the code used in the instructional videos from [**More while loop examples**](https://www.coursera.org/learn/python-crash-course/lecture/wXvhG/more-while-loop-examples)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. You will find additional information throughout the guide to explain the purpose of each concept covered, why the code is written in a certain way, and tips for running the code. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.





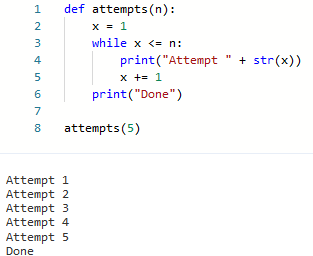
**About this code**

The username variable is assigned the value returned by the get\_username() function. The while loop starts. The loop will continue to iterate as long as the value of the username variable is not valid. The print() function prints the message "Invalid username". The username variable is assigned the value returned by the get\_username() function. The while loop ends.

The code you provided will give an error because the *valid\_username()* function is not defined. The *while* loop will try to call the *valid\_username()* function on line 3, but the function does not exist.

### More while loop examples

In the last video, we saw a very simple example of a while loop. We looked at the basic syntax of the loop and how it works. Let's now apply this knowledge to a similar example, but this time with a while loop inside a function. Can you work out what this function does?

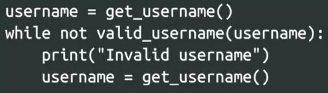


In this example, we start out by initializing a variable called x. In this case, we initialize it with a value of 1. Then, we enter our while loop which checks to see if the value inside of the x variable is less than the parameter n that the function received. If that comparison evaluates to true, then the code inside the while block is executed. Say we pass a value of 5 as a parameter to this function. In the first pass through the loop, x is always equal to 1, so the comparison: 1 smaller than or equal to 5 would be true and we then enter the body of the loop. In the body, we first print a message indicating that the current attempt number and then we increase the value of x by 1.

To increment the number we're using a slightly different expression than before. x +=1 is a shorthand version of x = x+1. You can use either expression since they both mean the same thing. The process continues until the result of the comparison isn't true anymore, which happens when x is bigger than n.

In our current example, this would be when the value of x is 6. Let's see it in action. In these past examples, we've used the simple conditions of a number being smaller, or smaller or equal than another number. These are common conditions, but they're by no means the only conditions you can have in a while loop.

It's common, for example, to call a separate function that evaluates the condition, like this.



In this case, there's a lot of code hidden behind functions and it's doing stuff we don't see. There's a get username function that asks the user for a username and a valid\_username function that validates that username. And all this is happening in just a handful of characters.

As you can see, you can pack a lot of punch into just a short line of code. In this case, the body of the while loop will be executed until the user enters a valid username. The important thing to remember is that the condition used by the while loop needs to evaluate to true or false. It doesn't matter if this is done by using comparison operators or calling additional functions. The conditions used in while loops can also become more complex if we use the logical operators that we encountered when looking into branching, and, or, and not. This lets us combine the values of several expressions to get the result we want.

Next, we're going to do a rundown of some of the most common pitfalls that you may come across when writing your own loops. Head on over to the next video to get started.

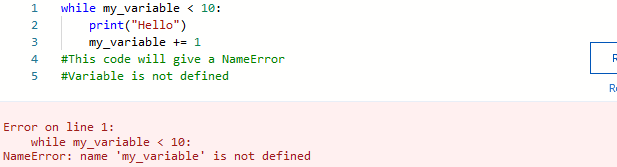
### Review: Why initializing variables matters

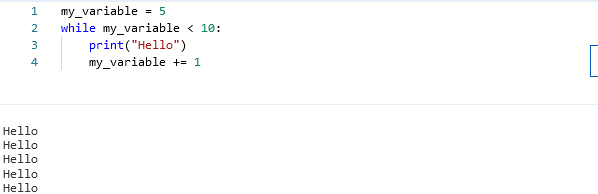
This reading contains the code used in the instructional videos from [**Why initializing variables matters**](https://www.coursera.org/learn/python-crash-course/lecture/oRAfO/why-initializing-variables-matters)

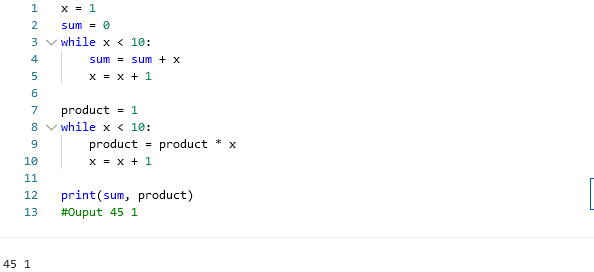
**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. You will find additional information throughout the guide to explain the purpose of each concept covered, why the code is written in a certain way, and tips for running the code. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.







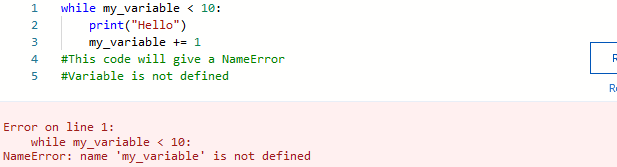
**About this code**

This code contains an error, but it will not give an error when the code is run. The problem here is that product should be set to product = *x*. In this code, x is never initialized so this code is false before it is even run. The product variable has the wrong value

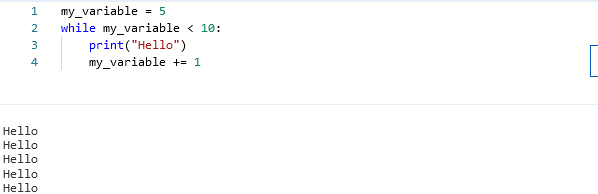
### Why Initializing Variables Matters

As we've called out earlier, writing loops allows us to get our computer to do repetitive work for us. Since one of the main benefits of writing scripts in IT is to save time by automating repetitive tasks, loops are super useful. So let's make sure you avoid some of the most common mistakes people make when writing loops. One of the most common errors is forgetting to initialize variables with the right value. We've all made this mistake when starting to code.

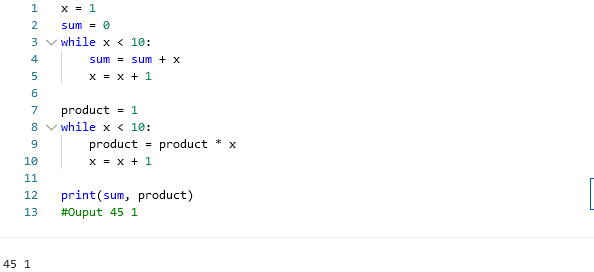
Remember how in the earlier examples we initialized our variable x to 0 in one case and to 1 in the other. When we forget to initialize the variable two different things can happen. The first possible outcome and the easiest to catch is that Python might raise an error telling us that we're using a variable we haven't defined, which looks like this.



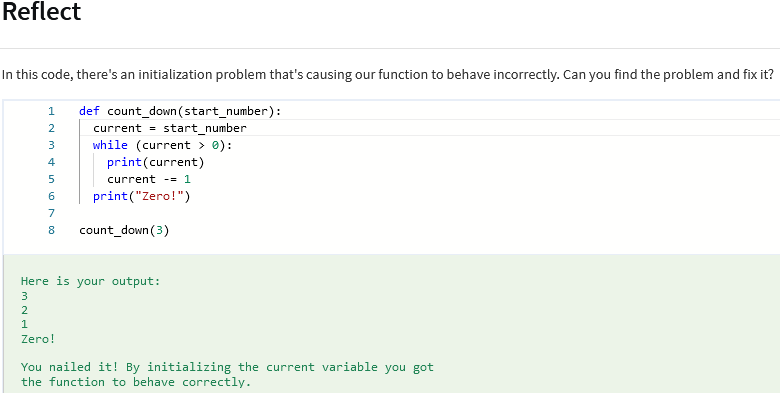
As we've done with other errors we've come across, we can look at the last line to understand what's going on. This error type is a name error, and the message that comes after it says we're using an undefined variable. It's straightforward to fix, we just need to initialize the variable before using it, like this:



Fixed. Now, there's a second issue we might face if we forget to initialize variables with the right value. We might have already used the variable in our program. In this case, if we reuse the variable without setting the correct value from the start, it will still have the value from before. This can lead to some pretty unexpected behavior. Check out this script. Can you spot the problem?



In the first block, we correctly initialize x to 1 and sum to 0 and then iterate until x equals 10, summing up all the values in between. So by the end of that block, sum equals the result of adding all the numbers from 1 to 10 and x is 10. In the second part of the code, the original intention was to get the product of all the numbers from 1 to 10. But if you look closely, you can see that we're initializing the product but forgetting to initialize x, so x is still 10. This means that when the while condition gets checked, x is already 10 at the start of the iteration. The while condition is false before it even starts and the body never executes. Let's see how this problem would look. In this case, it might be harder to catch the problem because python doesn't raise an error. The problem here is that our product variable has the wrong value. If you have a loop that's gone rogue and is not behaving as expected, it's a good idea to check if all the variables are correctly initialized. In this example, we need to set x back to 1 before starting the second loop. As always, the best way to learn is to practice it yourself.



(inserted the current = start\_number line to initialize it)

Make sense? Remember, if you ever feel stuck or a little unsure about something you can always ask for help in the discussion forums. These forums are there to let you get the help you need when you need it, so don't forget to use them.

So, to recap, whenever you're writing a loop check that you're initializing all the variables you want to use before you use them.

And don't worry if you don't get it right the first time, we've all been there when learning how to code. As we've called out before, the way to master programming is to practice, practice, practice. Keep practicing until you're comfortable and even then it's still okay to make mistakes. So don't feel like you can't loop back around to review and practice everything we've covered so far.

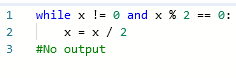
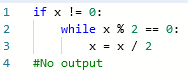
### Review: Infinite loops and how to break them

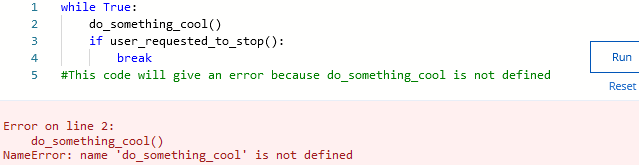
This reading contains the code used in the instructional videos from [**Infinite loops and how to break them**](https://www.coursera.org/learn/python-crash-course/lecture/qt7y9/infinite-loops-and-how-to-break-them)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. You will find additional information throughout the guide to explain the purpose of each concept covered, why the code is written in a certain way, and tips for running the code. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.





**About this code**

This code block is an example snippet of code. In your code, you could have an infinite loop that looks something like this. In Python, we use the break keyword which you can see here to signal that the current loop should stop running. We can use it not only to stop infinite loops but also to stop a loop early if the code has already achieved what's needed.

### Infinite Loops and How to Break Them

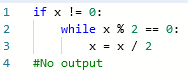
You may remember by now that while loops use a condition to check when to exit. The body of the while loop needs to make sure that the condition being checked will change. If it doesn't change, the loop may never finish and we get what's called an infinite loop, a loop that keeps executing and never stops.

Check out this example.



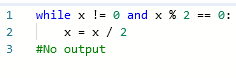
It uses the modulo operator that we saw a while back. This cycle will finish for positive and negative values of x. But what would happen if x was zero? The remainder of 0 divided by 2 is 0, so the condition would be true. The result of dividing 0 by 2 would also be zero, so the value of x wouldn't change. This loop would go on forever, and so we'd get an infinite loop. If our code was called with x having the value of zero, the computer would just waste resources doing a division that would never lead to the loop stopping. The program would be stuck in an infinite loop circling background endlessly.

To avoid this, we need to think about what needs to happen for a loop to be successful. In this example, we said that x needs to be different than zero. So we could nest this while loop inside an if statement just like this.



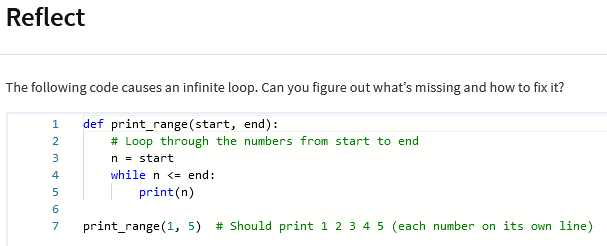
With this approach, the while loop is executed only when x is not zero.

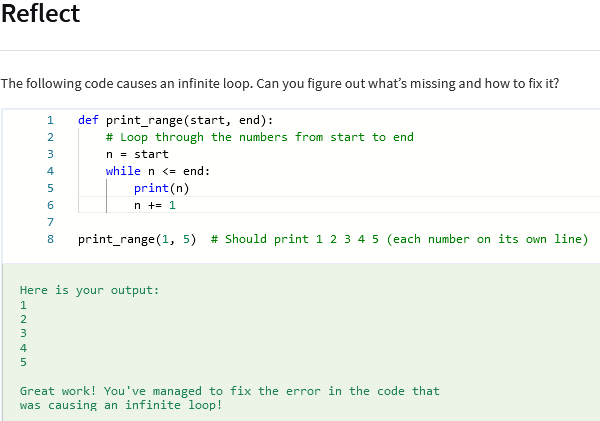
Alternatively, we could add the condition directly to the loop using a logical operator like in this example.



This makes sure we only enter the body of the loop for values of x that are both different than zero and even. Talking about infinite loops reminds me of one of the first times I used while loops myself. I wrote a script that emailed me as a way of verifying that the code worked, and while some condition was true, I forgot to exit the loop. Turns out those e-mails get sent faster than once per second. As you can imagine, I got about 500 e-mails before I realized what was going on. Infinitely grateful for that little lesson.

When you're done laughing at my story, remember, when you're writing loops, it's a good idea to take a moment to consider the different values a variable can take. This helps you make sure your loop won't get stuck. If you see that your program is running forever without finishing, have a second look at your loops to check there's no infinite loop hiding somewhere in the code.

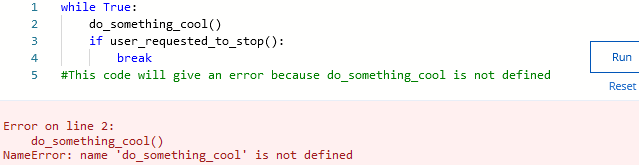




While you need to watch out for infinite loops, they are not always a bad thing. Sometimes you actually want your program to execute continuously until some external condition is met. If you've used the ping utility on Linux or macOS systems, or ping-t on a Windows system, you've seen an infinite loop in action. This tool will keep sending packets and printing the results to the terminal unless you send it the interrupt signal, usually pressing Ctrl+C. If you were looking at the program source code, you'll see that it uses an infinite loop to do this with a block of code with instructions to keep sending the packets forever.

One thing to call out is it should always be possible to break the loop by sending a certain signal. In the ping example, that signal is the user pressing Ctrl+C. In other cases, it could be that the user pressed the button on a graphical application, or that another program sent a specific signal, or even that a time limit was reached.

In your code, you could have an infinite loop that looks something like this.



In Python, we use the break keyword which you can see here to signal that the current loop should stop running. We can use it not only to stop infinite loops but also to stop a loop early if the code has already achieved what's needed.

So quick refresh. How do you avoid the most common pitfalls when writing while loops? First, remember to initialize your variables, and second, check that your loops won't run forever.

### Study guide: while loops

This study guide provides a quick-reference summary of what you learned in this segment and serves as a guide for the upcoming practice quiz.

In the *while* loops segment, you learned about the logical structure and syntax of *while* loops. You also learned about the importance of initializing variables and how to resolve infinite *while* loops with the *break* keyword.

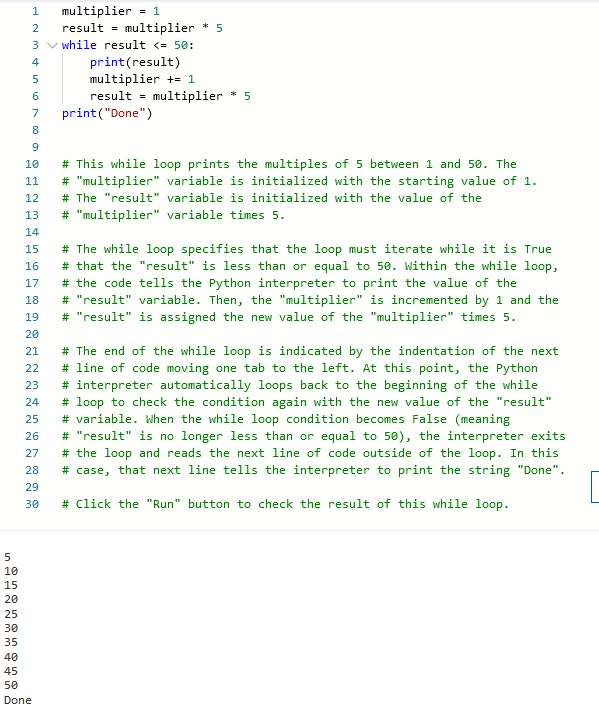
***while* loops**

A *while* loop executes the body of the loop while a specified condition remains *True*. They are commonly used when there’s an unknown number of operations to be performed, and a condition needs to be checked at each iteration.

Syntax:



Example *while* loop



**Common errors in Loops**

If you get an error message on a loop or it appears to hang, your debugging checklist should include the following checks:

* **Failure to initialize variables**. Make sure all the variables used in the loop’s condition are initialized before the loop.
* **Unintended infinite loops**. Make sure that the body of the loop modifies the variables used in the condition, so that the loop will eventually end for all possible values of the variables. You can often prevent an infinite loop by using the break keyword or by adding end criteria to the condition part of the *while* loop.

**Terms**

* *while loop* - Tells the computer to execute a set of instructions while a specified condition is *True*. In other words, *while* loops keep executing the same group of instructions until the condition becomes *False*.
* *infinite loop* - Missing a method for exiting the loop, causing the loop to run forever.
* *break* - A *break* statement in Python provides a way to exit out of a loop before the loop's condition is false. Once a break statement is encountered, the program's control flow jumps out of the loop and continues executing the code after the loop.
* *pass* - A *pass* statement in Python is a placeholder statement which is used when the syntax requires a statement, but you don't want to execute any code or command.

**Math concepts on the practice quiz**

The coding problems on the upcoming practice quiz will involve a few math concepts. Don’t worry if you are rusty on math. You will have plenty of support with these concepts on the quiz. The following is a quick overview of the math terms you will encounter on the quiz:

* **prime numbers** - Integers that have only two factors, which are the number itself multiplied by 1. The first ten prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23 and 29. Each of these prime numbers can be evenly divided only by itself and 1.
* **prime factors** - Prime numbers that are factors of an integer. For example, the prime numbers 2 and 5 are the prime factors of the number 10 (2x5=10). The prime factors of an integer will not produce a remainder when used to divide that integer.
* **divisor** - A number (denominator) that is used to divide another number (numerator). For example, if the number 10 is divided by 5, the number 5 is the divisor.
* **sum of all divisors of a number** - The result of adding all of the divisors of a number together.
* **multiplication table** - An integer multiplied by a series of numbers and their results formatted as a table or a list. For example:

4x1=4

4x2=8

4x3=12

4x4=16

4x5=20

**Coding skills**

The following are sample skill areas to study for *while* loops.

**Skill group 1**

* Initialize a variable
* Use a *while* loop that runs while a specific condition is true
* Ensure the *while* loop will not be an infinite loop
* Increment a value within a *while* loop

# This function counts the number of integer factors for a

# "given\_number" variable, passed through the function’s parameters.

# The "count" return value includes the "given\_number" itself as a

# factor (n\*1).

def count\_factors(given\_number):

# To include the "given\_number" variable as a "factor", initialize

# the "factor" variable with the value 1 (if the "factor" variable

# were to start at 2, the "given\_number" itself would be excluded).

factor = 1

count = 1

# This "if" block will run if the "given\_number" equals 0.

if given\_number == 0:

# If True, the return value will be 0 factors.

return 0

# The while loop will run while the "factor" is still less than

# the "given\_number" variable.

while factor < given\_number:

# This "if" block checks if the "given\_number" can be divided by

# the "factor" variable without leaving a remainder. The modulo

# operator % is used to test for a remainder.

if given\_number % factor == 0:

# If True, then the "factor" variable is added to the count of

# the "given\_number"’s integer factors.

count += 1

# When exiting the if block, increment the "factor" variable by 1

# to divide the "given\_number" variable by a new "factor" value

# inside the while loop.

factor += 1

# When the interpreter exits either the while loop or the top if

# block, it will return the value of the "count" variable.

return count

print(count\_factors(0)) # Count value should be 0

print(count\_factors(3)) # Should count 2 factors (1x3)

print(count\_factors(10)) # Should count 4 factors (1x10, 2x5)

print(count\_factors(24)) # Should count 8 factors (1x24, 2x12, 3x8, and 4x6).

Output:

0

2

4

8

**Skill group 2**

* Initialize variables to assign data types before they are used in a *while* loop
* Use the *break* keyword as an exit point for a *while* loop

# This function outputs an addition table. It is written to end after

# printing 5 lines of the addition table, but it will break out of the

# loop if the "my\_sum" variable exceeds 20.

# The function accepts a "given\_number" variable through its

# parameters.

def addition\_table(given\_number):

# The "iterated\_number" and "my\_sum" variables are initialized with

# the value of 1. Although the "my\_sum" variable does not need any

# specific initial value, it still must be assigned a data type

# before being used in the while loop. By initializing "my\_sum"

# with any integer, the data type will be set to int.

iterated\_number = 1

my\_sum = 1

# The while loop will run while it is True that the

# "iterated\_number" is less than or equal to 5.

while iterated\_number <= 5:

# The "my\_sum" variable is assigned the value of the

# "given\_number" plus the "iterated\_number" variables.

my\_sum = given\_number + iterated\_number

# Test to see if the "my\_sum" variable is greater than 20.

if my\_sum > 20:

# If True, then use the break keyword to exit the loop.

break

# If False, the Python interpreter will move to the next line

# in the while loop after the if-statement has ended.

# The print function will output the "given\_number" plus

# the "iterated\_number" equals "my\_sum".

print(str(given\_number), "+", str(iterated\_number), "=", str(my\_sum))

# Increment the "iterated\_number" before the while loop starts

# over again to print a new "my\_sum" value.

iterated\_number += 1

addition\_table(5)

addition\_table(17)

addition\_table(30)

# Expected output:

# 5 + 1 = 6

# 5 + 2 = 7

# 5 + 3 = 8

# 5 + 4 = 9

# 5 + 5 = 10

# 17 + 1 = 18

# 17 + 2 = 19

# 17 + 3 = 20

# None

Output:

5 + 1 = 6

5 + 2 = 7

5 + 3 = 8

5 + 4 = 9

5 + 5 = 10

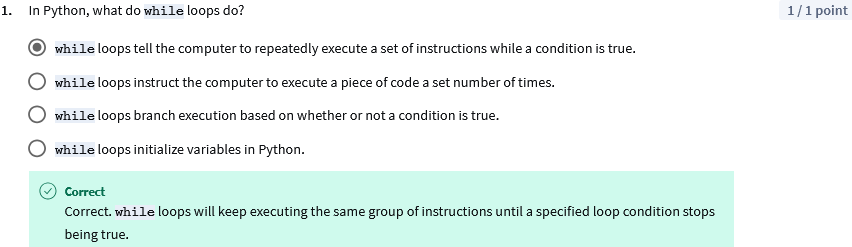
17 + 1 = 18

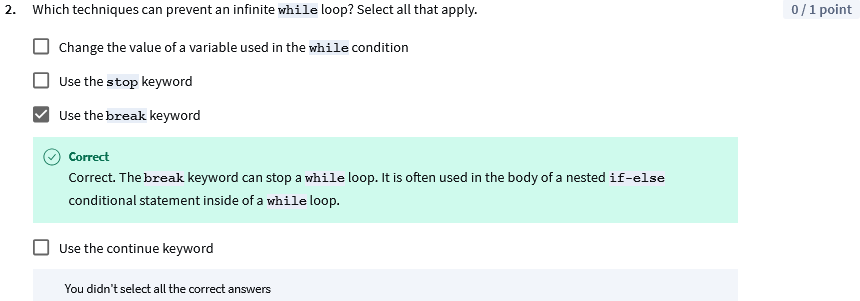
17 + 2 = 19

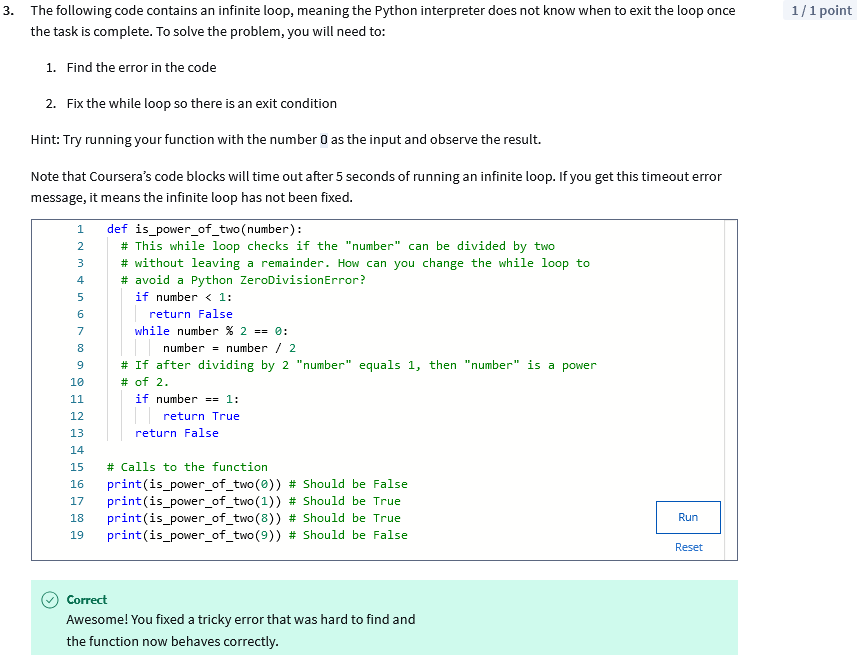
17 + 3 = 20

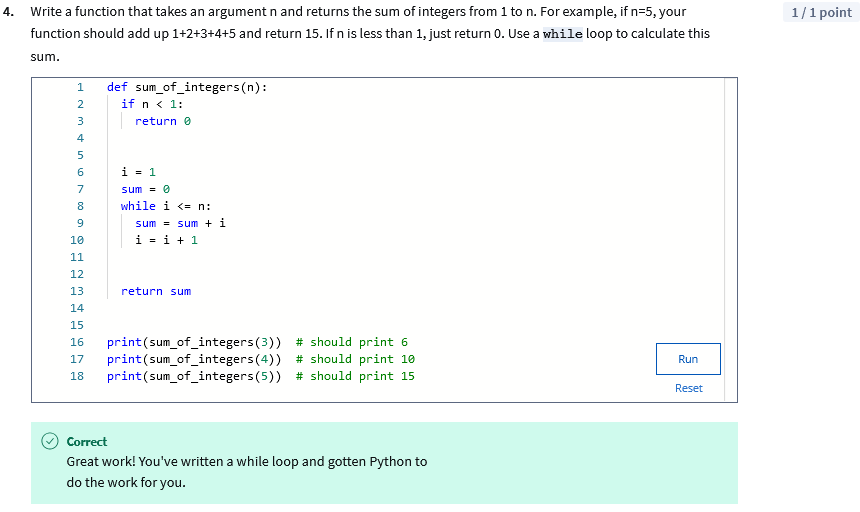
### Practice quiz: While loops













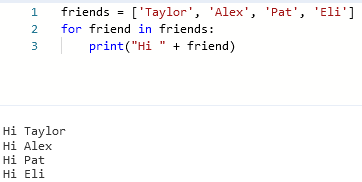
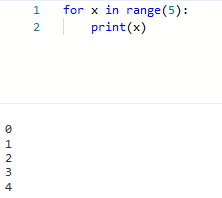
## For Loops

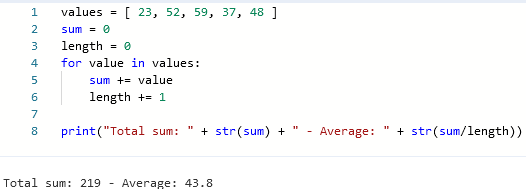
### Review: What is a for loop?

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.





### What is a for loop?

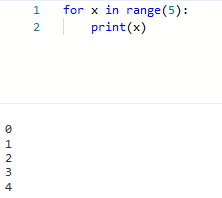
In this video, we're going to meet the for loop. A for loop iterates over a sequence of values.

A very simple example of a for loop is to iterate over a sequence of numbers, like this. Notice how the structure is kind of similar to the structures we've already seen. The first line indicates the distinguishing keyword. In this case, that's for. And it ends with a colon. The body of the loop is indented to the right, like we saw in the while loop, the if block, and the function definitions.

What's different in this case is that we have the **keyword “in”**. Also, between the for keyword and in keyword, we have the name of a variable. This variable will take each of the values in the sequence that loop iterates through. So in this example, it'll iterate through a sequence of numbers generated using the range function.

There are two important things I want to call out about this range function. First, in Python and a lot of other programming languages, a range of numbers will start with the value 0 by default.

Second, the list of numbers generated will be one less than the given value. In the simple example here, x will take the values 0, 1, 2, 3, and 4. Let's check this out.



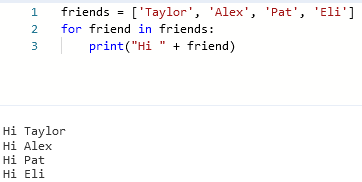
So there, we have a very basic for loop. It iterates over a sequence of numbers generated by the range function. When using a for loop, we point the variable defined between for and in, in this case, x, at each element of the sequence.

This means on the first iteration x points at 0. On the second iteration, it points at 1, and so on. Whatever code we put in the body of the loop will be executed on each of the values, one value at a time.

As we said earlier, the loop's body can do a lot of things with the values it iterates. For example, you could have a function to calculate the square of a number, and then use a for loop to sum the squares of the numbers in a range.

Iterating over numbers looks very similar to the while loop examples we showed before. So you may be wondering why have two loops that look like they do the same thing? Well, the power of the for loop is that we can use it to iterate over a sequence of values of any type, not just a range of numbers. Think all the way back to our very first Python example in this course. Remember our trusty "hi friends" script?

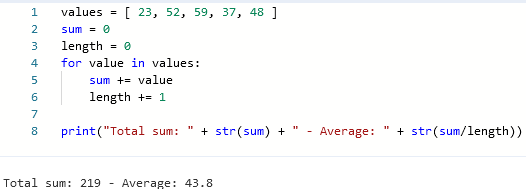
In it, we saw a for loop that iterated over a list of strings. It looks like this.



We'll talk a lot more about lists later on. But for now, you only need to know that we can construct lists using square brackets, and separate the elements in them with commas.

In this example, we're iterating a list of strings. And for each of the strings in the list, we're printing a greeting.

The sequence that the for loop iterates over could contain any type of element, not just strings. For example, we could iterate over a list of numbers to calculate the total sum and average. Here's one way of doing this.



Here, we're defining a list of values. After that, we're initializing two variables, sum and length, that will update in the body of the for loop. In the for loop, we're iterating over each of the values in the list, adding the current value to the sum of values, and then also adding 1 to length, which calculates how many elements there are in the list. Once we've gone through the whole list, we print out the sum and the average.

We'll keep using for loops in our examples every time we want to iterate over the elements of any sequence and operate with them. Some examples of sequences that we can iterate are

* the files in a directory
* the lines in a file
* the processes running on a machine.

And there's a bunch of others. So as an IT specialist, you'll use for loops to automate tons of stuff. For example, you might use them to:

* copy files to machines
* process the contents of files
* automatically install software

and a lot more.

A few weeks ago, I had to update a lot of files with different values depending on their contents. So I used a for loop in a script to iterate over all the files. Then, my script took different actions based on an if condition and updated all of those files for me. It would have taken me forever if I had done this manually file by file.

If you're wondering when you should use for loops and when you should use while loops, there's a way to tell.

Use for loops when there's a sequence of elements that you want to iterate.

Use while loops when you want to repeat an action until a condition changes.

And if whatever you're trying to do can be done with either for or while loops, just use whichever one's your favorite.

Next up, we've put together more examples to help get you more practice with for loops and discover some of the cool things you could do with them.

### Review: More for loops examples

This reading contains the code used in the instructional videos from [**More for loops examples**](https://www.coursera.org/learn/python-crash-course/lecture/00vRH/more-for-loop-examples)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

product = 1

for n in range(1,10):

product = product \* n

print(product)

**About this code**

This code first defines a variable product and initializes it to 1. Then, it uses a for loop to iterate over the range of numbers from 1 to 9. In each iteration of the loop, the variable n is assigned the current number in the range. The code then multiplies product by n. This means that after the first iteration, product will be equal to 1 \* 1 = 1. After the second iteration, product will be equal to 1 \* 2 = 2. And so on.

The loop will terminate when the value of n is 9. At this point, the value of product will be 1 \* 2 \* 3 \* ... \* 9 = 362880.

def to\_celsius(x):

return (x-32)\*5/9

for x in range(0,101,10):

print(x, to\_celsius(x))

### More for loop examples

In the last video, we talked about the range function and how it generates a sequence of numbers starting with zero. Sometimes though, we don't wanna start with zero. For these situations, the range function also allows us to specify the first element of the list to generate. We do that by passing two parameters to the function instead of one, like in the next example:

product = 1

for n in range(1,10):

product = product \* n

print(product)

In this example, we're calculating the product of all numbers from one to nine. For this operation, it's important that we start with one and not with zero. If we'd started with zero, the whole product would be zero.

Additionally, we can specify a third parameter to change the size of each step. This means that instead of going one by one, we could have a larger difference between the elements. Let's check out this example of when you might wanna do something like this.

def to\_celsius(x):

return (x-32)\*5/9

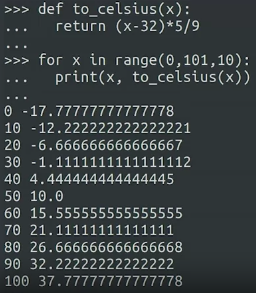
for x in range(0,101,10):

print(x, to\_celsius(x))

First, we're defining a function that converts a temperature value from Fahrenheit to Celsius, and we're simply using a conversion formula to do that. Then, we have a for loop that starts at zero and goes up to 100 in steps of 10.

**Notice that we're using 101 for the upper limit instead of 100. We're doing this because the range never includes the last element, and we want to include 100 in our range.**

The body of the for loop prints, the value in Fahrenheit, and the value in Celsius creating a conversion table. Let's see this in action.



That example got you feeling the heat? Don't worry, there's a quick rundown of what we've learned.

The range function can receive one, two, or three parameters.

If it receives one parameter, it will create a sequence, one by one, from zero until one less than the parameter received. If it receives two parameters, it will create a sequence, one by one, from the first parameter until one less than the second parameter. Finally, if it receives three parameters, it will create a sequence starting from the first number and moving towards the second number but this time, the jumps between the numbers will be the size of the third number. And again, it will stop before the second number.

This might sound like a lot to remember, but don't panic. As we've said before, you don't have to try to memorize it all. Just keep practicing. It'll soon become second nature.

### A Closer Look at the Range() Function

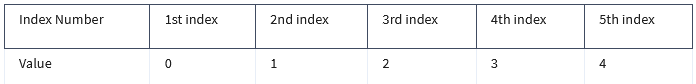
The **in** keyword, when used with the **range()** function, generates a sequence of integer numbers, which can be used with a **for** loop to control the start point, the end point, and the incremental values of the loop.

**Syntax:**

for n in range(x, y, z):

print(n)

The **range()** function uses a set of indices that point to integer values, which start at the number 0. The numeric values 0, 1, 2, 3, 4 correlate to ordinal index positions 1st, 2nd, 3rd, 4th, 5th. So, when a range call to the 5th index position is made using **range(5)** the index is pointing to the numeric value of 4.



The **range()** function can take up to three parameters: **range(start, stop, step)**

**Start** The first item in the **range()** function parameters is the starting position of the range. The default is the first index position, which points to the numeric value 0. This value is included in the range.

**Stop** The second item in the **range()** function parameters is the ending position of the range. There is no default index position, so this index number must be given to the **range()** parameters. For example, the line **for n in range(4)** will loop 4 times with the **n** variable starting at 0 and looping 4 index positions: 0, 1, 2, 3. As you can see, **range(4)** (meaning index position 4) ends at the numeric value 3. In Python, this structure may be phrased as “the end-of-range value is *excluded* from the range.” In order to include the value 4 in **range(4)**, the syntax can be written as **range(4+1)** or **range(5)**. Both of these ranges will produce the numeric values 0, 1, 2, 3, 4.

**Step** The third item in the **range()** function parameters is the incremental step value. The default increment is +1. The default value can be overridden with any valid increment. However, note that the loop will still end at the end-of-range index position, regardless of the incremental value. For example, if you have a loop with the range: **for n in range(1, 5, 6)**, the range will only produce the numeric value 1. This is because the incremental value of 6 exceeded the ending point of the range.

**Practice Exercise**

You can use the code block below to test the values of **n** with various **range()** parameters. A few suggestions to test are:

**range(stop)**

* range(3)
* range(3+1)

**range(start, stop)**

* range(2, 6)
* range(5,10+1)

**range(start, stop, step)**

* range(4, 15+1, 2)
* range(2\*2, 25, 3+2)
* range(10, 0, -2)

for n in range(5, 10+1):

print(n)

**Examples of the range() function in code:**

**Example 1**

# This loop iterates on the value of the "n" variable in a range

# of 0 to 10 (the value of the end-of-range index 11 is excluded).

# The incremental value for the loop is 2. The print() function will

# output the resulting value of "n" as the loop counts from 0 to 10

# (end-of-range index 11) in incremental steps of 2. This is one

# method that can be used in Python to print a list of even numbers.

for n in range(0,11,2):

print(n)

# The loop should print 0, 2, 4, 6, 8, 10

**Example 2**

# This loop iterates on the value of the "number" variable in a range

# of 2 to 7+1 (the value of the end-of-range index 7 is excluded, so

# +1 has been added to the parameter to include the numeric value 7 in

# the range). The incremental value for the loop is the default of +1.

# The print() function will output the resulting value of "number"

# multiplied by 3.

for number in range(2,7+1):

print(number\*3)

# The loop should print 6, 9, 12, 15, 18, 21

**Example 3**

# This loop iterates on the value of the "x" variable in a range

# of 2 to -1 (the end-of-range index -2 is excluded). The third

# parameter is also a negative number, making it a decremental value

# of -1. The print() function will output the resulting value of

# "x" as it starts at 2 and counts down to -1 (index -2).

for x in range(2, -2, -1):

print(x)

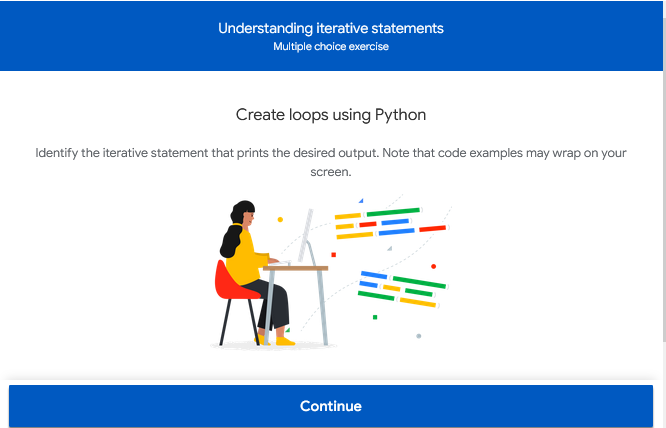
# The loop should print 2, 1, 0, -1

**Key takeaways**

The roles of the **range(start, stop, step)** function parameters are:

* **Start** - Beginning of range
  + value included in range
  + default = 0
* **Stop** - End of range
  + value excluded from range (to include, use stop+1)
  + no default
  + must provide the ending index number
* **Step** - Incremental value
  + default = 1

### Identify: Select the correct iterative statement



### Review: Nested for loops

This reading contains the code used in the instructional videos from [**Nested for loops**](https://www.coursera.org/learn/python-crash-course/lecture/f7w6D/nested-for-loops)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

for left in range(7):

for right in range(left, 7):

print("[" + str(left) + "|" + str(right) + "]", end=" ")

print()

—-------------

teams = [ 'Dragons', 'Wolves', 'Pandas', 'Unicorns']

for home\_team in teams:

for away\_team in teams:

if home\_team != away\_team:

print(home\_team + " vs " + away\_team)

—-------------

for element in long\_list:

do\_something(element)

for element1 in long\_list:

for element2 in long\_list:

do\_something(element1, element2)

### Nested for Loops

We're going to explore what happens when you get loops inside of loops.

Have you ever played dominoes before? There's a bunch of fun games you can play with these tiles. In case you're not familiar, each domino tile has two numbers represented by a collection of dots carved on each half of the tile. The numbers go from zero to six. Tiles can be rotated so that each combination of numbers is represented only once in a set of domino tiles. In other words, the two-three tile is the same as the three-two tile and there's only one per set.

Now, imagine we wanted to write a program that prints each domino tile in a set. If we take all of the tiles that have zero on the left, we can print tiles with numbers from zero to six on the right. That should be easy to do with a for loop. What about tiles that have one on the left? Well, we need to skip the one-zero tile because that one was already printed as zero-one. So we could print a list of tiles with one on the left and numbers from one to six on the right. When we look at two, we would need to skip both zero and one, and so on. Are you following along? How do you think we'd write the code for this? Turning this into code means that we'd need to write two for loops, one inside the other. This is what we call nested for loops. Check out how this looks in Python code.

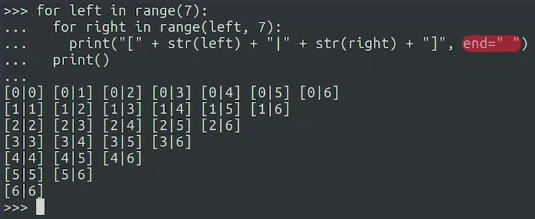
for left in range(7):

for right in range(left, 7):

print("[" + str(left) + "|" + str(right) + "]", end=" ")

print()

For left in range, seven. For right in range, left, seven. Print. We're gonna do left bracket, plus str left plus pipe plus str, right plus, close bracket, end equals. Space. And print.



In this code, we're using a new parameter that we pass to the print function. This parameter is called end. Normally, once print has taken the content we passed and written it to the screen, then it writes a special character that creates a new line called the newline character. If we want print to write something else instead of the new line character, we use the end parameter, like we see in this example.

Notice how the second for loop iterates over a different number of elements each time it's called as the value of left changes. Depending on what you want to achieve with your nested loops, you may want both loops to always go through the same number of elements, or you might want the second loop to connect to the first one.

Let's look at a different example. Let's say you run a local girls basketball league in your town. You have four teams that will play against each other in the league, both at home and away. You've stored the names of the teams in a list, like this. We want to write a script that will output all possible team pairings. For this, the order of the names matters because for each game, the first name will be the home team, and the second name is the away team. And of course, what we don't wanna do is have a team playing against itself. So what statement do we need to use to avoid that?

To do this, we need to use a conditional that makes sure we only print the pairing when the names are different. Check out what this looks like.

teams = [ 'Dragons', 'Wolves', 'Pandas', 'Unicorns']

for home\_team in teams:

for away\_team in teams:

if home\_team != away\_team:

print(home\_team + " vs " + away\_team)

For home\_team in teams: for away\_team in teams: if home\_team not equal to away\_team, print home\_team versus away\_team. Success.

As you can see, nested loops are super useful for solving certain problems, like pairing teams.

We've seen that nested loops are a handy tool, but we need to be careful not to just blindly apply them to any problem. Why? Well, because the longer the lists your code needs to iterate through, the longer it takes your computer to complete the task.

Let's say your manager asks you to do an operation that will run through a list of 10,000 elements. If the operation takes one millisecond per element, the whole loop would take one millisecond times 10,000 to complete, which is 10 seconds. Now, imagine we add a nested loop that has to go over the same 10,000 elements. This means that each iteration of the outside loop would do a full iteration of the inside loop, which again, would take 10 seconds to go through the whole list. So now the whole iteration takes 10,000 times 10 seconds, which is 100,000 seconds. That's over 27 hours. This doesn't mean we shouldn't use nested loops. They're a useful tool in solving problems that require them, but we need to be careful of where and how we use them. Throughout this course and ones coming up, we'll look at a lot of techniques that can help us pick the right tool to use for each type of problem. Up next, we'll look into some common errors that you might come across when writing your for loops and what to do about them.

### Strings and for loops

Over the past few modules we’ve been talking about using loops and their syntax in Python. As you have seen, a loop is an instruction that repeats multiple times as long as the condition set forth in the loop is met. You use loops to repeat a block of code. For example, if you want to send users an invitation, asking them to subscribe every time they open a new project, you would use a loop.

In this reading, you will learn the basics about *for* loops and how you can use them with strings.

***For* Loop Recap**

As a reminder, *for* loops enable you to iterate over a sequence of values, such as numbers, names, or lines in a file. You can even iterate over a list of strings.

**What are strings?**

At the beginning of this course we identified strings as one of the basic data types used by Python and other programming languages. Strings represent a sequence of characters and are often used to display output to the user.

You can recognize a string because it’s surrounded by single or double quotes like the following examples:

"Hello world."

‘Ostriches can’t fly.’

"567.89"

**Pro tip**: You can’t mix single and double quotes in the same string or you’ll get a syntax error.

**Note**: In the examples above, Python does not consider “567.89” a number. It is a string because it is surrounded by quotes.

**Using *for* loops with strings**

In an earlier video, you learned that a *for* loop iterates over the items in a collection until each item in the collection has been looped over.

**Pro tip:** Every *for* loop can also be written as a while loop.

Let’s see what that looks like. In the example below, we’re using a *for* loop to print out the characters in the string “Hello.”

*>>> greeting = "Hello"*

*>>> for c in greeting:*

*>>> print("The next character is: ", c)*

*The next character is: H*

*The next character is: e*

*The next character is: l*

*The next character is: l*

*The next character is: o*

Notice that the *for* loop will stop iterating after the “o” is printed. You can also use *for* loops with Python’s *range ()* function to generate a series of numbers, which you can see in the following example:

*>>> for i in range(0, 3):*

*>>> print("The next value is:", i)*

*The next value is 0*

*The next value is 1*

*The next value is 2*

Practically every Python script you write will use strings, which we’ll talk about in more detail in Module 4. For now, it’s enough for you to know that they are immutable. In other words, once you create a string, it can’t be changed. But you can use *for* loops to create a new string.

You can use *for* loops with strings to perform tasks and functions such as:

* Reading text from a file
* Searching for a value or specific data in a document or spreadsheet

**Pro tip:** If you modify or update a collection of items you are using with a *for* loop, make sure you modify your *for* loop as well.

**Key takeaways**

The real power of *for* loops is that they allow you to repeat sequences of any kind of data, not just a range of numbers or letters. They can help you find and use data faster. Over the next several videos and readings, you’ll get a chance to see what else you can do with loops and strings.

### Looping over a String

Looping over a string allows programmers to examine each character within a string individually, gather information about its occurrences, and perform a given operation. There are multiple ways to loop over a string—some ways are more beneficial than others—and which one you choose depends on the information you need.

In this reading, you will learn about looping over a string, how to do it, and see some examples along the way.

To loop over a string means to start with the first character in a string and iterate over each character until the end of the string. Strings are objects that contain a sequence of single-character strings. Yes, a single letter is classified as a string in Python. For example, *string[0]* is considered a string even though it is just a single character.

**Note**: Python does not use characters as a type like other programming languages do; it just supports strings with a length of 1.

***for* Loop**

The most direct—and common—way to loop over a string is to use the *for* loop. Let’s look at an example:

greeting = 'Hello'

for char in greeting:

print(char)

Notice that *greeting* is defined as the string *'Hello'*. The loop begins with *char = 0*, which is the first element in the string. It directly calls the elements of the string and prints each element on a new line, resulting in the output below:

*H*

*e*

*l*

*l*

*o*

What if you don’t want the elements of the string but the position of the string instead? You’re in luck because Python can work that magic too!

for i in range(len(greeting)):

print(i)

*len(greeting)* is an integer that tells Python how many characters are in the string. But because it’s an integer, you need to convert it to an iterable sequence by using the *range()* function. This loop does the same thing as the loop above, but instead of printing elements, it prints integers resulting in the output below:

*0*

*1*

*2*

*3*

*4*

***while* loop with indexing**

This *while* loop is the more “common” *while* loop that programmers often use. This type of loop involves an index variable to represent the current position within the sequence. Most of the time, this will start with 0 for the initial iteration. Let’s take a look at an example:

greeting = 'Hello'

index = 0

while index < len(greeting):

print(greeting[index])

index += 1

The initial index value is 0, and the *while* loop continues to execute as long as the index variable is less than the length of the *len(greeting)*. At each iteration, Python prints the value at the current index position *(greeting[index])*. Then, Python increments the index by 1 *(index += 1)* to move to the next position. The output of this example is:

*H*

*e*

*l*

*l*

*o*

**Pro Tip:** In any *while* loop, you can add conditional statements and stop the iteration process early so that the loop does not examine every character.

***while* loop with slicing**

Using a *while* loop with slicing accomplishes the same thing that a *while* loop with indexing does—like the example you explored above—this is just another way to write a *while* loop. You use this *while* loop in combination with string slicing to iterate over a portion of a sequence. Remember, it’s up to you to choose the method for looping over a string based on your level of comfort. There are multiple ways to write lines of code to execute the same result. Let’s explore how a *while* loop with slicing results in the same output.

greeting = 'Hello'

index = 0

while index < len(greeting):

print(greeting[index:index+1])

index += 1

This *while* loop continues to run as long as the index variable is less than the length of the string, which is determined by using *len(greeting)*. With each iteration, a substring of length 1 is extracted using *(greeting[index:index+1])* and printed. Then, the index is incremented by 1 *(index += 1)* to move to the next position. The output is:

*H*

*e*

*l*

*l*

*o*

**List comprehensions**

List comprehensions are a concise way to create lists in Python. Let’s look at an example:

numbers = [1, 2, 3, 4, 5]

squared\_numbers = [x \*\* 2 for x in numbers]

print(squared\_numbers)

This example starts with a list of numbers. The second line of code defines the type of transformation you want to execute on each element in the original list. In this case, you’re using this line of code to create a new list called *squared\_numbers* and apply *x \*\* 2* to square each element in the *numbers* list. The result of each squared element is then printed in a new list:

*[1, 4, 9, 16, 25]*

**Note:** List comprehensions can include conditional statements and nested loops, which allows for additional filtering of elements based on specific called conditions.

**Additional advanced string loop techniques**

There are additional ways to loop over a string in Python that you should learn, practice, and master. These additional looping techniques include the generator functions, *map()*, and *zip()*. The *map()* and *zip()* functions are extremely powerful string manipulation tools that demonstrate functional programming concepts. To learn more about these advanced techniques, see these resources:

* [Python - map() function](https://www.tutorialsteacher.com/python/python-map-function)
* [Python zip() method](https://www.tutorialsteacher.com/python/zip-method)

**Pro Tip:**  Looping over multiple strings at once can push the limits of *for* loops. Because of this, it’s important to be aware of other alternatives to simplify a *for* or *while* loop.

**Key takeaways**

When you loop over a string, you are able to examine each individual element in the string and manipulate it how you’d like. A *for* loop iterates over each element, and a *while* loop allows you to add conditional statements and stop the iteration process early. As you saw in this reading, there are a variety of methods to loop over a string. It’s up to you to familiarize yourself with each technique and write code according to the advantages and disadvantages of each.

### Slice and Join Strings

You’ve been learning all about strings. Now it’s time to learn how to extract parts of a string—what we call “slicing” a string—or create a longer string by joining two or more strings together.

Slicing a string is just like taking a slice out of a homemade apple pie. When it comes to strings, slices can be as small or as big as you want. If you want to put two or more of those slices back together, you join them together end-to-end—in Python speak, you “concatenate” them—to make one bigger, single string.

In this reading, you will learn when to slice and join strings, how to do it, and see some examples along the way.

**What is slicing and joining strings?**

When you **slice** a string, you extract a subset of the original string—sometimes referred to as indexing a string. **Joining** strings is the process of linking two or more strings together to create a bigger string.

**How to slice strings**

Bracket notation, *[ ]*, is used to specify the start of the index, ending index, or both. If you do not include the starting index, then the slice contains everything from the beginning of the string to the ending index. This is the same if you do not include the ending index. Let’s look at a couple of examples:

**Pro tip:** Remember that the indexes in Python start with 0, and not 1.

string1 = "Greetings, Earthlings"

print(string1[0]) # Prints “G”

print(string1[4:8]) # Prints “ting”

print(string1[11:]) # Prints “Earthlings”

print(string1[:5]) # Prints “Greet”

**How to join strings**

To join strings in Python, you use the plus operator, + , just as if you were adding two numbers together. The following example joins three strings together.

print("Hello" + " " + "world") #Prints “Hello world”

You can also use the *join()* function, which is very useful when you want to concatenate elements from a list of strings with a specific delimiter. In the following example, we have a list of strings called greetings and we join them with a space using *.join(greetings)*. The *join()* function concatenates all the strings in the list greetings, and places a space between each string.

greetings = ["Hello", "world"]

print(" ".join(greetings)) # Prints "Hello world"

You can also concatenate a combination of strings and variables like in the following example.

name = "Alice"

print("Hello, " + name + "!") # Prints "Hello, Alice!"

**How to combine slicing and joining strings**

Now you know how to slice strings and join strings. Now, let’s put the two operations together by taking an unformatted phone number, 2025551212, and return it as a properly formatted U.S. number. In this example, we’ll use *phonenum* to refer to the unformatted phone number.

# The first 3 digits are the area code:

area\_code = "(" + phonenum[:3] + ")"

This function slices the first three numbers from the list.

# The next 3 digits are called the “exchange”:

exchange = phonenum[3:6]

This function slices the numbers 4–6 from the list.

# The next 3 digits are the line number:

line = phonenum[-4:]

This negative index function counts backwards from the end of the numbers, slicing the last four numbers in the list.

# Put the pieces back together into a nicely formatted string:

return area\_code + " " + exchange + "-" + line

When you’re done, your code will look like this:

def format\_phone(phonenum):

area\_code = "(" + phonenum[:3] + ")"

exchange = phonenum[3:6]

line = phonenum[-4:]

return area\_code + " " + exchange + "-" + line

Finally, we’ll use the print function to join the three previously sliced numbers together in the correct format. With this function definition, when you call *print(format\_phone("2025551212"))*, it will print (202) 555-1212.

print(format\_phone("2025551212")) # Outputs: (202) 555-1212

**Key takeaways**

Slicing and joining strings can be beneficial to correctly format numbers, making it easier to manipulate data in a more efficient and meaningful way. Slicing strings allows you to access individual characters of a string by specifying the index. The index can be as long or as short as you like—you can even have a negative index, if counting backwards is your thing! Joining strings allows you to add two or more strings together, which is beneficial when needing to create a sentence or even properly format different numbers.

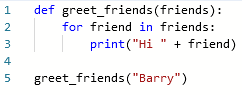
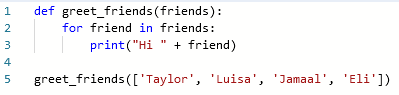
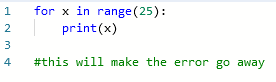
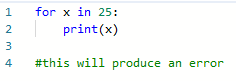
### Review: Common errors in for loops

This reading contains the code used in the instructional videos from [**Common errors in for loops**](https://www.coursera.org/learn/python-crash-course/lecture/8XtCn/common-errors-in-for-loops)

**Introduction**

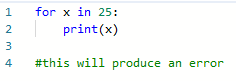
This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.



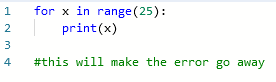
### Common Errors in for Loops

We've now seen how to write for loops, combine them with functions, nest a for loop inside a different loop, and even combine a nested loop with conditionals. Nice job, you're chugging right along. But before we're done with for loops, let's check out some common mistakes you may come across while trying this yourself. As we've called out already, for loops iterate over sequences. The interpreter will refuse to iterate over a single element. As you see here:

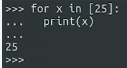


In this example, we're trying to iterate over the number 25. Python prints a TypeError telling us that integers are not iterable. There are two solutions to this problem, depending on what we're trying to do.

If we want to go from zero to 25, then we use the range function:



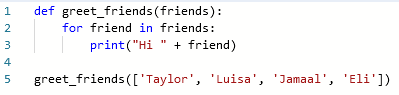
but if we're trying to iterate over a list that has 25 as the only element, then it needs to be a list and that means writing it between square brackets:

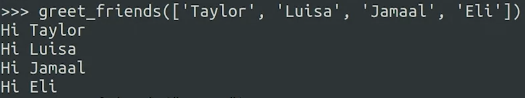


You might be wondering why you'd ever want to iterate over a list of one element and that's a good question. Well, this kind of issue usually happens when you have a function with a for loop inside it, which is iterating over the elements of a list received by parameter.

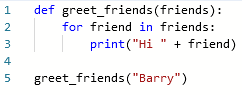
Say for example, you have a function that fixes the permissions of a list of files received by parameter, and you want to call this function to fix the permissions of just one specific file. To do that, you need to pass the file as the single element of a list.

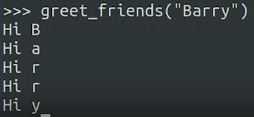
Let's check this out with some code we're familiar with, our friendliest of Python examples, hi, friends. We're going to modify it to have the greetings inside a function.





We've defined a greet friends function, that receives a list by parameter and iterates over that list, greeting each friend. But what if we only want to greet one friend instead of four? Well, we still need to define a list, but with only one element. But first, let's see what would happen if we don't do that: greet\_friends("Barry").





Not what we expected, right? Well, what's going on here? This happens because strings are iterable, the for loop will go over each letter of the string and do the operation we asked it to do, which in this case, print a greeting. Depending on what you're trying to do, you may actually want to iterate through the letters of a string. But in this case, we don't. So to sum it up, if you get an error that a certain type isn't iterable, you need to make sure the for loop is using a sequence of elements and not just one, and if you find your code iterating through each letter of a string when you want it to do it for the whole string, you probably want to have that string be a part of a list. We've now learned how to write while loops and for loops. You might remember, for loops are best when you want to iterate over a known sequence of elements but when you want to operate while a certain condition is true, while loops are the best choice. Next up, we've got a super useful cheat sheet for you that puts all this into one handy resource. After that, head over to the practice quiz to test your knowledge and check in on how you're doing.

### Study Guide: for Loops

This study guide provides a summary of what you learned in this segment and serves as a guide for the upcoming practice quiz.

In the *for* Loops segment, you learned about the logical structure and syntax of *for* loops. You took a closer look at the *range()* function. You learned about nested *for* loops and complex nested *for* loops with *if* statements. You also learned how to fix common errors in *for* loops.

***for* Loops vs. *while* Loops**

*for* loops and *while* loops share several characteristics. Both loops can be used with a variety of data types, both can be nested, and both can be used with the keywords break and continue. However, there are important differences between the two types of loops:

* *while* loops are used when a segment of code needs to execute repeatedly while a condition is true
* *for* loops iterate over a sequence of elements, executing the body of the loop for each element in the sequence

An important distinction is that *for* loops are suited for objects that have iterable structures. So lists, strings, ranges of integers. Individual integers are not iterable, but can be looped over by the use of the range() function, which is covered below. While loops do not iterate per se, rather they watch a truth condition

**Syntax**

The syntax of a *for* loop with the in keyword:

for variable in sequence:

body of loop

**Common *for* Loop Structures**

*for* Loop with *range()*

The in keyword with the *range()* function generates a sequence of integer numbers, which can be used with a *for* loop to configure the iterations of the code. The range of numbers [0, 1, 2] correlates to ordinal index positions (1st, 2nd, 3rd), rather than the cardinal (quantity) values of the numbers 0, 1, and 2. For example, range(5) means the five index positions in the range [0, 1, 2, 3, 4].

The *range()* function can take up to three parameters. The roles of the three possible *range(x,y,z)* parameters are:

* *x = Start* - Starting index position of the range
  + Default index position is 0.
  + The starting index position is included in the range.
  + Example syntax: *range(2, y, z)* or *range(x+3, y, z)*
* *y = Stop* - Ending index position of range
  + No default index position. Must include the ending index position in the *range()* parameters.
    - Example syntax: *range(y)*
  + The value of the ending index position is excluded from the range.
  + To include the ending index number, use the expression: *y+1 (index + 1)*
    - Example syntax: *range(x, y+1, z)*
    - Alternatively, if *y* = 10, you can write: *range(x, 11, z)*
* *z = Step* - Incremental value

Example of a *for* loop with the in keyword and the *range()* function:

# This loop iterates on the value of the "number" variable in a range

# of 1 to 6+1 (the upper range limit of 6 is excluded, so +1 has

# been added to it to include 6 in the range). The incremental value

# for the loop is 2 (number+2). The print() function will output the

# resulting value of "number" multiplied by 3.

for number in range(1, 6+1, 2):

print(number \* 3)

# The loop should print 3, 9, 15

**Common pitfalls when using the *range()* function:**

* Forgetting that **the upper limit of a** *range()* **isn’t included** in the range.
* **Iterating over non-sequences**. For example, strings are iterable letter by letter, but not word by word.

Example of a *range()* function where the value of the upper limit of the range is excluded:

**Nested *for* Loops**

The syntax of nested *for* loops:

for x in sequence:

# start of the outer loop body

for y in sequence:

# start of the inner loop body

# end of of the inner loop body

# continue body of the outer loop

# end of the outer loop body

Example of nested *for* loops:

# This code demonstrates the outer and inner loop iterations of a pair

# of nested for loops. Click "Run" to see the results. The outer loop

# will run twice for the range pointer positions [0, 1] in range(2).

# The inner loop will run 4 times for the range pointer positions

# [0, 1, 2, 3] in range(3+1) or range(4) each time the outer loop runs.

# So, the inner loop will execute 8 times in total.

for x in range(2):

print("This is the outer loop iteration number " + str(x))

for y in range(3+1):

print("Inner loop iteration number " + str(y))

print("Exit inner loop")

***for* loop with nested if Statement**

The syntax of a *for* Loop with nested if Statement:

for x in sequence:

# start of body of for loop

if condition is true:

# start of body of if-statement

# end of body of if-statement

# continue body of for loop

# end of body of for loop

# As a list comprehension:

[x for x in sequence if condition]

# This for loop iterates through the numbers 0 to 6. The if statement

# uses the modulo operator to test if the "x" variable is divisible by

# 2. If True, the if statement will print the value of "x" and exit

# back into the for loop for the next iteration of "x". Since no

# incremental value is specified in the range() parameters, the default

# increment is +1.

for x in range(7):

if x % 2 == 0:

print(x)

# The loop should print 0, 2, 4, 6

# As a list comprehension:

even\_numbers = [x for x in range(7) if x % 2 == 0]

print(even\_numbers)

**List comprehensions**

It is important to know that loops can be avoided sometimes; as you progress, you will develop a sense of when and how to do so. The concepts *for* loops are similar between other languages, but in Python, **list comprehensions** provide a concise way to create lists based on existing lists or sequences.

Here is a concrete example for better understanding. Let's say you have a sequence of numbers and you want to create a new list containing only the even numbers from the sequence.

With a traditional *for* Loop, you might write:

sequence = range(10)

new\_list = []

for x in sequence:

if x % 2 == 0:

new\_list.append(x)

With a list comprehension, you could achieve the same result in a more concise way:

sequence = range(10)

new\_list = [x for x in sequence if x % 2 == 0]

Both of these pieces of code will create a list of even numbers from 0 to 10: [0, 2, 4, 6, 8]. The list comprehension version does this in a single, compact line. These “one-liners” are very useful and dramatically reduce overhead.

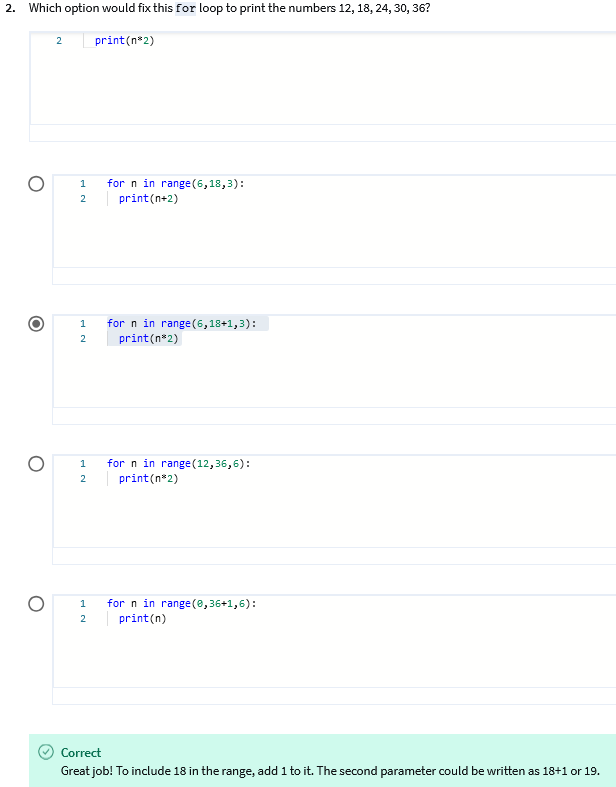
An example of a useful one-liner is:

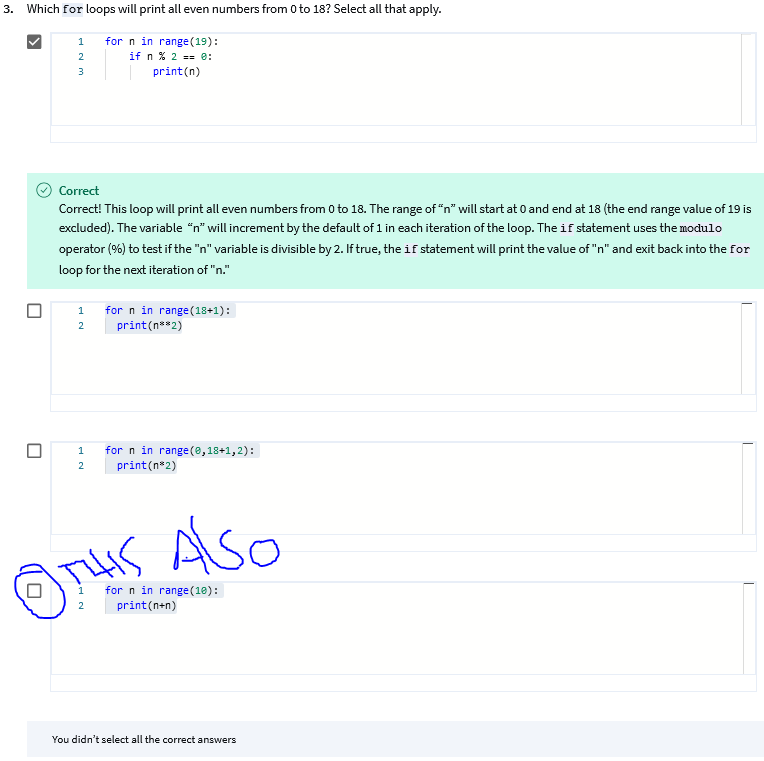
print("\*" \* 8)

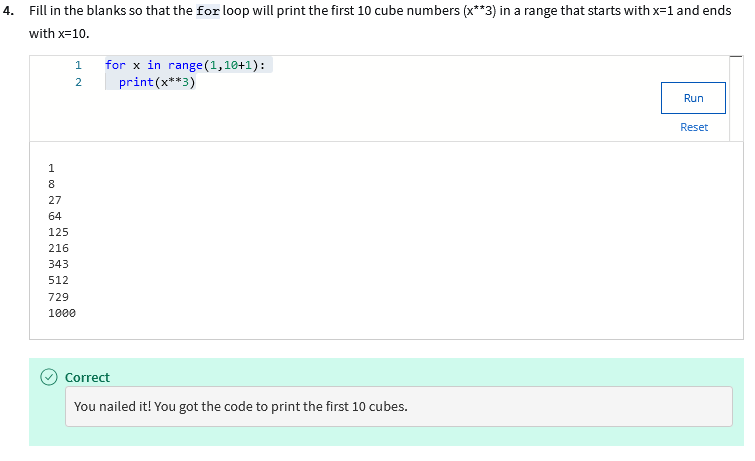
This prints “\*” 8 times. The number can be replaced by an integer variable, and to do this with a loop would be several lines of code and run more slowly.

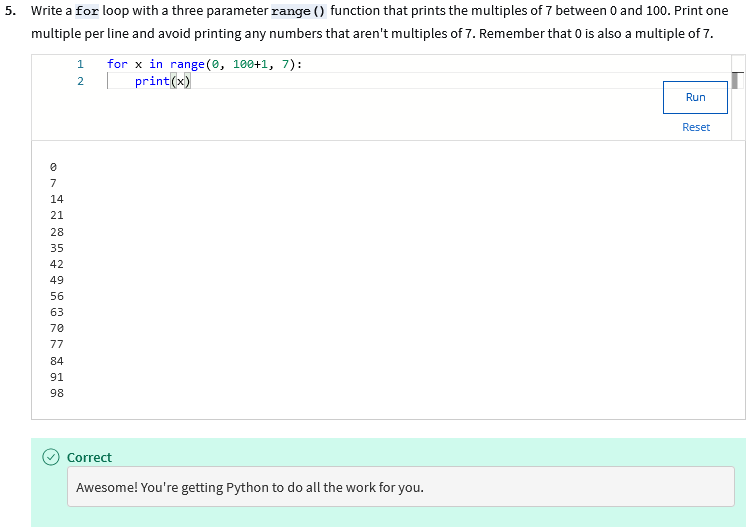
### Practice Quiz: For Loops

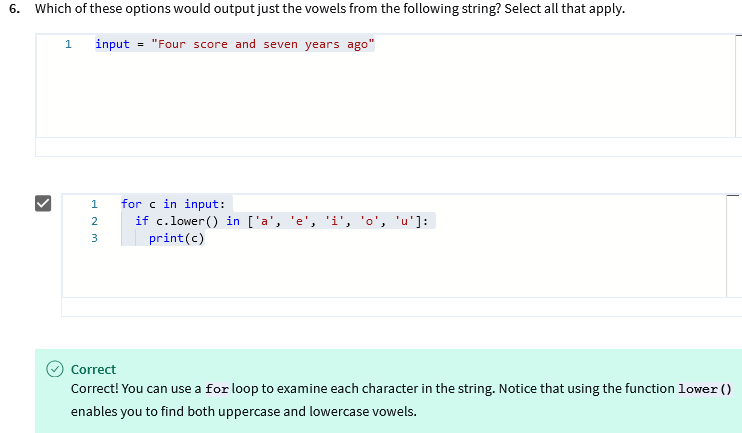
## 

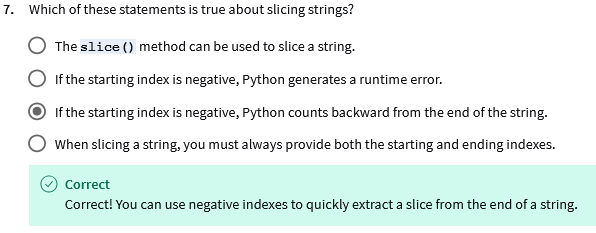












## Recursion (Optional)

### 

### Review: What is recursion?

This reading contains the code used in the instructional videos from [**What is recursion?**](https://www.coursera.org/learn/python-crash-course/lecture/7T0ET/what-is-recursion-optional)

**Introduction**

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

def factorial(n):

if n < 2:

return 1

return n \* factorial(n-1)

—---

def factorial(n):

print("Factorial called with " + str(n))

if n < 2:

print("Returning 1")

return 1

result = n \* factorial(n-1)

print("Returning " + str(result) + " for factorial of " + str(n))

return result

factorial(4)

### What is recursion? (Optional)

We've now discovered two looping techniques that we could use in Python: while loops and for loops. We use while loops when we want to do an operation repeatedly while a certain condition is true. We use for loops when we want to iterate over the elements of a sequence. Now, we're going to check out a third technique called recursion. But before we dive in, you may have noticed that this video is marked as optional. That's because while recursion is a very common technique used in software engineering, it's not used that much in automation. Still, we think it's valuable for you to know about recursion and to have an idea of how to use it. You may see it in code written by others or you may face a problem where recursion is the best way to solve it.

**Recursion is the repeated application of the same procedure to a smaller problem.**

Have you ever played with a Russian nesting doll? They are a great visual example of recursion. Each doll has a smaller doll inside it. When you open up the doll to find the smaller one inside, you keep going until you reach the smallest doll which can't be opened.

**Recursion lets us tackle complex problems by reducing the problem to a simpler one.**

Take our Russian nesting dolls, all nested inside each other. Imagine we want to find out how many dolls there are in total. We would need to open each doll one by one until we got to the last one and then count how many dolls we've opened. That's recursion in action.

Here's another example with a more complex problem. Imagine you're in a line of people and you want to know how many people are in front of you, and let me tell you I can't stand long lines. Anyway, if the line is long, it might be hard to count the people without leaving the line and losing your place. Instead you can ask the person in front of you how many people are in front of them. Since this person will be in the same situation as you, they'll have to ask the same question to the person in front of them and so on and so on until the question reaches the first person in the line. This person can confidently reply that there are no people in front of them. So then the second person in line can reply one, the person behind them replies two, and so on until the answer reaches you.

How does this translate into programming?

**In programming, recursion is a way of doing a repetitive task by having a function call itself. A recursive function calls itself.**

Usually with a modified parameter until it reaches a specific condition. This condition is called the base case. In our earlier examples, the base case would be the smallest Russian doll or the person at the front of the queue.

Let's check out an example of a recursive function to understand what we're talking about.

def factorial(n):

if n < 2:

return 1

return n \* factorial(n-1)

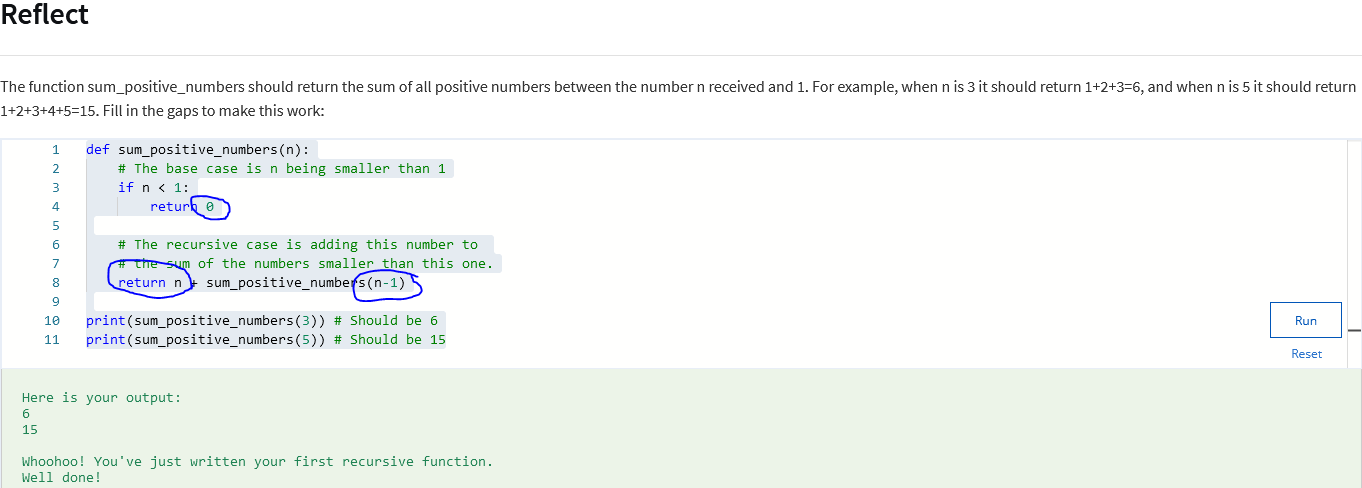
Here, we're defining a function called factorial.

At the beginning of the function, we have a conditional block defining the base case, where n is smaller than 2.

It simply returns the value 1.

After the base case, we have a line where the factorial function is calling itself with n minus 1. This is called the recursive case.

This creates a loop. Each time the function is executed, it calls itself with a smaller number until it reaches the base case. Once it reaches the base case, it returns the value 1. Then the previously called function multiplies that by two and the previously called function multiplies it by three and so on. This loop will keep going until the first factorial function called returns the desired result. It's a bit complex. Let's add a few print statements to see exactly how this works. So here we can see the function kept calling itself until it reached the base case. After that, each function returned the value of the previous function multiplied by n until the original function returned.



### Review: Recursion in Action in the IT context

This reading contains the code used in the instructional videos from [**Recursion in the IT context**](https://www.coursera.org/learn/python-crash-course/lecture/pQhjw/recursion-in-action-in-the-it-context)

Introduction

This follow-along reading is organized to match the content in the video that follows. It contains the same code shown in the next video. These code blocks will provide you with the opportunity to see how the code is written, allow you to practice running it, and can be used as a reference to refer back to.

You can follow along in the reading as the instructor discusses the code or review the code after watching the video.

def factorial(n):

if n < 2:

return 1

return n \* factorial(n-1)

factorial(1000)

#this will produce an error

### Recursion in Action in the IT Context

By now you've seen what a recursive function looks like, how to write a base case and the recursive case. You might be wondering why do we need recursive functions if I can just use a for or while loop? Well, solutions to some specific problems are easier to write and understand when using recursive functions. A lot of math functions like the factorial or the sum of all the previous numbers are good examples of this. If a math function is already defined in recursive terms, it's straightforward to just write the code as a recursive function. But it's not just about math functions. Let's check out a couple of examples of how this could help an IT specialist trying to automate tasks.

Let's say that you need to write a tool that goes through a bunch of directories in your computer and calculates how many files are contained in each. When listing the files inside a directory, you might find subdirectories inside them and you'd want to count the files in those subdirectories as well. This is a great time to use recursion. The base case would be a directory with no subdirectories. For this case, the function would just return the amount of files.

The recursive case would be calling the recursive function for each of the contained subdirectories. The return value of a given function call would be the sum of all the files in that directory plus all the files in the contained subdirectories. A directory of files that can contain other directories is an example of a recursive structure. Because directories can contain subdirectories that contain subdirectories that contain subdirectories, and so on.

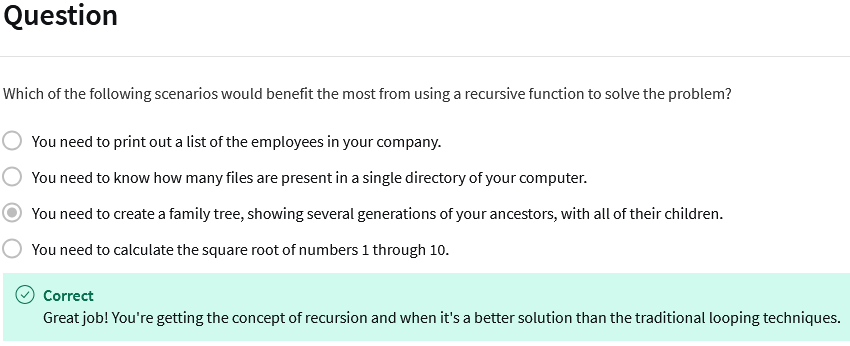
When operating over recursive structure, it's usually easier to use recursive functions than for or while loops.

Another IT-focused example of a recursive structure is anything that deals with groups of users that can contain other groups. We see this situation a lot when using administrative tools like active directory or LDAP. Say your group management software allows you to create groups that have both users and other groups as their members.

And you want to list all human users that are part of a given group. Here you would use a recursive function to go through the groups.

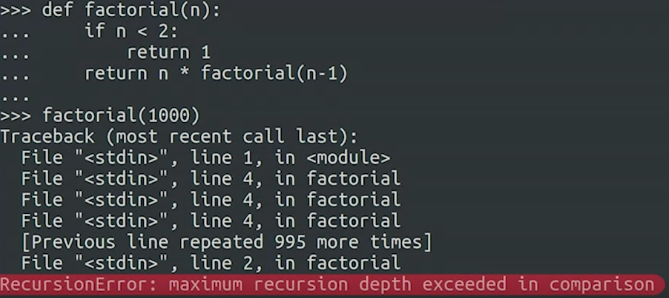
The base case would be a group that only includes users listing all of them. The recursive case would mean going through all the groups containing all the users in them and then listing any users contained in the current group.

It's important to call out that in some languages there's a maximum amount of recursive calls you can use.



In Python by default, you can call a recursive function 1,000 times until you reach the limit. That's fine for things like subdirectories or user groups that aren't thousands of levels deep. But it might not be enough for mathematical functions like the ones we saw in the last video.

Let's go back to our factorial example from the last video and try to call it with n equals 1,000.



See that error? It's telling us that we've reached the maximum limit for recursive calls. So while you can use recursion in a bunch of different scenarios, we only recommend using it when you need to go through a recursive structure that won't reach a thousand nested levels. All right, we've just added recursion to your growing scripting tool box. They're ready for you whenever the situation calls for it.

### Additional Recursion Sources

In the past videos, we visited the basic concepts of recursive functions.

A recursive function must include a recursive case and base case. The recursive case calls the function again, with a different value. The base case returns a value without calling the same function.

A recursive function will usually have this structure:

def recursive\_function(parameters):

if base\_case\_condition(parameters):

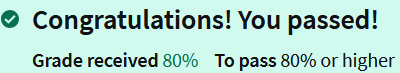
return base\_case\_value

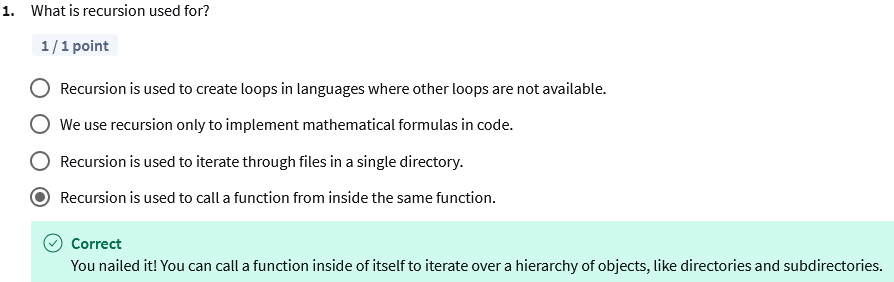
recursive\_function(modified\_parameters)

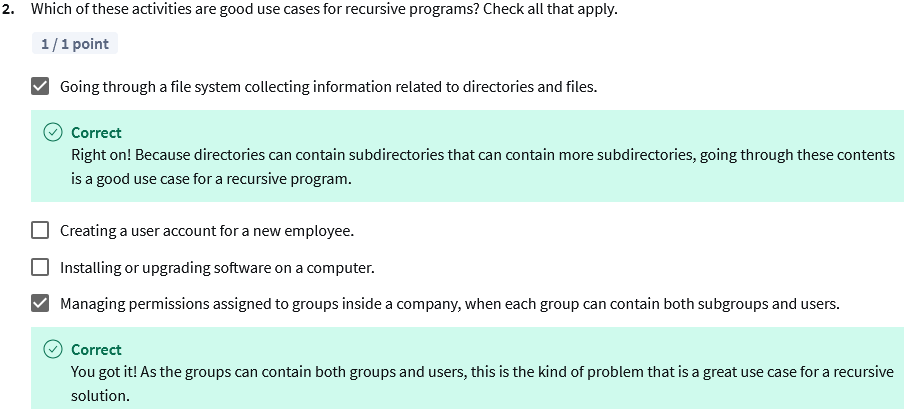
For more information on recursion, check out these resources:

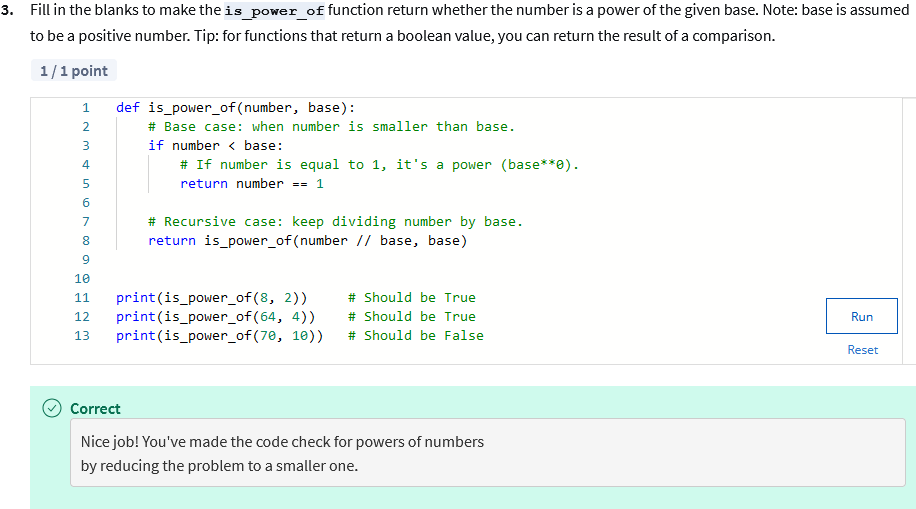
* [Wikipedia Recursion page](https://en.wikipedia.org/wiki/Recursion)
* See what happens when you [Search Google for Recursion](https://www.google.com/search?q=recursion)

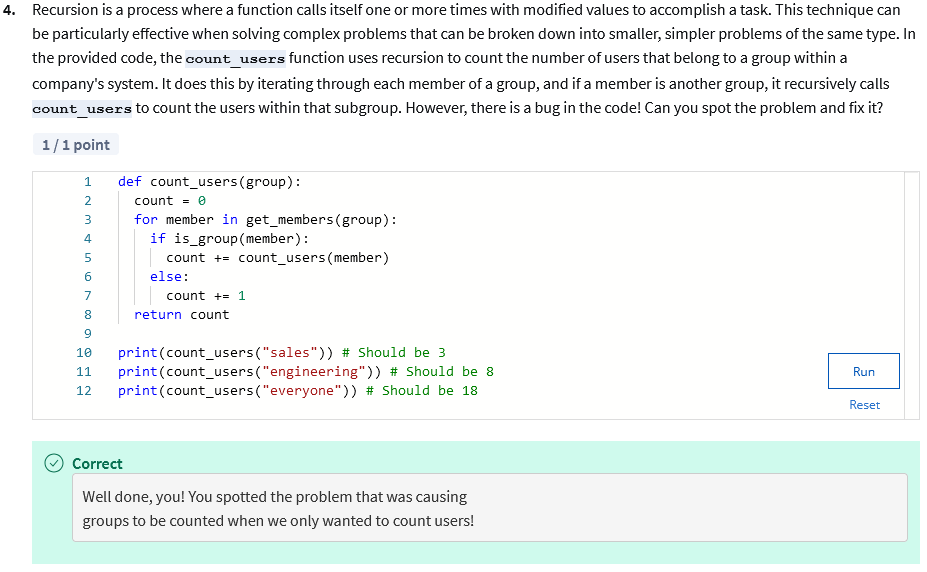
### Practice Quiz: Recursion

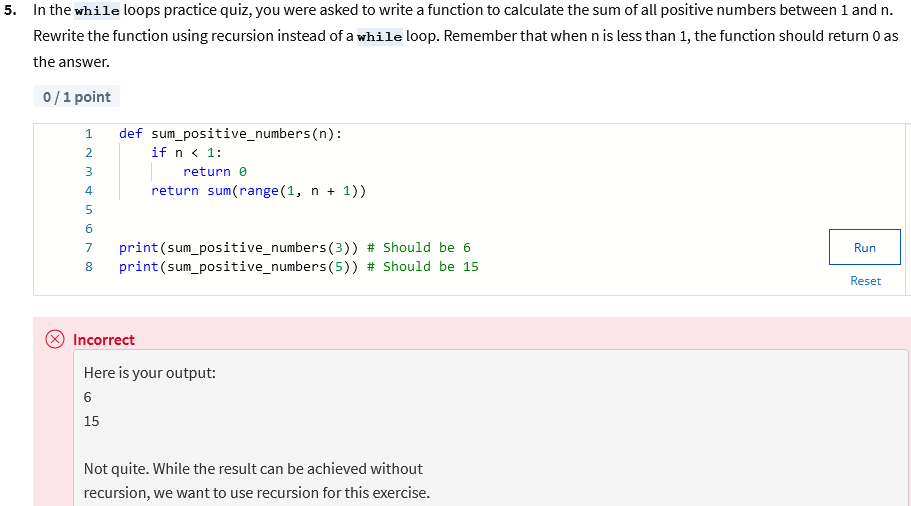








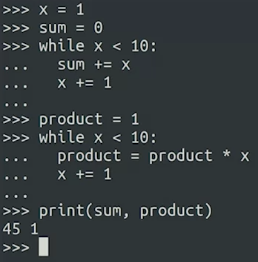




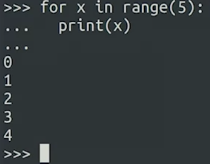
## Module Review

### Loops Wrap Up

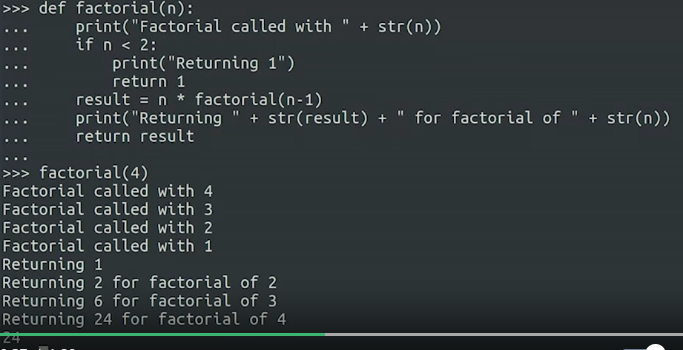
Wow, we've come a long way and you've learned a lot already. Now's a good time to stop and give yourself a big pat on the back. In this module, we've looked at ways we can use to tell a computer to do an action repetitively. Python gives us three different ways to perform repetitive tasks while loops, for loops, and recursion. We use while loops when we want to do an operation while a certain condition is true or alternatively until it becomes false.



We use for loops when we want to iterate over the elements of the sequence or a range of numbers.



And we use recursion when the problem is best solved in smaller steps and then combining those steps towards a larger solution.



If you're still not sure which is the best tool to choose for a specific problem don't worry, that's normal. As you keep practicing your automation skills, choosing between one option and another will become natural.

So next time you find yourself doing the same or similar things over and over again, that's your call to see if you can use a loop to get your computer to do the work for you. Up next it's test time again, with the next graded assessment. Like always remember you can take as much time as you need before taking the assessment. Go at your own pace, review everything we've covered, and practice the examples. So there's no chance loops will ever throw you for a loop.

### Glossary terms from course 1, module 3

**Terms and definitions from Course 1, Module 3**

**Break:** A way to exit out of a loop before the loop's condition is false

**Control statements:** Programming constructs that direct the flow of execution of a program by allowing you to make decisions, repeat actions, or choose between different code paths based on specific conditions.

**For loop:** This executes a block of code for a specified number of iterations or over a collection of items.

**Infinite loop:** A sequence that is missing a method for exiting the loop, causing the loop to run forever

**Iterators**: Variables that allow you to loop through a collection one item at a time

**Loop:** A sequence that makes the computer do repetitive tasks

**Programming:** The process of writing a program to behave in different ways

**Pass:** A placeholder statement which is used when the syntax requires a statement, but you don't want to execute any code or command

**Recursion:** The repeated application of the same procedure to a smaller problem

**While loop:** This is used when a segment of code needs to execute repeatedly while a condition is true

### Study Guide: Module 3 Graded Quiz

<https://github.com/GaJoDev/Python/blob/main/Coursera%20-%20Google%20IT%20Automation%20with%20Python%20Professional%20Certificate/1.%20Crash%20Course%20on%20Python/Module%203/study_guide_module3_graded_quiz.md>

It is time to prepare for the Module 3 Graded Quiz. Please review the following items from this module before beginning the Module 3 Graded Quiz. If you would like to refresh your memory on these materials, please also revisit the [**while** Loop Study Guide](https://www.coursera.org/learn/python-crash-course/supplement/iyNWi/study-guide-while-loops) and the [**for** Loop Study Guide](https://www.coursera.org/learn/python-crash-course/supplement/dqH6E/study-guide-for-loops)

located before the Practice Quizzes in Module 3. You will not be tested on the Recursion lesson content, which is optional in this module.

**Knowledge**

**Terms**

* **variables** - Know how to properly initialize or increment a variable. You will also need to recognize a coding error due to the failure to properly initialize or increment a variable.
* **infinite loops** - Know how to recognize infinite loops and use common solutions to prevent them. For example, check loop conditions, ranges, iterators, control statements, etc. to ensure that at least one of these controls are in place to prevent an infinite loop.
* **iterators** - Know the various options available for iterating a variable (e.g., using assignment operators, using the third **range()** function parameter). You will also need to analyze where the iteration should occur. A misplaced iterator could produce the wrong output or create an infinite loop.
* **control statements** - Know how and when to use the **break** and **continue** control statements to prevent infinite loops.

**Common Functions**

* **range() Function Parameters** - Know the roles of the three possible **range(x, y, z)** function parameters:
  + **x** Start of Range (included)
  + **y** End of Range (excluded index)
    - To include the end of range index, use the expression **y+1**
    - The end of range must be included in the **range()** parameters.
  + **z** Incremental value
  + **Example 1:** range(4, 12**+1**, **2**)
    - This example creates a range that starts at 4 and ends at 12 (without the **+1**, the range would end at 11).
    - The third parameter increments the range iteration by 2, as opposed to the default increment of 1. The **range(4, 12+1, 2)** expression would produce the values: 4, 6, 8, 10, 12
  + **Example 2:** range(10, 2**-1**, **-2**)
    - This example creates a range that starts at 10 and ends at 2**-1**, with a decremental value of **-2**. When counting down, to include the value of the end of the range index, use **-1** (end of range minus 1). This range produces the sequence: 10, 8, 6, 4, 2
* **print() Function Default Behavior** - Know the default behavior of the **print()** function is to insert a new line character after the print statement runs.
  + To override the insertion of the new line character and replace it with a space, add **end=" "** as the last item in the **print()** parameters. This makes it possible to add the next print output to the same line, separated by a space. You might use this technique when a print() function is part of a **for** or **while** loop. Example syntax: **print(x+1, end=" ")**

**Coding Skills**

**Skill 1: Using for loops with the range() function**

* Use a **for** loop with the **range()** function with the end-of-range value included in the range.

# This function will accept an integer variable "end" and count by 10

# from 0 to the "end" value.

def count\_by\_10(end):

# Initializeq the "count" variable as a string.

count = ""

# The range function parameters instruct Python to start the count

# at 0 and stop at the variable given as the upper end of the range.

# Since the value of the high end of a range is excluded by default,

# you can make Python include the "end" value by adding +1 to it.

# The third parameter tells Python to increment the count by 10.

for number in range(0,end+1,10):

# Although the variable "count" will hold a count of integers,

# this example will be converted to a string using "str(number)"

# in order to display the incremental count from 0 to the "end"

# value on the same line with a space " " separating each

# number.

count += str(number) + " "

# The .strip() method will trim the final space " " from the end of

# the string "count"

return count.strip()

# Call the function with 1 integer parameter.

print(count\_by\_10(100))

# Should print 0 10 20 30 40 50 60 70 80 90 100

* Use a set of nested **for** loops with the **range()** function to create a matrix of numbers.
* Include the upper range value in the **range()** function using end+1.

# This function uses a set of nested for loops with the range() function

# to create a matrix of numbers. The upper range value in the range()

# function should be included in the matrix. The matrix should consist

# of a set of numbers that fill both rows and columns.

def matrix(initial\_number, end\_of\_first\_row):

# It is an optional code style to assign the long variable names in the

# function parameters to shorter variable names.

n1 = initial\_number

n2 = end\_of\_first\_row+1 # include the upper range value with +1

# The first for loop will create the columns.

for column in range(n1, n2):

# The nested for loop will create the rows.

for row in range(n1, n2):

# To make the matrix of numbers easier to read, include a space

# between each number in the rows until the loop reaches the

# end of the row. You can override the default behavior of the

# print() function (which inserts a new line character after

# the print command runs) by using the "end=" "" parameter

# inside the print() function.

print(column\*row, end=" ")

# The row ends when the upper range value is encountered within the

# nested for loop. The outer (column) for loop should insert a new line

# to create the next row. Use the print() function new line default

# behavior with an empty print() function:

print()

# Call the function with 2 integer parameters.

matrix(1, 4)

# Should print:

# 1 2 3 4

# 2 4 6 8

# 3 6 9 12

# 4 8 12 16

* Predict the final value of a nested **for** loop with **range()** functions.

# For this example, the outer for loop uses an end of range index of

# 10. The value of index 10 will be 10-1, or 9.

for outer\_loop in range(10):

# Using the "outer\_loop" variable as the end of range for the

# inner loop, means the end of range index will be 9. The value

# of index 9 will be 9-1, or 8.

for inner\_loop in range(outer\_loop):

# The printed result is the value of "inner\_loop". Since

# there aren’t any calculations in this loop, there is a

# simple shortcut for solving what the final value printed

# by the "inner\_loop" will be. The solution is to simply use

# the value of the "inner\_loop" index, which is 8.

print(inner\_loop)

### 

Find and fix an error in a **for** loop with **range()** function.

# This function should count down by -2 from 11 to 1, so that it only

# prints odd numbers.

# This range(11, -2) tells the for loop to start at 11 and end at index

# position -2 (which corresponds to the numeric value of -1). Since the

# third incremental or decremental value is missing, the loop will

# increment by the default of +1 instead of the intended -2 decrement.

# Starting at index position 11 and incrementing by +1 will end the loop

# automatically, because the index is not counting down towards -2 as

# the end of the range.

# To fix this problem, the range() needs three parameters:

# First parameter should be the starting index position of 11.

# Second parameter should be the ending index position of 0 (value 1).

# Third parameter should be decrementing by -2.

# So, the range should be configured as range(11, 0, -2).

# Fix this loop with the corrected range parameters and click Run.

for n in range(11, -2):

if n % 2 != 0:

print(n, end=" ")

# Should print: 11, 9, 7, 5, 3, 1 once the problem is fixed.

**Skill 2: Using while loops**

* Use a **while** loop to print a sequence of numbers .

# For this example, the while loop will count down by threes starting

# from 18 and ending at 0.

starting\_number = 18

# The while loop will continue to loop until it reaches 0.

while starting\_number >= 0:

# To make the sequence of numbers easier to read, include a space

# between each number in the sequence. You can override the default

# behavior of the print() function by using the "end=" parameter with

# the print() function. The syntax for adding a space is: end=" "

print(starting\_number, end=" ")

# Decrement the "starting\_number" variable by -3.

starting\_number -= 3

# Should print 18 15 12 9 6 3 0

* Use a **while** loop to count the number of digits in a numerical value

# This function accepts a CEO's salary as a variable.

# It counts the number of digits in the salary and

# returns the sentence like:

# "The CEO has a 6-figure salary."

def X\_figure(salary):

# Initializes the counter as an integer.

tally = 0

# The if-statement checks if the variable "salary"

# is equal to 0.

if salary == 0:

# If true, then it increments the counter to

# show there is 1 digit in 0.

tally += 1

# The while loop starts to run while the "salary"

# is greater than or equal to 1 (the loop will

# not run if the "salary" is 0).

while salary >= 1:

# The body of the while loop counts the digits

# in "salary" by counting the number of times

# "salary" can be divided by 10 until "salary"

# is no longer >= 1.

salary = salary/10

# Add 1 to the counter to tally the number of

# times the loop runs.

tally += 1

# Return the results of the "tally" of the number

# of digits in "salary".

return tally

# Call the X\_figure function with 1 parameter, converted to a string,

# inside a print function with additional strings.

print("The CEO has a " + str(X\_figure(2300000)) + "-figure salary.")

# Should print"The CEO has a 7-figure salary."

**Skill 3: Using while loops with if-else statements**

* Use a function to accept two variable integers.
* Use nested **if-else** statements and **while** loops to count up or count down from the first variable to the second variable.

# "Going down: " plus the count up or down of the

# "floor" numbers.

floor = enter

elevator\_direction = ""

# If the passenger enters the elevator on a floor that

# is higher than the destination floor:

if enter > exit:

# Then the "elevator\_direction" string will be

# initialized with the string "Going down: ".

elevator\_direction = "Going down: "

# While the "floor" number is greater than or

# equal to the exit floor number:

while floor >= exit:

# The "floor" number is converted to a string

# and is appended to the string variable

# "elevator\_direction".

elevator\_direction += str(floor)

# If the "floor" number is still greater than

# the exit floor number:

if floor > exit:

# A pipe | character is added between each

# floor number in the string variable

# "elevator\_direction" to provide a visual

# divider between numbers. The if-statement

# above (if floor > exit) prevents the pipe

# character from appearing after the "floor"

# number is no longer greater than the "exit"

# variable.

elevator\_direction += " | "

# Decrement the "floor" number as the elevator

# goes down.

floor -= 1

# Else, it is implied that the passenger is entering the

# elevator on a floor that is lower than the destination

# floor.

else:

# The "elevator\_direction" string will be initialized

# with the string "Going up: ".

elevator\_direction = "Going up: "

# While the "floor" number is less than or equal to the

# "exit" floor number:

while floor <= exit:

# Convert the the "floor" number to a string and append

# it to the string variable "elevator\_direction".

elevator\_direction += str(floor)

# If the entry floor number is still less than the exit

# floor number:

if floor < exit:

# The pipe | character is added between each

# floor number in the string variable

# "elevator\_direction" to provide a visual

# divider between numbers. The if-statement

# above (if floor < exit) prevents the pipe

# character from appearing after the "floor"

# number is no longer less than the "exit"

# variable.

elevator\_direction += " | "

# Increments the "floor" number as the elevator goes up.

floor += 1

# Returns the string holding the elevator direction (Going down or

# Going up) along with the floor countdown or count up.

return elevator\_direction

# Call the function with 2 integer parameters.

print(elevator\_floor(1,4)) # Should print Going up: 1 | 2 | 3 | 4

print(elevator\_floor(6,2)) # Should print Going down: 6 | 5 | 4 | 3 | 2

**Reminder: Correct syntax is critical**Using precise syntax is critical when writing code in any programming language, including Python. Even a small typo can cause a syntax error and the automated Python-coded quiz grader will mark your code as incorrect. This reflects real life coding errors in the sense that a single error in spelling, case, punctuation, etc. can cause your code to fail. Coding problems caused by imprecise syntax will always be an issue whether you are learning a programming language or you are using programming skills on the job. So, it is critical to start the habit of being precise in your code now.   
No credit will be given if there are any coding errors on the automated graded quizzes - including minor errors. Fortunately, you have 3 optional retake opportunities on the graded quizzes in this course. Additionally, you have unlimited retakes on practice quizzes and can review the videos and readings as many times as you need to master the concepts in this course.   
Now, before starting the graded quiz, please review this list of common syntax errors coders make when writing code.  
**Common syntax errors:**

* + Misspellings
  + Incorrect indentations
  + Missing or incorrect key characters:
    - Parenthetical types - ( curved ), [ square ], { curly }
    - Quote types - "straight-double" or 'straight-single', “curly-double” or ‘curly-single’
    - Block introduction characters, like colons - :
  + Data type mismatches
  + Missing, incorrectly used, or misplaced Python reserved words
  + Using the wrong case (uppercase/lowercase) - Python is a case-sensitive language

**Resources**For additional Python practice, the following links will take you to several popular online interpreters and codepads:

* [Welcome to Python](https://www.python.org/shell/)
* [Online Python Interpreter](https://www.onlinegdb.com/online_python_interpreter)
* [Create a new Repl](https://repl.it/languages/python3)
* [Online Python-3 Compiler (Interpreter)](https://www.tutorialspoint.com/execute_python3_online.php)
* [Compile Python 3 Online](https://rextester.com/l/python3_online_compiler)
* [Your Python Trinket](https://trinket.io/python3)

### Module 3 Graded Assessment

