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.

You'll want to watch out for a common mistake: forgetting to initialize variables. If you try to use a variable without first initializing it, you'll run into a **NameError**. This is the Python interpreter catching the mistake and telling you that you're using an undefined variable. The fix is pretty simple: initialize the variable by assigning the variable a value before you use it.

Another common mistake to watch out for that can be a little trickier to spot is forgetting to initialize variables with the correct value. If you use a variable earlier in your code and then reuse it later in a loop without first setting the value to something you want, your code may wind up doing something you didn't expect. Don't forget to initialize your variables before using them!

Another easy mistake that can happen when using loops is introducing an infinite loop. An infinite loop means the code block in the loop will continue to execute and never stop. This can happen when the condition being evaluated in a *while* loop doesn't change. Pay close attention to your variables and what possible values they can take. Think about unexpected values, like zero.

In the Coursera code blocks, you may see an error message that reads "Evaluation took more than 5 seconds to complete." This means that the code encountered an infinite loop, and it timed out after 5 seconds. You should take a closer look at the code and variables to spot where the infinite loop is.

For loops allow you to iterate over a sequence of values. Let's take the example from the beginning of the video:

```
for x in range(5):
  print(x)
```

Similar to *if* statements and *while* loops, *for* loops begin with the keyword ***for*** with a colon at the end of the line. Just like in function definitions, *while* loops and *if* statements, the body of the *for* loop begins on the next line and is indented to the right. But what about the stuff in between the *for* keyword and the colon? In our example, we're using the *range()* function to create a sequence of numbers that our *for* loop can iterate over. In this case, our variable **x** points to the current element in the sequence as the *for* loop iterates over the sequence of numbers. Keep in mind that in Python and many programming languages, a range of numbers will start at 0, and the

list of numbers generated will be one less than the provided value. So *range(5)* will generate a sequence of numbers from 0 to 4, for a total of 5 numbers.

Bringing this all together, the range(5) function will create a sequence of numbers from 0 to 4. Our *for* loop will iterate over this sequence of numbers, one at a time, making the numbers accessible via the variable **x** and the code within our loop body will execute for each iteration through the sequence. So for the first loop, **x** will contain 0, the next loop, 1, and so on until it reaches 4. Once the end of the sequence comes up, the loop will exit and the code will continue.

The power of *for* loops comes from the fact that it can iterate over a sequence of any kind of data, not just a range of numbers. You can use *for* loops to iterate over a list of strings, such as usernames or lines in a file.

Not sure whether to use a *for* loop or a *while* loop? Remember that a *while* loop is great for performing an action over and over until a condition has changed. A *for* loop works well when you want to iterate over a sequence of elements.

Previously we had used the *range()* function by passing it a single parameter, and it generated a sequence of numbers from 0 to one less than we specified. But the *range()* function can do much more than that. We can pass in two parameters: the first specifying our starting point, the second specifying the end point. Don't forget that the sequence generated won't contain the last element; it will stop one before the parameter specified.

The *range()* function can take a third parameter, too. This third parameter lets you alter the size of each step. So instead of creating a sequence of numbers incremented by 1, you can generate a sequence of numbers that are incremented by 5.

To quickly recap the *range()* function when passing one, two, or three parameters:

- One parameter will create a sequence, one-by-one, from zero to one less than the parameter.
- Two parameters will create a sequence, one-by-one, from the first parameter to one less than the second parameter.
- Three parameters will create a sequence starting with the first parameter and stopping before the second parameter, but this time increasing each step by the third parameter.

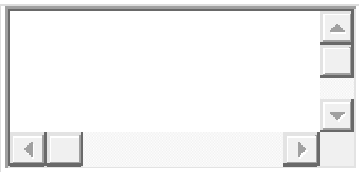
Loops Cheat Sheet

Check out below for a run down of the syntax for while loops and for loops.

While Loops

A while loop executes the body of the loop while the condition remains True.

Syntax:



```
while condition:
    body
```

Things to watch out for!

- **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.**

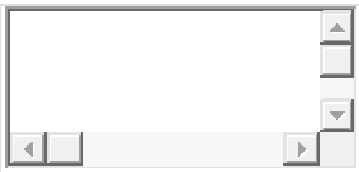
Typical use:

While loops are mostly used when there's an unknown number of operations to be performed, and a condition needs to be checked at each iteration.

For Loops

A for loop iterates over a sequence of elements, executing the body of the loop for each element in the sequence.

Syntax:



```
for variable in sequence

    body
```

The range() function:

range() generates a sequence of integer numbers. It can take one, two, or three parameters:

- range(n): 0, 1, 2, ... n-1
- range(x,y): x, x+1, x+2, ... y-1
- range(p,q,r): p, p+r, p+2r, p+3r, ... q-1 (if it's a valid increment)

Common pitfalls:

- **Forgetting that the upper limit of a range() isn't included.**
- **Iterating over non-sequences.** Integer numbers aren't iterable. Strings are iterable letter by letter, but that might not be what you want.

Typical use:

For loops are mostly used when there's a pre-defined sequence or range of numbers to iterate.

Break & Continue

You can interrupt both while and for loops using the break keyword. We normally do this to interrupt a cycle due to a separate condition.

You can use the continue keyword to skip the current iteration and continue with the next one. This is typically used to jump ahead when some of the elements of the sequence aren't relevant.

If you want to learn more, check out this wiki page on for loops.

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
- See what happens when you Search Google for Recursion