More on loops in Python

Previously, you explored iterative statements. An **iterative statement** is code that repeatedly executes a set of instructions. Depending on the criteria, iterative statements execute zero or more times. We iterated through code using both *for* loops and *while* loops. In this reading, you'll recap the syntax of loops. Then, you'll learn how to use the *break* and *continue* keywords to control the execution of loops.

for loops

If you need to iterate through a specified sequence, you should use a *for* loop.

The following *for* loop iterates through a sequence of usernames. You can run it to observe the output:

```
for i in ["elarson", "bmoreno", "tshah", "sgilmore"]:
    print(i)
```

Output

```
elarson
bmoreno
tshah
sgilmore
```

The first line of this code is the loop header. In the loop header, the keyword *for* signals the beginning of a *for* loop. Directly after *for*, the loop variable appears. The **loop variable** is a variable that is used to control the iterations of a loop. In *for* loops, the loop variable is part of the header. In this example, the loop variable is *i*.

The rest of the loop header indicates the sequence to iterate through. The *in* operator appears before the sequence to tell Python to run the loop for every item in the sequence. In this example, the sequence is the list of usernames. The loop header must end with a colon (*:*).

The second line of this example *for* loop is the loop body. The body of the *for* loop might consist of multiple lines of code. In the body, you indicate what the loop should do with each iteration. In this case, it's to *print(i)*, or in other words, to display the current value of the loop variable during that

iteration of the loop. For Python to execute the code properly, the loop body must be indented further than the loop header.

Note: When used in a *for* loop, the *in* operator precedes the sequence that the *for* loop will iterate through. When used in a conditional statement, the *in* operator is used to evaluate whether an object is part of a sequence. The example *if* "*elarson*" *in* ["*tshah*", "*bmoreno*", "*elarson*"] evaluates to *True* because "*elarson*" is part of the sequence following *in*.

Looping through a list

Using *for* loops in Python allows you to easily iterate through lists, such as a list of computer assets. In the following *for* loop, *asset* is the loop variable and another variable, *computer_assets*, is the sequence. The *computer_assets* variable stores a list. This means that on the first iteration the value of *asset* will be the first element in that list, and on the second iteration, the value of *asset* will be the second element in that list. You can run the code to observe what it outputs:

```
computer_assets = ["laptop1", "desktop20", "smartphone03"]
for asset in computer_assets:
    print(asset)
```

output

```
laptop1
desktop20
smartphone03
```

Note: It is also possible to loop through a string. This will return every character one by one. You can observe this by running the following code block that iterates through the string "*security*":

```
string = "security"
for character in string:
    print(character)
```

output

```
s
e
c
u
r
```

```
i
t
y
```

Using range()

Another way to iterate through a *for* loop is based on a sequence of numbers, and this can be done with *range()*. The *range()* function generates a sequence of numbers. It accepts inputs for the start point, stop point, and increment in parentheses. For example, the following code indicates to start the sequence of numbers at *0*, stop at *5*, and increment each time by *1*:

range(0, 5, 1)

Note: The start point is inclusive, meaning that *0* will be included in the sequence of numbers, but the stop point is exclusive, meaning that *5* will be excluded from the sequence. It will conclude one integer before the stopping point.

When you run this code, you can observe how *5* is excluded from the sequence:

```
for i in range(0, 5, 1):
    print(i)
```

output

```
0
1
2
3
4
```

You should be aware that it's always necessary to include the stop point, but if the start point is the default value of *0* and the increment is the default value of *1*, they don't have to be specified in the code. If you run this code, you will get the same results:

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

output

```
0
1
2
```

```
3
4
```

Note: this is the last time i put output under the code and above the output, asume if theres two code blocks next to eachother without a space that its teh output

Note: If the start point is anything other than *0* or the increment is anything other than *1*, they should be specified.

while loops

If you want a loop to iterate based on a condition, you should use a *while* loop. As long as the condition is *True*, the loop continues, but when it evaluates to *False*, the *while* loop exits. The following *while* loop continues as long as the condition that *i < 5* is *True*:

```
i = 1
while i < 5:
    print(i)
    i = i + 1
```

```
1
2
3
4
```

In this *while* loop, the loop header is the line *while i < 5:*. Unlike with *for* loops, the value of a loop variable used to control the iterations is not assigned within the loop header in a *while* loop. Instead, it is assigned outside of the loop. In this example, *i* is assigned a starting value of *1* in a line preceding the loop.

The keyword *while* signals the beginning of a *while* loop. After this, the loop header indicates the condition that determines when the loop terminates. This condition uses the same comparison operators as conditional statements. Like in a *for* loop, the header of a *while* loop must end with a colon (*:*).

The body of a *while* loop indicates the actions to take with each iteration. In this example, it is to display the value of *i* and to increment the value of *i* by *1*. In order for the value of *i* to change with each iteration, it's necessary to indicate this in the body of the *while* loop. In this example, the loop iterates four times until it reaches a value of *5*.

Integers in the loop condition

Often, as just demonstrated, the loop condition is based on integer values. For example, you might want to allow a user to log in as long as they've logged in less than five times. Then, your loop variable, *login_attempts*, can be initialized to *0*, incremented by *1* in the loop, and the loop condition can specify to iterate only when the variable is less than *5*. You can run the code below and review the count of each login attempt:

```
login_attempts = 0
while login_attempts < 5:
    print("Login attempts:", login_attempts)
    login_attempts = login_attempts + 1
```

```
Login attempts: 0
Login attempts: 1
Login attempts: 2
Login attempts: 3
Login attempts: 4
```

The value of *login_attempts* went from *0* to *4* before the loop condition evaluated to *False*. Therefore, the values of *0* through *4* print, and the value *5* does not print.

Boolean values in the loop condition

Conditions in *while* loops can also depend on other data types, including comparisons of Boolean data. In Boolean data comparisons, your loop condition can check whether a loop variable equals a value like *True* or *False*. The loop iterates an indeterminate number of times until the Boolean condition is no longer *True*.

In the example below, a Boolean value is used to exit a loop when a user has made five login attempts. A variable called *count* keeps track of each login attempt and changes the *login_status* variable to *False* when the *count* equals *4*. (Incrementing *count* from *0* to *4* represents five login attempts.) Because the *while* condition only iterates when *login_status* is *True*, it will exit the loop. You can run this to explore this output:

```
count = 0
login_status = True
while login_status == True:
    print("Try again.")
    count = count + 1
    if count == 4:
        login_status = False
```

```
Try again.
Try again.
Try again.
Try again.
```

The code prints a message to try again four times, but exits the loop once *login_status* is set to *False*.

Managing loops

You can use the *break* and *continue* keywords to further control your loop iterations. Both are incorporated into a conditional statement within the body of the loop. They can be inserted to execute when the condition in an *if* statement is *True*. The *break* keyword is used to break out of a loop. The *continue* keyword is used to skip an iteration and continue with the next one.

break

When you want to exit a *for* or *while* loop based on a particular condition in an *if* statement being *True*, you can write a conditional statement in the body of the loop and write the keyword *break* in the body of the conditional.

The following example demonstrates this. The conditional statement with *break* instructs Python to exit the *for* loop if the value of the loop variable *asset* is equal to "*desktop20*". On the second iteration, this condition evaluates to *True*. You can run this code to observe this in the output:

```
computer_assets = ["laptop1", "desktop20", "smartphone03"]
for asset in computer_assets:
    if asset == "desktop20":
        break
    print(asset)
```

```
laptop1
```

As expected, the values of "*desktop20*" and "*smartphone03*" don't print because the loop breaks on the second iteration.

continue

When you want to skip an iteration based on a certain condition in an *if* statement being *True*, you can add the keyword *continue* in the body of a conditional statement within the loop. In this example, *continue* will execute when the loop variable of *asset* is equal to "*desktop20*". You can run this code to observe how this output differs from the previous example with *break*:

```
computer_assets = ["laptop1", "desktop20", "smartphone03"]
for asset in computer_assets:
    if asset == "desktop20":
        continue
    print(asset)
```

```
laptop1
smartphone03
```

The value "*desktop20*" in the second iteration doesn't print. However, in this case, the loop continues to the next iteration, and "*smartphone03*" is printed.

Infinite loops

If you create a loop that doesn't exit, this is called an infinite loop. In these cases, you should press *CTRL-C* or *CTRL-Z* on your keyboard to stop the infinite loop. You might need to do this when running a service that constantly processes data, such as a web server.

Key takeaways

Security analysts need to be familiar with iterative statements. They can use *for* loops to perform tasks that involve iterating through lists a predetermined number of times. They can also use *while* loops to perform tasks based on certain conditions evaluating to *True*. The *break* and *continue* keywords are used in iterative statements to control the flow of loops based on additional conditions.

Revision #1
Created 11 December 2023 07:26:40 by naruzkurai
Updated 19 December 2023 03:38:35 by naruzkurai