Nested Loops

by Sophia

WHAT'S COVERED

In this lesson, you will learn about loops created within another loop known as nested loops. Specifically, this lesson covers:

1. Nested Loops

At a high level, a nested loop is just a loop within another loop. This is very similar to the idea of nested conditional statements from Challenge 1.3. There, we used if statements nested in other if statements. This same nesting idea can be applied to loops as well.

A nested loop can be quite useful when you have a series of statements that includes a loop that you want to have repeated. The functionality is no different than using a single loop except that the "outer" loop has one or more "inner" loops within it. There are many situations where using a nested loop is beneficial, such as looping through 2D data collection types. Another great example could be prompting a teacher for numeric grades to calculate the average grade for a student. However, instead of prompting for a single student's grades, it could prompt for the entire class. To handle this, you would add an outer loop that loops across all of the students while the inner loop prompts the grades for each student.

The format of nested `while` loops would look something like this:

EXAMPLE

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

And the format of nested `for` loops would look something like this:

EXAMPLE

```
for <variable> in <iterable>:
  for <variable> in <iterable>:
    <statement(s)>
  <statement(s)>
```

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

If you notice in these nested examples, the inner loop would be executed one time for each iteration of the outer loop.

THINK ABOUT IT

What do you think would happen if the indentations were not coded correctly? That's right, exception errors would occur during runtime. Remember, it is very important to ensure that we have the indentations correct, as the indentations determine which loop a line of code is part of. Using correct indentations will become more important and complex as the number of loops (or conditional statements for that matter) increases.

On that note, there is no limit on nested loops used or whether or not we have a `for` loop as the outer loop and a `while` loop as the inner loop. In a nested loop, the number of total iterations (all loops included) will be the total number of iterations in the outer loop multiplied by the iterations of the inner loop. In each iteration of the outer loop, the inner loop will execute all of its iterations. Then for each iteration of an outer loop, the inner loop will re-start and complete its execution before the outer loop can continue to its next iteration.

These nested loops are especially useful when it comes to working with multi-dimensional data collection types, such as printing out multi-dimensional lists or iterating a list that has a nested list.

Let's go back to revisit some of the multiple dimension lists that we introduced in Challenge 2.1.

EXAMPLE

```
multiplesList = [[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
print(multiplesList)
```

The output with four nested lists.

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Remember this example? It's four nested lists within a list. Accessing individual elements can be a bit of a challenge if we wanted to output items within the list. We can do the same and more with loops.

Let's see how we can rebuild this list but also add to it by extending it to store the multiples of values here. To do so, we need to identify that the first list within the outer list stores the multiples of 1, the second list stores the multiples of 2, and so forth. That pattern is consistent and being that we have a set number for each (4 outer lists and 5 elements per list), we can use the `for` loop for that purpose.

EXAMPLE

```
multiples = []
```

```
for outer in range(1,5):
  multiples.append([])
  for inner in range(1,6):
    multiples[outer-1].append( outer * inner)
print(multiples)
```

In this example program, we are first declaring the variable multiples as an empty list. Then, we have the outer loop where we create a variable called `outer` and set it to the range of 1 going to the value of 5. In the outer `for` loop, we first append (using the `.append()` method) an empty list to multiples. Within the inner loop, we created a variable called `inner` and set it to the value of 1 going up to 6 which exits when the value hits 6. Next, we are appending to the list at the position of the outer variable minus 1 (since the list starts at 0) the value of the outer loop multiplied by the inner loop.

TRY IT

Directions: Nested loops can look a little tricky. Go ahead and enter the program above into the IDE and see if you get the output below.

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

Did you get the same output? If not, check your indentations.

Let's see why this works and what the values are within the nested loop by adding in outputs within the nested loop to make it easier to see what is happening during each iteration.

TRY IT

Directions: Now add the modified `print()` function below so we can see each loop being built.

```
multiples = []
for outer in range(1,5):
  multiples.append([])
  for inner in range(1,6):
    print("Outer: ",outer, ", Inner: ", inner, "Outer x inner: ",inner * outer)
    multiples[outer-1].append( outer * inner)
print(multiples)
```

TRY IT

Directions: Go ahead and run the program to see each iteration of the loop being built.

```
Outer:  1 ,  Inner:  1 Outer x inner:  1
Outer:  1 ,  Inner:  2 Outer x inner:  2
Outer:  1 ,  Inner:  3 Outer x inner:  3
Outer:  1 ,  Inner:  4 Outer x inner:  4
Outer:  1 ,  Inner:  5 Outer x inner:  5
```

```
Outer:  2 ,  Inner:  1 Outer x inner:  2
Outer:  2 ,  Inner:  2 Outer x inner:  4
Outer:  2 ,  Inner:  3 Outer x inner:  6
Outer:  2 ,  Inner:  4 Outer x inner:  8
Outer:  2 ,  Inner:  5 Outer x inner:  10
Outer:  3 ,  Inner:  1 Outer x inner:  3
Outer:  3 ,  Inner:  2 Outer x inner:  6
Outer:  3 ,  Inner:  3 Outer x inner:  9
Outer:  3 ,  Inner:  4 Outer x inner:  12
Outer:  3 ,  Inner:  5 Outer x inner:  15
Outer:  4 ,  Inner:  1 Outer x inner:  4
Outer:  4 ,  Inner:  2 Outer x inner:  8
Outer:  4 ,  Inner:  3 Outer x inner:  12
Outer:  4 ,  Inner:  4 Outer x inner:  16
Outer:  4 ,  Inner:  5 Outer x inner:  20
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

You can see that the outer loop remains the same value of 1 for each iteration of the inner loop going to 5. Once the inner loop reaches 6, the loop exits and goes back to the outer loop for the next iteration when the value is 2 and so on.

Now let's also add a nested loop for the output to make it easier to navigate and access each element. This is important as it's not practical to update individual elements one at a time manually.

TRY IT

Directions: Let's add the updated program below. We notice that the first snippet is the same as before, but without that modified `print()` function line. The second snippet of code is what will make the output a little easier to read, by outputting each element in the order that they appear.

```
multiples = []
for outer in range(1,5):
  multiples.append([])
  for inner in range(1,6):
    multiples[outer-1].append( outer * inner)
print(multiples)

for outerList in multiples:
  for innerValue in outerList:
    print (innerValue," ",end ='')
  print()
```

To break down the second snippet of code (the second `for` loop), we created a variable called `outerList` and set it to each list within the multiples in the outer list. Then in the inner for loop, we created a variable `innerValue` and set it to each of the `outerList` values. Within the inner `for` loop, we output each element with a space and use the `end=' '` to skip over the output of the new line.

KEY CONCEPT

The **end** as a parameter of the `print()` function is something that we haven't looked at before. By default, the `print()` function outputs and automatically creates a new line, meaning it acts much like the Enter key on your keyboard and moves to the next line after completion. Using the end parameter of the `print()` function, we are able to change the default operation of the `print()` function. There are essentially two ways we can use the end parameter:

1. end='\n'
   a. Using "\n" for a new line is exactly what the `.print()` function does by default. You would not need to enter the end parameter if you just needed a new line afterwards since `print()` will do that anyways.
2. end=' '
   a. Using a space (or any character(s)) will keep the `print()` function from creating a new line. In our case, entering a space is ideal to keep the output on a single line.

After the inner loop, we're including a single `print()` function to create a new line. Remember that a function can have many parameters or no parameters. A `print()` function call without parameters would simply output a new line.

TRY IT

Directions: Go ahead and let's run this program.

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

Did you get the same output?

Without the `print()` function call without parameters, we would simply have each element on a single line.

TRY IT

Directions: Try removing the final `print()` and run the program. Place it back in. See the difference?

```
[[1, 2, 3, 4, 5], [2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 4 8 12 16 20
```

THINK ABOUT IT

The benefit of using this type of range functionality is that now we have the ability to customize it to any number we wish for the range of the outer and inner loop.

For example, as a 3X3.

TRY IT

Directions: Try building a 3X3 (3 nested lists) and run it. Visually, you can think of this as being a table that consists of rows and columns. In a 3X3 nested list, you would have 3 rows and 3 columns.

```
multiples = []
for outer in range(1,4):
  multiples.append([])
  for inner in range(1,4):
    multiples[outer-1].append( outer * inner)
print(multiples)

for outerList in multiples:
  for innerValue in outerList:
    print (innerValue," ",end ='')
  print()
```

The output with the nested loops.

```
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
1  2  3
2  4  6
3  6  9
```

Did you get three nested lists as the output? Looks similar to a Tic-Tac-Toe board, wouldn't you say?

Let’s make it bigger. How about a 10X10?

TRY IT

Directions: Try building a 10X10 (10 nested lists) and run it.

```
multiples = []
for outer in range(1,11):
  multiples.append([])
  for inner in range(1,11):
    multiples[outer-1].append( outer * inner)
print(multiples)

for outerList in multiples:
  for innervalue in outerList:
    print (innervalue," ",end ='')
  print()
```

The output with ten nested loops.

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [2, 4, 6, 8, 10, 12, 14, 16, 18, 20], [3, 6, 9, 12, 15, 18,
 21, 24, 27, 30], [4, 8, 12, 16, 20, 24, 28, 32, 36, 40], [5, 10, 15, 20, 25, 30, 35, 40, 45,
 50], [6, 12, 18, 24, 30, 36, 42, 48, 54, 60], [7, 14, 21, 28, 35, 42, 49, 56, 63, 70], [8, 1
6, 24, 32, 40, 48, 56, 64, 72, 80], [9, 18, 27, 36, 45, 54, 63, 72, 81, 90], [10, 20, 30, 40,
 50, 60, 70, 80, 90, 100]]
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

We can customize this to any format that we wish using these nested loops. The best part is that we do not have to manually set each number. What can be helpful as well here is to recognize the pattern and create the code to reflect that pattern.

TERM TO KNOW

end
The end parameter is a function of the `print()` function. Using the end parameter, we are able to change the default operation of the `print()` function, namely to prevent the `print()` function from creating a new line.

SUMMARY

In this lesson, we learned about using loops within other loops, also known as **nested loops**. Using loops within other loops is similar to the nested conditional statements. We learned how important it is to ensure that we get the indentations right as that determines which loop a line of code is in. We identified that a nested loop can be quite useful when you have a series of statements that includes a loop that you want to have repeated. Finally, we had the opportunity to create and run some looping programs.

Best of luck in your learning!

TERMS TO KNOW

end

The end parameter is a function of the print() function. Using the end parameter, we are able to change the default operation of the print() function, namely to prevent the print() function from creating a new line.