Nested Loops

Definition

Though there are no new rules delineating nested loops, many beginning programmers think of nested loops as being different than regular loops. The term "nested loop" typically refers to any set-up where a loop resides inside of another loop. If there are just two loops, some refer to the setup as a "double loop" setup. It's important to note that nested loops are different than two separate loops. In the example below with two separate loops, assume that all necessary variables are declared.

```
for i in range(n):
    stmt1
for j in range(n):
    stmt2
```

In this setup, stmt1 runs exactly n times, followed by stmt2 running exactly n times.

A nested loop setup would look like this instead:

```
for i in range(n):
    for j in range(n):
        stmt1
```

In this situation, let's carefully trace through the order of execution, paying particular attention to the sequence of values for the two variables i and j.

First, we execute the initialization statement i = 0, setting i to 0. Next, we check the Boolean condition i < n, which will be true so long as n is positive. Next, we must execute the ENTIRE body of this loop before we get to the point where we execute the increment statement, i++.

We have seen a loop like this:

```
    for j in range(n)
        stmt1
```

before. Basically, it will go through each value of j starting at 0 and ending at n-1. While all of this occurs, i stays set at 0. After this loop finishes, we THEN go to the statement i++ and change i to 1. From this point, we once again execute the statement:

```
    for j in range(n):
        stmt1
```

which will go through each value of j starting at 0 and ending at n-1.

Repeating this pattern, we find that the nested loop structure will execute the given statement n^2 times, and the following chart shows the values for i and j for each iteration for n = 4:

i	j
0	0
0	1
0	2
0	3
1	0
1	1
1	2
1	3
2	0
2	1
2	2
2	3
3	0
3	1
3	2
3	3

Tracing Examples

Let's determine the output of the following segment of code:

```
for i in range(4):
    for j in range(4):
        print(i,end=" ")
```

When i is 0, we run the WHOLE inner for loop. Notice that when we run it, we print out the value of i (which is currently 0) multiple times. In particular, we print it four times, so our output begins:

```
0 0 0 0
```

Next, we set i to 1, and repeat the inner for loop again. We know that this for loop will simply output the value of i (which is now 1), four times. After this iteration for i, the output is

```
0 0 0 0 1 1 1 1
```

Naturally, when we repeat this process for i equals 2 and i equals 3, our final output will be:

```
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

How would this output change if the inner loop ran five times?

```
for i in range(4):
    for j in range(5):
        print(i,end=" ")
```

In a nutshell, each time we printed i, we would be printing it 5 times not 4, so we would have:

```
0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Now, consider the following segment of code:

```
for i in range(4):
    for j in range(5):
        print(j,end=" ")
```

This time, instead of printing out the value i, we will be printing out the value of j. Notice that each time through, j goes from 0 to 4, inclusive. In fact, the value of I doesn't matter at all with respect to what gets printed out. Instead, i only controls how many times the sequence 0 1 2 3 4 gets printed. Thus, the output of this code segment is:

```
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
```

Now imagine that we would like four our printout to be a bit different:

```
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

The only difference here is that we need to insert a printf to print out a newline AFTER we are done printing out zero through four. This should appear after the value four is printed each time:

```
for i in range(4):
    for j in range(5):
        print(i,end=" ")
    print(" ")
```

Notice that a block of statements is now needed because we need to encompass two statements inside of the for-i loop.

Tracing Example: Inner Loop depends on the Outer Loop

Let’s make a slight change to our original example:

```
for i in range(1,5):
    for j in range(i):
        print(i,end=" ")
```

One change is that the first loop goes from 1 to 4 instead of 0 to 3. The second change is that the inner loop runs i times, a variable, instead of 4, a constant. In essence, what we are saying is that we want the inner loop to run exactly i times instead of four times. This means the first time around, it will run once. The second time it will run twice, etc.

The corresponding output is as follows:

```
1 2 2 3 3 3 4 4 4 4
```

If we insert new line characters like so:

```
for i in range(1,5):
    for j in range(i):
        print(i,end=" ")
    print();
```

The output is as follows:

```
1
2 2
3 3 3
4 4 4 4
```

The key idea here is that the inner loop *depends on* the value of a variable that changes in the outer loop, so its actions are different each time around (in a predictable pattern.)

How can we edit the previous code segment to produce the following output?

```
1
1 2
1 2 3
1 2 3 4
```

Stars Example: Right Triangle

Now that we've gotten a reasonable idea of the interaction of nested loops, we can attack the follow problem:

Write a program that asks the user for a positive integer, n, and then prints out a right triangle of side length n with the star character, '*'. For example, if 6 were entered for n, the appropriate output would be:

```
*
**
***
****
*****
******
```

If we take a look at the last two examples, the general shape of the output of this design is the exact same as those examples.

Thus, we should be able to model our solution almost exactly using those molds. In fact, the only difference is that our print will print out a star instead of a number. The basic idea is as follows:

```
Loop through each row (1 through n):

For each row

        Print out row number of stars.
        Print a new line character.
```

We can now create our program that will initially prompt for the user to enter n. We will assume they will enter a reasonable value (greater than zero and less than 80, since most monitors display only 80 columns.)

```
n = int(input("How many rows do you want?\n"))
for i in range(1,n+1):
    for j in range(i):
        print("*",end="")
    print()
```

Stars Example: Backwards Right Triangle

Now, let's attempt to write a program to print out a different shape, one that looks like this (for n = 6):

```
******
*****
****
***
**
*
```

The only difference here is that the number of stars on the first row starts at 6 and counts down. There's no reason that all for loops must count up, so to speak. We can simply arrange for our for loop to start at n and count down to 1, as follows:

```
for in range(n,0,-1):
```

We can clearly see that this loop will start at n, and after each iteration, will decrement i by one, until it equals 1. Notice that we do want to print exactly one star on the last line, so this value of i is included in the loop.

Here is our adjusted program:

```
n = int(input("How many rows do you want?\n"))
for i in range(n,0,-1):
    for j in range(i):
        print("*",end="")
    print()
```

Stars Example: Left-Justified Right Triangle

Now let's consider the changes we need to make to create a design of the following type (for n = 6):

```
     *
    **
   ***
  ****
 *****
******
```

The key difference with this example is that we need to print spaces before some of the stars. For example, on the first line, we must print n-1 spaces before we print one star. On the second line, we must print n-2 spaces before we print 2 stars. The pattern persists until we get to the last row where we print 0 spaces followed by n stars.

This indicates that we'll keep our plan of counting down with the outer loop. The value of the outer loop counter will equal the number of spaces we want on that row. After we print those spaces though, we must calculate the number of stars we want on that row. In particular, the pattern seems to be that

stars = n – spaces

Using this information, here is our basic game plan:

Go through each row, starting at n-1 downto 0

For each row do the following:

1) Print out row number of spaces
2) Print out n – row number of stars
3) Print out a newline character

Thus, our outer loop will now contain three statements inside of it. In the past, we didn't print out spaces, since trailing ones simply aren't necessary.

Here is the program:

```
n = int(input("How many rows do you want?\n"))
for i in range(n-1,-1,-1):

    for j in range(i):
        print(" ",end="")
    for j in range(n-i):
        print("*",end="")
    print()
```

Some of our key changes are as follows:

1) The i loop starts at n-1 instead of n and ends at 0 instead of 1.

2) We calculated the number of stars needed and stored this in a variable stars. This isn't necessary as we could have put n – i instead the Boolean expression in the j loop, but adding this variable makes the code easier to read.

3) Notice we can reuse the variable j is the star loop because we have FINISHED using it to print out spaces.