Python Lecture: Nested Loops

(Examples: mult, stars, primetest, diamond, checkerboard)

Loops Inside of Loops

I. Analogy - Nested Loops and Tables, Multiplication Table

In most of the loop situations we have encountered so far, a loop index marches linearly in some direction, eventually stopping when it gets too big or too small. If we were to visualize each unique value of the loop index, we most likely would visualize each of the values in the order the variable takes them, in one long line.

For example, the loop structure

```
for i in range(10, 40, 3):
```

we might visualize the different values i equals in sequence as follows:

10 13 16 19 22 25 28 31 34 37

But, imagine that we wanted to visualize a table of some sort, such as this one:

1 2 3 4 5

2 4 6 8 10

3 6 9 12 15

4 8 12 16 20

5 10 15 20 25

You might recognize this as a multiplication chart. Each row of the chart shows a sequence of values. Naturally, we might think of designing a loop just to print one row.

But then, we'd have to write five loops in a row just to print out this chart. One might initially come up with this to get the chart on the previous page:

```
for j in range(1, 6):
    print(j,end="\t")
print()
for j in range(1, 6):
    print(2*j,end="\t")
print()
for j in range(1, 6):
    print(3*j,end="\t")
print()
for j in range(1, 6):
    print(4*j,end="\t")
print()
for j in range(1, 6):
    print(5*j,end="\t")
print()
```

But, this seems to be a bit wasteful. What if we want to print a multiplication chart for all positive integers upto 20, would we really want to write out 20 loops, one after the other?

What we notice here is that the print itself in each of the different loops is pretty similar. In fact, the only thing that changes between each different print between successive for loops is what we multiply j by. The values we multiply j by follows a relatively straight-forward pattern: 1, 2, 3, 4, 5. If we think about our inner loop, we see that part of the point of loops is to encapsulate patterns so we don't have th physically type out so many lines of code. Thus, we can simply create a loop with a different variable to go through the integers 1, 2, 3, 4 and 5. Our resulting code is as follows:

```
for i in range(1, 6):
    for j in range(1, 6):
        print(i*j,end="\t")
    print()
```

We've found the underlying pattern in the repeated code and abstracted it away, expressing the repeated for loops by putting our initial for loop that we wrote multiple times as a single line of code inside of ***another*** for loop! Needless to say, this shortens our code ***and*** makes it much more flexible to modify. (We can very easily changet his version to print out a 20 x 20 multiplication chart. The same edit on the old version would be quite tedious!)

II. Stars - a More Complicated Nested Pattern

Consider the task of printing a right triangle pattern of stars, where each row has one more star than the previous row. Here is the design for n = 10 rows:

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

Let's write this program without the use of Python's * operator which multiplies strings. (If we use it, we can print this design without a nested loop structure.)

On the first row, we want one star. On the second row, we want two stars. In general, let's consider the problem of printing i stars on a single row. We can do it as follows:

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

Now, we simply see that we want i to be 1, then 2, then 3, etc. But this is just a loop itself. So, our solution simply becomes:

```
for i in range(1,11):
    for j in range(i):
        print("*",end="")
    print()
```

If we carefully trace through the code, we can see that when i is set to 1, j assumes just the value 0. When i is then set to 2, j assumes the values 0 and 1. When i is set to 3, j assumes the values 0, 1 and 2. The pattern continues until the last iteration of the i loop when i=10 and then j assumes the values 0,1,2,3,4,5,6,7,8 and 9. A good way to visualize the movement of loop indexes of nested loops is to list each row for the outer variable, and on that row, list the values of the inner loop variable:

```
i= 1: j=0
i= 2: j=0,1
i= 3: j=0,1,2
...
i=10: j=0,1,2,3,4,5,6,7,8,9
```

III. Prime Number Example

Now, let's consider the problem of printing out all of the prime numbers in a range. (Normally, we could do this with a prime sieve, but for this example, we'll use a less efficient but more straight-forward strategy that utilizes a nested loop structure. The prime sieve does as well, but it also uses a list, which we haven't seen yet.)

First, let's simplify our problem to determining if a particular number num is prime or not. We must try dividing i by each integer, starting at 2, ending at num-1. If any of these divisions produces an integer, then i is not prime. Technically, we can show that we can stop our trial division at $\sqrt{num}$, but for this exam we'll just try all of the divisions upto num-1. Let's say for this simplified problem, if the number is prime, we print it and if it's not, we do nothing.

First, let's think about trial division. If one number divides equally into another one, what that really means is that there's no remainder when the division is carried out. Luckily, the mod operator (%) provides us the remainder of a division. Thus, this is the preferred way to check for divisibility between integers. To determine whether or not a value is prime, we'll use a boolean variable (a variable that can be either True or False). Initially, this variable will be set to True (indicating that by default, until proven otherwise, we assume num is prime. But, if we find proof that num isn't prime, we'll just change the value of this boolean variable to False.

Here is our segment of code that solves the problem for a single variable num:

```
isPrime = True

for div in range(2, num):
    if num%div == 0:
        isPrime = False
            break

    if isPrime:
        print(num,end=" ")
```

Now, if we want to check to see which integers from integer start to integer end is prime and just print out the ones that are, we can simply just put this whole segment of code inside a loop where num ranges from start to end inclusive!

Here is the full program (primetest):

```
def main():

    start = int(input("What is the starting point of your range?\n"))
    end = int(input("What is the ending point of your range?\n"))

    print("All the primes in between",start,"and",end,"are:")
    for num in range(start, end+1):

        isPrime = True

        for div in range(2, num):
            if num%div == 0:
                isPrime = False
                break

        if isPrime:
            print(num,end=" ")

    print()

main()
```

IV. Diamonds are a Woman's Best Friend

Now we come to an example that requires more precise control with a nested loop structure. Let's define a diamond of 2n-1 carats to be composed of 2n-1 rows, where the first row has 1 star, the second 2 stars, the n^{th} row has n stars, the n+1 row has n-1 stars, the n+2 row has n-2 stars, until the last row which has one star, with spaces on some rows so the shape looks like a diamond. Here is a 7 carat diamond:

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

First, we note that the first n rows follow one pattern while the next n-1 rows follow a different pattern. It makes sense for us to split our problem into two separate problems, one for printing the top of the design and the other for printing the bottom portion of the design.

What makes this design more difficult than the prior star example is that we have to print some spaces before stars on some rows. For the example with n = 4 (7 carat diamond), on the first row we have 3 spaces and 1 star. On the second row we have 2 spaces and 3 stars. On the third row we have 1 space and 5 stars. On the fourth row

we have 0 spaces and 7 stars. The pattern is that the spaces start at n-1 and decrease by 1 each time while the stars start at 1 and increase by 2 each time. While one loop variable can be used to control both of these changes, it tends to be easier for beginning programmers to use two variables, one for spaces and one for stars.

So, one strategy that can work is for us to initialize our variables for spaces and stars before our outer for loop. Then, within the for loop, we have two separate inner for loops - one that prints spaces, followed by one that prints stars. After both of these loops complete, we update our variables for both spaces and stars, accordingly. In the code segment below that prints this part of the star design, carat is the original input (required to be an odd integer):

```
spaces = (carats-1)//2
stars = 1

for i in range((carats+1)//2):

    for j in range(spaces):
        print(" ", end="")

    for j in range(stars):
        print("*", end="")

    spaces = spaces - 1
    stars = stars + 2
    print()
```

In terms of designing this code, it really helps to be able to know how to write a loop that "prints spaces number of spaces." That way, you can put a note to yourself to do this inside the main loop and later, fill in the loop that accomplishes this task.

Now, the second portion of the design starts with 1 space and carats-2 stars on a row. (This is the n+1 row of the whole design.) From there, the spaces increment by 1 for each subsequent row and the stars decrement by 2 for each subsequent row.

Roughly speaking, we can use the same general strategy we used above to print the top portion of the design, but we just have to modify the parts of this design that are different for the bottom half. This includes the initial values for spaces and stars, the number of time the outer loop runs (n-1 not n), as well as the increments for both spaces and stars. The second portion of the program is included on the next page:

```
spaces = 1
stars = carats - 2

for i in range((carats+1)//2):

    for j in range(spaces):
        print(" ", end="")

    for j in range(stars):
        print("*", end="")

    spaces = spaces + 1
    stars = stars - 2
    print()
```

V. Checkerboard with the Turtle

This last example utilizes the Python Turtle. We will display a red and black checkerboard with the Python Turtle. Our usual loop structure to print out an 8 by 8 board would be a nested loop structure, where both loops run 8 times. Unlike our previous example however where we just always print out a star or something very simple related to our loop indices, here we have to do a couple things that are more complicated:

1. Every other square will be red and the color of the starting square is different on different rows.

2. We have to contend with pixel values instead of just drawing things at row i and column j.

There is a very elegant solution to the first issue. Let our basic looping structure look as follows:

```
SIZE = 8
for row in range(SIZE):
    for col in range(SIZE):
```

With this looping structure, here are the values or row and col at each location of the checkerboard for the first two rows. Red squares are drawn in red:

(0, 0) (0, 1) (0,2) (0,3) (0, 4) (0, 5) (0,6) (0,7)

(1, 0) (1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7)

The key pattern is that for each red square, the sum of the row and col variables is even and for each black square, this sum is odd. This makes a great deal of sense once you think about it. If the top left is red and it's (0, 0), then if we move directly right or down, we are adding precisely one to the sum of row and col, and by the definition of how a checkerboard is arranged, we change color as well. Getting from any red square to any other red square requires an even number of moves up, down, left or right. Thus, the sum of row and col will always keep the same parity (odd or even) when moving between two red squares. Same for two black squares. Thus, we can add an if statement inside of the nested loop structure to set our fill color accordingly:

```
SIZE = 8
for row in range(SIZE):
    for col in range(SIZE):
        if (row+col)%2 == 0:
            turtle.fillcolor("red")
        else:
            turtle.fillcolor("black")
```

Our second issue is when row = 0 and col = 0, we might not want to necessarily draw our square from pixel (0, 0). We want to keep our main loop as we have above because it nicely takes care of both the design of squares we have to draw and we've solved the color issue utilizing the parity of the row and col variables. Thus, we must solve our problem by mapping each (row, col) pair to different pixel values (x, y). One nice way to solve this problem is to set up three constants:

```
LEN = 50
STARTX = -200
STARTY = -200
```

STARTX and STARTY represent the pixel coordinates of the bottom left corner of the checkerboard and LEN represents the length of the side of a single square of the checkerboard. With these constants, we can set the x and y coordinate of the bottom left coordinates of a single square to draw as follows:

```
x = STARTX + col*LEN
y = STARTY + row*LEN
```

Once we know where the bottom left corner of the single square is, we simply need to run a loop that draws the four sides of the square. This loop runs four times and in the body of the loop the turtle goes forward and then turns left, since we're starting at the bottom left corner:

```
for i in range(4):
    turtle.forward(LEN)
    turtle.left(90)
```

Here is the program in its entirety:

```
import turtle

SIZE = 8
LEN = 50
STARTX = -200
STARTY = -200
def main():

    # Draw fast and make our color red.
    turtle.speed(0)

    # Double loop through each of the 8 x 8 squares.
    for row in range(SIZE):
        for col in range(SIZE):

            # Set the fill color based on the parity of row+col.
            if (row+col)%2 == 0:
                turtle.fillcolor("red")
            else:
                turtle.fillcolor("black")
            turtle.begin_fill()

            # Calculate where to go.
            x = STARTX + col*LEN
            y = STARTY + row*LEN

            # Move the pen there.
            turtle.penup()
            turtle.setpos(x,y)
            turtle.pendown()

            # Draw the square.
            for i in range(4):
                turtle.forward(LEN)
                turtle.left(90)

            # This ends the fill of this square.
            turtle.end_fill()

main()
```