

# CHAPTER 11

## Nested and Variable Loops

We already saw that, within the body of a loop (which is a block of code), we can put other things that have their own blocks. If you look at the number-guessing program from chapter 1, you'll see this:

```
while guess != secret and tries < 6:
    guess = input("What's yer guess? ")
    if guess < secret:
        print "Too low, ye scurvy dog!"
    elif guess > secret:
        print "Too high, landlubber!"
    tries = tries + 1
```

while loop block

if block

elif block

The outer, light gray block is a **while** loop block, and the dark gray blocks are **if** and **elif** blocks within that **while** loop block.

You can also put a loop within another loop. These loops are called *nested loops*.

### Nested loops

Remember the multiplication table program you wrote for the “Try it out” section in chapter 8? Without the user-input part, it might look something like this:

```
multiplier = 5
for i in range(1, 11):
    print i, "x", multiplier, "=", i * multiplier
```

What if you wanted to print three multiplication tables at once? That's the kind of thing a *nested loop* is perfect for. A nested loop is one loop inside another loop. For *each* iteration of the outer loop, the inner loop goes through *all* of its iterations.

To print three multiplication tables, you'd just enclose the original loop (which prints a single multiplication table) in an outer loop (which runs three times). This makes the program print three tables instead of one. The following listing shows what the code looks like.

#### Listing 11.1 Printing three multiplication tables at once

<pre>for multiplier in range(5, 8):     for i in range(1, 11):         print i, "x", multiplier, "=", i * multiplier     print</pre>	<div style="border-left: 1px solid black; padding-left: 5px;">This inner loop prints a single table</div>	<div style="border-left: 1px solid black; padding-left: 5px;">This outer loop runs 3 iterations, with values 5, 6, 7</div>
--	---	--

Notice that we had to indent the inner loop and the `print` statement an extra four spaces from the beginning of the outer `for` loop. This program will print the 5 times, 6 times, and 7 times tables, up to 10 for each table:

```
>>> ===== RESTART =====
>>>
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
7 x 5 = 35
8 x 5 = 40
9 x 5 = 45
10 x 5 = 50

1 x 6 = 6
2 x 6 = 12
3 x 6 = 18
4 x 6 = 24
5 x 6 = 30
6 x 6 = 36
7 x 6 = 42
8 x 6 = 48
9 x 6 = 54
10 x 6 = 60

1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

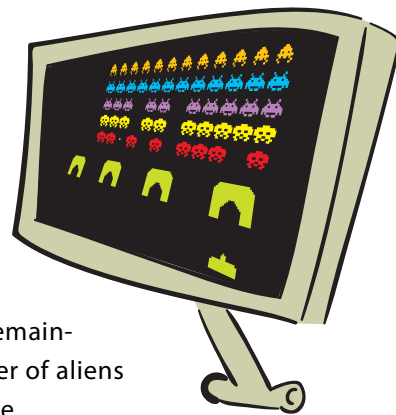
Although you might think it's pretty boring, a good way to see what's going on with nested loops is to just print some stars to the screen and count them. We'll do that in the next section.

## Variable loops

Fixed numbers, like the ones you've used in the `range()` function, are also called *constants*. If you use constants in the `range()` function of a `for` loop, the loop will run the same number of times whenever the program is run. In that case, we say the number of loops is *hard-coded*, because it's defined in your code and never changes. That's not always what you want.

Sometimes you want the number of loops to be determined by the user or by another part of the program. For that, you need a variable.

For example, let's say you were making a space-shooter game. You'd have to keep redrawing the screen as aliens got wiped out. You'd have some sort of counter to keep track of how many aliens were left, and whenever the screen was updated, you'd need to loop through the remaining aliens and draw their images on the screen. The number of aliens would change every time the player wiped out another one.



Because you haven't learned how to draw aliens on the screen yet, here's a simple example program that uses a variable loop:

```
for i in range(1, numStars):
    print '*',
```

```
>>> ===== RESTART =====
>>>
How many stars do you want? 5
* * * *
```

The program asked the user how many stars he wanted, and then it used a variable loop to print that many. Well, almost! We asked for five stars and only got four! Oops, we forgot that the `for` loop stops one short of the second number in the `range`. So we need to add 1 to the user's input:

```
numStars = int(raw_input ("How many stars do you want? "))
for i in range(1, numStars + 1):
    print '*',
```

← Adds 1, so if he asks for 5 stars, he gets 5 stars

Another way to do the same thing is to start the loop counting at 0, instead of 1. (We mentioned that back in chapter 8.) This is very common in programming, and you'll see why in the next chapter. Here's how that would look:

```
numStars = int(raw_input ("How many stars do you want? "))
for i in range(0, numStars):
    print '*',
```

```
>>> ===== RESTART =====
>>>
How many stars do you want? 5
* * * * *
```

## Variable nested loops

Now let's try a *variable nested loop*. That's just a nested loop where one or more of the loops uses a variable in the `range()` function. Here's an example.

### Listing 11.2 A variable nested loop

```
numLines = int(raw_input ('How many lines of stars do you want? '))
numStars = int(raw_input ('How many stars per line? '))
for line in range(0, numLines):
    for star in range(0, numStars):
        print '*',
    print
```

Try running this program to see if it makes sense. You should see something like this:

```
>>> ===== RESTART =====
>>>
How many lines of stars do you want? 3
How many stars per line? 5
*****
*****
*****
```

The first two lines ask the user how many lines she wants and how many stars per line. It remembers the answers using the variables `numLines` and `numStars`. Then we have the two loops:

- The inner loop (`for star in range (0, numStars):`) prints each star and runs once for each star on a line.
- The outer loop (`for line in range (0, numLines):`) runs once for each line of stars.

The second `print` command is needed to start a new line of stars. If we didn't have that, all the stars would print on one line because of the comma in the first `print` statement.

You can even have nested-nested loops (or *double-nested loops*). They look like this.

### Listing 11.3 Blocks of stars with double-nested loops

```
numBlocks = int(raw_input ('How many blocks of stars do you want? '))
numLines = int(raw_input ('How many lines in each block? '))
numStars = int(raw_input ('How many stars per line? '))
for block in range(0, numBlocks):
    for line in range(0, numLines):
        for star in range(0, numStars):
            print '*',
        print
    print
```

Here's the output:

```
>>> ===== RESTART =====
>>>
How many blocks of stars do you want? 3
How many lines of stars in each block? 4
How many stars per line? 8
* * * * *
* * * * *
* * * * *
* * * * *

* * * * *
* * * * *
* * * * *
* * * * *

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

We say the loop is nested “three deep.”

## Even more variable nested loops

The next listing shows a trickier version of the program from listing 11.3.

### Listing 11.4 A trickier version of blocks of stars

```
numBlocks = int(raw_input('How many blocks of stars do you want? '))
for block in range(1, numBlocks + 1):
    for line in range(1, block * 2 ):
        for star in range(1, (block + line) * 2):
            print '*',
        print
    print
```

Formulas for number  
of lines and stars

Here's the output:

```
>>> ===== RESTART =====
>>>
How many blocks of stars do you want? 3
* * *

* * * * *
* * * * *
* * * * *

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

In listing 11.4, the loop variables of the outer loops are used to set the ranges for the inner loops. So instead of each block having the same number of lines and each line having the same number of stars, they're different each time through the loop.

You can nest loops as deep as you want. It can get a bit hairy keeping track of what's going on, so it sometimes helps to print out the values of the loop variables, as shown next.

#### Listing 11.5 Printing the loop variables in nested loops

```
numBlocks = int(raw_input('How many blocks of stars do you want? '))
for block in range(1, numBlocks + 1):
    print 'block = ', block
    for line in range(1, block * 2):
        for star in range(1, (block + line) * 2):
            print '*',
        print ' line = ', line, 'star = ', star
    print
```

← Displays variables

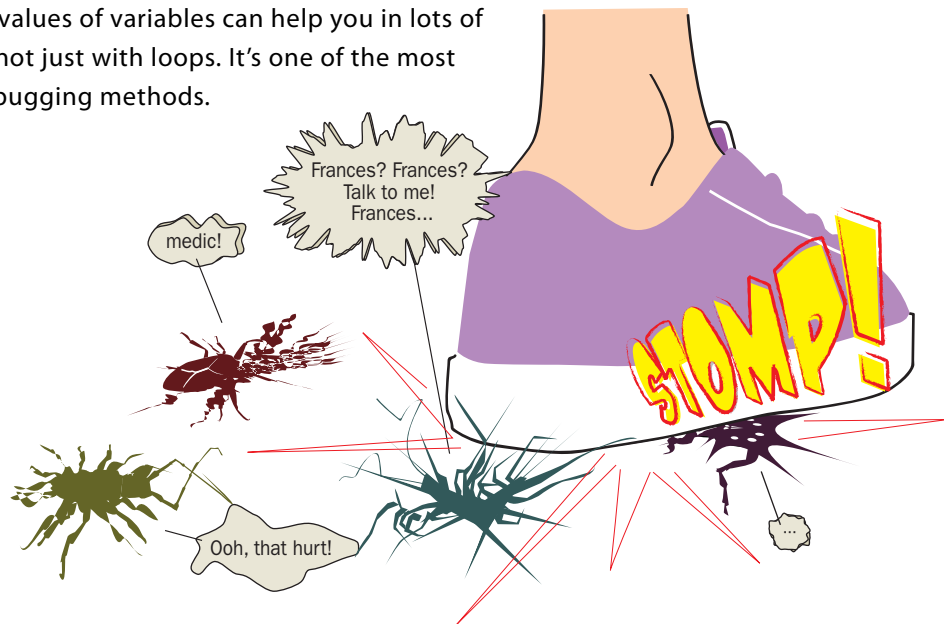
Here's the output of the program:

```
>>> ===== RESTART =====
>>>
How many blocks of stars do you want? 3
block = 1
* * *   line = 1   star = 3

block = 2
* * * * *   line = 1   star = 5
* * * * *   line = 2   star = 7
* * * * *   line = 3   star = 9

block = 3
* * * * * *   line = 1   star = 7
* * * * * *   line = 2   star = 9
* * * * * *   line = 3   star = 11
* * * * * *   line = 4   star = 13
* * * * * *   line = 5   star = 15
```

Printing the values of variables can help you in lots of situations—not just with loops. It’s one of the most common debugging methods.



## Using nested loops

So what can we do with all these nested loops? Well, one of the things they’re good for is figuring out all the possible *permutations* and *combinations* of a series of decisions.

### WORD BOX

*Permutation* is a mathematical term that means a unique way of combining a set of things. *Combination* means something very similar. The difference is that, with a combination, the order doesn’t matter, but with a permutation, the order does matter.

If I asked you to pick three numbers from 1 to 20, you could pick

- 5, 8, 14
- 2, 12, 20

and so on. If we tried to make a list of all the permutations of three numbers from 1 to 20, these two would be separate entries:

- 5, 8, 14
- 8, 5, 14

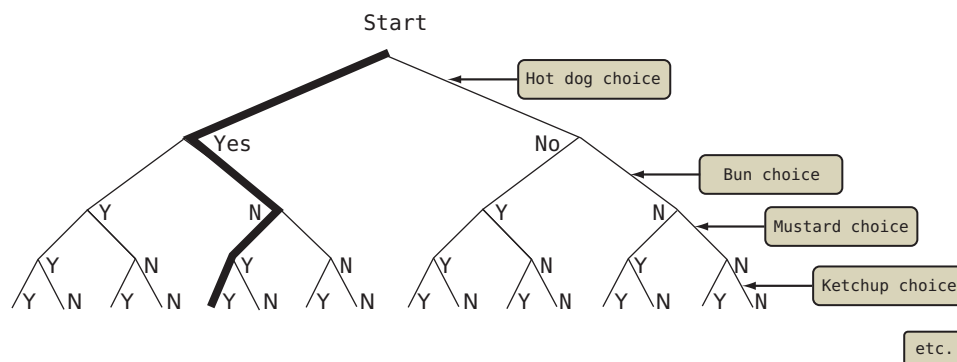
That’s because, with permutations, the order in which they appear matters. If we made a list of all the combinations, all these would count as a single entry:

- 5, 8, 14
- 8, 5, 14
- 8, 14, 5

That’s because order doesn’t matter for combinations.

The best way to explain this is with an example. Let's imagine we're running a hot dog stand at our school's spring fair, and we want to make a poster showing how to order all possible combinations of hot dog, bun, ketchup, mustard, and onions by number. So we need to figure out what all the possible combinations are.

One way to think about this problem is to use something called a *decision tree*. The next figure shows a decision tree for the hot dog problem.



Each decision point has only two choices, Yes or No. Each different path down the tree describes a different combination of hot dog parts. The path I highlighted says "Yes" for hot dog, "No" for bun, "Yes" for mustard, and "Yes" for ketchup.

Now we're going to use nested loops to list all the combinations—all the paths through the decision tree. Because there are five decision points, there are five levels in our decision tree, so there will be five nested loops in our program. (The figure only shows the first four levels of the decision tree.)

Type the code in the following listing into an IDLE editor window, and save it as **hotdog1.py**.

#### Listing 11.6 Hot dog combinations

```
print "\tDog \tBun \tKetchup\tMustard\tOnions"
count = 1

for dog in [0, 1]:
    for bun in [0, 1]:
        for ketchup in [0, 1]:
            for mustard in [0, 1]:
                for onion in [0, 1]:
                    print "#", count, "\t",
                    print dog, "\t", bun, "\t", ketchup, "\t",
                    print mustard, "\t", onion
                    count = count + 1
```

dog loop  
bun loop  
ketchup loop  
mustard loop  
onion loop



See how the loops are all one inside the other? That's what nested loops really are—loops inside other loops:

- The outer (**dog**) loop runs twice.
- The **bun** loop runs twice for each iteration of the **dog** loop. So it runs  $2 \times 2 = 4$  times.
- The **ketchup** loop runs twice for each iteration of the **dog** loop. So it runs  $2 \times 2 \times 2 = 8$  times.

And so on.

The innermost loop (that's the one farthest in—the **onion** loop) runs  $2 \times 2 \times 2 \times 2 \times 2 = 32$  times. This covers all the possible combinations. So there are 32 possible combinations.

If you run the program in listing 11.6, you should get something like this:

```
>>> ===== RESTART =====
>>>
```

	Dog	Bun	Ketchup	Mustard	Onions
# 1	0	0	0	0	0
# 2	0	0	0	0	1
# 3	0	0	0	1	0
# 4	0	0	0	1	1
# 5	0	0	1	0	0
# 6	0	0	1	0	1
# 7	0	0	1	1	0
# 8	0	0	1	1	1
# 9	0	1	0	0	0
# 10	0	1	0	0	1
# 11	0	1	0	1	0
# 12	0	1	0	1	1
# 13	0	1	1	0	0
# 14	0	1	1	0	1
# 15	0	1	1	1	0
# 16	0	1	1	1	1
# 17	1	0	0	0	0
# 18	1	0	0	0	1
# 19	1	0	0	1	0
# 20	1	0	0	1	1
# 21	1	0	1	0	0
# 22	1	0	1	0	1
# 23	1	0	1	1	0
# 24	1	0	1	1	1
# 25	1	1	0	0	0
# 26	1	1	0	0	1
# 27	1	1	0	1	0
# 28	1	1	0	1	1
# 29	1	1	1	0	0
# 30	1	1	1	0	1
# 31	1	1	1	1	0
# 32	1	1	1	1	1

The five nested loops run through all possible combinations of `dog`, `bun`, `ketchup`, `mustard`, and `onion`.

In listing 11.6, we used the `tab` character to line everything up. That's the `\t` parts. We haven't talked about *print formatting* yet, but if you want to know more about it, you can have a peek at chapter 21.

We used a variable called `count` to number each combination. So, for example, a hot dog with a bun and mustard would be #27. Of course, some of the 32 combinations don't make sense. (A hot dog with no bun but with mustard and ketchup would be a little messy.) But you know what they say: "The customer is always right!"



## Counting calories

Because everyone is concerned about nutrition these days, let's add a calorie count for each combination on the menu. (You might not care about the calories, but I bet your parents do!) That will let us use some of Python's math abilities, which we learned about back in chapter 3.

We already know which items are in each combination. All we need now are the calories for each item. Then we can add them all up in the innermost loop.

Here's some code that sets how many calories are in each item:

```
dog_cal = 140
bun_cal = 120
mus_cal = 20
ket_cal = 80
onion_cal = 40
```

Now we just need to add them up. We know there's either 0 or 1 of each item in each menu combination. So we can just multiply the quantity by the calories for every item, like this:

```
tot_cal = (dog * dog_cal) + (bun * bun_cal) + \
          (mustard * mus_cal) + (ketchup * ket_cal) + \
          (onion * onion_cal)
```



Because the order of operations is multiplication first, then addition, I didn't really need to put in the parentheses. I just put them in to make it easier to see what's going on.

**Long lines of code**

Did you notice the backslash (\) characters at the end of the lines in the previous code? If you have a long expression that won't fit on a single line, you can use the backslash character to tell Python, "This line isn't done. Treat whatever is on the next line as if it's part of this line." Here we used two backslashes to split our long line into three short lines. The backslash is called a *line-continuation character*, and several programming languages have them.

You can also put an extra set of parentheses around the whole expression, and then you can split your expression over multiple lines without using the backslash, like this:

```
tot_cal = ((dog * dog_cal) + (bun * bun_cal) +
           (mustard * mus_cal) + (ketchup * ket_cal) +
           (onion * onion_cal))
```

Putting this all together, the new calorie-counter version of the hot dog program is shown next.

**Listing 11.7 Hot dog program with calorie counter**

```
dog_cal = 140
bun_cal = 120
ket_cal = 80
mus_cal = 20
onion_cal = 40

print "\tDog \tBun \tKetchup\tMustard\tOnions\tCalories"
count = 1
for dog in [0, 1]:
    for bun in [0, 1]:
        for ketchup in [0, 1]:
            for mustard in [0, 1]:
                for onion in [0, 1]:
                    total_cal = (bun * bun_cal)+(dog * dog_cal) + \
                                (ketchup * ket_cal)+(mustard * mus_cal) + \
                                (onion * onion_cal)
                    print "#", count, "\t",
                    print dog, "\t", bun, "\t", ketchup, "\t",
                    print mustard, "\t", onion,
                    print "\t", total_cal
                    count = count + 1
```

*Lists calories for each part of the hot dog*

*Prints headings*

*dog is the outer loop*

*Calculates calories in the inner loop*

*Nested loops*

Try running the program in listing 11.7 in IDLE. The output should look like this:

```
>>> ===== RESTART =====
>>>
```

	Dog	Bun	Ketchup	Mustard	Onions	Calories
# 1	0	0	0	0	0	0
# 2	0	0	0	0	1	40
# 3	0	0	0	1	0	20
# 4	0	0	0	1	1	60
# 5	0	0	1	0	0	80
# 6	0	0	1	0	1	120
# 7	0	0	1	1	0	100
# 8	0	0	1	1	1	140
# 9	0	1	0	0	0	120
# 10	0	1	0	0	1	160
# 11	0	1	0	1	0	140
# 12	0	1	0	1	1	180
# 13	0	1	1	0	0	200
# 14	0	1	1	0	1	240
# 15	0	1	1	1	0	220
# 16	0	1	1	1	1	260
# 17	1	0	0	0	0	140
# 18	1	0	0	0	1	180
# 19	1	0	0	1	0	160
# 20	1	0	0	1	1	200
# 21	1	0	1	0	0	220
# 22	1	0	1	0	1	260
# 23	1	0	1	1	0	240
# 24	1	0	1	1	1	280
# 25	1	1	0	0	0	260
# 26	1	1	0	0	1	300
# 27	1	1	0	1	0	280
# 28	1	1	0	1	1	320
# 29	1	1	1	0	0	340
# 30	1	1	1	0	1	380
# 31	1	1	1	1	0	360
# 32	1	1	1	1	1	400

Just imagine how tedious it would be to work out the calories for all these combinations by hand, even if you had a calculator to do the math. It's way more fun to write a program to figure it out for you. Looping and a bit of math in Python make it a snap!

```
python3 11.7.py
```

## What did you learn?

In this chapter, you learned about

- Nested loops
- Variable loops
- Permutations and combinations
- Decision trees

## Test your knowledge

- 1 How do you make a variable loop in Python?
- 2 How do you make a nested loop in Python?
- 3 What's the total number of stars that will be printed by the following code?

```
for i in range(5):
    for j in range(3):
        print '*',
    print
```

- 4 What will the output from the code in question 3 look like?
- 5 If a decision tree has four levels and two choices per level, how many possible choices (paths through the decision tree) are there?

## Try it out

- 1 Remember the countdown-timer program we created in chapter 8? Here it is, to refresh your memory:

```
import time
for i in range (10, 0, -1):
    print i
    time.sleep(1)
print "BLAST OFF!"
```

Modify the program to use a variable loop. The program should ask the user where the countdown should start, like this:

```
Countdown timer:  How many seconds?  4
4
3
2
1
BLAST OFF!
```

- 2 Take the program you wrote in question #1, and have it print a row of stars beside each number, like this:

```
Countdown timer:  How many seconds?  4
4 * * * *
3 * * *
2 * *
1 *
BLAST OFF!
```

(Hint: You probably need to use a nested loop.)