3.3 While-loops

A *loop* is a block of code that a program will execute over and over until some condition is met. Loops are among the most important concepts in all of programming. Most programs spend most of their execution time inside loops. If we didn't have loops the runtime of a program would be proportional to its length; loops allow us to have short programs that perform lengthy computations.

Python has two loop constructs, which we call **while**-loops and **for**-loops. In this section we will introduce **while**-loops and in the next section we present **for**-loops. The remainder of the chapter provides examples of these useful constructs.

A **while**-loop has the structure:

```
while <condition >:
        statement_block
```

When this statement is executed, the statement_block is executed over and over until the condition finally becomes False. If the condition is False when we first get to the loop, the statement_block is not executed at all.

Program 3.3.1 prints the integers from 1 to 3.

```
# This prints the numbers 1 to 3
def main():
    x = 1
    while x <= 3:
        print(x)
        x = x + 1
main()
```

Program 3.3.1: First loop example

We will now step through this program one instruction at a time. You want to think of loops functionally in terms of what they do, but occasionally it is useful to simulate the way the computer deals with the loop.

At the start we set x = 1.
We test x <= 3. This is True, so we enter the loop body.
We print x, which is 1.
We set x = 1+1, which is 2.

We return to the top of the loop and test x <= 3. This is True so we enter the loop body.
We print x, which is now 2.
We set x = 2+1, which is 3.

We return to the top of the loop and test x <= 3. This is True so we enter the loop body.
We print x, which is now 3.
We set x = 3+1, which is 4.

We return to the top of the loop and test x <= 3. This is False, so we leave the loop.

Of course, this is a lot of work to print the numbers 1, 2, and 3, but we can easily change the program to Program 3.3.2, which prints the numbers from 1 to 10,000.

```
# This prints the numbers 1 to 10,000
def main():
    x = 1
    while x <= 10000:
        print(x)
        x = x + 1
main()
```

Program 3.3.2: A longer-running version of program 3.3.1

In both Program 3.3.1 and Program 3.3.2 we give an initial value to variable x before the top of the loop. This is important. Since x is part of the loop condition, it must have a value before the condition can be evaluated. If we left off the assignment statement x=1 we would get an error for an unknown variable x at the start of the loop. In general, every variable mentioned in the condition of a **while**-loop must be given a value before we get to the loop; if we fail to do this the program will crash with an ”unknown variable” error message.

Here is another example. Program 3.3.3 reads a positive integer from the user and then prints all the ways it can be factored into a product of two integers. For example, if you enter 6 it prints

1 * 6 = 6
2 * 3 = 6
3 * 2 = 6
6 * 1 = 6

```
# This reads a positive integer and prints all
# the ways it can be factored.

def main():
    n = eval(input("Enter a positive number: "))
    factor = 1
    while factor <= n:
        if n % factor == 0:
            print("%d * %d = %d"%(factor,n/factor,n))
        factor = factor + 1

main()
```

Program 3.3.3: This factors a number entered by the user.

Variable factor plays the role that x did in the previous programs: it takes on all of the values between 1 and an upper bound, which in this case is n. It is very important in this program that the line that increments variable factor: factor = factor + 1 is not inside the **if**-statement. If it was, then once we got a value of factor that did not divide evenly into n we would never change from this value, and the program would never halt. If this should ever happen to you, there are two ways to stop a runaway program. The <Ctrl>-C combination generates a "keyboard interrupt" that will stop any running program. You can also close the shell window in which the program is running. It is better, however, to avoid this situation by thinking carefully about your programs. Every time through a loop you should make some progress towards eventually terminating.

Input loops

One very common use for a loop is to collect input from a user. The loop reads data and processes it until some condition regarding the input is satisfied. The loop condition in this situation is almost always dependant on computations in in the body of the loop, so it might seem difficult to set up this condition so that it has a value at the start, before the body of the loop has been executed.

Program 3.3.4 shows a way to handle this situation. In this program the user is repeatedly asked to enter numbers. When the user responds with a 0 the loop ends and the average of the user's numbers is printed.

```
def main():
    total = 0.0
    count = 0
    done = False
    while not done:
        x = eval(input("Enter a number, or 0 to exit: "))
        if x ==0:
            done = True
        else:
            total = total + x
            count = count + 1

    print("%d numbers were read for an average of %.2f"
        % (count, total/count))

main()
```

Program 3.3.4: Computing an average of data supplied by the user

This program bases the termination condition for the loop on variable done, which reports whether we are done with the input phase of the program. Before the loop starts we have done no input so we can't be done: this variable is always initialized to False. Each time we read new input we immediately check it against the exit condition and adjust the value of done accordingly.

The following program illustrates an input loop that reads text strings rather than numbers. The idea of the loop is the same as Program 3.3.4.

```
# This is a guessing game based on
# a Brothers Grimm tale
def main():
    done = False
    while not done:
        name = input( "What is my name? " )
        if name.lower() == "rumpelstiltskin":
            done = True
        else:
            print( "Wrong! Guess again." )
    print( "Drat!  You guessed it." )
main()
```

Program 3.3.5: What's my name?

The program repeatedly prompts the user to enter a name. If the name is "rumpelstiltskin" the loop halts and the program prints `"You guessed it."` For any other input there is a `"Guess again"` message. This program accepts any capitalization of the name: "Rumpelstiltskin", "rumpelstiltskin" or even "RumpelStiltSkin". It does this by comparing name.lower(), which is the translates all of the letters to lower0-case, to the lower-case version of the correct answer: "rumpelstiltskin". Note how easy this is; with very little work you can make any input be case-insinsitive.

For a final example, we will nest one loop inside another. This is very common in programming, and also a common source of errors. To do this correctly, we need to be in control of our code and know what each element of the program is doing.

The problem we will address with the program is this: *Write a program that inputs numbers and prints their factorizations into prime numbers. The program should end when the user inputs the number 0.* For example, for input 84 the program should print `2 2 3 7`.

We will write this program in stages. The first stage just handles the user input getting numbers from the user until we get the number 0. This is similar to programs 3.3.5 and 3.3.6.

```
# This gets numbers from the user and
# prints their prime factorizations

def main():
    done = False
    while not done:
        n = eval( input("Enter a number, or 0 to exit: ") )
        if n == 0:
            done = True
        else:
            print( "factorization of %d goes here" % n )

main()
```

The user-input loop

Note the way we left a marker for where the next portion of the code goes the " factorization goes here" line will be replaced by a new block of code. Note also that, although this doesn't completely solve the problem we started with, it is a complete, working program. We can run it to test out our code so far. This is important. As you start to develop longer and longer programs, you should try to keep your programs in a working state as you generate more code. This allows you to test out your code is small portions that are easy to understand. If we run this much of our program we get the following interaction:

```
Enter a number, or 0 to exit: 2
factorization goes here
Enter a number, or 0 to exit: 5
factorization goes here
Enter a number, or 0 to exit: 0
>>>
```

Our next step is to replace the " factorization goes here" line with code that actually computes the prime factorization. We don't need to actually generate prime numbers for this. If we start looking for factors at 2 and replace n by its quotient with each successful factor, the only factors we will find will be prime numbers. For example, suppose we start with n = 90. 2 is a factor, so we write down 2 and reduce n to 45. 2 is no longer a factor of this, so we try the next number, 3, as a factor. 3 divides evenly into 45, so we write down 3 and reduce n to 15. 3 is still a factor of this, so we write down 3 again and reduce n to 5. 3 is no longer a factor, so we try 4, which is also not a factor. We eventually test 5 as a factor; this is successful so we write down 5 and reduce n to 1. We can now stop, since there are no more possible factors. Here is the code that implements this:

```
factor = 2
while factor <= n:
        if n % factor == 0:
                print( factor, end=" ")
                n = n/factor
        else:
                factor = factor + 1
print
```

Finding factors

Note that we print each successful factor with the statement

```
print( factor, end=" " )
```

so we don't actually terminate the output line. At the end of the factorization we have one **print** statement, which completes the line of output.

Here is our current version of the program.

```
# This gets numbers from the user and
# prints their prime factorizations

def main():
    done = False
    while not done:
        n = eval(input( "Enter a number, or 0 to exit: "))
        if n == 0:
            done = True
        else:
            factor = 2
            while factor <= n:
                if n % factor == 0:
                    print( factor, end= " ")
                    n = n/factor
                else:
                    factor = factor + 1
            print
main()
```

User-input plus factoring

If we run this we get the following interactions:

```
Enter a number, or 0 to exit: 34
2 17
Enter a number, or 0 to exit: 160
2 2 2 2 2 5
Enter a number, or 0 to exit: 3344
2 2 2 2 11 19
Enter a number, or 0 to exit: 0
>>>
```

This works, but it seems awkward to have the same factor printed over and over again, as in the factorization of 160, which is 2x2x2x2x2x5. With a little more thought we can have a third loop count how many times each factor divides into n. If the count is 1 we just print the factor; if it is more than 1 we print the factor and the count, with the format factor\^count. Here is that code:

```
factor = 2
while factor <= n:
    if n % factor == 0:
        count = 0
        while n%factor == 0:
            count = count + 1
            n = n/factor
            if count == 1:
                print( factor , end=" ")
            else:
                print( "%d^%d"%(factor ,count), end=" " )
            else:
                factor = factor + 1
print
```

Counting the factors

and here is the final version of the program:

```
# This gets numbers from the user and
# prints their prime factorizations

def main():
    done = False
    while not done:
        n = eval(input "Enter a number, or 0 to exit: "))
        if n == 0:
            done = True
        else:
            factor = 2
            while factor <= n:
                if n % factor == 0:
                    count = 0
                    while n%factor == 0:
                        count = count + 1
                        n = n/factor
                    if count == 1:
                        print(factor, end=" ")
                    else:
                        print("%d^%d"%(factor, count), end=" ")
                else:
                    factor = factor + 1
            print( )

main()
```

Program 3.3.6: The final version

This time we get the following interactions:

```
Enter a number, or 0 to exit: 34
2 17
Enter a number, or 0 to exit: 84
2^2 3 7
Enter a number, or 0 to exit: 3344
2^4 11 19
Enter a number, or 0 to exit: 0
>>>
```