Chapter 9

While loops

We have already learned about for loops, which allow us to repeat things a specified number of times. Sometimes, though, we need to repeat something, but we don't know ahead of time exactly how many times it has to be repeated. For instance, a game of Tic-tac-toe keeps going until someone wins or there are no more moves to be made, so the number of turns will vary from game to game. This is a situation that would call for a while loop.

9.1 Examples

Example 1 Let's go back to the first program we wrote back in Section 1.3, the temperature converter. One annoying thing about it is that the user has to restart the program for every new temperature. A while loop will allow the user to repeatedly enter temperatures. A simple way for the user to indicate that they are done is to have them enter a nonsense temperature like −1000 (which is below absolute 0). This is done below:

```
temp = 0
while temp!=-1000:
    temp = eval(input('Enter a temperature (-1000 to quit): '))
    print('In Fahrenheit that is', 9/5*temp+32)
```

Look at the `while` statement first. It says that we will keep looping, that is, keep getting and converting temperatures, as long as the temperature entered is not −1000. As soon as −1000 is entered, the while loop stops. Tracing through, the program first compares `temp` to −1000. If `temp` is not −1000, then the program asks for a temperature and converts it. The program then loops back up and again compares `temp` to −1000. If `temp` is not −1000, the program will ask for another temperature, convert it, and then loop back up again and do another comparison. It continues this process until the user enters −1000.

We need the line `temp=0` at the start, as without it, we would get a name error. The program would

get to the `while` statement, try to see if `temp` is not equal to −1000 and run into a problem because `temp` doesn't yet exist. To take care of this, we just declare `temp` equal to 0. There is nothing special about the value 0 here. We could set it to anything except −1000. (Setting it to −1000 would cause the condition on the while loop to be false right from the start and the loop would never run.)

Note that is natural to think of the while loop as continuing looping until the user enters -1000. However, when we construct the condition, instead of thinking about when to stop looping, we instead need to think in terms of what has to be true in order to keep going.

A while loop is a lot like an if statement. The difference is that the indented statements in an if block will only be executed once, whereas the indented statements in a while loop are repeatedly executed.

Example 2 One problem with the previous program is that when the user enters in −1000 to quit, the program still converts the value −1000 and doesn't give any message to indicate that the program has ended. A nicer way to do the program is shown below.

```
temp = 0
while temp!=-1000:
    temp = eval(input('Enter a temperature (-1000 to quit): '))
    if temp!=-1000:
        print('In Fahrenheit that is', 9/5*temp+32)
    else:
        print('Bye!')
```

Example 3 When first met if statements in Section 4.1, we wrote a program that played a simple random number guessing game. The problem with that program is that the player only gets one guess. We can, in a sense, replace the if statement in that program with a while loop to create a program that allows the user to keep guessing until they get it right.

```
from random import randint
secret_num = randint(1,10)
guess = 0
while guess != secret_num:
    guess = eval(input('Guess the secret number: '))
print('You finally got it!')
```

The condition `guess!=secret_num` says that as long as the current guess is not correct, we will keep looping. In this case, the loop consists of one statement, the input statement, and so the program will keep asking the user for a guess until their guess is correct. We require the line `guess=0` prior to the while loop so that the first time the program reaches the loop, there is something in `guess` for the program to use in the comparison. The exact value of `guess` doesn't really matter at this point. We just want something that is guaranteed to be different than `secret_num`. When the user finally guesses the right answer, the loop ends and program control moves to the **`print`** statement after the loop, which prints a congratulatory message to the player.

Example 4 We can use a while loop to mimic a for loop, as shown below. Both loops have the exact same effect.

```
for i in range(10):                    i=0
    print(i)                           while i<10:
                                           print(i)
                                           i=i+1
```

Remember that the for loop starts with the loop variable `i` equal to 0 and ends with it equal to 9. To use a while loop to mimic the for loop, we have to manually create our own loop variable `i`. We start by setting it to 0. In the while loop we have the same **`print`** statement as in the for loop, but we have another statement, `i=i+1`, to manually increase the loop variable, something that the for loop does automatically.

Example 5 Below is our old friend that converts from Fahrenheit to Celsius.

```
temp = eval(input('Enter a temperature in Celsius: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

A program that gets input from a user may want to check to see that the user has entered valid data. The smallest possible temperature is absolute zero, -273.15 °C. The program below takes absolute zero into account:

```
temp = eval(input('Enter a temperature in Celsius: '))
if temp<-273.15:
    print('That temperature is not possible.'))
else:
    print('In Fahrenheit, that is', 9/5*temp+32)
```

One way to improve this is to allow the user to keep reentering the temperature until they enter a valid one. You may have experienced something similar using an online form to enter a phone number or a credit card number. If you enter an invalid number, you are told to reenter it. In the code below, the while loop acts very similarly to the if statement in the previous example.

```
temp = eval(input('Enter a temperature in Celsius: '))
while temp<-273.15:
    temp = eval(input('Impossible.  Enter a valid temperature: '))
print('In Fahrenheit, that is', 9/5*temp+32)
```

Note that we do not need an **`else`** statement here, like we had with the if statement.. The condition on the while loop guarantees that we will only get to the **`print`** statement once the user enters a valid temperature. Until that point, the program will be stuck in the loop, continually asking the user for a new temperature.

Example 6 As mentioned before, it is a valuable skill is to be able to read code. One way to do so is to pretend to be the Python interpreter and go through the code line by line. Let's try it with the

code below.

```
i = 0
while i<50:
    print(i)
    i=i+2
print('Bye!')
```

The variable `i` gets set to 0 to start. Next, the program tests the condition on the while loop. Because `i` is 0, which is less than 50, the code indented under the `while` statement will get executed. This code prints the current value of `i` and then executes the statement `i=i+2` which adds 2 to `i`.

The variable `i` is now 2 and the program loops back to the `while` statement. It checks to see if `i` is less than 50, and since `i` is 2, which is less than 50, the indented code should be executed again. So we print `i` again, add 2 to it, and then loop back to check the while loop condition again. We keep doing this until finally `i` gets to 50.

At this point, the `while` condition will finally not be true anymore and the program jumps down to the first statement after the `while`, which prints `Bye!`. The end result of the program is the numbers 0, 2, 4, . . . , 48 followed by the message, `Bye!`.

9.2 Infinite loops

When working with while loops, sooner or later you will accidentally send Python into a never-ending loop. Here is an example:

```
i=0
while i<10:
    print(i)
```

In this program, the value of `i` never changes and so the condition `i<10` is always true. Python will continuously print zeroes. To stop a program caught in a never-ending loop, use `Restart Shell` under the `Shell` menu. You can use this to stop a Python program before it is finished executing.

Sometimes a never-ending loop is what you want. A simple way to create one is shown below:

```
while True:
    # statements to be repeated go here
```

The value **`True`** is called a boolean value and is discussed further in Section 10.2.

9.3 The `break` statement

The `break` statement can be used to break out of a for or while loop before the loop is finished.

Example 1 Here is a program that allows the user to enter up to 10 numbers. The user can stop early by entering a negative number.

```
for i in range(10):
    num = eval(input('Enter number: '))
    if num<0:
        break
```

This could also be accomplished with a while loop.

```
i=0
num=1
while i<10 and num>0:
    num = eval(input('Enter a number: '))
```

Either method is ok. In many cases the `break` statement can help make your code easier to understand and less clumsy.

Example 2 Earlier in the chapter, we used a while loop to allow the user to repeatedly enter temperatures to be converted. Here is, more or less, the original version on the left compared with a different approach using the `break` statement.

```
temp = 0
while temp!=-1000:
    temp = eval(input(': '))
    if temp!=-1000:
        print(9/5*temp+32)
    else:
        print('Bye!')
```

```
while True:
    temp = eval(input(': '))
    if temp==-1000:
        print('Bye')
        break
    print(9/5*temp+32)
```

9.4 The else statement

There is an optional `else` that you can use with `break` statements. The code indented under the `else` gets executed only if the loop completes without a `break` happening.

Example 1 This is a simple example based off of Example 1 of the previous section.

```
for i in range(10):
    num = eval(input('Enter number: '))
    if num<0:
        print('Stopped early')
        break
else:
    print('User entered all ten values')
```

The program allows the user to enter up to 10 numbers. If they enter a negative, then the program prints `Stopped early` and asks for no more numbers. If the user enters no negatives, then the program prints `User entered all ten values`.

Example 2 Here are two ways to check if an integer `num` is prime. A prime number is a number whose only divisors are 1 and itself. The approach on the left uses a while loop, while the approach on the right uses a for/break loop:

```
i=2
while i<num and num%i!=0:
    i=i+1
if i==num:
    print('Prime')
else:
    print('Not prime')
```

```
for i in range(2, num):
    if num%i==0:
        print('Not prime')
        break
else:
    print('Prime')
```

The idea behind both approaches is to scan through all the integers between 2 and `num`-1, and if any of them is a divisor, then we know `num` is not prime. To see if a value `i` is a divisor of `num`, we just have to check to see if `num%i` is 0.

The idea of the while loop version is we continue looping as long as we haven't found a divisor. If we get all the way through the loop without finding a divisor, then `i` will equal `num`, and in that case the number must be prime.

The idea of the for/break version is we loop through all the potential divisors, and as soon as we find one, we know the number is not prime and we print `Not prime` and stop looping. If we get all the way through the loop without breaking, then we have not found a divisor. In that case the `else` block will execute and print that the number is prime.

9.5 The guessing game, more nicely done

It is worth going through step-by-step how to develop a program. We will modify the guessing game program from Section 9.1 to do the following:

- The player only gets five turns.
- The program tells the player after each guess if the number is higher or lower.
- The program prints appropriate messages for when the player wins and loses.

Below is what we want the program to look like:

```
Enter your guess (1-100): 50
LOWER. 4 guesses left.

Enter your guess (1-100): 25
```

```
LOWER. 3 guesses left.

Enter your guess (1-100): 12
LOWER. 2 guesses left.

Enter your guess (1-100): 6
HIGHER. 1 guesses left.

Enter your guess (1-100): 9
LOWER. 0 guesses left.

You lose.  The correct number is 8
```

First, think about what we will need in the program:

- We need random numbers, so there will be an import statement at the beginning of the program and a `randint` function somewhere else.
- To allow the user to guess until they either guess right or run out of turns, one solution is to use while loop with a condition that takes care of both of these possibilities.
- There will be an input statement to get the user's guess. As this is something that is repeatedly done, it will go inside the loop.
- There will be an if statement to take care of the higher/lower thing. As this comparison will be done repeatedly and will depend on the user's guesses, it will go in the loop after the input statement.
- There will be a counting variable to keep track of how many turns the player has taken. Each time the user makes a guess, the count will go up by one, so this statement will also go inside the loop.

Next start coding those things that are easy to do:

```
from random import randint

secret_num = randint(1,100)
num_guesses = 0

while #some condition goes here#
    guess = eval(input('Enter your guess (1-100): '))
    num_guesses = num_guesses + 1
    # higher/lower if statement goes here
```

For the while loop, we want to continue looping as long as the user has not guessed the secret number and as long as the player has not used up all of their guesses:

```
while guess != secret_num and num_guesses <= 4:
```

The higher/lower if statement can be done like this:

```
    if guess < secret_num:
        print('HIGHER.', 5-num_guesses, 'guesses left.\n')
    elif guess > secret_num:
        print('LOWER.', 5-num_guesses, 'guesses left.\n')
    else:
        print('You got it!')
```

Finally, it would be nice to have a message for the player if they run out of turns. When they run out of turns, the while loop will stop looping and program control will shift to whatever comes outside of the loop. At this point we can print the message, but we only want to do so if the reason that the loop stopped is because of the player running out of turns and not because they guessed correctly. We can accomplish this with an if statement after the loop. This is shown below along with the rest of the completed program.

```
from random import randint

secret_num = randint(1,100)
num_guesses = 0
guess = 0

while guess != secret_num and num_guesses <= 4:
    guess = eval(input('Enter your guess (1-100): '))
    num_guesses = num_guesses + 1
    if guess < secret_num:
        print('HIGHER.', 5-num_guesses, 'guesses left.\n')
    elif guess > secret_num:
        print('LOWER.', 5-num_guesses, 'guesses left.\n')
    else:
        print('You got it!')

if num_guesses==5 and guess != secret_num:
    print('You lose.  The correct number is', secret_num)
```

Here is an alternative solution using a for/break loop:

```
from random import randint

secret_num = randint(1,100)

for num_guesses in range(5):
    guess = eval(input('Enter your guess (1-100): '))
    if guess < secret_num:
        print('HIGHER.', 5-num_guesses, 'guesses left.\n')
    elif guess > secret_num:
        print('LOWER.', 5-num_guesses, 'guesses left.\n')
    else:
        print('You got it!')
        break
else:
    print('You lose.  The correct number is', secret_num)
```

9.6 Exercises

1. The code below prints the numbers from 1 to 50. Rewrite the code using a while loop to accomplish the same thing.

```
for i in range(1,51):
    print(i)
```

2. (a) Write a program that uses a while loop (not a for loop) to read through a string and print the characters of the string one-by-one on separate lines.

   (b) Modify the program above to print out every second character of the string.

3. A good program will make sure that the data its users enter is valid. Write a program that asks the user for a weight and converts it from kilograms to pounds. Whenever the user enters a weight below 0, the program should tell them that their entry is invalid and then ask them again to enter a weight. [Hint: Use a while loop, not an if statement].

4. Write a program that asks the user to enter a password. If the user enters the right password, the program should tell them they are logged in to the system. Otherwise, the program should ask them to reenter the password. The user should only get five tries to enter the password, after which point the program should tell them that they are kicked off of the system.

5. Write a program that allows the user to enter any number of test scores. The user indicates they are done by entering in a negative number. Print how many of the scores are A's (90 or above). Also print out the average.

6. Modify the higher/lower program so that when there is only one guess left, it says `1 guess`, not `1 guesses`.

7. Recall that, given a string `s`, `s.index('x')` returns the index of the first x in `s` and an error if there is no x.

   (a) Write a program that asks the user for a string and a letter. Using a while loop, the program should print the index of the first occurrence of that letter and a message if the string does not contain the letter.

   (b) Write the above program using a for/break loop instead of a while loop.

8. The GCD (greatest common divisor) of two numbers is the largest number that both are divisible by. For instance, gcd(18, 42) is 6 because the largest number that both 18 and 42 are divisible by is 6. Write a program that asks the user for two numbers and computes their gcd. Shown below is a way to compute the GCD, called Euclid's Algorithm.

   - First compute the remainder of dividing the larger number by the smaller number
   - Next, replace the larger number with the smaller number and the smaller number with the remainder.
   - Repeat this process until the smaller number is 0. The GCD is the last value of the larger number.

Chapter 13

Functions

Functions are useful for breaking up a large program to make it easier to read and maintain. They are also useful if you find yourself writing the same code at several different points in your program. You can put that code in a function and call the function whenever you want to execute that code. You can also use functions to create your own utilities, math functions, etc.

13.1 Basics

Functions are defined with the `def` statement. The statement ends with a colon, and the code that is part of the function is indented below the `def` statement. Here we create a simple function that just prints something.

```
def print_hello():
    print('Hello!')

print_hello()
print('1234567')
print_hello()
```

```
Hello!
1234567
Hello!
```

The first two lines define the function. In the last three lines we call the function twice.

One use for functions is if you are using the same code over and over again in various parts of your program, you can make your program shorter and easier to understand by putting the code in a function. For instance, suppose for some reason you need to print a box of stars like the one below at several points in your program.

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

Put the code into a function, and then whenever you need a box, just call the function rather than typing several lines of redundant code. Here is the function.

```
def draw_square():
    print('*' * 15)
    print('*', ' '*11, '*')
    print('*', ' '*11, '*')
    print('*' * 15)
```

One benefit of this is that if you decide to change the size of the box, you just have to modify the code in the function, whereas if you had copied and pasted the box-drawing code everywhere you needed it, you would have to change all of them.

13.2 Arguments

We can pass values to functions. Here is an example:

```
def print_hello(n):
    print('Hello ' * n)
    print()

print_hello(3)
print_hello(5)
times = 2
print_hello(times)
```

```
Hello Hello Hello
Hello Hello Hello Hello Hello
Hello Hello
```

When we call the `print_hello` function with the value 3, that value gets stored in the variable `n`. We can then refer to that variable `n` in our function's code.

You can pass more than one value to a function:

```
def multiple_print(string, n):
    print(string * n)
    print()

multiple_print('Hello', 5)
multiple_print('A', 10)
```

```
HelloHelloHelloHelloHello
AAAAAAAAAA
```

13.3 Returning values

We can write functions that perform calculations and return a result.

Example 1 Here is a simple function that converts temperatures from Celsius to Fahrenheit.

```
def convert(t):
    return t*9/5+32

print(convert(20))
```

```
68
```

The `return` statement is used to send the result of a function's calculations back to the caller.

Notice that the function itself does not do any printing. The printing is done outside of the function. That way, we can do math with the result, like below.

```
print(convert(20)+5)
```

If we had just printed the result in the function instead of returning it, the result would have been printed to the screen and forgotten about, and we would never be able to do anything with it.

Example 2 As another example, the Python `math` module contains trig functions, but they only work in radians. Let us write our own sine function that works in degrees.

```
from math import pi, sin

def deg_sin(x):
    return sin(pi*x/180)
```

Example 3 A function can return multiple values as a list.

Say we want to write a function that solves the system of equations $ax + by = e$ and $cx + dy = f$. It turns out that if there is a unique solution, then it is given by $x = (de - bf)/(ad - bc)$ and $y = (af - ce)/(ad - bc)$. We need our function to return both the x and y solutions.

```
def solve(a,b,c,d,e,f):
    x = (d*e-b*f)/(a*d-b*c)
    y = (a*f-c*e)/(a*d-b*c)
    return [x,y]
```

```
xsol, ysol = solve(2,3,4,1,2,5)
print('The solution is x = ', xsol, 'and y = ', ysol)
```

```
The solution is x =  1.3 and y =  -0.2
```

This method uses the shortcut for assigning to lists that was mentioned in Section 10.3.

Example 4 A `return` statement by itself can be used to end a function early.

```
def multiple_print(string, n, bad_words):
    if string in bad_words:
        return
    print(string * n)
    print()
```

The same effect can be achieved with an if/else statement, but in some cases, using `return` can make your code simpler and more readable.

13.4 Default arguments and keyword arguments

You can specify a default value for an argument. This makes it optional, and if the caller decides not to use it, then it takes the default value. Here is an example:

```
def multiple_print(string, n=1)
    print(string * n)
    print()

multiple_print('Hello', 5)
multiple_print('Hello')
```

```
HelloHelloHelloHelloHello
Hello
```

Default arguments need to come at the end of the function definition, after all of the non-default arguments.

Keyword arguments A related concept to default arguments is *keyword arguments*. Say we have the following function definition:

```
def fancy_print(text, color, background, style, justify):
```

Every time you call this function, you have to remember the correct order of the arguments. Fortunately, Python allows you to name the arguments when calling the function, as shown below:

```
fancy_print(text='Hi', color='yellow', background='black',
            style='bold', justify='left')
```

```
fancy_print(text='Hi', style='bold', justify='left',
            background='black', color='yellow')
```

As we can see, the order of the arguments does not matter when you use keyword arguments.

When defining the function, it would be a good idea to give defaults. For instance, most of the time, the caller would want left justification, a white background, etc. Using these values as defaults means the caller does not have to specify every single argument every time they call the function. Here is a example:

```
def fancy_print(text, color='black', background='white',
                style='normal', justify='left'):
    # function code goes here

fancy_print('Hi', style='bold')
fancy_print('Hi', color='yellow', background='black')
fancy_print('Hi')
```

Note We have actually seen default and keyword arguments before—the `sep`, `end` and `file` arguments of the **`print`** function.

13.5 Local variables

Let's say we have two functions like the ones below that each use a variable `i`:

```
def func1():
    for i in range(10):
        print(i)

def func2():
    i=100
    func1()
    print(i)
```

A problem that could arise here is that when we call `func1`, we might mess up the value of `i` in `func2`. In a large program it would be a nightmare trying to make sure that we don't repeat variable names in different functions, and, fortunately, we don't have to worry about this. When a variable is defined inside a function, it is *local* to that function, which means it essentially does not exist outside that function. This way each function can define its own variables and not have to worry about if those variable names are used in other functions.

Global variables On the other hand, sometimes you actually do want the same variable to be available to multiple functions. Such a variable is called a *global* variable. You have to be careful using global variables, especially in larger programs, but a few global variables used judiciously are fine in smaller programs. Here is a short example:

```
def reset():
    global time_left
    time_left = 0

def print_time():
    print(time_left)

time_left=30
```

In this program we have a variable `time_left` that we would like multiple functions to have access to. If a function wants to change the value of that variable, we need to tell the function that `time_left` is a global variable. We use a `global` statement in the function to do this. On the other hand, if we just want to use the value of the global variable, we do not need a `global` statement.

Arguments We finish the chapter with a bit of a technical detail. You can skip this section for the time being if you don't want to worry about details right now. Here are two simple functions:

```
def func1(x):
    x = x + 1

def func2(L):
    L = L + [1]

a=3
M=[1,2,3]
func1(a)
func2(M)
```

When we call `func1` with `a` and `func2` with `L`, a question arises: do the functions change the values of `a` and `L`? The answer may surprise you. The value of `a` is unchanged, but the value of `L` is changed. The reason has to do with a difference in the way that Python handles numbers and lists. Lists are said to be *mutable* objects, meaning they can be changed, whereas numbers and strings are *immutable*, meaning they cannot be changed. There is more on this in Section 19.1.

If we want to reverse the behavior of the above example so that `a` is modified and `L` is not, do the following:

```
def func1(x):
    x = x + 1
    return x

def func2(L):
    copy = L[:]
    copy = copy + [1]

a=3
M=[1,2,3]
a=func1(a)  # note change on this line
```

```
func2(M)
```

13.6 Exercises

1. Write a function called `rectangle` that takes two integers `m` and `n` as arguments and prints out an $m \times n$ box consisting of asterisks. Shown below is the output of `rectangle(2,4)`

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

2. (a) Write a function called `add_excitement` that takes a list of strings and adds an exclamation point (`!`) to the end of each string in the list. The program should modify the original list and not return anything.

   (b) Write the same function except that it should not modify the original list and should instead return a new list.

3. Write a function called `sum_digits` that is given an integer `num` and returns the sum of the digits of `num`.

4. The *digital root* of a number n is obtained as follows: Add up the digits n to get a new number. Add up the digits of that to get another new number. Keep doing this until you get a number that has only one digit. That number is the digital root.

   For example, if $n = 45893$, we add up the digits to get $4+5+8+9+3 = 29$. We then add up the digits of 29 to get $2+9 = 11$. We then add up the digits of 11 to get $1+1 = 2$. Since 2 has only one digit, 2 is our digital root.

   Write a function that returns the digital root of an integer n. [Note: there is a shortcut, where the digital root is equal to n mod 9, but do not use that here.]

5. Write a function called `first_diff` that is given two strings and returns the first location in which the strings differ. If the strings are identical, it should return -1.

6. Write a function called `binom` that takes two integers n and k and returns the binomial coefficient $\binom{n}{k}$. The definition is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

7. Write a function that takes an integer n and returns a random integer with exactly n digits. For instance, if n is 3, then 125 and 593 would be valid return values, but 093 would not because that is really 93, which is a two-digit number.

8. Write a function called `number_of_factors` that takes an integer and returns how many factors the number has.

9. Write a function called `factors` that takes an integer and returns a list of its factors.

10. Write a function called `closest` that takes a list of numbers `L` and a number `n` and returns the largest element in `L` that is not larger than `n`. For instance, if `L=[1,6,3,9,11]` and `n=8`, then the function should return 6, because 6 is the closest thing in `L` to 8 that is not larger than 8. Don't worry about if all of the things in `L` are smaller than `n`.