

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Repeated execution of a sequence of statements is called **iteration**. Because iteration is so common, Python provides several language features to make it easier. We've already seen the `for` statement in a previous chapter. This is a very common form of iteration in Python. In this chapter we are going to look at the `while` statement — another way to have your program do iteration.

## The `for` loop revisited

Recall that the `for` loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. We saw this example in an earlier chapter.

```
1 for f in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:  
2     print("Hi", f, "Please come to my party on Saturday")  
3
```

**ActiveCode: 1** (ch07\_for1)

Run

We have also seen iteration paired with the update idea to form the accumulator pattern. For example, to compute the sum of the first `n` integers, we could create a `for` loop using the `range` to produce the numbers 1 thru `n`. Using the accumulator pattern, we can start with a running total variable initialized to 0 and on each iteration, add the current value of the loop variable. A function to compute this sum is shown below.

```
1 def sumTo(aBound):  
2     theSum = 0  
3     for aNumber in range(1, aBound+1):  
4         theSum = theSum + aNumber  
5  
6     return theSum  
7  
8 print(sumTo(4))  
9  
10 print(sumTo(1000))  
11
```

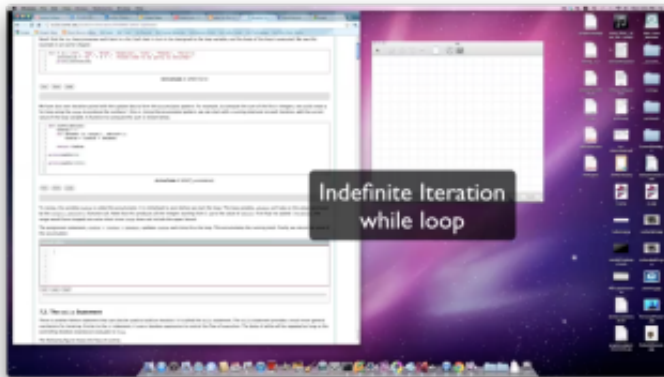
**ActiveCode: 2** (ch07\_summation)

Run

To review, the variable `theSum` is called the accumulator. It is initialized to zero before we start the loop. The loop variable, `aNumber` will take on the values produced by the `range(1,aBound+1)` function call. Note that this produces all the integers from 1 up to the value of `aBound`. If we had not added 1 to `aBound`, the range would have stopped one value short since `range` does not include the upper bound.

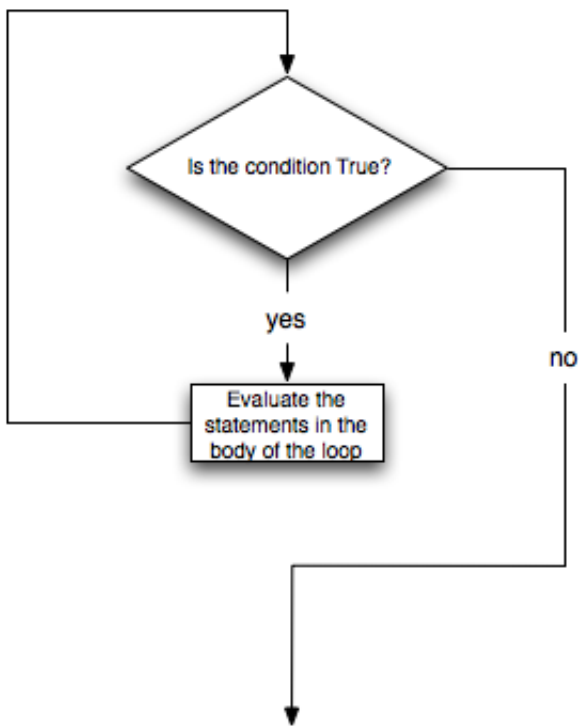
The assignment statement, `theSum = theSum + aNumber`, updates `theSum` each time thru the loop. This accumulates the running total. Finally, we return the value of the accumulator.

## The while Statement



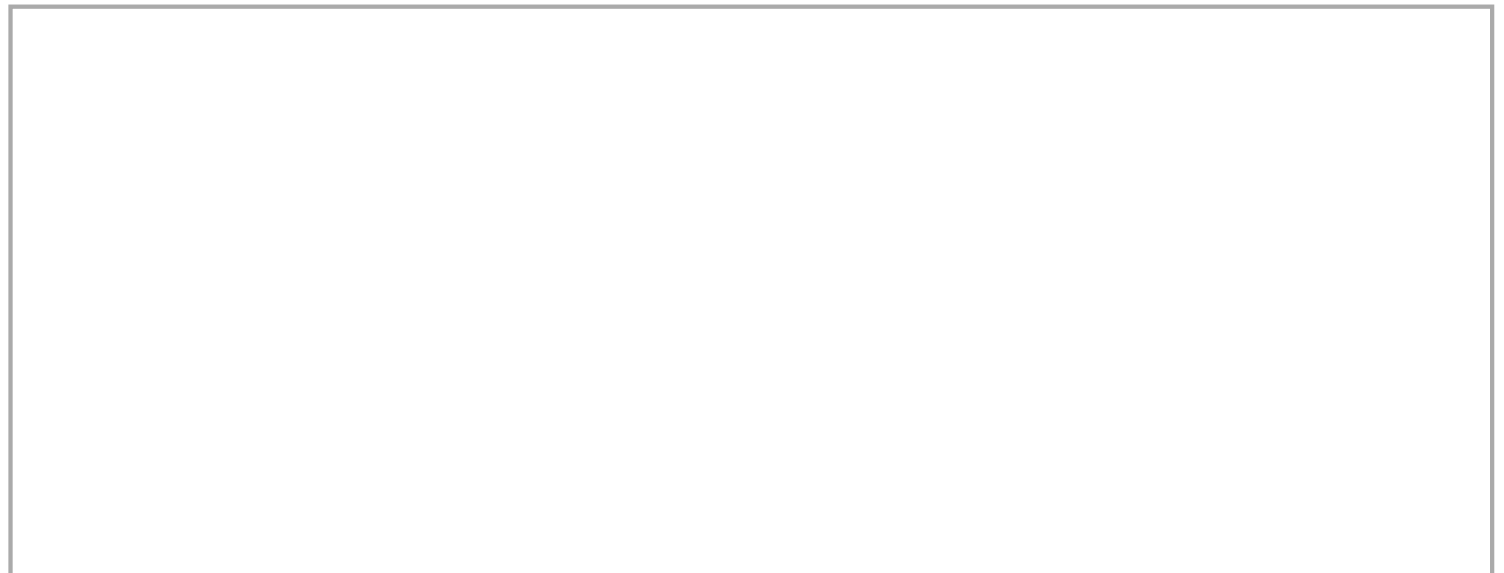
There is another Python statement that can also be used to build an iteration. It is called the `while` statement. The `while` statement provides a much more general mechanism for iterating. Similar to the `if` statement, it uses a boolean expression to control the flow of execution. The body of `while` will be repeated as long as the controlling boolean expression evaluates to `True`.

The following figure shows the flow of control.



We can use the `while` loop to create any type of iteration we wish, including anything that we have previously done with a `for` loop. For example, the program in the previous section could be rewritten using `while`. Instead of relying on the `range` function to produce the numbers for our summation, we will need to produce them ourselves. To do this, we will create a variable called `aNumber` and initialize it to 1, the first number in the summation. Every iteration will add `aNumber` to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to `True` as long as we want to keep adding values to our running total. In this case, as long as `aNumber` is less than or equal to the bound, we should keep going.

Here is a new version of the summation program that uses a `while` statement.



```
1 def sumTo(aBound):
2     """ Return the sum of 1+2+3 ... n """
3
4     theSum = 0
5     aNumber = 1
6     while aNumber <= aBound:
7         theSum = theSum + aNumber
8         aNumber = aNumber + 1
9     return theSum
10
11 print(sumTo(4))
12
13 print(sumTo(1000))
14
```

**ActiveCode: 3** (ch07\_while1)

Run

You can almost read the `while` statement as if it were in natural language. It means, while `aNumber` is less than or equal to `aBound`, continue executing the body of the loop. Within the body, each time, update `theSum` using the accumulator pattern and increment `aNumber`. After the body of the loop, we go back up to the condition of the `while` and reevaluate it. When `aNumber` becomes greater than `aBound`, the condition fails and flow of control continues to the `return` statement.

The same program in codelens will allow you to observe the flow of execution.

---

```
➔ 1 def sumTo(aBound):
2     """ Return the sum of 1+2+3 ... n """
3
4     theSum = 0
5     aNumber = 1
6     while aNumber <= aBound:
7         theSum = theSum + aNumber
8         aNumber = aNumber + 1
9     return theSum
10
11 print(sumTo(4))
```

---



&lt;&lt; First

&lt; Back

Step 1 of 19

Forward &gt;

Last &gt;&gt;

→ line that has just executed

→ next line to execute

Frames

Objects

### CodeLens: 1 (ch07\_while2)

#### Note

The names of the variables have been chosen to help readability.

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `False` or `True`.
2. If the condition is `False`, exit the `while` statement and continue execution at the next statement.
3. If the condition is `True`, execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is `False` the first time through the loop, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes `False` and the loop terminates. Otherwise the loop will repeat forever. This is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions written on the back of the shampoo bottle (lather, rinse, repeat) create an infinite loop.

In the case shown above, we can prove that the loop terminates because we know that the value of `aBound` is finite, and we can see that the value of `aNumber` increments each time through the loop, so eventually it will have to exceed `aBound`. In other cases, it is not so easy to tell.

#### Note

Introduction of the `while` statement causes us to think about the types of iteration we have seen. The `for` statement will always iterate through a sequence of values like the list of names for the party or the list of numbers created by `range`. Since we know that it will iterate once for each value in the collection, it is often said that a `for` loop creates a **definite iteration** because we definitely know how many times we are going to iterate. On the other hand, the `while` statement is dependent on a condition that needs to evaluate to `False` in order for the loop to terminate. Since we do not necessarily know when this will

happen, it creates what we call **indefinite iteration**. Indefinite iteration simply means that we don't know how many times we will repeat but eventually the condition controlling the iteration will fail and the iteration will stop. (Unless we have an infinite loop which is of course a problem)

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop you have to control the loop variable yourself. You give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

So why have two kinds of loop if `for` looks easier? This next example shows an indefinite iteration where we need the extra power that we get from the `while` loop.

### Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

1	
2	

**ActiveCode: 4** (scratch\_07\_01)

Run

### Check your understanding

itr-1: True or False: You can rewrite any for-loop as a while-loop.

- ☒ a) True  
☐ b) False

Check Me

Compare Me

itr-2: The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

```
n = 10
answer = 1
while ( n > 0 ):
    answer = answer + n
    n = n + 1
print answer
```

- ☒ a) n starts at 10 and is incremented by 1 each time through the loop, so it will always be positive
- ☐ b) answer starts at 1 and is incremented by n each time, so it will always be positive
- ☐ c) You cannot compare n to 0 in while loop. You must compare it to another variable.
- ☐ d) In the while loop body, we must set n to False, and this code does not do that.

[Check Me](#)[Compare Me](#)

## Randomly Walking Turtles

Suppose we want to entertain ourselves by watching a turtle wander around randomly inside the screen. When we run the program we want the turtle and program to behave in the following way:

1. The turtle begins in the center of the screen.
2. Flip a coin. If its heads then turn to the left 90 degrees. If its tails then turn to the right 90 degrees.
3. Take 50 steps forward.
4. If the turtle has moved outside the screen then stop, otherwise go back to step 2 and repeat.

Notice that we cannot predict how many times the turtle will need to flip the coin before it wanders out of the screen, so we can't use a for loop in this case. In fact, although very unlikely, this program might never end, that is why we call this indefinite iteration.

So based on the problem description above, we can outline a program as follows:

```
create a window and a turtle

while the turtle is still in the window:
    generate a random number between 0 and 1
    if the number == 0 (heads):
        turn left
    else:
        turn right
    move the turtle forward 50
```

Now, probably the only thing that seems a bit confusing to you is the part about whether or not the turtle is still in the screen. But this is the nice thing about programming, we can delay the tough stuff and get *something* in our program working right away. The way we are going to do this is to delegate the work of deciding whether the turtle is still in the screen or not to a boolean function. Lets call this boolean function `isInScreen`. We can write a very simple version of this boolean function by having it always return `True`, or by having it decide randomly, the point is to have it do something simple so that we can focus on the parts we already know how to do well and get them working. Since having it always return true would not be a good idea we will write our version to decide randomly. Lets say that there is a 90% chance the turtle is still in the window and 10% that the turtle has escaped.

1	
---	--

**ActiveCode: 5** (iter\_randwalk1)

Run

Now we have a working program that draws a random walk of our turtle that has a 90% chance of staying on the screen. We are in a good position, because a large part of our program is working and we can focus on the next bit of work – deciding whether the turtle is inside the screen boundaries or not.

We can find out the width and the height of the screen using the `window_width` and `window_height` methods of the screen object. However, remember that the turtle starts at position 0,0 in the middle of the screen. So we never want the turtle to go farther right than `width/2` or farther left than `negative width/2`. We never want the turtle to go further up than `height/2` or further down than `negative height/2`. Once we know what the boundaries are we can use some conditionals to check the turtle position against the boundaries and return `False` if the turtle is outside or `True` if the turtle is inside.

Once we have computed our boundaries we can get the current position of the turtle and then use conditionals to decide. Here is one implementation:



```
def isInScreen(wn,t):
    leftBound = -(wn.window_width()/2)
    rightBound = wn.window_width()/2
    topBound = wn.window_height()/2
    bottomBound = -(wn.window_height()/2)

    turtleX = t.xcor()
    turtleY = t.ycor()

    stillIn = True
    if turtleX > rightBound or turtleX < leftBound:
        stillIn = False
    if turtleY > topBound or turtleY < bottomBound:
        stillIn = False

    return stillIn
```

There are lots of ways that the conditional could be written. In this case we have given `stillIn` the default value of `True` and use two `if` statements to possibly set the value to `False`. You could rewrite this to use nested conditionals or `elif` statements and set `stillIn` to `True` in an `else` clause.

Here is the full version of our random walk program.

```
1 import random
2 import turtle
3
4 def isInScreen(w,t):
5     leftBound = - w.window_width()/2
6     rightBound = w.window_width()/2
7     topBound = w.window_height()/2
8     bottomBound = -w.window_height()/2
9
10    turtleX = t.xcor()
11    turtleY = t.ycor()
12
13    stillIn = True
14    if turtleX > rightBound or turtleX < leftBound:
15        stillIn = False
```

**ActiveCode: 6** (iter\_randwalk2)

Run

We could have written this program without using a boolean function. You might want to try to rewrite it using a complex condition on the while statement. However, using a boolean function makes the program much more readable and easier to understand. It also gives us another tool to use if this was a larger program and we needed to have a check for whether the turtle was still in the screen in another part of the program. Another advantage is that if you ever need to write a similar program, you can reuse this function with confidence the next time you need it. Breaking up this program into a couple of parts is another example of functional decomposition.

### Check your understanding

itr-3: Which type of loop can be used to perform the following iteration: You choose a positive integer at random and then print the numbers from 1 up to and including the selected integer.

- ☒ a) a for-loop or a while-loop
- ☐ b) only a for-loop
- ☐ c) only a while-loop

[Check Me](#)[Compare Me](#)

itr-4: In the random walk program in this section, what does the isInScreen function do?

- ☒ a) Returns True if the turtle is still on the screen and False if the turtle is no longer on the screen.
- ☐ b) Uses a while loop to move the turtle randomly until it goes off the screen.
- ☐ c) Turns the turtle right or left at random and moves the turtle forward 50.
- ☐ d) Calculates and returns the position of the turtle in the window.

[Check Me](#)[Compare Me](#)

## The $3n + 1$ Sequence

As another example of indefinite iteration, let's look at a sequence that has fascinated mathematicians for many years. The rule for creating the sequence is to start from some given number, call it  $n$ , and to generate the next term of the sequence from  $n$ , either by halving  $n$ , whenever  $n$  is even, or else by multiplying it by three and adding 1 when it is odd. The sequence terminates when  $n$  reaches 1.

This Python function captures that algorithm. Try running this program several times supplying different values for  $n$ .

```
1 def seq3np1(n):
2     """ Print the 3n+1 sequence from n, terminating when it reaches 1."""
3     while n != 1:
4         print(n)
5         if n % 2 == 0:           # n is even
6             n = n // 2
7         else:                   # n is odd
8             n = n * 3 + 1
9         print(n)                # the last print is 1
10
11 seq3np1(3)
12
```

**ActiveCode: 7** (ch07\_indef1)**Run**

```
3
10
5
16
8
4
2
1
```

The condition for this loop is  $n \neq 1$ . The loop will continue running until  $n == 1$  (which will make the condition false).

Each time through the loop, the program prints the value of  $n$  and then checks whether it is even or odd using the remainder operator. If it is even, the value of  $n$  is divided by 2 using integer division. If it is odd, the value is replaced by  $n * 3 + 1$ . Try some other examples.

Since  $n$  sometimes increases and sometimes decreases, there is no obvious proof that  $n$  will ever reach 1, or that the program terminates. For some particular values of  $n$ , we can prove termination. For example, if the starting value is a power of two, then the value of  $n$  will be even each time through the loop until it reaches 1.

You might like to have some fun and see if you can find a small starting number that needs more than a hundred steps before it terminates.

**Lab**

- Experimenting with the  $3n+1$  Sequence  
(<http://dcs.asu.edu/faculty/abansal/CST100/Labs/IterationsRevisited-ExperimentingWith3n+1SequenceLab.html>) In this guided lab exercise we will try to learn more about this sequence.

Particular values aside, the interesting question is whether we can prove that this sequence terminates for *all* values of  $n$ . So far, no one has been able to prove it or disprove it!

Think carefully about what would be needed for a proof or disproof of the hypothesis “*All positive integers will eventually converge to 1*”. With fast computers we have been able to test every integer up to very large values, and so far, they all eventually end up at 1. But this doesn’t mean that there might not be some as-yet untested number which does not reduce to 1.

You’ll notice that if you don’t stop when you reach one, the sequence gets into its own loop: 1, 4, 2, 1, 4, 2, 1, 4, and so on. One possibility is that there might be other cycles that we just haven’t found.

### Choosing between `for` and `while`

Use a `for` loop if you know the maximum number of times that you’ll need to execute the body. For example, if you’re traversing a list of elements, or can formulate a suitable call to `range`, then choose the `for` loop.

So any problem like “iterate this weather model run for 1000 cycles”, or “search this list of words”, “check all integers up to 10000 to see which are prime” suggest that a `for` loop is best.

By contrast, if you are required to repeat some computation until some condition is met, as we did in this  $3n + 1$  problem, you’ll need a `while` loop.

As we noted before, the first case is called **definite iteration** — we have some definite bounds for what is needed. The latter case is called **indefinite iteration** — we are not sure how many iterations we’ll need — we cannot even establish an upper bound!

### Check your understanding

itr-5: Consider the code that prints the  $3n+1$  sequence in ActiveCode box 6. Will the while loop in this code always terminate for any value of  $n$ ?

- ☐ a) Yes.
- ☐ b) No.
- ☒ c) No one knows.

Check Me

Compare Me

Correct!! That this sequence terminates for all values of  $n$  has not been proven or disproven so no one knows whether the while loop will always terminate or not.

## Newton's Method

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of  $n$ . If you start with almost any approximation, you can compute a better approximation with the following formula:

```
better = 1/2 * (approx + n/approx)
```

Execute this algorithm a few times using your calculator. Can you see why each iteration brings your estimate a little closer? One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer.


The following implementation of Newton's method requires two parameters. The first is the value whose square root will be approximated. The second is the number of times to iterate the calculation yielding a better result.

```
1 def newtonSqrt(n, howmany):
2     approx = 0.5 * n
3     for i in range(howmany):
4         betterapprox = 0.5 * (approx + n/approx)
5         approx = betterapprox
6     return betterapprox
7
8 print(newtonSqrt(10,3))
9 print(newtonSqrt(10,5))
10 print(newtonSqrt(10,10))
11
```

**ActiveCode: 8** (chp07\_newtonsdef)

Run



```
3.16231942215
3.16227766017
3.16227766017
```



You may have noticed that the second and third calls to `newtonSqrt` in the previous example both returned the same value for the square root of 10. Using 10 iterations instead of 5 did not improve the value. In general, Newton's algorithm will eventually reach a point where the new approximation is no better than the previous. At that point, we could simply stop. In other words, by repeatedly applying this formula until the better approximation gets close enough to the previous one, we can write a function for computing the square root that uses the number of iterations necessary and no more.

This implementation, shown in codelens, uses a `while` condition to execute until the approximation is no longer changing. Each time thru the loop we compute a "better" approximation using the formula described earlier. As long as the "better" is different, we try again. Step thru the program and watch the approximations get closer and closer.

```
1 def newtonSqrt(n):
2     approx = 0.5 * n
3     better = 0.5 * (approx + n/approx)
4     while better != approx:
5         approx = better
6         better = 0.5 * (approx + n/approx)
7     return approx
8
9 print(newtonSqrt(10))
```



&lt;&lt; First

&lt; Back

Program terminated

Forward &gt;

Last &gt;&gt;

 line that has just executed next line to execute

Program output:

3.16227766017

Frames

Objects

Global variables

newtonSqrt

function

newtonSqrt(n)

### CodeLens: 2 (chp07\_newtonwhile)

#### Note

The `while` statement shown above uses comparison of two floating point numbers in the condition. Since floating point numbers are themselves approximation of real numbers in mathematics, it is often better to compare for a result that is within some small threshold of the value you are looking for.

---

© Copyright 2013 Brad Miller, David Ranum, Created using Runestone Interactive.