

Iteration

Note

In the Notes, any code will be in **blue**, input is **green** and any output in **red**.

Updating variables

A common pattern in assignment statements is an assignment statement that updates a variable, where the new value of the variable depends on the old.

```
x = x + 1
```

This means “get the current value of x, add 1, and then update x with the new value.”

If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
x = x + 1
```

NameError: name 'x' is not defined

Before you can update a variable, you have to initialize it, usually with a simple assignment:

```
x = 0
```

```
x = x + 1
```

Updating a variable by adding 1 is called an increment; subtracting 1 is called a decrement.

The while statement

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. Because iteration is so common, Python provides several language features to make it easier.

One form of iteration in Python is the while statement. Here is a simple program that counts down from five and then says “Blastoff!”.

```
n = 5
```

```
while n > 0:
```

```
    print(n)
```

```
    n = n - 1
```

```
print('Blastoff!')
```

You can almost read the while statement as if it were English. It means, “While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!” More formally, here is the flow of execution for a while statement:

- 1. Evaluate the condition, yielding True or False.**
- 2. If the condition is false, exit the while statement and continue execution at the next statement.**
- 3. If the condition is true, execute the body and then go back to step 1.**

This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. For the above loop, we would say, “It had five iterations”, which means that the body of the loop was executed five times.

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. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

Infinite loops

An endless source of amusement for programmers is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop because there is no iteration variable telling you how many times to execute the loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we have to get to 0. Other times a loop is obviously infinite because it has no iteration variable at all.

Sometimes you don’t know it’s time to end a loop until you get halfway through the body. In that case you can write an infinite loop on purpose and then use the break statement to jump out of the loop.

This loop is obviously an infinite loop because the logical expression on the while statement is simply the logical constant True:

NOTE DON’T CODE THIS!!!!

```
n = 10
```

```
while True:
```

```
    print(n, end=' ')
```

```
    n = n - 1
```

```
print('Done!')
```

Finishing iterations with continue

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true by virtue of the fact that the expression is the constant value True.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops as long as we carefully add code to the body of the loop to explicitly exit the loop using break when we have reached the exit condition.

For example, suppose you want to take input from the user until they type done. You could write:

```
while True:
    line = input('> ')
    if line != 'done':
        print(line)
    else:
        print(line)
        print('Done!')
        break
```

The loop condition is True, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> gdagds
gdagds
> sakjdka
sakjdka
> lksdajdlksaj
lksdajdlksaj
> done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens").

Finishing iterations with continue

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types “done”, but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    elif line != 'done':
        print(line)
    else:
        print('Done!')
        break
```

Here is a sample run of this new program with continue added.

```
> sdfdsf
sdfdsf
> fsdf
fsdf
> #fdsfsdf
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the continue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

Definite loops using for

Sometimes we want to loop through a *set* of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a *definite* loop using a for statement. We call the while statement an *indefinite* loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

The syntax of a for loop is similar to the while loop in that there is a for statement and a loop body:

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print('Happy New Year:', friend)
print('Done!')
```

In Python terms, the variable `friends` is a list of three strings and the `for` loop goes through the list and executes the body once for each of the three strings in the list resulting in this output:

Happy New Year: Joseph

Happy New Year: Glenn

Happy New Year: Sally

Done!

Translating this `for` loop to English is not as direct as the `while`, but if you think of `friends` as a *set*, it goes like this: “Run the statements in the body of the `for` loop once for each friend *in* the set named `friends`.”

Looking at the `for` loop, *for* and *in* are reserved Python keywords, and `friend` and `friends` are variables.

```
for friend in friends:

    print('Happy New Year:', friend)
```

In particular, `friend` is the *iteration variable* for the `for` loop. The variable `friend` changes for each iteration of the loop and controls when the `for` loop completes. The *iteration variable* steps successively through the three strings stored in the `friends` variable.

Loop patterns

Often we use a `for` or `while` loop to go through a list of items or the contents of a file and we are looking for something such as the largest or smallest value of the data we scan through.

These loops are generally constructed by:

- **Initializing one or more variables before the loop starts**
- **Performing some computation on each item in the loop body, possibly changing the variables in the body of the loop**
- **Looking at the resulting variables when the loop completes**

We will use a list of numbers to demonstrate the concepts and construction of these loop patterns.

Counting and summing loops

For example, to count the number of items in a list, we would write the following for loop:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
    print('Count: ', count)
```

We set the variable `count` to zero before the loop starts, then we write a for loop to run through the list of numbers. Our *iteration* variable is named `itervar` and while we do not use `itervar` in the loop, it does control the loop and cause the loop body to be executed once for each of the values in the list.

In the body of the loop, we add 1 to the current value of `count` for each of the values in the list. While the loop is executing, the value of `count` is the number of values we have seen “so far”.

Once the loop completes, the value of `count` is the total number of items. The total number “falls in our lap” at the end of the loop. We construct the loop so that we have what we want when the loop finishes.

Another similar loop that computes the total of a set of numbers is as follows:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
    print('Total: ', total)
```

In this loop we *do* use the *iteration variable*. Instead of simply adding one to the count as in the previous loop, we add the actual number (3, 41, 12, etc.) to the running total during each loop iteration. If you think about the variable `total`, it contains the “running total of the values so far”. So before the loop starts `total` is zero because we have not yet seen any values, during the loop `total` is the running total, and at the end of the loop `total` is the overall total of all the values in the list.

As the loop executes, `total` accumulates the sum of the elements; a variable used this way is sometimes called an *accumulator*.

Neither the counting loop nor the summing loop are particularly useful in practice because there are built-in functions **`len()`** and **`sum()`** that compute the number of items in a list and the total of the items in the list respectively.

Maximum and minimum loops

To find the largest value in a list or sequence, we construct the following loop:

```
largest = None
print('Before:', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest:', largest)
```

When the program executes, the output is as follows:

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

The variable `largest` is best thought of as the “largest value we have seen so far”. Before the loop, we set `largest` to the constant `None`. `None` is a special constant value which we can store in a variable to mark the variable as “empty”.

Before the loop starts, the largest value we have seen so far is `None` since we have not yet seen any values. While the loop is executing, if `largest` is `None` then we take the first value we see as the largest so far. You can see in the first iteration when the value of `itervar` is 3, since `largest` is `None`, we immediately set `largest` to be 3.

After the first iteration, `largest` is no longer `None`, so the second part of the compound logical expression that checks `itervar > largest` triggers only when we see a value that is larger than the “largest so far”. When we see a new “even larger” value we take that new value for `largest`. You can see in the program output that `largest` progresses from 3 to 41 to 74.

At the end of the loop, we have scanned all of the values and the variable `largest` now does contain the largest value in the list.

To compute the smallest number, the code is very similar with one small change:

```
smallest = None

print('Before:', smallest)

for itervar in [3, 41, 12, 9, 74, 15]:

    if smallest is None or itervar < smallest:

        smallest = itervar

    print('Loop:', itervar, smallest)

print('Smallest:', smallest)
```

Again, `smallest` is the “smallest so far” before, during, and after the loop executes. When the loop has completed, `smallest` contains the minimum value in the list.

Again as in counting and summing, the built-in functions **`max()`** and **`min()`** make writing these exact loops unnecessary.

The following is a simple version of the Python built-in `min()` function:

```
values = [3, 41, 12, 9, 74, 15]

print(min(values))
```

Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is “debugging by bisection.” For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.