Chapter 6. Loop with while and for

For a' that, an' a' that, Our toils obscure, an' a' that ...

—Robert Burns, For a' That and a' That

Testing with `if`, `elif`, and `else` runs from top to bottom. Sometimes, we need to do something more than once. We need a *loop*, and Python gives us two choices: `while` and `for`.

Repeat with while

The simplest looping mechanism in Python is `while`. Using the interactive interpreter, try this example, which is a simple loop that prints the numbers from 1 to 5:

```
>>> count = 1
>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
>>>
```

We first assigned the value `1` to `count`. The `while` loop compared the value of `count` to `5` and continued if `count` was less than or equal to `5`. Inside the loop, we printed the value of `count` and then *incremented* its value by one with the statement `count += 1`. Python goes back to the top of the loop, and again compares `count` with `5`. The value of `count` is now `2`, so the contents of the `while` loop are again executed, and `count` is incremented to `3`.

This continues until `count` is incremented from `5` to `6` at the bottom of the loop. On the next trip to the top, `count <= 5` is now `False`, and the `while` loop ends. Python moves on to the next lines.

Cancel with break

If you want to loop until something occurs, but you're not sure when that might happen, you can use an *infinite loop* with a `break` statement. This time, let's read a line of input from the keyboard via Python's `input()` function and then print it with the first letter capitalized. We break out of the loop when a line containing only the letter `q` is typed:

```
>>> while True:
...     stuff = input("String to capitalize [type q to quit]: ")
...     if stuff == "q":
...         break
...     print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
>>>
```

Skip Ahead with continue

Sometimes, you don't want to break out of a loop but just want to skip ahead to the next iteration for some reason. Here's a contrived example: let's read an integer, print its square if it's odd, and skip it if it's even. We even added a few comments. Again, we use `q` to stop the loop:

```
>>> while True:
...     value = input("Integer, please [q to quit]: ")
...     if value == 'q':      # quit
...         break
...     number = int(value)
...     if number % 2 == 0:   # an even number
...        continue
...     print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
Integer, please [q to quit]: q
>>>
```

Check break Use with else

If the `while` loop ended normally (no `break` call), control passes to an optional `else`. You use this when you've coded a `while` loop to check for something, and breaking as soon as it's found. The `else` would be run if the `while` loop completed but the object was not found:

```
>>> numbers = [1, 3, 5]
>>> position = 0
>>> while position < len(numbers):
...     number = numbers[position]
...     if number % 2 == 0:
...         print('Found even number', number)
...         break
...     position += 1
... else:  # break not called
...     print('No even number found')
...
No even number found
```

NOTE

This use of `else` might seem nonintuitive. Consider it a *break checker*.

Iterate with for and in

Python makes frequent use of *iterators*, for good reason. They make it possible for you to traverse data structures without knowing how large they are or how they are implemented. You can even iterate over data that is created on the fly, allowing processing of data *streams* that would otherwise not fit in the computer's memory all at once.

To show iteration, we need something to iterate over. You've already seen strings in Chapter 5, but have not yet read the details on other *iterables* like lists and tuples (Chapter 7) or dictionaries (Chapter 8). I'll show two ways to walk through a string here, and show iteration for the other types in their own chapters.

It's legal Python to step through a string like this:

```
>>> word = 'thud'
>>> offset = 0
>>> while offset < len(word):
...     print(word[offset])
...     offset += 1
...
t
h
u
d
```

But there's a better, more Pythonic way:

```
>>> for letter in word:
...     print(letter)
...
t
h
u
d
```

String iteration produces one character at a time.

Cancel with break

A `break` in a `for` loop breaks out of the loop, as it does for a `while` loop:

```
>>> word = 'thud'
>>> for letter in word:
...     if letter == 'u':
...         break
...     print(letter)
...
t
h
```

Skip with continue

Inserting a `continue` in a `for` loop jumps to the next iteration of the loop, as it does for a `while` loop.

Check break Use with else

Similar to `while`, `for` has an optional `else` that checks whether the `for` completed normally. If `break` was *not* called, the `else` statement is run.

This is useful when you want to verify that the previous `for` loop ran to completion instead of being stopped early with a `break`:

```
>>> word = 'thud'
>>> for letter in word:
...     if letter == 'x':
...         print("Eek! An 'x'!")
...         break
...     print(letter)
... else:
...     print("No 'x' in there.")
...
t
h
u
d
No 'x' in there.
```

NOTE

As with `while`, the use of `else` with `for` might seem nonintuitive. It makes more sense if you think of the `for` as looking for something, and `else` being called if you didn't find it. To get the same effect without `else`, use some variable to indicate whether you found what you wanted in the `for` loop.

Generate Number Sequences with range()

The `range()` function returns a stream of numbers within a specified range. without first having to create and store a large data structure such as a list or tuple. This lets you create huge ranges without using all the memory in your computer and crashing your program.

You use `range()` similar to how to you use slices: `range(` *`start`*, *`stop`*, *`step`* `)`. If you omit *`start`*, the range begins at `0`. The only required value is *`stop`*; as with slices, the last value created will be just before *`stop`*. The default value of *`step`* is `1`, but you can go backward with `-1`.

Like `zip()`, `range()` returns an *iterable* object, so you need to step through the values with `for ... in`, or convert the object to a sequence like a list. Let's make the range `0, 1, 2`:

```
>>> for x in range(0,3):
...     print(x)
...
0
1
2
>>> list( range(0, 3) )
[0, 1, 2]
```

Here's how to make a range from `2` down to `0`:

```
>>> for x in range(2, -1, -1):
...     print(x)
...
2
1
0
>>> list( range(2, -1, -1) )
[2, 1, 0]
```

The following snippet uses a step size of `2` to get the even numbers from `0` to `10`:

```
>>> list( range(0, 11, 2) )
[0, 2, 4, 6, 8, 10]
```

Other Iterators

Chapter 14 shows iteration over files. In Chapter 10, you can see how to enable iteration over objects that you've defined yourself. Also, Chapter 11 talks about `itertools`—a standard Python module with many useful shortcuts.

Coming Up

Chain individual data into *lists* and *tuples*.

Things to Do

6.1 Use a `for` loop to print the values of the list `[3, 2, 1, 0]`.

6.2 Assign the value `7` to the variable `guess_me`, and the value `1` to the variable `number`. Write a `while` loop that compares `number` with `guess_me`. Print `'too low'` if `number` is less than `guess me`. If `number` equals `guess_me`, print `'found it!'` and then exit the loop. If `number` is greater than `guess_me`, print `'oops'` and then exit the loop. Increment `number` at the end of the loop.

6.3 Assign the value `5` to the variable `guess_me`. Use a `for` loop to iterate a variable called `number` over `range(10)`. If `number` is less than `guess_me`, print `'too low'`. If it equals `guess_me`, print `found it!` and then break out of the for loop. If `number` is greater than `guess_me`, print `'oops'` and then exit the loop.