For Loop

A loop takes a few lines of code, and runs them again and again. Most algorithms have a lines of code that need to be run thousands or millions of times, and loops are the way to do this.

For Loop - aka Foreach

The "for" loop is probably the single most useful type of loop. The for-loop, aka "foreach" loop, looks at each element in a collection once. The collection can be any type of collection-like data structure, but the examples below use a list.

Here is a for-loop example that prints a few numbers:

```
>>> for num in [2, 4, 6, 8]:
        print(num)
2
4
6
8
```

Loop Syntax: the loop begins with the keyword `for` followed by a variable name to use in the loop, e.g. `num` in this example. Then the keyword `in` and a collection of elements for the loop, e.g. the list `[2, 4, 6, 8]`. Finally there is colon `:` followed by the indented "body" lines controlled by the loop.

Loop Operation: the loop runs the body lines again and again, once for each element in the collection. Each run of the body is called an "iteration" of the loop. For the first iteration, the variable is set to the first element, and the body lines run (in this case, essentially `num = 2`. For the second iteration, `num = 4` and so on, once for each element.

The main story of the for loop is that if we have a collection of numbers or strings or pixels, the for-loop is an easy way to write code that looks at each value once. Now we'll look at a few features and slightly subtle features of the loop.

Rule: Do Not Change Collection While Looping

For the for-loop to work properly, the body lines should not add or remove elements while the loop is running. This is a pretty reasonable rule — it's hard to see how the loop could work if elements were appearing and disappearing on the fly. Violating this rule does not produce a clear error message; it just means the loop may not see the series elements correctly.

Zero Iterations is Ok

Suppose `urls` is a list of urls we want to print with a foreach:

```
for url in urls:
    print(url)
print('All done')
```

What does the above code do if the number of urls in the list is zero, i.e. the list is empty? That's actually a valid loop input, and in that case the for-loop just runs the body lines zero times, skipping directly to the "All done" line. Sometimes programmers feel they need to add an extra if-statement to guard against the list being empty, but in fact the for-loop skips over the empty collection fine.

Loop Controls The Variable, Not You

Usually variables only change when we see an assignment with an equal sign **=**

The for-loop is different, since for each iteration, the loop sets the variable to point to the next value. Mostly this is very convenient, but it does mean that setting the variable to something at the end of the loop has basically no effect:

```
for num in [2, 4, 6, 8]:
    print(num)
    num = 100    # No effect on output,
                 # the loop resets num on each
iteration
```

Loop Break

The `break` directive in a loop exits the loop immediately. Loops have their standard way of exiting. The `break` provides an extra way to exit the loop if some special condition occurs. Usually the `break` is put inside an if that checks for some condition.

This example loops over a list of numbers, printing each one in the usual way. However, the if/break structure checks each number before printing, and breaks out of the loop if the number is 6.

```
nums = [12, 1, 6, 13, 6, 0]
for num in nums:
    if num == 6:
        break  # exit loop immediately
    print(num)
print('All done')

12
1
All done
```

Most loops do not use `break`, but it is an option to exit the loop early.

Loop Continue

The `continue` directive directs the loop run to go back to the top of the loop immediately to start the next iteration. In effect, it skips the current iteration. The continue directive is very rarely used. We mention it here for completeness since it goes with `break`.

Here is the above example changed to use `continue`. In effect it skips over iterations where num is 6.

```
nums = [12, 1, 6, 13, 6, 0]
for num in nums:
    if num == 6:
        continue  # jump to top of loop
    print(num)
print('All done')

12
1
13
0
All done
```

Standard Loop Phrases

Looping over collections is very common. Here are the most important patterns.

1. Loop Over a List

The for-loop will see each element in the list once. There are no index numbers in this form.

```
words = ['hi', 'there!', ...]
for word in words:
    # use word in here
    print(word)
```

2. Loop Over range()

Combine a for-loop with range() to loop over a series of numbers. (See also range())

```
for i in range(10):
  print(i)
# 0 1 2 3 4 5 6 7 8 9
```

3. Loop Over All Index Numbers

For a linear collections (e.g. list, string), index numbers 0, 1, ... len-1 indentify each element in the collection. The following loop combines for/range/len to run a variable i through all the index numbers in a string or list.

Code inside the loop can use `[i]` or a slice or whatever to access into the collection based on `i`. Use this form if the algorithm needs access to the index numbers, not just the bare elements.

```
# string s, loop though all its chars by index number
s = 'Hello'
for i in range(len(s)):  # i.e. range(5)
  print(i, s[i])

0 H
1 e
2 l
3 l
4 0
```

It's standard to use the one-letter variable names `i, j, k` for looping through index numbers like this (echoing use of subscripts i, j, k in mathematical writing before computers existed).