# FOR LOOP AND THE RANGE() BUILT IN FUNCTION

The for loop in Python is more like a foreach iterative-type loop in a shell scripting language than a traditional for conditional loop that works like a counter. Python's for takes an iterable (such as a sequence or iterator) and traverses each element once.

```
>>> print 'I like to use the Internet for:'
I like to use the Internet for:
>>> for item in ['e-mail', 'net-surfing', 'homework',
'chat']:
... print item
 ...
e-mail
net-surfing
homework
chat
```

Our output in the previous example may look more presentable if we display the items on the same line rather than on separate lines. print statements by default automatically add a NEWLINE character at the end of every line. This can be suppressed by terminating the print statement with a comma ( , ).

```
print 'I like to use the Internet for:'
 for item in ['e-mail', 'net-surfing', 'homework', 'chat']:
 print item,
print
```

The code required further modification to include an additional print statement with no arguments to flush our line of output with a terminating NEWLINE; otherwise, the prompt will show up on the same line immediately after the last piece of data output. Here is the output with the modified code:

I like to use the Internet for:

e-mail net-surfing homework chat

Elements in print statements separated by commas will automatically include a delimiting space between them as they are displayed.

Providing a string format gives the programmer the most control because it dictates the exact output layout, without having to worry about the spaces generated by commas. It also allows all the data to be grouped together in one placethe tuple or dictionary on the right-hand side of the format operator.

```
>>> who = 'knights'
>>> what = 'Ni!'
>>> print 'We are the', who, 'who say', what, what, what, what
We are the knights who say Ni! Ni! Ni! Ni!
>>> print 'We are the %s who say %s' % \
... (who, ((what + ' ') * 4))
We are the knights who say Ni! Ni! Ni! Ni!
```

Using the string format operator also allows us to do some quick string manipulation before the output, as you can see in the previous example. We conclude our introduction to loops by showing you how we can make Python's for statement act more like a traditional loop, in other words, a numerical counting loop. Because we cannot change the behavior of a for loop (iterates over a sequence), we can manipulate our sequence so that it is a list of numbers. That way, even though we are still iterating over a sequence, it will at least appear to

perform the number counting and incrementing that we envisioned.

```
>>> for eachNum in [0, 1, 2]:
... print eachNum
...
0
1
2
```

Within our loop, eachNum contains the integer value that we are displaying and can use it in any numerical calculation we wish. Because our range of numbers may differ, Python provides the range() built-in function to generate such a list for us. It does exactly what we want, taking a range of numbers and generating a list.

```
>>> for eachNum in range(3):
... print eachNum
...
0
1
2
```

For strings, it is easy to iterate over each character:

```
>>> foo = 'abc'
>>> for c in foo:
... print c
...
a
b
c
```

The range() function has been often seen with len() for indexing into a string. Here, we can display both elements and their corresponding index value:

```
>>> foo = 'abc'
 >>> for i in range(len(foo)):
 ... print foo[i], '(%d)' % i
 ...
a (0)
 b (1)
 c (2)
```

However, these loops were seen as restrictiveyou either index by each element or by its index, but never both. This led to the enumerate() function (introduced in Python 2.3) that does give both:

```
 >>> for i, ch in enumerate(foo):
 ... print ch, '(%d)' %% i
 ...
a (0)
 b (1)
c (2)
```