
# Python 101 (part 3): A Twist In The Tail

## By Vikram Vaswani

This article copyright Melonfire 2000−2002. All rights reserved.

# Table of Contents

# The Big Picture

Last time out, I taught you a little bit about two of Python's basic data types, strings and numbers. I also gave you a crash course in Python operators and expressions, and demonstrated how they could be used in conjunction with the "if" family of conditional statements to add control routines to your Python programs.

However, "if" statements are only one piece of the jigsaw; Python also allows you to control program execution with the "while" and "for" loops, which coincidentally also happen to be the subject of today's discussion. Along the way, I'll be demonstrating yet another of Python's built−in data structures, visiting the Superhero Hall Of Fame, hooking up with some old flames, and examining prime numbers and factorials.

Let's get started, shall we?

# Here Comes A Hero

Thus far, the variables you've used contain only a single value − for example,

```
Python 1.5.2 (#1, Aug 25 2000, 09:33:37) [GCC 2.96 20000731
(experimental)] on linux-i386
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> i=0
>>> alpha=63453473458348383L
>>> name="god"
>>>
```

For simple Python programs, this is usually more than enough. However, as your Python programs grow in complexity, you're going to need more advanced data structures to store and manipulate information. And that's exactly where lists come in.

Unlike string and number objects, which typically hold a single value, a list can best be thought of as a "container" variable, which can contain one or more values. For example,

```
>>> superheroes = ["Spiderman", "Superman", "Human Torch",
"Batman"]
>>>
```

Here, "superheroes" is a list containing the values "Spiderman", "Superman", "Human Torch", and "Batman".

Lists are particularly useful for grouping related values together − names, dates, phone numbers of ex−girlfriends et al. The various elements of the list are accessed via an index number, with the first element starting at zero. So, to access the element "Superman", you would use the notation

```
>>> superheroes[0]
'Spiderman'
>>>
```

while

```
>>> superheroes[3]
'Batman'
>>>
```

− essentially, the list name followed by the index number enclosed within square braces. Geeks refer to this as "zero−based indexing".

Defining a list is simple − simply assign values (enclosed in square braces) to a variable, as illustrated below:

```
>>> oldFlames = ["Jennifer", "Susan", "Tina", "Bozo The
Clown"]
>>>
```

The rules for choosing a list name are the same as those for any other Python variable − it must begin with a letter, and can optionally be followed by more letters and numbers.

If you've worked with other programming languages, it should now be obvious that lists in Python are the equivalent of arrays in Perl, PHP and C. However, unlike these languages, Python does not restrict lists to elements of a specific object type, and can mix strings, numbers and even other lists within a single list "container".

```
>>> allMixedUp = ["ding dong", 23, "abracadabra", 26346.3, [4,
"four"]]
>>> allMixedUp
['ding dong', 23, 'abracadabra', 26346.3, [4, 'four']]
>>> allMixedUp[0]
'ding dong'
>>> allMixedUp[4]
[4, 'four']
>>> allMixedUp[4][0]
4
>>> allMixedUp[4][1]
'four'
>>>
```

# Making Friends And Influencing People

Lists can be concatenated with the + operator,

```
>>> oldFlames = ["Jennifer", "Susan", "Tina", "Bozo The
Clown"]
>>> superheroes = ["Spiderman", "Superman", "Human Torch",
"Batman"]
>>> strangeFriends = oldFlames + superheroes
>>> strangeFriends
['Jennifer', 'Susan', 'Tina', 'Bozo The Clown', 'Spiderman',
'Superman',
'Human Torch', 'Batman']
>>>
```

and repeated with the * operator, in much the same manner as strings and numbers.

```
>>> oldFlames * 3
['Jennifer', 'Susan', 'Tina', 'Bozo The Clown', 'Jennifer',
'Susan',
'Tina', 'Bozo The Clown', 'Jennifer', 'Susan', 'Tina', 'Bozo
The Clown']
>>>
```

"Slices" of a list can be extracted using notation similar to that used for extracting substrings − take a look:

```
>>> oldFlames
['Jennifer', 'Susan', 'Tina', 'Bozo The Clown']
>>> oldFlames[0]
'Jennifer'
>>> oldFlames[0:2]
['Jennifer', 'Susan']
>>> oldFlames[0:5]
['Jennifer', 'Susan', 'Tina', 'Bozo The Clown']
>>> oldFlames[0:]
['Jennifer', 'Susan', 'Tina', 'Bozo The Clown']
>>> oldFlames[:3]
```

```
['Jennifer', 'Susan', 'Tina']
>>> oldFlames[-4]
'Jennifer'
>>> oldFlames[-1]
'Bozo The Clown'
>>>
```

The built-in len() function can be used to calculate the number of elements in a list,

```
>>> superheroes
['Spiderman', 'Superman', 'Human Torch', 'Batman']
>>> len(superheroes)
4
>>>
```

while the "in" and "not in" operators can be used to test for the presence of a particular element in a list. A match returns 1 (true), while a failure returns 0 (false).

```
>>> superheroes
['Spiderman', 'Superman', 'Human Torch', 'Batman']
>>> "batman" in superheroes
0
>>> "Batman" in superheroes
1
>>> "Batma" in superheroes
0
>>> "Incredible Hulk" in superheroes
0
>>> "Incredible Hulk" not in superheroes
1
>>>
```

# We Don't Need Another Hero

Unlike strings, lists are "mutable", which means that the elements contained within a list can be changed at will. For example, any list element can be altered simply by assigning a new value to it via its index.

```
>>> superheroes
['Spiderman', 'Superman', 'Human Torch', 'Batman']
>>> superheroes[3] = "Captain America"
>>> superheroes
['Spiderman', 'Superman', 'Human Torch', 'Captain America']
>>>
```

You can alter more than one value at a time by using list slices.

```
>>> superheroes
['Spiderman', 'Superman', 'Human Torch', 'Captain America']
>>> superheroes[0:2] = ["Incredible Hulk", "Green Lantern"]
>>> superheroes
['Incredible Hulk', 'Green Lantern', 'Human Torch', 'Captain
America']
>>>
```

The built−in append() method makes it easy to add items to a list,

```
>>> superheroes
['Incredible Hulk', 'Green Lantern', 'Human Torch', 'Captain
America']
>>> superheroes.append("Spawn")
>>> superheroes
['Incredible Hulk', 'Green Lantern', 'Human Torch', 'Captain
America',
'Spawn']
>>>
```

while the del() method makes it just as easy to remove them.

```
>>> superheroes
['Incredible Hulk', 'Green Lantern', 'Human Torch', 'Captain
America',
'Spawn']
>>> del superheroes[4]
>>> superheroes
['Incredible Hulk', 'Green Lantern', 'Human Torch', 'Captain
America']
>>> del superheroes[0:2]
>>> superheroes
['Human Torch', 'Captain America']
>>>
```

Note that there's also a remove() method, which allows you to remove an element by value rather than index.

```
>>> superheroes
['Incredible Hulk', 'Green Lantern', 'Human Torch', 'Captain
America']
>>> superheroes.remove("Green Lantern")
>>> superheroes.remove("Captain America")
>>> superheroes
['Incredible Hulk', 'Human Torch']
>>>
```

Finally, the sort() and reverse() methods allow you to rearrange the contents of a list.

```
>>> oldFlames
['Jennifer', 'Susan', 'Tina', 'Bozo The Clown']
>>> oldFlames.sort()
>>> oldFlames
['Bozo The Clown', 'Jennifer', 'Susan', 'Tina']
>>> oldFlames.reverse()
>>> oldFlames
['Tina', 'Susan', 'Jennifer', 'Bozo The Clown']
>>>
```

A number of other list methods are also available − take a look at the Python Library Reference at http://www.python.org/doc/current/lib/lib.html for more information and examples.

Let's move on to loops.

# Looping The Loop

For those of you unfamiliar with the term, a "loop" is a programming construct that allows you to execute a set of statements over and over again, until a pre−defined condition is met. It's one of the most basic constructs available in a programming language, and comes in handy when you need to perform a repetitive task over and over again.

Unlike its counterparts, which offer a variety of different loop variants, Python keeps things simple with only two types of loops: the "while" loop and the "for" loop. The former is simpler to read and understand, and it usually looks like this:

```
while (condition):
do this!
```

In English, this would roughly translate to

```
while (living with Mom and Dad):
curfew is 11 PM
```

while the Python equivalent would look like this

```
while (livingWithParents == 1):
curfew = 2300
```

As with conditional statements, so long as the specified condition remains true, the indented code block will continue to execute. However, as soon as the condition becomes false − you move out and get your own place, say − the loop will be broken and the indented statements will stop executing.

You'll notice that Python uses indentation to decide which statements belong to the "while" block − you probably remember this from last time. As with the "if" statement, if the code block consists of only a single statement, Python allows you to place it on the same line as the "while" statement. For example,

```
while (livingWithParents == 0): curfew = "Huh? What curfew?"
```

is a perfectly valid "while" loop.

Here's a simple example which demonstrates the "while" loop.

```
#!/usr/bin/python

# initialize a variable
response = ""

# while loop
while response != "y":

# keep asking the question
response = raw_input("Would you like to receive unsolicited
commercial email from people you don't know, advertising
products you have
no interest in, on a regular basis (once every 15 minutes)?
[y/n] ")

print "Thank you for your cooperation!"
```

And here's the output.

```
Would you like to receive unsolicited commercial email from
people you
don't know, advertising products you have no interest in, on a
regular
basis (once every 15 minutes)? [y/n] n
Would you like to receive unsolicted commerial email from
people you don't
know, advertising products you have no interest in, on a
regular basis
(once every 15 minutes)? [y/n] n
Would you like to receive unsolicted commerial email from
people you don't
know, advertising products you have no interest in, on a
regular basis
(once every 15 minutes)? [y/n] n
Would you like to receive unsolicted commerial email from
people you don't
```

```
know, advertising products you have no interest in, on a
regular basis
(once every 15 minutes)? [y/n] y
Thank you for your cooperation!
```

Python allows you to add an "else" clause to your "while" loop as well; this clause is executed if the loop is executed without encountering a single "break" statement (more on this later.)

Consequently, the example above could be rewritten to read:

```
#!/usr/bin/python

# initialize a variable
response = ""

# while loop
while response != "y":

# keep asking the question
response = raw_input("Would you like to receive unsolicited
commercial email from people you don't know, advertising
products you have
no interest in, on
a regular basis (once every 15 minutes)? [y/n] ")

else:
print "Thank you for your cooperation!"
```

How about something a little more constructive?

```
#!/usr/bin/python

# get a number
num = input("Gimme a number: ")

# assign the number to a "temp" variable
tmpnum = num
factorial = 1

# calculate the factorial
while (num != 1):
```

```
factorial = factorial * num
num = num - 1

print "The factorial of", tmpnum, "is", factorial
```

In case you flunked math class, the factorial of a number X is the product of all the numbers between 1 and X. And here's what the output looks like:

```
Gimme a number: 7
The factorial of 7 is 5040
```

And if you have a calculator handy, you'll see that

```
7*6*5*4*3*2*1 = 5040
```

Once the user enters a number, a "while" loop is used to calculate the product of that number and the variable "factorial" (initialized to 1) − this value is again stored in the variable "factorial". Next, the number is reduced by 1, and the process is repeated, until the number becomes equal to 1. At this stage, the value of "factorial" is printed.

# Twist And Turn

In most programming languages, a "for" loop is used to execute a set of statements a certain number of times. Unlike a "while" loop, which continues to run for so long as the specified conditional expression evaluates as true, a "for" loop comes with a specific limit on the number of times it can iterate.

Python's "for" loop conforms to this basic requirement; however, as with most things in Python, there's a twist in the tail. A Python "for" loop is designed only to iterate over built−in "sequence objects" like strings and lists, and is structured like this:

```
for temp_var in sequence_obj:
do this!
```

Or, in English, "take each element of the sequence sequence_obj, place it in the variable temp_var, and execute the indented code block on temp_var".

An example might help to make this clearer:

```
>>> superheroes = ['Incredible Hulk', 'Green Lantern', 'Human
Torch',
'Captain America']
>>> for myhero in superheroes:
... print myhero, "rocks!"
...
Incredible Hulk rocks!
Green Lantern rocks!
Human Torch rocks!
Captain America rocks!
>>>
```

In this case, I've first initialized a list containing four elements. Next, I've used a "for" loop to iterate through the list; on each successive iteration, one element of the list is assigned to the temporary variable "myhero" and then printed to the console via a print() call. Once all the elements of the list have been processed, the loop is automatically terminated.

You can use a "for" loop with any "sequence object" − this next example does something similar with a string.

```
>>> str = "abracadabra"
>>> for char in str:
... print char, "_",
...
a _ b _ r _ a _ c _ a _ d _ a _ b _ r _ a _
>>>
```

---

# Within Range()

While on the topic of the "for" loop, it's worth mentioning the range() function, a built−in Python function whose sole raison d'etre is to return a range of numbers, given a starting and ending point. This range is always returned as a list − and as you'll see, this can combine quite effectively with the "for" loop in certain situations.

```
>>> range(30,40)
[30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
>>>
```

You can omit the first argument to have Python generate a range from 0 to the specified end point.

```
>>> range(40)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39]
>>>
```

You can skip certain numbers in the range by adding an optional "step" argument (by default, this is 1).

```
>>> range (25,500,25)
[25, 50, 75, 100, 125, 150, 175, 200, 225, 250, 275, 300, 325,
350, 375,
400, 425, 450, 475]
>>> range (100,1,−10)
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
>>>
```

What does this have to do with anything? Well, the range() function, in combination with the "for" loop, can come in handy when you need to perform a series of actions a specified number of times,

```
>>> for x in range(1,12):
... print "It is now", x, "o'clock"
...
It is now 1 o'clock
It is now 2 o'clock
It is now 3 o'clock
It is now 4 o'clock
It is now 5 o'clock
It is now 6 o'clock
It is now 7 o'clock
It is now 8 o'clock
It is now 9 o'clock
It is now 10 o'clock
It is now 11 o'clock
>>>
```

---

or to generate indices (corresponding to list elements) in a "for" loop.

---

```
>>> flavours = ["Strawberry", "Blueberry", "Blackcurrant",
"Pineapple",
"Mango", "Grape", "Orange", "Banana"]
>>> for temp in range(2,5): print flavours[temp]
...
Blackcurrant
Pineapple
Mango
>>>
```

---

Here's a more interesting example.

---

```
#!/usr/bin/python

# get a number
num = input("Gimme a number: ")

# for loop
for count in range(2,num):
# if factor exists, num cannot be prime!
if num % count == 0:
print num, "is not a prime number."
```

```
break
else:
# if we get this far, num is prime!
print num, "is a prime number."
```

This is a simple piece of code to test whether or not a number is prime. A "for" loop is used, in conjunction with the range() function, to divide the user−specified number (num) by all numbers within the range 2 to (num). If no factors are found, it implies that the number is a prime number.

Here's what it looks like:

```
Gimme a number: 23
23 is a prime number.
Gimme a number: 45
45 is not a prime number.
Gimme a number: 11
11 is a prime number.
Gimme a number: 111
111 is not a prime number.
```

As with the "while" loop, Python allows you to add an "else" clause to a "for" loop; it is executed only if the loop is completed without encountering a "break" statement even once.

# Just Passin' Through

Python's loops also come with a bunch of control statements, which can be used to modify their behaviour. I've listed the important ones below, together with examples:

**break:**

The "break" keyword is used to exit a loop when it encounters an unexpected situation. A good example of this is the dreaded "division by zero" error − when dividing one number by another one (which keeps decreasing), it is advisable to check the divisor and use the "break" statement to exit the loop as soon as it becomes equal to zero.

```
for x in range(10, -1, -1):

# check for division by zero and exit
if x == 0:
break

print 100 / x
```

In this case, the "break" statement ensures that the loop is terminated whenever an attempt is made to divide by zero. If you'd like to see what happens without the "break" statement, simply comment out the "if" test in the code above.

**continue:**

The "continue" keyword is used to skip a particular iteration of the loop and move back to the top of the loop − it's demonstrated in the following example:

```
for x in range(10):
if x == 7:
continue
print x
```

In this case, Python will print a string of numbers from 1 to 10 − however, when it hits 7, the "continue" statement will cause it to skip that particular iteration and go back to the top of the loop. So your string of numbers will not include 7 − try it and see for yourself.

**pass:**

The "pass" statement essentially means "do nothing". Since Python uses indentation rather than braces to distinguish blocks of code, it generates a syntax error if it doesn't find an expected code block within a loop or conditional statement. The "pass" statement is used as a placeholder in such situations.

```
if var == "neo":
call_neo_func()
elif var == "trinity":
# insert code later
pass
elif var == "agent":
# insert code later
pass
```

In this case, the "pass" statement is used to avoid an error when the program is run (try omitting it and see what happens.)

And that's about it for the moment. In this article, you found out a little more about adding flow control to your Python programs with the "for" and "while" loops, and you also learnt about the ancillary "break", "continue" and "pass" statements. You saw how the range() function can be used to generate number ranges, which can then be used in combination with a "for" loop. And you now know a little more about Python's data structures, after that crash course in list objects.

In the next article, we'll be continuing our tour of built−in Python objects with a look at dictionaries and tuples, powerful and flexible data structures which let you do weird and wonderful things with your code. We'll also re−visit numbers, strings and lists for a look at some more of the functions built into these objects. See you then!