

Scientific Programming in Python
Notes
23 September 2009

Lists

Summary For a list, `x`, the most general slice notation is `x[start:stop:step]`, but all slice arguments are optional. With this notation the values at the following indices will be returned in a new list—`start`, `start+step`, `start+2*step`, etc. up to but *excluding* the `stop` index.

- The first index in a list is 0.
- Negative integers are used to index from the end of the list.
- Leaving out the `start` index defaults to slicing from index 0.
- Leaving out the `stop` index defaults to slicing to the end of the list.
- Leaving out the `step` value defaults to a step size of 1.

A Python list is an ordered sequence of values. These values can be of different types and some of the values may even be lists themselves. The values in a list are enclosed in square brackets (`[]`) and separated by commas.

```
[1, 2, 3, 4, 5]           # A list of integers
[0.4, 1.3, 10, 3.14159265] # A list of integers and floats
["bits", "and", "pieces"] # A list of strings
[3, 3.1416, "pi"]        # A mixed list
[12345, [1, 2, 3, 4, 5], "12345"] # A list that contains another list
```

To access the entries in a list, we use an integer index. The index of the first element is 0.

```
>>> x = [1, 5, 9, 'abcd', ['a', 'b'], 10.5]
>>> print x[3]
abcd
>>> print x[0]
1
>>> print x[4][1]
b
```

We can use negative indices to access values in reverse order. The index of the last element in the list is `-1`. The index of the second to last element is `-2`, and so on.

```
>>> x = [1, 5, 9, 'abcd', ['a', 'b'], 10.5]
>>> print x[-1]
10.5
>>> print x[-2]
['a', 'b']
>>> print x[-2][-1]
b
```

Slicing

A slice from a list is simply an ordered subset. The colon symbol (`:`) is used to define a slice. The notation `s[a:b]` means that we start from index `a` and stop slicing *before* index `b`. **Note that the first index is `a` but the last index is `b-1`.** This is different from most other languages and can cause a lot of confusion if you are not aware of it.

```
>>> x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']
>>> print x[4:10]
['e', 'f', 'g', 'h', 'i', 'j']
```

We can leave out either index to start from the beginning of the list or stop at the end of the list. This returns all of the values starting from index 3.

```
>>> print x[3:]
['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']
```

This returns all of the values before index 3.

```
>>> print x[:3]
['a', 'b', 'c']
```

Finally, we can combine negative indices with the colon operator.

```
>>> print x[-2:] # all values from the second to last index
['l', 'm']
>>> print x[:-2] # all values except the last two
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
>>> print x[-3:-1]
['k', 'l']
>>> print x[2:-2]
['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
```

List operators

- Concatenate two lists (+)

```
>>> [1, 2] + ['a', 'b']
[1, 2, 'a', 'b']
```

- Delete an entry from a list (del)

```
>>> x = [1, 2, 3]
>>> del x[1] # Delete the value at index 1
>>> print x
[1, 3]
```

- Repeat a list (*)

```
>>> [1, 10, 100] * 3
[1, 10, 100, 1, 10, 100, 1, 10, 100]
```

Lists can also be compared using the normal comparison operators. To evaluate a comparison, the lists are compared element by element.

Functions

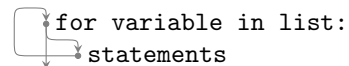
For a list, `x`,

- `len(x)` returns the number of elements in `x`,
- `x.sort()` sorts the list,
- `x.append(value)` adds the element `value` to the end of the list,

- `x.count(value)` counts the number of times that `value` appears in the list,
- `x.index(value)` returns the index of the first element in list that equals `value`,
- `x.insert(index, value)` places the element `value` *before* the given `index`,
- `x.reverse()` reverses the order of the list,
- `x.sum()` computes the sum of the values in `x`.

The for loop

The general form of a `for` loop is shown on the right. Each of the values in the given list is assigned, in order, to the variable exactly once and the statements inside the loop are executed. When the end of the list is reached, the loop ends. The length of the list determines how many times the loop repeats.



The example below uses the `range` function (more on this in the next section) to iterate through each of the indices of a list and to print its values.

```
x = [3, 1, 4, 1]
for i in range(len(x)):
    print "x[" + str(i) + "] =", x[i]
```

The output of this code is

```
x[ 0 ] = 3
x[ 1 ] = 1
x[ 2 ] = 4
x[ 3 ] = 1
```

The following example is a bit more advanced and uses a `for` loop to evaluate a list of functions for one input value.

```
from __future__ import division
from scipy import *

x = 1.25
print "x =", x

for f in [sin, cos, tan]:
    print f.__name__ + "(x) =", f(x)
```

This produces the output below. Notice how easy it would be to evaluate even more functions—simply add more entries to the list in the `for` statement.

```
x = 1.25
sin(x) = 0.948984619356
cos(x) = 0.315322362395
tan(x) = 3.00956967386
```

Generating lists with range

It is often useful to generate series of integers in a `for` loop. These can then be used as indices into other lists. The `range` function is used for this. This function has three forms.

```
range(stop)
range(start, stop)
range(start, stop, step)
```

In the first form, the result is a list of consecutive integers starting at 0 and ending *before* the `stop` argument.

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
```

In the second form, the result is a list of consecutive integers starting from the `start` argument and ending *before* the `stop` argument.

```
>>> range(3, 10)
[3, 4, 5, 6, 7, 8, 9]
```

In the third form, the result is a list of equally spaced integers starting from `start`, stopping before `stop` and spaced by `step`.

```
>>> range(1, 11, 2)
[1, 3, 5, 7, 9]
>>> range(2, 11, 2)
[2, 4, 6, 8, 10]
>>> range(5, -1, -1)
[5, 4, 3, 2, 1, 0]
```

List comprehension

Summary `[expression for variable in list]` produces a new list by iterating over `list` and evaluating `expression` for each value in it.

List comprehension is used to build a new, modified list from another list. The example below shows how we can use a `for` loop to compute the squares of the first 10 positive integers.

```
x = range(1, 11)
y = []
for value in x:
    y.append(value**2)
```

Using list comprehension, we can compute exactly the same result with the following code.

```
x = range(1, 11)
y = [value**2 for value in x]
```

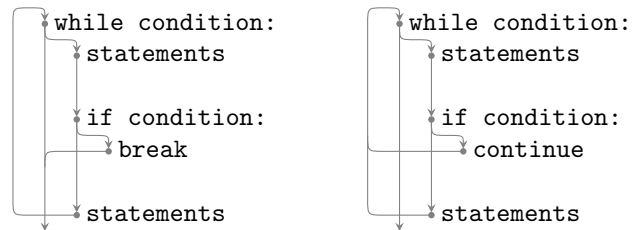
In both examples, the same elements are present. We start from a list `x`, we iterate through the list using a variable `value` and generate a new list `y`. The advantages of the second method is that it is easier to read, uses only one line of code, and is somewhat faster to execute.

The break and continue statements

All the loops have conditions and will terminate when those conditions are false. However, the conditions are always evaluated at the beginning of the loop. We can interrupt a loop in the middle using the `break` and `continue` statements.

- The `break` statement takes us out of the loop block and to the first statement after the loop.
- The `continue` statement takes us out of the loop block and back to the beginning of the loop.

Compare the two structures below to see the difference between `break` and `continue`.



In the next example, the `for` loop will end before all of the values in the list have been used. When `x` is `-0.4`, the `if` condition is `True` and the `break` terminates the loop.

```
from __future__ import division
from scipy import *

x_list = [1, 2.5, 1000, -0.4, 1e-6]
for x in x_list:
    if x <= 0:
        print "Cannot calculate the log of a negative number:", x
        break
    print "log(", x, ") =", log(x)
```