Lists

You've seen lists before in the construction

for j in range(10):

What exactly are these ranges?

```python
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> type(range(6))
<type 'list'>
```

List support the same method of indexing and some of the same functions and operations as strings. (See the table in Section 5.6 of the Python Standard Library Reference for operations supported by all sequence types.)
You have the very useful empty list:

```python
>>> s=[]
>>> len(s)
0
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in
<module>
```

There are a few useful operations you can perform when you have a list of numbers...
But you can make lists of any objects, including objects of different types, some of which may themselves be lists. (The following is a weird example, and I'm not sure why you would do it, but you can.)

```python
>>> s=[3,-7,5,2,11,26]
>>> sum(s)
40
>>> max(s)
26
>>> min(s)
-7
```

It is tempting to think of a string as a list of characters. But there is an important difference: lists are immutable. You can change the contents of individual cells.

```python
>>> import math
>>> t=[2,9,5]
>>> s=['spirit level',math.pi,t,43*75]
>>> s[3]
3225
>>> s[2][2]
5
>>> s[0][5]
't'
>>> s[1]
3.141592653589793
```
You can even do this sort of thing with slices of the list, which allows you to insert and delete elements from the list.

```python
>>> s=range(5,9)
>>> s
[5, 6, 7, 8]
>>> s[2]=-11
>>> s
[5, 6, -11, 8]
```

```python
>>> s=[1,2,3,4,5]
>>> s[5:5]=[6,7,8,9]
>>> s
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[2:5]=[
]
>>> s
[1, 2, 6, 7, 8, 9]
```

There is a bunch of convenient methods for carrying out these and other mutating operations on lists. See the table in Section 5.6.4 of the Standard Library Reference. Here is a brief sample:
There are important differences between these mutating operations and the operations that work on both mutable and immutable sequence types, even if they might appear to have the same effect. The mutating operations modify the object itself; the others create new objects.
As an exercise, try to predict what each of the following statements does. They are all intended to set change the value of `s` from `[1,2,3,4]` to `[1,2,3,4,5]`, but they do not all do this; some of them do something else, and some of them cause a runtime error:

(a) `s.append(5)`  
(b) `s.extend(5)`  
(c) `s.append([5])`  
(d) `s.extend([5])`  
(e) `s = s+5`  
(f) `s=s+[5]`  
(g) `s=s.append(5)`

One consequence of the mutability of lists is that when a list is passed as an argument to a function, the list can be altered as a result of executing the function. This cannot happen with the data types (int, float, str) that
we have seen so far. In class I will draw state diagrams to explain the behavior of the example below.

```
def f(x,y,z):
    x=7
    y=[7]
    z[0]=7

>>> a=6
>>> b=[6]
>>> c=[6]
>>> f(a,b,c)
>>> print a
  6
>>> print b
  [6]
>>> print c
  [7]
```

**List Comprehension**

Suppose you have a list `u` of floats and you want to create a list of the square roots of the numbers in the list. We can do it with the following code.
Python offers a nice shortcut, called list comprehension, inspired by a standard mathematical notation for sets: If \( L \) is a set of numbers, then the set of square roots is denoted by:

\[
\{\sqrt{x} | x \in L\}
\]

In Python, we can mimic this notation and write, in place of our construction of the list \( v \) above

```python
v=[math.sqrt(x) for x in u]
```

As another example, here we take a list of strings and convert them all to upper-case.
We can go a little further with our square root example. Here we use list comprehension with the random number generator to create a list containing both positive and negative values, and then use list comprehension again with 'if' to filter out negative values before finding the square root:

```
u=[random.random()-0.5 for x in range(8)]

>>> u
[0.22717774659875167, 0.1791265809279945, -0.1342798505673375, 0.45442208410686136, -0.11774922039818136, 0.38142814415542703, 0.4954685313885897, -0.1713835869379009]

>>> v=[math.sqrt(x) for x in u if x>=0]

>>> v
[0.476631667641536, 0.423233482758624, 0.6741083622881868, 0.6175986918342906, 0.7038952559781816]
```

The general form for list comprehension is

```
[e(x) for x in my_list if c(x)]
```

where x is a new variable, e is any expression (usually involving x) and c is a boolean-valued expression (also typically involving x). What does [3 for x in range(10)] do? How about [x for x in range(10) if x<0]?