Lists

How lists are like strings

A Python list is a new type. Lists allow many of the same operations as strings. (See the table in Section 4.6 of the Python Standard Library Reference for operations supported by all sequence types, which include lists and strings.) These include indexing, slices, addition for concatenation, multiplication for repetition, and the count and index methods. The Python Shell session below illustrates this.
>>> s=[2,5,-1.3]

>>> len(s)
3

>>> s[2]
-1.3

>>> s[3]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    s[3]
IndexError: list index out of range

>>> s[1:]
[5, -1.3]

>>> s=s+[6,4,-1,2]

>>> s
[2, 5, -1.3, 6, 4, -1, 2]

>>> for num in s[3:6]:
    print(num)

6
4
-1

>>> s=s*2

>>> s
[2, 5, -1.3, 6, 4, -1, 2, 2, 5, -1.3, 6, 4, -1, 2]

>>> s.count(2)
4

Figure 1. Some basic operations on lists; all the operations displayed here can also be performed with strings.
There is also an empty list.

```python
>>> s=[]
>>> len(s)
0
```

*Figure 2. The empty list.*

**How lists are different from strings**

**Lists are heterogeneous**

You can have any type of object as a list element, including another list, and you can even have different types of objects in the same list. The list `s` in the code below has 4 elements, all of different types.
>>> 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

Figure 3. The elements of the list s include a string, an int, a float, and another list.

Lists are mutable

Unlike strings, lists can be modified by assigning new values to their elements:

>>>s=[1,2,3]
>>> s[1]='two'
>>> s
[1, 'two', 3]

Figure 4. Changing an element of a list; recall that this operation cannot be done with strings.

You can even do this sort of thing with slices of the list, which allows you to replace whole ranges of elements with other elements:
Mutability of lists permits a bunch of convenient methods for carrying out these and other such mutating operations on lists. See the table in Section 4.6.4 of the Standard Library Reference. Here is a brief sample.

```python
>>> s=[1,2,3]
>>> s.append(4)
>>> s
[1, 2, 3, 4]
>>> s.extend([5,6,7])
>>> x=s.pop()
>>> x
7
>>> s.insert(0,x)
>>> s
[7, 1, 2, 3, 4, 5, 6]
```

Figure 6. Methods that modify lists. Observe that these methods do not create a new list, and do not return the altered list as a value.

By the way, the append method, which allows you to attach an item to the end of a list, is probably the one that you will use the most.
There are important differences between these mutating operations and some of the string methods we saw, as well as operations, like concatenation, that work on both mutable and immutable sequence types. The mutating operations modify the object itself and do not return the new list as a value. (In contrast, a string method like s.lower() creates a new string, which it returns as a value, and leaves s unchanged.)

```python
>>> s1=[1,2,3]
>>> t1=s1
>>> s1.append(4)
>>> t1
[1, 2, 3, 4]
>>> s2=[1,2,3]
>>> t2=s2
>>> s2=s2+[4]
>>> s2
[1, 2, 3, 4]
>>> t2
[1, 2, 3]
```

Figure 7. After appending a new element, s1 and t1 still point to the same list, which has now been changed. But concatenating s2 with [4] leads to the creation of a new list, while the value of t2 is unchanged.
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)

Figure 8 Test yourself---what do these statements do?

**Special operations on lists**

If the items in a list can be compared to one another with < and >, then built-in functions min and max (which also work for strings) return the minimum and maximum elements of a list.

The method sort sorts the list in place, modifying it in the process.

If the elements in a list are numbers, then the built-in function sum computes the sum of the list elements.

```python
>>> s=[-9,14.2,7.3,1.00003,-46.8]  
>>> min(s)  
-46.8  
>>> max(s)  
14.2  
>>> sum(s)  
-33.299699999999995  
>>> s.sort()  
>>> s  
[-46.8, -9, 1.00003, 7.3, 14.2]
```
Lists as function arguments

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.
Figure 10. Passing a list to a function can change the list if the function applies list-mutating operations.
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
>>> u = [4, 7, 3, 9, 1.5, 2, 4]
>>> import math
>>> v=[]
>>> for x in u:
    v.append(math.sqrt(x))
>>> v
[2.0, 2.6457513110645907,
1.7320508075688772, 3.0, 1.224744871391589,
1.4142135623730951, 2.0]
```

Figure 11. Making a list of the square roots of the elements of a list.

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.

$$\left\{ \sqrt{x} \mid x \in L \right\}$$

In Python, we can mimic this notation and write, in place of our construction of the list $v$ above
As another example, here we take a list of strings and convert them all to upper-case.

```python
>>> strings = ['father', 'mother', 'brother', 'sister', 'cousin', 'uncle', 'aunt']
>>> w = [s.upper() for s in strings]
>>> w
['FATHER', 'MOTHER', 'BROTHER', 'SISTER', 'COUSIN', 'UNCLE', 'AUNT']
```

Figure 13 Another list comprehension example.

We can go a little farther 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:

```python
>>> 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]
```

Figure 14 List comprehension with 'if'.

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

Figure 12. The same operation done in a single line with list comprehension.
The general form for list comprehension is

```
[e(x) for x in my_sequence 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 \( x \)).

\( \text{my_sequence} \) can be an object of any sequence type (a list, a string, a range..)

What does

```
[3 for x in range(10)]
```

do? How about

```
[x for x in range(10) if x<0]
```