Scalar objects – objects without accessible internal structure.

Non-Scalar – structured objects (e.g. strings)

Tuples – ordered sequences of elements (pg. 56). The types of elements can be anything. Here is an example of coding with tuple:

```
t = ()
t1= (1, 'two', 3)
t2 = (t1, 3.25)
```

Tuples can be:

Concatenated

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print (t1 + t2)
```

indexed

```
print (t1 + t2)[3]
```

Sliced

```
print (t1 + t2)[2:5]
```

print (t1 + t2)[3]

produces in the shell...
(1, 'two', 3)

because...

```
T1                  T2
(1, 'two', 3, (1, 'two', 3), 3.25)
 0   1    2     **3**
```

print (t1 + t2)[2:5]

produces in the shell...

(3, (1, 'two', 3), 3.25)

because...

```
T1                  T2
(1, 'two', 3, (1, 'two', 3), 3.25)
 0   1    2     3         4
```

5.2 Lists and Mutability

List: An ordered sequence of values, where each value is identified by an index.

The difference between a list and a tuple:

Use square brackets for **lists** []
Use parentheses for **tuples**

Here is an example of a list:

```
L = ['I did it all', 4, love]
for I in range (len(L)):
        print L[i]
```

produces the output:

```
I did it all
4
love
```

pg. 58 “Lists differ from tuples in one hugely important way: lists are **mutable.** The types **tuple, int, float, and str** are all **<u>immutable</u>.”**

Objects of **immutable** types cannot be modified.

Objects of type **<u>list</u>** CAN BE MODIFIED!!

pg. 58 – 59:
“However, if you keep repeating the mantra 'In Python a variable is merely a name, i.e. a label that can be attached to an object,' it will bring you clarity.”

```
Univs = [Techs, Ivys]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

print 'Univs = ', Univs

produces... Univs = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]

print 'Univs1 = ', Univs1

produces... Univs1 = [['MIT', 'Caltech'], ['Harvard, 'Yale', 'Brown']]

print Univs == Univs1

produces... True

pg. 59
The **append** method has a **side effect.** Rather than create a new list, it mutates the existing list... by adding a new element.

Example: Techs**.append**('RPI')

Aliasing – Two distinct paths to the same list object.

As with tuples, a FOR statement can be used to iterate over the elements of a list. Here is an example:

```
for e in Univs:
        print 'Univs contains', e
        print ' which contains'
        for u in e:
                print '     ', u
```

will print

```
Univs contains ['MIT', 'Caltech', 'RPI']
  which contains
        MIT
        Caltech
        RPI
Univs contains ['Harvard', 'Yale', 'Brown']
 which contains
        Harvard
        Yale
        Brown
```

LIST OF METHODS ASSOCIATED WITH LISTS: (pg. 61)

L. append(e) adds the object e to the end of L

L.count(e) returns the number of times that e occurs in L

L.insert(i, e) inserts the object e into L at index I.

L.extend(L1) append the items in list L1 to the end of L

L.remove(e) deletes the first occurrence of e from L.

L.index(e) returns the index of the first occurrence of e in L.

L.pop(i) remove and return the item at index i. If i is omitted, it defaults to -1.

L.sort() has the side effect of sorting the elements of L.

L.reverse() has the side effect of reversing the order of the elements in L.

LIST COMPREHENSION

```
L = [x**2 for x in range(1,7)]
print L
```

will print the list

[1, 4, 9, 16, 25, 36]

The FOR clause in a list comprehension can be followed by one or more if and for statements that are applied to the values produced by the FOR clause. These additional clauses modify the sequence of values generated by the first FOR clause and produce a new sequence of values, to which the operation associated with the comprehension is applied.

For example,the code
```
mixed = [1, 2, 'a', 3, 4.0]
print [x**2 for x in mixed if type(x) == int]
```

squares the integers in mixed, and then prints [1, 4, 9].

DO NOT WRITE SUBTLE CODE... it will be hard for programmers to see what you are doing.

First-class objects- objects that can be treated like objects of any other type, e.g. int or list.

Unary function - a function that has only one parameter

Unusual explanation of maps on pg. 64

5.4 Strings, tuples, and lists

We have looked at three different sequence types: str, tuple, and list. They are similar in that objects of all of these types can be operated upon...

seq[i] returns the ith element in the sequence

len(seq) returns the length of the sequence.

Seq1 + seq2 concatenates the two sequences.

N * seq returns a sequence that repeats seq n times.

Seq[start:end] returns a slice of the sequence.

E in seq tests whether e is contained in the sequence.

For e in seq iterates over the elements of the sequence.

STR = characters = 'a','b', 'c' **IMMUTABLE**
TUPLE = any type = (), (3,), ('abc', 4) **IMMUTABLE**
LIST = any type = [], [3], ['abc', 4] **MUTABLE**

The following code incrementally builds a list containing all of the even numbers in another list:

```
evenElems = []
for e in L:
   if e%2 == 0:
   evenElems.append(e)
```

```
evenElems = [] #defines variable to be populated
for e in L: #sets up a for loop that will examine all elements in L
   if e%2 == 0: #specifies that if the item in list L is an EVEN number
   evenElems.append(e) #if item is even, append that number to the empty variable above
```

<u>WAYS TO USE STRINGS:</u>

s.count(s1) counts how many times the string s1 occurs in s

s.find(s1) returns the index of the first occurrence of the substring s1 in s, and -1 if s1 is not in s

s.rfind(s1) same as find, but starts at the end of s (the “r” in “rfind” stands for reverse)

s.index(s1) same as find, but raises an exception (see chapter 7) if s1 is not in s.

s.rindex(s1) same as index, but starts from the end of S.

s.lower() converts all uppercase to lowercase

s.replace(old, new) replaces all occurrences of the string old with the string new

s.rstrip() removes trailing white space

s.split(d) splits s using d as a delimiter. Returns a list of substrings of s. For example, the value of
'David Guttag plays basketball'.split(' ') is
['David', 'Guttag', 'plays', 'basketball'].

5.5 DICTIONARIES

Think of a dictionary as a set of key/value pairs. Literals of type **dict** are enclosed in curly braces, and each element is written as a key followed by a colon followed by a value.

For example, the code:

```
monthNumbers = {'Jan':1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5:
'May' }
print 'The third month is ' + monthNumbers[3]
dist = monthNumbers['Apr'] – monthNumbers['Jan']
print 'Apr and Jan are', dist, 'months apart'
```

will print

```
The third month is Mar
Apr and June are 3 months apart
```

The entries in a dict are unordered and cannot be accessed with an index. That's why

monthNumbers[1] unambiguously refers to the entry with the key 1 rather than the second entry.

When a **for** statement is used to iterate over a dictionary, the value assigned to the iteration variable is a key, not a key/value pair. For example, the code

```
keys = []
for e in monthNumbers:
    keys.append(e)
keys.sort()
print keys
```

prints [1, 2, 3, 4, 5, 'Apr', 'Feb', 'Jan', 'May', 3, 4]

*Like **lists, dictionaries** are **mutable.**

We add elements to a dictionary by assigning a value to an unused key, e.g.,

```
    FtoE['Blanc'] = 'white'
```

HERE ARE SOME OF THE MORE USEFUL OPERATORS FOR DICTIONARIES

len(d) returns the number of items in d.

d.keys() returns a list containing the keys in d.

d.values() returns a list containing the values in d.

k in d returns True if key k is in d.

d[k] returns the item in d with key k.

d.get(k, v] returns d[k] if k in d, and v otherwise.

d[k] = v associates the value v with the key k. if there is already a value associated with k, that value is replaced.

del d[k] remove the key k from d.

for k in d iterates over the keys in d.