Python Containers 2

October 26, 2015

Reference: Think Python, p. 115-124, 102-106

Contents

1 Learning Objectives:

The learning objectives for this section are very straightforward:

1. Learn to use tuples
2. Learn to use dictionaries

2 Overview

In this topic we consider the last two of the Python containers we will cover: tuples and dictionaries. We've dealt with tuples already to some degree, but we're going to look at them in more detail.

Tuples and dictionaries are both containers for data, but beyond that there are significant differences. Tuples are immutable sequences, whereas dictionaries are not sequences at all, but are fully changeable. As you'll see, tuples are a straightforward extension of lists, but dictionaries will require that you master a few new concepts.

3 Tuples

These are the last of the three sequence types (strings, lists, and tuples) Another container of ordered components Like lists, the elements of a tuple can be any type and can differ from element to element Unlike lists, tuples are immutable—the contents of a tuple can't be changed! There is no convention on pronunciation. People mostly say it one of two ways, either like the last syllable of quintuple (two-pull), or like the last syllable of multiple (tuh-puhl).

3.1 Tuple creation and assignment

You can create a tuple simply by using a set of comma separated items during assignment:

```
In [1]: a = 117, 155, 222
        print 'a contains the values', a, 'and is type', type(a)
```

```
a contains the values (117, 155, 222) and is type <type ' tuple' >
```

But for stylistic reasons, and for ease of reading we strongly recommend that you use parentheses:

```
In [2]: a = (117, 155, 222)
        print 'a contains the values', a, 'and is type', type(a), 'just as above.'
```

```
a contains the values (117, 155, 222) and is type <type ' tuple' > just as above.
```

Creating a tuple with only a single member requires a special notation, a trailing comma, otherwise Python assumes you've just put a number in brackets.

```
In [4]: a = (117)
        print 'a contains the value', a, 'and is type', type(a)
        b = (117, )
        print 'b contains the value', b, 'and is type', type(b), 'because of the trailing comma.'
```

```
a contains the value 117 and is type <type ' int' >
b contains the value (117,) and is type <type ' tuple' > because of the trailing comma.
```

Tuples can be empty as well. To make things a bit confusing, you don't need a comma here because in the course of normal mathematical operations you'd never expect to see empty brackets.

```
In [5]: a = ()
        print 'a contains the value', a, 'and is type', type(a)
```

```
a contains the value () and is type <type ' tuple' >
```

3.2 Example 8.1:

Tuples can be constructed from other variables as well:

```
In [1]: lastName  = 'Jones'
        firstName = 'Jane'
        midName   = 'P'
        a = (lastName, firstName, midName)
        print a
```

```
(' Jones' , ' Jane' , ' P' )
```

Here we have an excelent case, where the variables could themselves come from a data file (e.g., a CSV, or from user input), but where the tuple's immutability helps us retain a constant order in the naming scheme, meaning that we can always draw from `a[2]` to get the first name. It also means that no one can accidentaly re-assign a new last name to 'Jane P. Jones'. In your program, once the tuple is assigned, a `('r','o','s','e')` is a `('r','o','s','e')` is a `('r','o','s','e')`, and if anyone tries to change it into a `('r','i','s','e')` your program will bomb.

Values used are those at time of tuple creation—there is no dynamic update (same as we saw with lists)

Tuples are also created by the `tuple()` function:

```
In [9]: print tuple('abcd')          #a is ('a','b','c', 'd')
        print tuple([1,2,3,4])       #a is (1,2,3,4)
```

```
(' a' , ' b' , ' c' , ' d' )
(1, 2, 3, 4)
```

Note: the function `tuple()` requires a sequence, so while the first statement here works, the second crashes:

```
In [6]: print (1,2,3)
        print tuple(1,2,3) #bombs---looks to Python like 3 arguments
```

```
(1, 2, 3)
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-6-011db97787b1> in <module>()
          1 print (1,2,3)
    ----> 2 print tuple(1,2,3) #bombs---looks to Python like 3 arguments


        TypeError: tuple() takes at most 1 argument (3 given)
```

Tuples are just a box. A box with imutable members (unless the members themselves contain mutable elements). Tuple members can be of differing types, including tuples:

```
In [ ]: a = ('Jones', 'Jane','P.', 37215,"12-14-192",(424, 17, 5923))
```

The reason I mentioned it was possible to have mutable elements inside tuples is because tuples can contain lists:

```
In [ ]: a = ("Jones", "Jane", 'P.', 37215,"12-14-192", [424, 17, 5923])
```

In this case, while tuples themselves are immutable, any lists contained as tuple elements *are* mutable.

3.3 Accessing Tuples

To access tuple elements we use the same syntax as for strings and lists. Elements can be accessed by index:

```
In [2]: a = (1,2,3,4,5)

        print a[0]
        print a[-2]
        print a[1:3]
        print a[2:]
        print "And then we crash!"
        print a[5]
```

```
1
4
(2, 3)
(3, 4, 5)
And then we crash!


        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-2-c73b566cce67> in <module>()
          6 print a[2:]
          7 print "And then we crash!"
    ----> 8 print a[5]


        IndexError: tuple index out of range
```

We can access nested items within tuples in same way as we saw for lists

```
In [ ]: a = (0, ['x','y','z'], 2, 3)

        b = a[1]      #b is the list ['x','y','z']
        c = a[1][2]    #c is the string 'z'
```

But individual elements within tuples are immutable, except if the element itself is mutable:

```
In [50]: a = (1,2,3)
         a[1] = 4 #error
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-50-eef606f22b4d> in <module>()
  1 a = (1,2,3)
----> 2 a[1] = 4 #error


TypeError: ' tuple'  object does not support item assignment
```

So we can change elements inside a list (below), but we can't actually change the list itself:

```
In [53]: a = (0,[1,2,3],4)
         print 'Original:', a
         a[1][1] = 9   #ok, changes 2 to 9
         print 'Changed:', a
         a[1]    = 9   #error

Original: (0, [1, 2, 3], 4)
Changed: (0, [1, 9, 3], 4)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-53-39db0e808a74> in <module>()
  3 a[1][1] = 9   #ok, changes 2 to 9
  4 print ' Changed:' , a
----> 5 a[1]    = 9   #error


TypeError: ' tuple'  object does not support item assignment
```

3.4 Tuple Operators and Functions

3.4.1 Operations

Concatenation

```
In [54]: a = (1,2,3,4)
         b = a + (5,6,7)
         print b

(1, 2, 3, 4, 5, 6, 7)
```

Replication

```
In [55]: b = a*2
         print b

(1, 2, 3, 4, 1, 2, 3, 4)
```

Membership (in, not in)

```
In [57]: print 3 in a        #True
         print 'z' in a      #False
         print '3' not in a #True
```

```
True
False
True
```

3.4.2 Functions

A number of functions work on tuples:

```
In [43]: print len((1,[2,3,5,6],7)) #returns 3
         print max((10,21,17))    #returns 21
         print min(('a','Z','b') ) #returns 'Z' (ascii Z comes before a-z)
         print sum((1.5, 2., 3.5)) #returns 7.0
```

```
3
21
Z
7.0
```

3.5 Tuple Methods

There are only two native tuple methods (largely because the immutability of tuples means fewer 'manipulation'methods like `.pop`.

```
a.count(object)   #return number times object appears in a
a.index(object)   #return subscript of object, bomb if absent
```

3.6 Example 8.2

```
In [44]: a = (1,5,3,5,0)
         n = a.count(5)
         print n
```

```
2
```

3.7 Example 8.3

```
In [58]: a = (1,5,3,5,0)

         print a.index(0)
         print a.index('3') #this bombs, there is no string  in a
```

```
4
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-58-3b8cd47a12cd> in <module>()
  2
  3 print a.index(0)
```

```
----> 4 print a.index(' 3' ) #this bombs, there is no string  in a


    ValueError: tuple.index(x): x not in tuple
```

3.8 Exercise 8.1:

Write a function to concatenate two tuples and return the concatenation as a new tuple. However, if either tuple is empty, return a tuple containing nothing but a single 0 (a zero inside a tuple).

Go to Exercise 8.1

3.9 When to use tuples vs. lists?

If mutability required, you have no choice: list If you won't modify the contents you can use either, but tuple is the better choice: Prevents accidental change to the object The reader knows the object never changes More efficiently handled internally by Python (though this is probably not a serious concern)

3.10 Sequence Unpacking

You have seen how to pack values into a tuple:

```
In [ ]: aTuple = (1,2,3)
        bTuple = (1, 2., '3', [40,41])
```

It is very common to pack and return function values as tuple so that the `return` value from the function includes multiple elements that are in a fixed position within the return variable.

3.11 Example 8.4:

Create a function that returns the square and cube of an input parameter `x`. Calculate these separately, but return them packed together in a tuple:

```
In [47]: def func(x):
             # func returns the square and cube of a value x.
             xSquared = x*x
             xCubed   = x*x*x
             return (xSquared,xCubed) #return a tuple

         cTuple = func(3)

         print 'The square is always the first element:', cTuple[0], 'and the cube the second:',cTuple[
         print type(cTuple)
```

```
The square is always the first element: 9 and the cube the second: 27
<type ' tuple' >
```

Instead of passing the function output into a single variable you can pass it to multiple comma separated variables. The function output is a tuple, so we expect this to work directly on tuples as well:

```
In [49]: square,cube = func(3)
         print 'The square is always the first element:', square, 'and the cube the second:',cube

         elementone, elementtwo = (12, 24)
         print 'And now we unpack element one of the tuple:', elementone, ' and the second:', elementtwo
```

```
The square is always the first element: 9 and the cube the second: 27
And now we unpack element one of the tuple: 12  and the second: 24
```

3.12 Exercise 8.2:

Write a function to calculate the area and perimeter of a circle with given radius. Return the result using a tuple. Test your function. Retrieve the area and perimeter by unpacking.

Go to Exercise 8.2

Unpacking works for other sequences

```
In [59]: aString = '123'
         a,b,c   = aString

         aList = ['Jones','Paula',42]
         lastName,firstName,age = aList
```

The only rule for unpacking is that the number of variables on the left hand side must match the number of elements in the sequence:

```
In [60]: lastName,firstName,midName, age = aList  #bombs


         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-60-05c763491842> in <module>()
        ----> 1 lastName,firstName,midName, age = aList  #bombs


         ValueError: need more than 3 values to unpack
```

3.13 Example 8.5 (See BoundingBoxExamples.py):

Here things start to get more complicated, and, maybe a bit more 'geography'-ish.

```
In [62]: def boundingBox(pts):
             '''Find bounding box for sequence of points   pts[0], pts[1]...
                For each member p of pts, the x- and y-coordinates
                are p[0] and p[1] respectively'''
             xmin = ymin =  1.e300
             xmax = ymax = -1.e300
             for p in pts:
                xmin = min(xmin,p[0])
                ymin = min(ymin,p[1])
                xmax = max(xmax,p[0])
                ymax = max(ymax,p[1])

             #return corners as list of 4 values [x,y, x,y]
             return [xmin,ymin,xmax,ymax]

         #list of 3 points
         wellLocations = [ [-5,5], [10,-15], [12,3] ]

         print "      points:",wellLocations
         print "bounding box:",boundingBox(wellLocations), '\n'

         llx,lly,urx,ury = boundingBox(wellLocations)
```

```
        print 'lower left  corner: %.0f, %.0f' % (llx,lly)
        print 'upper right corner: %.0f, %.0f\n\n' % (urx,ury)
```

```
points: [[-5, 5], [10, -15], [12, 3]]
bounding box: [-5, -15, 12, 5]

lower left  corner: -5, -15
upper right corner: 12, 5
```

Note the formatting in the final print statements and the unpacking that we do. You don't have to unpack, and, it might be better here to use a tuple, given that the order of the x & y coordinates actually matter here.

Here's another version (look especially at the comment above the `return` statement):

3.14 Example 8.6:

```
In [63]: def boundingBox(pts):
             '''Find bounding box for sequence of points   pts[0], pts[1]...
                For each member p of pts, the x- and y-coordinates
                are p[0] and p[1] respectively'''
             xmin = ymin =  1.e300
             xmax = ymax = -1.e300
             for p in pts:
                xmin = min(xmin,p[0])
                ymin = min(ymin,p[1])
                xmax = max(xmax,p[0])
                ymax = max(ymax,p[1])

             #return corners as list of tuples (2 tuples, not 4 values!!)
             return [(xmin,ymin),(xmax,ymax) ]

         #list of 3 points
         wellLocations = [ [-5,5], [10,-15], [12,3] ]

         print "      points:",wellLocations
         print "bounding box:",boundingBox(wellLocations),'\n'

         ll,ur = boundingBox(wellLocations)

         print 'lower left  corner:', ll
         print 'upper right corner:', ur
```

```
points: [[-5, 5], [10, -15], [12, 3]]
bounding box: [(-5, -15), (12, 5)]

lower left  corner: (-5, -15)
upper right corner: (12, 5)
```

So it makes sense to use tuples in this case. We don't want the elements in the bounding box to get over-written by accident, or else the rest of our mapping will get messed up.

3.15 Exercise 8.3:

Change the code (Example 8.5 or Example 8.6) to return the bounding box with a tuple or nested tuple.

Go to Exercise 8.3

3.15.1 A Quick aside:

The example has a long verbatim text string at start of function definition (`'''Find bounding box...'''`). This string doesn't print out, what's it doing there?

This is called a document string (or Docstring). Python will display that string if we issue built-in command `help(function)`. There are general guidelines issued for Python code here.

3.16 Example 8.7:

```
In [64]: help(boundingBox)

Help on function boundingBox in module __main__:

boundingBox(pts)
    Find bounding box for sequence of points   pts[0], pts[1]...
    For each member p of pts, the x- and y-coordinates
    are p[0] and p[1] respectively
```

4 Dictionaries

Dictionaries are the last container type we will cover (although Python has others, they are much less useful to us). Here are some key points to keep in mind about dictionaries:

- There is no order to dictionary elements, thus we can't call them sequences
- They are the only Python 'mapping type', which refers to access method:

Recall sequences In a sequence (string, tuple, list) we use a numeric index to identify individual elements. The index is offset from the beginning of the sequence, meaning `x[7]` gives us element 8 of the sequence `x` (seven positions away from the first element).

In Python there is no fixed rule about item order in the computer's memory. A list doesn't store objects sequentially in the computer memory, they're stored wherever there's space.

This came up in the Piazza discussion earlier, but Python lists (for example) are actually stored in memory as a set of addresses pointing to a different place in memory where the values are stored. This may seem complicated, but it's a neat trick. It helps Python save memory space, since if can assign the same memory address to two objects with similar values. This kind of blows my mind, and really shows how well planned out Python is. I'll just show you a quick example:

```
In [67]: a = 10
         # This prints the physical memory address of the data in the variable:
         print id(a)

         b = 'Cheese curds'
         c = [a, 10, 'Cheese curds']

         print id(b), 'is different than',id(a), 'because they\'re storing different values.\n'
         print 'But if we look at the addresses of the variables in c:', [id(x) for x in c], '\n'
         print 'we see repetition within the list and between b, c and a.  Cool'

30846052
52434128 is different than 30846052 because they' re storing different values.

But if we look at the addresses of the variables in c: [30846052, 30846052, 52435648]

we see repetition within the list and between b, c and a.  Cool
```

Anyway, we don't worry about that: we give Python an index, it finds the element.

In general there is no relation between the index and the data value. Index [0] isn't more important than index [3], or bigger, or smaller, or whatever. They're just where they are and that's it.

Dictionaries use a 'key'to obtain values rather than a numerical index. This is more like a phone book or a calendar. A person's name in a phone book is the key, the phone number is the value for that entry.

Python implements dictionaries using what is known as a hash table: When given a key Python applies hash function to the key, and uses the result to locate value in a table. All this is Python's job. The structure of the hash table is a black box to us, but it does have implications. In particular, the order of the keys in the hash table may not be the same as the order you assigned them. You'll definitely see this, and it often confuses people.

A key can be a string or numeric, and is usually associated with the data value(s). That is, the key is usually related to the value in a meaningful way. Regardless, we think of dictionary as consisting of key-value pairs. A key is used to 'map'into the container and obtain values, hence the term 'mapping type'.

4.1 Dictionary creation and assignment

The correct syntax for dictionaries are a set of `key:value` pairs separated by commas and enclosed by curly braces:

```
a = {key1:value1, key2:value2, key3:value3 ...}
```

Examples:

```
In [68]: a = {'Sally': 2621804, 'Bob':2634460,  'Jean':2620272}
         print a

         b = {'Sally':'262-1804', 'Bob': '263-4460', 'Jean': '262-0272'}
         print b

         c = {'Madison':(43.1,-89.4), 'Los Angeles':(34.1,-118.1)}
         print c

         d = {0: 'cero', 1: 'uno', 2: 'dos', 3: 'tres', 8: 'ocho'}
         print d

         e = {}  #an empty dictionary
         print e
```

```
{' Bob' : 2634460, ' Sally' : 2621804, ' Jean' : 2620272}
{' Bob' : ' 263-4460' , ' Sally' : ' 262-1804' , ' Jean' : ' 262-0272' }
{' Madison' : (43.1, -89.4), ' Los Angeles' : (34.1, -118.1)}
{0: ' cero' , 1: ' uno' , 2: ' dos' , 3: ' tres' , 8: ' ocho' }
{}
```

See what happened with `b`? The order changed. Why? The hash table's inscrutable logic. It doesn't matter though, because the order of the dictionary doesn't matter. We're going to get things out of it using the key, not the value order.

4.1.1 Rules for `key:value` pairs:

- Keys must be immutable (strings, ints, floats, complex, tuples having immutable elements)
- Keys must be unique—no duplicates w/in dictionary. The interpreter will enforce this by overwriting duplicate keys:

```
In [69]: x = {'key1':1, 'key2':2, 'key1':3}
         print x
```

```
{' key2' : 2, ' key1' : 3}
```

Yonks. If you're not careful, particularly if you're creating large dictionary objects, you might be expecting to have 1 associated with `key1`, but you overwrote it in your dictionary argument.

Take care with numeric keys: 1.0 and 1 are the same (they hash to same value)

- The value in the `key:value` pair can be anything, including other dictionaries:
- Repeated keys? No. Repeated values? Fine.

```
In [ ]: a = {'k1':1, 'k2':2, 'k3':1}
```

The dictionary above has 3 elements, three keys, but only 2 distinct values. That's fine, there's no reason to worry about that.

Keys and values can be of varying type:

```
In [ ]: a = {'k1':1, 2:2, 3: 'three '}
```

Just as we saw the use of the `tuple()` function, the `dict()` function can be used to construct dictionaries from list (or tuple) pairs (a straight sequence won't work):

```
In [72]: aList = [ ['k1',1], ['k2',2], ['k3',3]]
         aDict = dict(aList)
         print aDict

         b = [1,2,3,4]
         bDict = dict(b)
```

```
{' k3' : 3, ' k2' : 2, ' k1' : 1}
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-72-b7a86b226a2d> in <module>()
          4
          5 b = [1,2,3,4]
    ----> 6 bDict = dict(b)


        TypeError: cannot convert dictionary update sequence element #0 to a sequence
```

If `dict()` is given a dictionary the function will simply return a 'shallow'copy of the original. Later on we will see what this means.

4.2 Exercise 8.4:

Construct a dictionary to store the courses you took last semester by course number (e.g., 'GEOG378'). Each course should have a course title and professor name. (Hint: you may want to use tuples to store the information for each course.)

Go to Exercise 8.4

4.2.1 Accessing Dictionary Elements

To illustrate, suppose we need a dictionary that stores ascii characters and their corresponding number. As you know, in the ascii table 'A'is 65, and 65 is 'A'. So this is a small part of the ascii table:

```
In [ ]: asciiDict = {'A':65, 'a':97, 65:'A', 97:'a'}
```

We can do a better job of this using the `string` module and the function `ord`:

```
In [81]: import string
         def string_pairs(x):
           '''A function that takes in a string element and returns its UNICODE value:'''
           return (x, ord(x))

         pairs = [string_pairs(x) for x in string.ascii_lowercase]
         print pairs
         asciiDict = dict(pairs)
         print '\nAnd now the dictionary:\n'
         print asciiDict
```

```
[(' a' , 97), (' b' , 98), (' c' , 99), (' d' , 100), (' e' , 101), (' f' , 102), (' g' , 103), (' h' ,

And now the dictionary:

{' a' : 97, ' c' : 99, ' b' : 98, ' e' : 101, ' d' : 100, ' g' : 103, ' f' : 102, ' i' : 105, ' h' : 10
```

Individual elements in the dictionary can be extracted using `dictName[key]`:

```
In [82]: print asciiDict['a']

         print (asciiDict['a'], asciiDict['b'])

         print 35 + asciiDict['a'] + asciiDict['b']  # adding the integers. . .
```

```
97
(97, 98)
230
```

If the key doesn't exist, your program will bomb with a `KeyError`

```
In [83]: x = asciiDict['Z'] #error
```

```
        ---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call last)

        <ipython-input-83-7aa35936f5da> in <module>()
    ----> 1 x = asciiDict[' Z' ] #error


        KeyError: ' Z'
```

You can create a dictionary piecewise, by assigning values to keys using `dictName[key] = value`.

```
In [84]: asciiDict = {}  #empty
         asciiDict[65] = 'A'
         asciiDict[97] = 'a'
         asciiDict['A'] = 65
         asciiDict['a'] = 97   #asciiDict now has 4 entries

         print asciiDict
```

```
{' A' : 65, 65: ' A' , ' a' : 97, 97: ' a' }
```

Unlike in a tuple where the values are immutable, if the key already exists, the associated value will be changed:

```
In [85]: asciiDict['a'] = 32  #'a' is now paired with 32
         print asciiDict
```

```
{' A' : 65, 65: ' A' , ' a' : 32, 97: ' a' }
```

Individual elements can be deleted using the del statement (note here the key is a number, but not the position of the value):

```
In [86]: del asciiDict[97]  #asciiDict now has just 3 entries
         print asciiDict
```

```
{' A' : 65, 65: ' A' , ' a' : 32}
```

Attempting to delete using a nonexistent key is a KeyError:

```
In [87]: del asciiDict[33]  #error
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-87-357bf6bbd749> in <module>()
----> 1 del asciiDict[33]  #error


KeyError: 33
```

Bottom line: `[key]` is like an index, except that it is a key, not an offset from the start of the dictionary.

4.2.2 Dictionary Operators

Let's build another dictionary for illustration in this section. This time we want a dictionary that has r,g,b values for various colors. Each key is the color name as a string. The values are lists of 3 integers:

```
In [91]: colorDict = { 'red':[255,   0,   0],
                       'grn':[  0, 255,   0],
                       'blu':[  0,   0, 255],
                       'yel':['rabbit']}
```

Dictionaries have no concatenation or replication operator.

Python has comparison operators for dictionaries (`<`, `<=`, ...), but you are unlikely to use them. You can also test membership (`in`, `not in`), but these only refer to the names of the elements, not the values themselves.

4.3 Example 8.8:

```
In [20]: print 'blu' in colorDict              #True
         print 'mauve' in colorDict            #False
         print 'mauve' not in colorDict        #True
         print 'rabbit' in colorDict
         del colorDict['yel']
         print 'After a bit of house cleaning:\n', colorDict
```

```
True
False
True
False
After a bit of house cleaning:
{' grn' : [0, 255, 0], ' blu' : [0, 0, 255], ' red' : [255, 0, 0]}
```

Very important: `in` refers to the keys, not to the values (which is why the 'rabbit'returns False)!

4.4 Example 8.9

You can print all keys of dictionary using a `for` loop:

```
In [21]: for x in colorDict:
                 print x
```

```
grn
blu
red
```

4.5 Example 8.10

Or you can print the values by calling them using their keys:

```
In [22]: for x in colorDict:
                 print colorDict[x]
```

```
[0, 255, 0]
[0, 0, 255]
[255, 0, 0]
```

4.6 Example 8.11:

```
In [89]: for x in colorDict:
                 print x,colorDict[x]
```

```
grn [0, 255, 0]
blu [0, 0, 255]
yel [' rabbit' ]
red [255, 0, 0]
```

In this case `x` is a terrible choice for the loop variable. For clarity's sake use `key` or `k` for the loop variable:

```
In [23]: for key in colorDict:
                 print key
```

```
grn
blu
red
```

-OR-

```
In [24]: for k in colorDict:
                 print colorDict[k]
```

```
[0, 255, 0]
[0, 0, 255]
[255, 0, 0]
```

Basically the same thing, but stylistically a bit cleaner.

4.7 Exercise 8.5:

Using the course dictionary you created above, print out all the courses above 300.

Go to Exercise 8.5

4.8 % More things to know that you don't have to know:

Dictionaries use a special version of format operator % (Don't learn this until you are a formatting expert. I am mentioning this now because you might see it in other people's programs)

Old way using % to format string values looked like this:

```
In [93]: fmtString = 'Red is %s, Green is %s'
         values    = (colorDict['red'],colorDict['grn'])
         print fmtString % values
```

```
Red is [255, 0, 0], Green is [0, 255, 0]
```

Special for dictionaries (note the lack of quotes around the dictionary names here):

```
In [95]: fmtString = 'Red is %(red)s, Green is %(grn)s'
         values    = colorDict
         print fmtString % values
```

```
Red is [255, 0, 0], Green is [0, 255, 0]
```

-or-

```
In [96]: print 'Red is %(red)s, Green is %(grn)s' % colorDict
```

```
Red is [255, 0, 0], Green is [0, 255, 0]
```

4.9 Dictionary Functions

As always, you can use `dir(colorDict)` to get the functions available to any object.

```
In [97]: print len(colorDict) #returns 3
         print hash(x)        #returns an int if x is hashable
                              #(i.e, suitable as key), error otherwise
```

```
4
-1668490562
```

4.10 Dictionary Methods

Let `d` be a dictionary e.g.,

```
In [40]: d = {'key1': 457, 'key2' : 88}

         print 'We\'ve made the dictionary:\n', d

         d.clear()            #empty the dictionary (now d equals {})

         print 'Now we\'ve emptied it:\n', d, '\n'

         d = {'key1': 457, 'key2' : 88}
         print 'Made it again and made a copy:\n', d.copy(), '\n'              #return a copy of d

         import string
         sequence = list(string.lowercase)
         value = (0,0,0)

         d.clear()           #empty the dictionary (now d equals {})
         d = d.fromkeys(sequence, value) # create new dictionary
                                 # using sequence to assign each key
                                 # and value for ALL values (same for all)

         print 'Emptied d, then created a new dictionary from a sequence of characters with a set of va
```

```
We' ve made the dictionary:
{' key2' : 88, ' key1' : 457}
Now we' ve emptied it:
{}

Made it again and made a copy:
{' key2' : 88, ' key1' : 457}

Emptied d, then created a new dictionary from a sequence of characters with a set of values.
{' a' : (0, 0, 0), ' c' : (0, 0, 0), ' b' : (0, 0, 0), ' e' : (0, 0, 0), ' d' : (0, 0, 0), ' g' : (0, 0
```

4.11 Example 8.12:

Here we use a sequence for the key and assign a single value to every single element. Where do you think all those values are stored in memory? Imagine how much space this could save if you're working on very massive projects.

```
In [98]: aDict = {}.fromkeys('abcd',1)
         print aDict
         print [id(aDict[k]) for k in aDict]
```

```
{' a' : 1, ' c' : 1, ' b' : 1, ' d' : 1}
[30846160, 30846160, 30846160, 30846160]
```

Note the print order of elements. This has no relation to order of sequence 'abcd'. Dictionaries aren't ordered like sequences, and so there is no reason for 'b'to precede 'c'. Why does this happen? It has to do with the way Python manages memory, specifically the way it allocates and recovers values from memory. That's as much as I can tell you. . .

Another example, this time using a list rather than a string:

4.12 Example 8.13:

```
In [42]: a = [0,1,2,3]
         aDict = {}.fromkeys(a)
         print aDict

{0: None, 1: None, 2: None, 3: None}
```

`fromkeys()` found a sequence, but not a value to assign and so it used `None` (a special Python value) for all values.

```
d.get(key)          #return value for key, or None if absent
d.get(key,default)   #return value for key, or default if absent
```

4.13 Example 8.14:

```
In [110]: c = colorDict.get('mauve','N/A')
          #c will equal 'N/A'
          print c

          print d.has_key('mauve')        #return True if key exists, False otherwise

N/A
False
```

`has_key()` is now effectively replaced by `in`, but you will see it in older programs and it continues to be used.

List returning methods:

d.items() #return list of (key,value) pairs d.keys() #return list of keys d.values() #return list of values

4.14 Example 8.12

We said that the lists are unsorted because of the hash table, but it is possible to sort them:

```
In [111]: keyList = colorDict.keys()
          keyList.sort()
          for key in keyList:
             print key,colorDict[key]

blu [0, 0, 255]
grn [0, 255, 0]
red [255, 0, 0]
yel [' rabbit' ]
```

```
d.update(d2)         #add all key,value pairs from dictionary d2
                        #to existing dictionary d

d.setdefault(key,val) #like get, but if a key is not present it
                      #adds a new item (key,val) to d
                      #and returns val.  If the key is present, dict is
                      #not changed and None is returned
```

4.15 Example 8.15

This will illustrate the value of using a dictionary rather than a list. We will ask the user for a county name and print the high school graduation rate for that county. Version 1 will use a list to store data. Version 2 will use a dictionary.

```
In [113]: #Version 1 --- database is a list
          #store name, population, pop change (%) and high school grad %-age
          clist  = [ ["Dane",      43826,   8.7, 92.2],
                     ["Bayfield",  15147,   0.9, 86.9],
                     ["Adams",     20843,   4.6, 76.7],
                     ["Milwaukee", 915097, -2.7, 80.2]
                   ]

          #print the database
          print "County data as list:"
          for c in clist: print c

          while True :
              s = raw_input("\nEnter county (Return to exit) >")
              if len(s) < 1 : break

              #loop through the list looking for the county specified by the user
              for c in clist:
                  if c[0] == s:
                      print c[3],"percent high school graduates"
                      break

              else:
                  print s,"is not in database"
```

```
County data as list:
[' Dane' , 43826, 8.7, 92.2]
[' Bayfield' , 15147, 0.9, 86.9]
[' Adams' , 20843, 4.6, 76.7]
[' Milwaukee' , 915097, -2.7, 80.2]

Enter county (Return to exit) >Dane
92.2 percent high school graduates

Enter county (Return to exit) >
```

```
In [114]: #Version 2 --- database is a dictionary
          cdict = {
                    "Dane":     {"Pop": 43826, "PopChange2000-06": 8.7, "HsGrads":92.2},
                    "Bayfield": {"Pop": 15147, "PopChange2000-06": 0.9, "HsGrads":86.9},
                    "Adams":    {"Pop": 20843, "PopChange2000-06": 4.6, "HsGrads":76.7},
                    "Milwaukee":{"Pop":915097, "PopChange2000-06":-2.7, "HsGrads":80.2}
                   }

          print "\n\nCounty data as dictionary:"
          for c in cdict: print "%10s"%c , cdict[c]

          while True :
              s = raw_input("\nEnter county (Return to exit) >")
```

```
            if len(s) < 1 : break

            if s in cdict: print cdict[s]["HsGrads"],"percent high school graduates"

            else         : print s,"is not in database"

County data as dictionary:
  Bayfield {' HsGrads' : 86.9, ' PopChange2000-06' : 0.9, ' Pop' : 15147}
 Milwaukee {' HsGrads' : 80.2, ' PopChange2000-06' : -2.7, ' Pop' : 915097}
      Dane {' HsGrads' : 92.2, ' PopChange2000-06' : 8.7, ' Pop' : 43826}
     Adams {' HsGrads' : 76.7, ' PopChange2000-06' : 4.6, ' Pop' : 20843}

Enter county (Return to exit) >Bayfield
86.9 percent high school graduates

Enter county (Return to exit) >Dane
92.2 percent high school graduates

Enter county (Return to exit) >
```

Note: 1. Value lookup is much easier w/ dictionary, no loop needed 2. You can reference graduation rate by ['HsGrads'] rather than by subscript [3]. The key is much more descriptive than the index.

4.16 Example 8.16

We can store complex data (like a gridded field of temperature values) in a dictionary. We use the (lon,lat) tuple as key to retrieve values.

```
In [115]: import random

          '''This program creates a gridded spatial database of Temperature (T).
          Temperature values are stored in a dictionary. A longitude, latitude tuple
          is used as the dictionary key. The longitude/latitude grid is uniform
          with 1 degree spacing. Temperatures are assigned to each grid point using
          a random number generator.'''

          LonLatT = {}  #dictionary to hold longitude,latitude, and temperature

          for lon in range(-180,180):
             for lat in range(-90,90):
                LonLatT[(lon,lat)] = random.randint(-20,100)

          while True :
              s = raw_input("\nEnter lon,lat (Return to exit) >")
              if len(s) < 1 : break

              s = s.split(',') #split into fields

              lon = round(float(s[0]))
              lat = round(float(s[1]))

              if (lon,lat) in LonLatT:
                  print "T at %d,%d: %d" % (lon,lat,LonLatT[(lon,lat)])

              else:
```

```
                print "Sorry, don't have point within 1/2 degree of %s,%s" \
                      % (s[0],s[1])

Enter lon,lat (Return to exit) >45, -100
Sorry, don' t have point within 1/2 degree of 45, -100

Enter lon,lat (Return to exit) >0, 45
T at 0,45: 36

Enter lon,lat (Return to exit) >34, 45
T at 34,45: 18

Enter lon,lat (Return to exit) >
```

4.17 Exercises 8.6:

We have `d[key]`, so what is the point of `d.get(key)`? Why would one ever use `d.get()`? For example, why would we do this:

```
In [ ]: colorValue = colorDict.get('blu')
```

instead of

```
In [ ]: colorValue = colorDict['blu']
```

Using your course dictionary, write a program to prompt the user for a course number and print out the associated course title and professor.

Go to Exercise 8.6