DICTIONARIES, TUPLES and SETS

Python is renowned for its data types that allow us to store a collection of data. The most popular of these is the list, which you have already seen. A list is a collection of data that occupies cells that are numbered from zero. Now we will see other data types that are related to the list.

A list is not always the best choice for storing a lot of data. For example, let's suppose that we wanted to keep track of the populations of the 50 states and the District of Columbia. It's true that we could use a list for this purpose: a list with 51 cells, each containing the population of one state. Suppose we call this list `pop`. Which state would be stored in pop[0]? It would be intuitive to store the populations in alphabetical order by state. Then, Alabama would be the first state. However, if we alphabetized by the 2-letter abbreviation, then Alaska (AK) would come before Alabama (AL). Somewhere we would need to carefully document which way the data is sorted.

And even if it were clear to us how the states are listed, how would we look up the population of a certain state, such as New Jersey? The cells are numbered, not named. It turns out that if we alphabetize by state abbreviation, then the population of New Jersey is stored in pop[32]. We would somehow have to associate the string "NJ" with the numerical index 32. There are a few ways we can get around this problem. But in Python, the most elegant solution is to use a dictionary instead of a list.

A dictionary is a data type in Python that is similar to a list. It can hold a lot of data. But instead of indexing the cells by numbers starting at zero, we can index by almost any data we want! In other words, with a dictionary, it is legal to reference a cell like this: pop["NJ"]. We don't really care that NJ's population would have been the 33rd number in the list. In a dictionary, the data is not ordered, and not consecutive. In other words, the order in which we create or store the data in the dictionary is irrelevant. We simply associate the string "NJ" with a number representing the population.

Here is a simple dictionary in Python. Let's store the telephone dialing codes for a few countries.

D = { "Finland" : 358, "India" : 91, "USA" : 1, "France" : 33, "New Zealand" : 64 }

This dictionary called D contains five elements. Notice the punctuation of the colon and the comma. The comma separates entries in the dictionary. The colon indicates a key followed by a value. The key "Finland" is associated with the number 358. The key "India" is associated with the number 91. And so on. And we can use the bracket notation just as in a list: D["Finland"] equals 358.

Don't rely on Finland being at the "beginning" of the dictionary. The value 358 is obtained by D["Finland"]. There is no notion of a "first" or "last" or "next" or "previous" element in the dictionary. Yes, we can traverse the dictionary by means of a for-loop, but don't assume any particular order of the elements. Internally, dictionaries are optimized so that Python can quickly find your data.

Creating a dictionary

There are basically two ways to create a dictionary. The example above illustrates one way: you specify all the key-value pairs at once. But it is more likely that you will want to build the dictionary one element at a time. Of course, once a dictionary has been created, you can continue to add elements.

Here is how you can build a dictionary from scratch. We will create an empty dictionary and add two key-value pairs to it:

```
d = { }
d["Tina"] = 3
d["Kevin"] = 2
```

Now, if you ask Python to print the contents of the dictionary, print(d), here is what we get:

```
{'Tina': 3, 'Kevin': 2}
```

The word "in"

With the dictionary, the word "in" is used for looking up a key value. In the previous example, `"Kevin" in d` returns the value True. But `"James" in d` would return False. In fact, if we were to ask for the value of d["James"], this would generate an exception called a KeyError. So, when in doubt, it is wise to verify that the key actually exists in the dictionary before trying to retrieve its corresponding value.

Traversing a dictionary

Here is how we use a for-loop to traverse a dictionary. Let's say the dictionary is called D.

```
for key in D:
    print (key, D[key])
```

In the dialing code example earlier, the output would say:

```
Finland 358
India 91
USA 1
France 33
New Zealand 64
```

Questions:

1. How would you modify the code above to print the keys and values in neat columns? In other words, the dialing codes should be vertically aligned and right justified.

2. How would you modify the code above to print only the key-value pairs for dialing codes that are 2 digit numbers?

Sparse array example

Suppose we wanted to store the values of tuition and room/board at Furman for various years of the past. But we only want to store data for a handful of years, not every year of the university's existence. As an illustration, let's say we wanted to store this data in a dictionary called `cost`:

Year	Tuition	Room&Board
1920	90	230
1951	150	700
1960	650	750
1986	6656	3176
2000	18768	5144
2010	37728	9572

How does this data become a dictionary? The first column, the year, would be the key. And the remaining columns would be the value corresponding to a year. For example, we want to associate both the values $90 and $230 to the year 1920. This means that we want to store a list of 2 elements [90, 230] to the number 1920. We basically want to say: cost[1920] equals [90, 230].

This data would first reside in an input text file. We would read each line of the file, and expect to find three numbers on the line: the year, the tuition amount, and the room/board amount. We would tokenize to isolate these numbers. After reading a line and tokenizing, we would temporarily hold these three values in variables year, tuition, roomBoard. The code to initialize a new element of the dictionary would say:

```
cost[year] = [tuition, roomBoard]
```

In this example, the value at each key is itself a list. In other words, we have a list inside each cell of the dictionary. As mentioned before, cost[1920] would be the list [90, 230]. If we just wanted to see one of these values, we would know that tuition is stored in the [0] element of the list, and the room/board is stored in the [1] element. Therefore, cost[1920][0] would be the tuition and cost[1920][1] would be the room and board charge in 1920.

Dictionary within a dictionary

In the example above, we accessed the tuition value for 1920 with the expression cost[1920][0]. This relies on us remembering that tuition goes in [0] and room & board goes in [1]. We can again use a dictionary to avoid having to remember the order in which the data is stored. It would be more convenient for us to write this expression: cost[1920]["tuition"] to grab the tuition value for 1920.

So, the solution is to store a dictionary within a dictionary. In other words, our cost dictionary will consist of key-value pairs, in which the key is a year number, and the value is a dictionary of two elements, containing both the tuition labeled with the string "tuition" and the room and board amount labeled with the string "roomboard".

As an illustration of this dictionary-within-dictionary idea, let's convert a 2-dimensional list into such a dictionary.

```
L = [[1920,    90,  230], \
     [1951,   150,  700], \
     [1960,   650,  750], \
     [1986,  6656, 3176], \
     [2000, 18768, 5144], \
     [2010, 37728, 9572]]

cost = {}

for year in L:
    key = year[0]
    d = {}
    d["tuition"] = year[1]
    d["roomboard"] = year[2]
    cost[key] = d
```

Now that our `cost` dictionary has been created, we can traverse it and print it out as follows:

```
for year in cost:
    print (year, cost[year]["tuition"], cost[year]["roomboard"])
```

Practice:

Write a loop that will traverse the `cost` dictionary and print the years in which the room & board charge exceeded the tuition.

Copying a dictionary

You may recall that you cannot copy lists by simply assigning one to another using =. In other words, L2 = L for lists means for L and L2 to point to the same list. This is called aliasing, and can be a difficult bug to find. The same is true for dictionaries. If D is a dictionary, you cannot copy it into a new dictionary simply by saying D2 = D. As with lists, you must use a loop, and copy the elements one by one.

Here is an example program where copying dictionaries is useful. We want to have similar lunch and dinner menus. The program performs the following tasks:

- Create a dinner menu.
- Copy each item from the dinner menu into a new lunch menu.
- Add a new item to only the lunch menu.
- Add $1 to the price of each dinner entrée.
- Print out both menus to see the results.

```
dinner = { "ribeye steak" : 42.99, "crab cakes" : 32.99, \
           "stuffed tomato" : 25.99, "pork loin" : 27.99 }

# Let's create a lunch menu that includes another item
lunch = {}

for key in dinner:
    lunch[key] = dinner[key]

# Add new item to lunch
lunch["airline chicken"] = 27.99

# Let's add $1 to the dinner prices.
for key in dinner:
    dinner[key] += 1.00

print("Lunch:")
for key in lunch:
    print (key, lunch[key])

print("\nDinner:")
for key in dinner:
    print (key, dinner[key])
```

The output looks like this:

```
Lunch:
ribeye steak 42.99
crab cakes 32.99
stuffed tomato 25.99
pork loin 27.99
airline chicken 27.99

Dinner:
ribeye steak 43.99
crab cakes 33.99
stuffed tomato 26.99
pork loin 28.99
```

<u>Deleting an element</u>

We probably won't do this often, but it is possible to remove an element from a dictionary. We use the del statement. For example, `del lunch["airline chicken"]` would remove the key-value pair associated with the key value "airline chicken" from the dictionary called `lunch`.

What can the key look like?

In the examples above, you have seen strings used as the key when we stored data in a dictionary. But the key can also be other types as well, such as int, float and boolean. The keys do not all have to be the same type, but it would make sense for them to be. However, complex data types like lists and dictionaries cannot be used as the key. For example, if you wanted to keep track of the tuition and room/board at several colleges, and you wanted to specify values of Furman, you may be tempted to say: cost[["Furman", 1920]] = [90, 230]. However, this is illegal because the key cannot be a list. To get around this limitation, we would use a tuple instead. Tuples are described next.

Practice:

1. Suppose we needed to sort the data inside a dictionary. How could we do it?

2. Show how we can use a dictionary to keep track of a stock portfolio. An input text file has a list of buy and sell orders. A line from this file would have three pieces of information: the word "buy" or "sell", a stock symbol, and a number of shares. Each time a stock symbol appears for the first time, we would create a new entry in the dictionary.

3. Show how a dictionary can help us solve this problem. We want to read the Furman course schedule, and find out when each classroom is in use. What other dictionary-type problems can we formulate based on the data in the course schedule?

4. An operating system keeps track of each time a person logs in to the computer. This login data can be stored in a text file, with one login session stored per line. Each line would show the user's name, and the time of logging in and logging out. How can a dictionary help us maintain the data? What useful questions could we ask of the dictionary?

5. Data from the census can tell us how many people live in each geographical area of the United States. And by "geographical area" this could mean a county, a census tract, a block, or a square grid of latitude and longitude. How can a dictionary help us maintain this data? Consider how we could use the dictionary to determine:
   a. The population density of each county or tract
   b. The total population within 10 miles of a given point

6. What do problems 2-5 have in common that suggests that a dictionary is an appropriate data type to use?

The tuple data type

A tuple is like a list but for the following exceptions:

To display or type out an entire tuple, you would use parentheses instead of square brackets. So, [1, 2, 3, 4] is a list, while (1, 2, 3, 4) is a tuple.

A tuple is immutable. It is not meant to be updated. We cannot change or move around any of the data inside the tuple. We cannot add or remove values from a tuple. You can think of a tuple as a constant list. Because a tuple cannot change, Python can optimize its use. And this is why…

Tuples can be used as the key for a dictionary. ☺ So, earlier when we wanted to have multi-valued keys, we can do this using a tuple: `cost[("Furman", 1920)] = [90, 230]` is legal.

Not as many list functions apply to tuples. We can use: len, max, min, count and index. But we cannot use reverse or sort because that would change the tuple.

Multiple assignment with tuples

Tuples are sometimes used to assign values to several variables. For example, let's say you want to assign 7, 8, and 9 to the variables x, y and z, respectively. You can do so succinctly with a tuple like this:

```
(x, y, z) = (7, 8, 9)
```

Rather than writing three separate assignment statements:

```
x = 7
y = 8
z = 9
```

However, as a matter of programming style many people prefer not to use tuples unnecessarily for this purpose. In fact, once the statement `(x, y, z) = (7, 8, 9)` executes, we have no access to the tuple anymore because we did not assign the tuple itself to a variable.

The set data type

Essentially, a set is simply a list in which the elements do not repeat. For example, let's say you have a favorite author, and you want to collect all the titles written by this author. You don't care if you have two of the same title. You simply want to know which titles you have. This is one example where a set may be appropriate.

We can create a set by using the set() function on a list. For example,

```
L = [ 1, 3, 5, 7, 9 ]
S = set(L)
```

creates a set with five elements. Alternatively, you could simply use the conventional curly brace notation like this:

```
S = { 1, 3, 5, 7, 9 }
```

The same set could have been created if we duplicated some elements:

```
S = { 1, 3, 5, 1, 3, 5, 7, 7, 9, 9 }
```

The duplicate values are ignored. We simply care about which values exist inside the set.

Set operations

The following operations are defined on sets in Python. They are analogous to the standard set operations you are probably already familiar with.

For the examples in the table, let's assume that we have these sets:

S = { 1, 3, 5, 7, 9 }

T = { 1, 2, 3, 4 }

Operator	Meaning	Example
in	Is an element of	4 in S returns False 4 in T returns True
not in	Is not an element of	2 not in S returns True
&	And (intersection)	S & T is the set { 1, 3 }
|	Or (union)	S | T is the set { 1, 2, 3, 4, 5, 7, 9 }
–	And not: Must be in first set and not second	S – T is the set { 5, 7, 9 }
^	Exclusive or: i.e. either set but not both	S ^ T is the set { 2, 4, 5, 7, 9 }
==	Sets are equal	T == { 1, 2, 3, 4 } returns True
!=	Sets are not equal: one set has an element the other does not have	S != T returns True

Note that there is no set complement operation. In other words, it does not make sense to list all data that does not belong to a set.

Practice:

1. Show how sets can be used to help us solve this problem: Both a husband and wife have full-time jobs. They have a young child. On days when both parents must work, child care is needed. On which days can we determine that one parent will be able to stay home? And which of these days is a weekday (Monday – Friday)?

2. Show how sets can be used to help determine when two people can schedule a one-hour meeting tomorrow.