Arrays, Lists, Tuples and Dictionaries

# Arrays

One of the most fundamental data structures in any language is the array. Python doesn't have a native array data structure, but it has the list which is much more general and can be used as a multidimensional array quite easily.

# List basics – Square Brackets

A list in Python is just an ordered collection of items which can be of any type. By comparison an array is an ordered collection of items of a single type - so in principle a list is more flexible than an array but it is this flexibility that makes things slightly harder when you want to work with a regular structure. A list is also a dynamic mutable type and this means you can add and delete elements from the list at any time.

To define a list you simply write a comma separated list of items in square brackets:

months = ['January','February','March','April','May','June',\

'July','August','September','October','November','December']

*Notice that a back slash can be used to continue one line onto the next to make it easier to read in the editor.*

This looks like an array because you can use "slicing" notation to pick out an individual element - indexes start from 0. For example

print (months[2])

will display the third element, i.e. the value *March* in this case. Similarly to change the third element you can assign directly to it:

months[2]=”MAR”

The slicing notation looks like array indexing but it is a lot more flexible. For example

months[2:5]

is a sublist from the third element to the fifth i.e. from months[2] to months[4]. notice that the final element specified i.e. [5] is not included in the slice.

Also notice that you can leave out either of the start and end indexes and they will be assumed to have their maximum possible value. For example

months[5:]

is the list from List[5] to the end of the list and

months[:5]

is the list up to and not including months[5] and

months[:] or months - without any brackets

is the entire list.

List slicing is more or less the same as string slicing except that you can modify a slice. For example:

months[0:2]=[JAN, FEB]

has the same effect as

months[0]=’JAN’
months[1]=’FEB’

Finally is it worth knowing that the list you assign to a slice doesn't have to be the same size as the slice - it simply replaces it even if it is a different size.

# Tuple basics – ‘Round’ Brackets

A tuple is a sequence of immutable (unchangeable) Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use brackets, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between brackets also. For example −

tup1 = ('physics', 'chemistry', 1997, 2000)

tup2 = (1, 2, 3, 4, 5)

tup3 = "a", "b", "c", "d"

The empty tuple is written as two brackets containing nothing −

tup1 = ()

To write a tuple containing a single value you have to include a comma, even though there is only one value −

tup1 = (50,)

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

### Accessing Values in Tuples:

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

tup1 = ('physics', 'chemistry', 1997, 2000)

tup2 = (1, 2, 3, 4, 5, 6, 7)

print "tup1[0]: ", tup1[0]

print "tup2[1:5]: ", tup2[1:5]

When the above code is executed, it produces the following result −

tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]

### Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

tup1 = (12, 34.56)

tup2 = ('abc', 'xyz')

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2

print (tup3)

When the above code is executed, it produces the following result −

(12, 34.56, 'abc', 'xyz')

# Dictionary basics - Curly Brackets

A dictionary is a data structure in Python that allows data to be stored in a way that allows it to be found very quickly. Data is stored using a unique 'Key' followed by the data associated with that key. The structure is not limited to one data type but it can contain a mixture of all data types. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly brackets.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

For example suppose you want to keep track of how old people are. You can do this using a dictionary with almost no additional code.

For example to store a person's age you would use:

age={}
age["Lucy"]=19

The first instruction sets age to be an empty dictionary. After you have created an empty dictionary you can then store something in it, which is what the second line does.

Notice that you have to specify the person's name or some other identifier as part of the assignment. To get Lucy's age back all we have to do is:

print (age["Lucy"])

The dictionary is a machine that when you give it two pieces of data - a key and a value - it stores the value in association with the key. Then, if later you give it the key, it gives you the value you stored earlier.

The value can be any type of data that you care to work with, and this is one of the powerful features of a dictionary.

So to summarize:

A dictionary stores a value in association with a key

You store the value using:

dict[key]=value

You retrieve it using

dict[key]

### Initialisation

That's all there is to a dictionary in principle but in practice there are some minor complications. There is also a lot of similarity between a dictionary and a list - enough to occasionally confuse the beginner.

For example, if you wanted to store Lucy's age in a list you would write:

age\_list[0]=19

and to retrieve the age:

print (age\_list[0])

Notice that in this case the integer zero plays the role of the name "Lucy".

In both cases we are using an index to access the data in a data structure. In the case of a list you can only use integer indexes, but for a dictionary you can use a much wider range of types - although in practice you mostly use strings or integers.

There are some other important differences between lists and dictionaries. In particular you can't use an element of a list unless it has already been created in one way or another, but for a dictionary it is permissible.

ie if age\_list is a list then age\_list[0]=19

will result in a runtime error unless there is already an element zero.

However, age["Lucy"]=19 never causes an error and will either create the new entry or update an existing one.

To make the list assignment work we have to initialize the array so that it already has a zero element. For example, you can create a list with four elements all zero:

age\_list=[0,0,0,0]

age\_list[0]=19

This works, but of course you have to avoid trying to assign to other elements that don't exist such as age\_list[4],

This is a real distinction between lists and dictionaries:

 a list assignment updates existing elements

 a dictionary assignment creates and or updates elements.

To initialize a dictionary you use curly brackets to distinguish it from a list which uses square brackets.

That is:

age={ }

is a dictionary

age=[ ]

is a list

Just as you can initialize a list by writing its elements within square brackets,

eg

age\_list=[19,20,25,18]

you can initialize a dictionary using curly brackets:

age\_dict={"lucy":19,"harry":20",
 "ian":25,"sue":28}

Notice that the initialization is a tiny bit more complicated because you have to specify the key and the value. The way that you do this is to use the notation key:value, but apart from this the two usages are similar. Also notice that we do need quotes around each of the keys because in this case they are strings.

You must specify a key when you store something in a dictionary and nearly all operations with a dictionary involve using a key.

You can, however, use len(dict) which returns the number of items stored in the dictionary, dict. Also del(key) will delete the entry, i.e. the key and the value corresponding to key - but you will get an error if the key isn't present.