Chapter 6

Data Structures

Up to this point we have restricted our data to simple values: numbers and strings. In the real world data is far more complex than this. Consider a program that represents people. The program might need to keep track of physical characteristics, like height and hair color. Or it might need to track personal data, like name, age and social security number. Or it might need student id-numbers and transcripts (which are themselves lists of courses and grades). Most programs that deal with objects in the world store multiple items of simple data for each object. *Data structures* are program constructs that hold multiple items of related data, where each item can be of any type. Python has a rich, easy-to-use set of data structures that will enable you to write sophisticated programs with relatively little effort. In this chapter we will look at three important data structures in Python: lists, tuples and dictionaries.

6.1 Concepts

Data structures allow us to package together multiple items of simple data. For example, we might have a list of numbers: `[2, 3, 5]`, or a list with a name, age and serial number triple: `["Bob", 59, "123-45-6789"]`, or even a list of lists: `[ [1, 2], [3, 4, 5], [6] ]`. You can think of the concept of a list as a template for any sequence of data, of any type whatsoever. All of the structures we will look at in this chapter are templates for assembling simpler data into packages so that one element of the structure might be composed of many individual data items. These structures differ in the mechanisms for how they are created, how data is inserted into them, and how that data can be accessed. Your job as a programmer is to choose the data structure that fits the situation you are coding in the most natural way possible.

There are several concepts that are common to all three of the structures we will look at. One of these concepts is the notion of *indexing* the structure. This is the mechanism through which data can be retrieved from a structure. We have seen indexing before, with strings: if `s` is a string, then `s[i]` is the *ith* letter making up `s`. All three of our new structures are indexed in the same way, using square brackets and index values: `s[i]` is the value in the structure associated with index `i`. What differs from structure to structure is the kind of index the structure allows: some use numeric indexes and some use other kinds of data for indexes.

Another important concept in Python concerns whether a structure is *mutable* or *immutable*. Once immutable structures are created, the values in them cannot be changed. For example, strings are immutable. Suppose we create a string and store it in variable `x` with the statement

```
x = "Hi, Mom"
```

Then `x[1]` refers to the second element of this string, the letter `'i'`, but we are not allowed to write `x[1] = 'o'`; an error will result. We could write

```
x = x + '!'
```

to extend the string, but this creates a new string `"Hi, Mom!"` and stores that in `x`; the old string was not modified.

Lists, on the other hand, are mutable structures. Suppose we start with the list `L = [1, 3, 5]`. Then `L[1]` is the value 3. This time we are allowed to modify the contents of the list:

```
L[1] = 6
```

changes the list to `[1, 6, 5]`. Lists come with an `append()` method that inserts a value onto the end of the list. If `L` is this list `[1, 6, 5]` then

```
L.append(3)
```

turns `L` into `[1, 6, 5, 3]`. There is no `append` operation for our immutable strings. The closest thing to `append` for strings is the concatenation operator +, which creates a new list from the content of two old lists.

One place where this matters is passing a structure to a function. Suppose we want a function `StringAppend(string, tail)` that concatenates the tail on to the string. We would need to write this:

```
def StringAppend(string, tail):
        return string + tail
```

and call it in the following way:

```
s = StringAppend("bob", "by")
```

On the other hand, we could write a list function

```
def AddToEnd(list, x):
        list.append(x)
```

We might call this as follows:

```
L = [1, 3]
AddToEnd(L,5)
```

This changes `L` to the list `[1, 3, 5]`. There is no way for a function to modify a string because strings are immutable, but a function can modify the contents of a list – lists are mutable.