CHAPTER 12

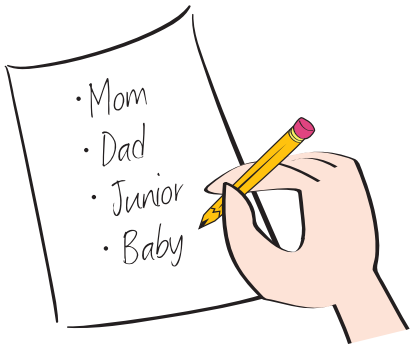
Collecting Things Together—Lists and Dictionaries

We've seen that Python can store things in its memory and retrieve them, using names. So far, we have stored *strings* and *numbers* (both *integers* and *floats*). Sometimes it's useful to store a bunch of things together in a kind of group or *collection*. Then you can do things to the whole collection at once and keep track of groups of things more easily. One of the kinds of collections is a *list*, and another is a *dictionary*. In this chapter, we're going to learn about lists and dictionaries—what they are and how to create, modify, and use them.

Lists are very useful, and they're used in many, many programs. We'll use a lot of them in the examples in upcoming chapters when we start doing graphics and game programming, because the many graphical objects in a game are often stored in a list.

What's a list?

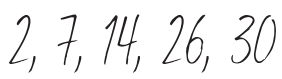
If I asked you to make a list of the members of your family, you might write something like this:



In Python, you'd write this:

```
family = ['Mom', 'Dad', 'Junior', 'Baby']
```

If I asked you to write down your lucky numbers, you might write this:



In Python, you'd write this:

```
luckyNumbers = [2, 7, 14, 26, 30]
```

Both **`family`** and **`luckyNumbers`** are examples of Python lists, and the individual things inside lists are called *items*. As you can see, lists in Python aren't much different from lists you make in everyday life. Lists use square brackets to show where the list starts and ends, and they use commas to separate the items inside.

Creating a list

Both **`family`** and **`luckyNumbers`** are variables. We said before that you can assign different kinds of values to variables. We have already used them for numbers and strings, and they can also be assigned a list.

You create a list like you create any other variable—by assigning something to it, just like we did with **`luckyNumbers`**. You can also create an empty list, like this:

```
newList = []
```

There are no items inside the square brackets, so the list is empty. But what good is an empty list? Why would you want to create one?

Well, quite often, you don't know ahead of time what's going to be in a list. You don't know how many items will be in it, or what those items will be. You just know you'll be using a list to hold them. Once you have an empty list, the program can add things to it. So how do you do that?

Adding things to a list

To add things to a list, you use **`append()`**. Try this in interactive mode:

```
>>> friends = []
>>> friends.append('David')
>>> print friends
```

Makes a new, empty list

Adds an item, "David", to the list

You'll get this result:

```
['David']
```

Try adding another item:

```
>>> friends.append('Mary')
>>> print friends
['David', 'Mary']
```

Remember that you have to create the list (empty or not) before you start adding things to it. It's like if you're making a cake: you can't just start pouring ingredients together—you have to get a bowl out first to pour them into. Otherwise you'll end up with stuff all over the counter!

What's the dot?

Why did we use a dot between **`friends`** and **`append()`**? Well, that starts getting into a pretty big topic: objects. You'll learn more about objects in chapter 14, but for now, here's a simple explanation.

WORD BOX

Append means to add something to the end.

When you append something to a list, you add it to the end of the list.

Many things in Python are *objects*. To do something with an object, you need the object's name (the variable name), then a dot, and then whatever you want to do to the object. So to *append* something to the **`friends`** list, you'd write this:

```
friends.append(something)
```

Lists can hold anything

Lists can hold any kind of data that Python can store. That includes numbers, strings, objects, and even other lists. The items in a list don't have to be the same type or kind of thing. That means a single list can hold both numbers and strings, for example. A list could look like this:

```
my_list = [5, 10, 23.76, 'Hello', myTeacher, 7, another_list]
```

Let's make a new list with something simple, like the letters of the alphabet, so it's easier to see what's going on as we learn about lists. Type this in interactive mode:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
```

Getting items from a list

You can get single items from a list by their index number. The list index starts from 0, so the first item in our list is `letters[0]`:

```
>>> print letters[0]
a
```

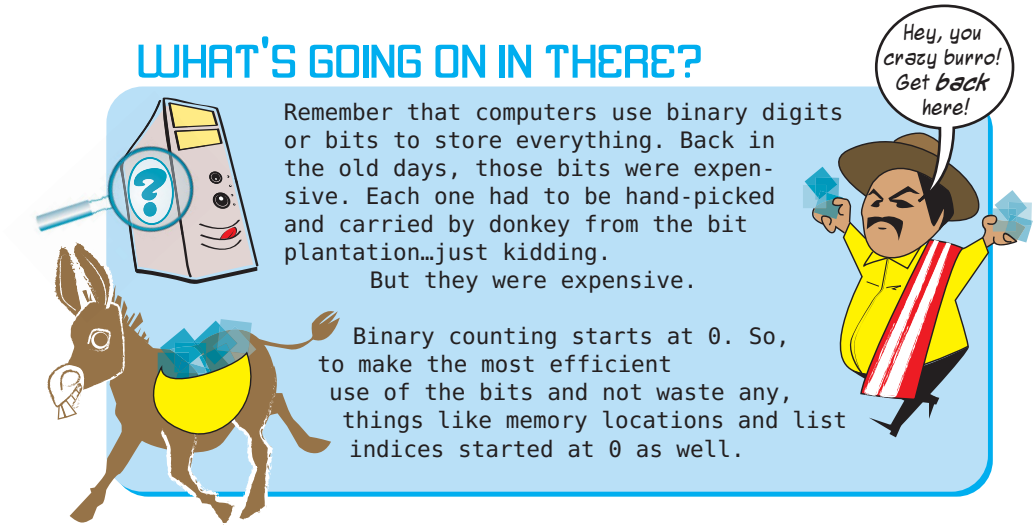
Let's try another one:

```
>>> print letters[3]
d
```

Why does the index start from 0, not 1?

That's a question a lot of programmers, engineers, and computer scientists have argued about since computers were invented. I'm not going to get in the middle of that argument, so let's just say the answer is "because," and move on …

Okay, okay! Have a look at "WHAT'S GOING ON IN THERE?" to see an explanation of why the index starts at 0 instead of 1.



You'll quickly get used to indices starting at 0. It's very common in programming.

BIG FANCY WORD ALERT!

Index means the position of something. The plural of *index* is *indices* (but some people also use *indexes* as the plural for index).

If you're the fourth person in line, your index in line is 4. But if you're the fourth person in a Python list, your index is 3, because Python list indices start at 0!

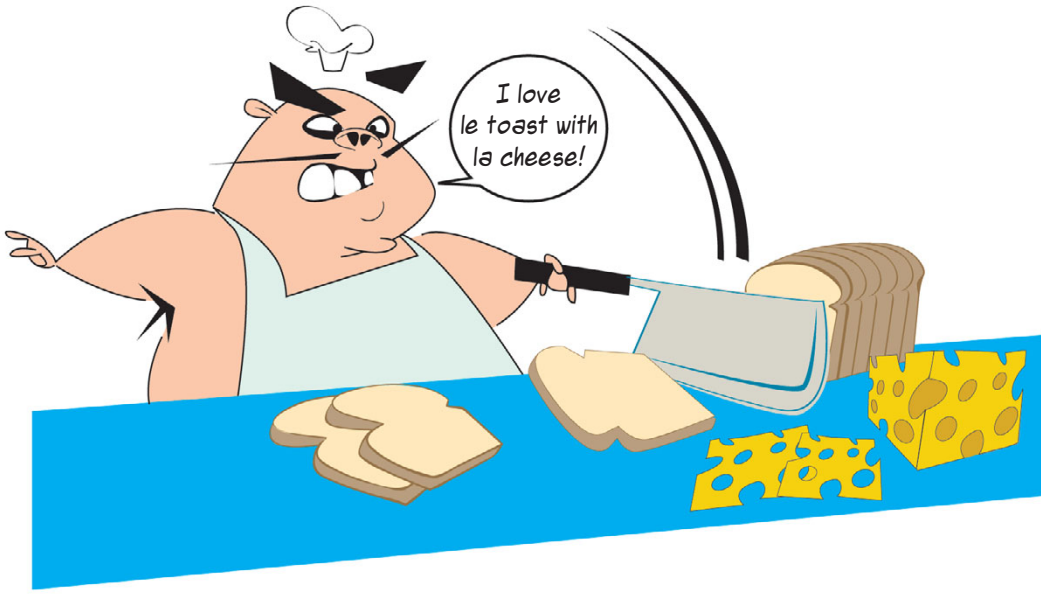
"Slicing" a list

You can also use indices to get more than one item from a list at a time. This is called *slicing* a list:

```
>>> print letters[1:4]
['b', 'c', 'd']
```

Similar to the `range()` in our `for` loops, slicing gets the items starting with the first index, but it stops *before* getting to the second index. That's why we got back three items, not four, in the previous example. One way to remember this is that the number of items you get

back is always the difference between the two index numbers. (4 – 1 = 3, and we got three items back.)



Here's one other thing that is important to remember about slicing a list: What you get back when you slice a list is another (usually smaller) list. This smaller list is called a *slice* of the original list. The original list isn't changed. The slice is a partial *copy* of the original.

Look at the difference here:

```
>>> print letters[1]
b
>>> print letters[1:2]
['b']
```

In the first case, we got back an item. In the second case, we got back a list containing the item. It's a subtle difference, but you need to know about it. In the first case, we used a single index to get one *item* out of the list. In the second case, we used *slice notation* to get a one-item *slice* of the list.

To really see the difference, try this:

```
>>> print type(letters[1])
<type 'str'>
>>> print type(letters[1:2])
<type 'list'>
```

Displaying the `type` of each one tells you for certain that in one case you get a single item (a *string*, in this case), and in the other case you get a *list*.

The smaller list you get back when you slice a list is a copy of items from the original list. That means you can change it and the original list won't be affected.

Slice shorthand

There are some shortcuts you can take when using slices. They don't really save you much typing, but programmers are a lazy bunch, so they use shortcuts a lot. I want you to know what the shortcuts are, so you can recognize them when you see them in other people's code and understand what's going on. That's important, because looking at other people's code and trying to understand it is a good way to learn a new programming language, or programming in general.

If the slice you want includes the start of the list, the shortcut is to use a colon followed by the number of items you want, like this:

```
>>> print letters[:2]
['a', 'b']
```

Notice that there is no number before the colon. This will give you everything from the start of the list up to (but not including) the index you specify.

You can do something similar to get the end of a list:

```
>>> print letters[2:]
['c', 'd', 'e']
```

Using a number followed by a colon gives you everything from the index you specify to the end of the list.

If you don't put any numbers in, and just use a colon, you get the whole list:

```
>>> print letters[:]
['a', 'b', 'c', 'd', 'e']
```

Remember I said that slices make a copy of the original? So `letters[:]` makes a copy of the whole list. This is handy if you want to make some changes to a list but keep the original unchanged.

Modifying items

You can use the index to change one of the list items:

```
>>> print letters
['a', 'b', 'c', 'd', 'e']
>>> letters[2] = 'z'
>>> print letters
['a', 'b', 'z', 'd', 'e']
```

But you can't use the index to add new items to the list. Right now, there are five items in the list, with indices from 0 to 4. So you could *not* do something like this:

```
letters[5] = 'f'
```

It would not work. (Try it if you want.) It's like trying to change something that isn't there yet. To add items to a list, you have to do something else, and that's where we're going next. But before we do, let's change our list back to the way it was:

```
>>> letters[2] = 'c'
>>> print letters
['a', 'b', 'c', 'd', 'e']
```

Other ways of adding to a list

You already saw how to add things to a list using `append()`. But there are other ways. In fact, there are three methods for adding things to a list—`append()`, `extend()`, and `insert()`:

- `append()` adds one item to the end of the list.
- `extend()` adds multiple items to the end of the list.
- `insert()` adds one item somewhere in the list, not necessarily at the end. You tell it where to add the item.

Adding to the end: `append()`

You already saw how `append()` works. It adds one item to the end of a list:

```
>>> letters.append('n')
>>> print letters
['a', 'b', 'c', 'd', 'e', 'n']
```

Let's add one more:

```
>>> letters.append('g')
>>> print letters
['a', 'b', 'c', 'd', 'e', 'n', 'g']
```

Notice that the letters are not in order. That's because `append()` adds the item to the end of the list. If you want the items in order, you'll have to *sort* them. We'll get to sorting very soon.

Extending the list: `extend()`

`extend()` adds several items to the end of a list:

```
>>> letters.extend(['p', 'q', 'r'])
>>> print letters
['a', 'b', 'c', 'd', 'e', 'n', 'g', 'p', 'q', 'r']
```

Notice that what's inside the round brackets of the `extend()` method is a list. A list has square brackets, so for `extend()`, you could have both round and square brackets.

Everything in the list you give to `extend()` gets added to the end of the original list.

Inserting an item: `insert()`

`insert()` adds a single item somewhere in the list. You tell it at what position in the list you want the item added:

```
>>> letters.insert(2, 'z')
>>> print letters
['a', 'b', 'z', 'c', 'd', 'e', 'n', 'g', 'p', 'q', 'r']
```

Here, we added the letter *z* at index 2. Index 2 is the third position in the list (because indices start at 0). The letter that used to be in the third position, *c*, got bumped over by one place, to the fourth position. Every other item in the list also got bumped one position.

The difference between `append()` and `extend()`

Sometimes `append()` and `extend()` look very similar, but they do different things. Let's go back to our original list. First, try using `extend()` to add three items:

```
>>> letters = ['a','b','c','d','e']
>>> letters.extend(['f', 'g', 'h'])
>>> print letters
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Now, we'll try to use `append()` to do the same thing:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters.append(['f', 'g', 'h'])
>>> print letters
['a', 'b', 'c', 'd', 'e', ['f', 'g', 'h']]
```

What happened here? Well, we said before that `append()` adds *one* item to a list. How did it add three? It didn't. It added one item, which happens to be *another list containing three items*. That's why we got the extra set of square brackets inside our list. Remember that a list can hold anything, including other lists. That's what we've got.

`insert()` works the same way as `append()`, except that you tell it where to put the new item. `append()` always puts it at the end.

Deleting from a list

How do you delete or remove things from a list? There are three ways: `remove()`, `del`, and `pop()`.

Deleting with `remove()`

`remove()` deletes the item you choose from the list and throws it away:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> letters.remove('c')
>>> print letters
['a', 'b', 'd', 'e']
```

You don't need to know where in the list the item is. You just need to know it's there somewhere. If you try to remove something that isn't in the list, you'll get an error:

```
>>> letters.remove('f')
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    letters.remove('f')
ValueError: list.remove(x): x not in list
```

So how can you find out if a list contains a certain item? That's coming right up. First, let's look at the other ways to delete something from a list.

Deleting with `del`

`del` lets you delete an item from the list using its index, like this:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> del letters[3]
>>> print letters
['a', 'b', 'c', 'e']
```

Here, we deleted the fourth item (index 3), which was the letter *d*.

Deleting with `pop()`

`pop()` takes the *last* item off the list and gives it back to you. That means you can assign it a name, like this:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> lastLetter = letters.pop()
>>> print letters
['a', 'b', 'c', 'd']
>>> print lastLetter
e
```

You can also use `pop()` with an index, like this:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> second = letters.pop(1)
>>> print second
b
>>> print letters
['a', 'c', 'd', 'e']
```

Here, we popped the second letter (index 1), which was *b*. The item we popped was assigned to `second`, and it was also removed from `letters`.

With nothing inside the parentheses, `pop()` gives you the last item and removes it from the list. If you put a number in the parentheses, `pop(n)` gives you the item at that index and removes it from the list.

Searching a list

Once you have several items in a list, how do you find them? Two things you'll often need to do with a list are

- Find out whether an item is in a list or not
- Find out where an item is in the list (its index)

The `in` keyword

To find out whether something is in a list, you use the `in` keyword, like this:

```
if 'a' in letters:
    print "found 'a' in letters"
else:
    print "didn't find 'a' in letters"
```

The `'a' in letters` part is a *Boolean* or *logical* expression. It'll return the value `True` if *a* is in the list, and `False` otherwise.

WORD BOX

Boolean is a kind of arithmetic that only uses two values: 1 and 0, or true and false. It was invented by mathematician George Boole, and it is used when combining true and false conditions (represented by 1 and 0) together with `and`, `or`, and `not`, like we saw in Chapter 7.

You can try this in interactive mode:

```
>>> 'a' in letters
True
>>> 's' in letters
False
```

This is telling us that the list called **`letters`** does have an item *a*, but it does not have an item *s*. So *a* is in the list, and *s* isn't in the list. Now you can combine **`in`** and **`remove()`**, and write something that won't give you an error, even if the value isn't in the list:

```
if 'a' in letters:
    letters.remove('a')
```

This code only removes the value from the list if the value is in the list.

Finding the index

To find where in the list an item is located, you use the **`index()`** method, like this:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> print letters.index('d')
3
```

So we know that *d* has index 3, which means it's the fourth item in the list.

Just like **`remove()`**, **`index()`** will give you an error if the value isn't found in the list, so it's a good idea to use it with **`in`**, like this:

```
if 'd' in letters:
    print letters.index('d')
```

Looping through a list

When we first talked about loops, you saw that loops iterate through a *list* of values. You also learned about the **`range()`** function and used it as a shortcut for generating lists of numbers for your loops. You saw that **`range()`** gives you a *list* of numbers.

But a loop can iterate through any list—it doesn't have to be a list of numbers. Let's say we wanted to print our list of letters with one item on each line. We could do something like this:

```
>>> letters = ['a', 'b', 'c', 'd', 'e']
>>> for letter in letters:
        print letter

a
b
c
d
e
```

This time, our loop variable is **`letter`**. (Before, we used loop variables like **`looper`** or **`i`**, **`j`**, and **`k`**.) The loop iterates over (loops through) all the values in the list, and each time through, the current item is stored in the loop variable, **`letter`**, and then is displayed.

Sorting lists

Lists are an *ordered* type of collection. This means the items in a list have a certain order, and each one has a place (its index). Once you have put items in a list in a certain order, they stay in that order unless you change the list with **`insert()`**, **`append()`**, **`remove()`**, or **`pop()`**. But that order might not be the order you want. You might want a list *sorted* before you use it.

To sort a list, you use the **`sort()`** method:

```
>>> letters = ['d', 'a', 'e', 'c', 'b']
>>> print letters
['d', 'a', 'e', 'c', 'b']
>>> letters.sort()
>>> print letters
['a', 'b', 'c', 'd', 'e']
```

`sort()` automatically sorts strings alphabetically and numbers numerically, from smallest to largest.

It's important to know that **`sort()`** modifies the list in place. That means it changes the original list you give it. It *does not* create a new, sorted list. That means you can't do this:

```
>>> print letters.sort()
```

If you do, you'll get "None." You have to do it in two steps, like this:

```
>>> letters.sort()
>>> print letters
```

Sorting in reverse order

There are two ways to get a list sorted in reverse order. One is to sort the list the normal way and then *reverse* the sorted list, like this:

```
>>> letters = ['d', 'a', 'e', 'c', 'b']
>>> letters.sort()
>>> print letters
['a', 'b', 'c', 'd', 'e']
>>> letters.reverse()
>>> print letters
['e', 'd', 'c', 'b', 'a']
```

Here you saw a new list method called **`reverse()`**, which reverses the order of items in a list.

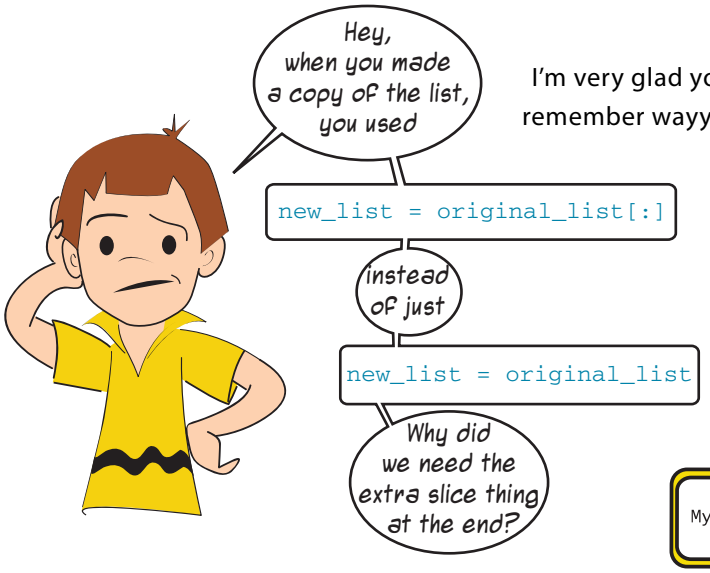
The other way is to add a parameter to **`sort()`** to make it sort in descending order (from largest to smallest):

```
>>> letters = ['d', 'a', 'e', 'c', 'b']
>>> letters.sort (reverse = True)
>>> print letters
['e', 'd', 'c', 'b', 'a']
```

The parameter is called **`reverse`**, and it does exactly what you'd expect—it makes the list sort in reverse order.

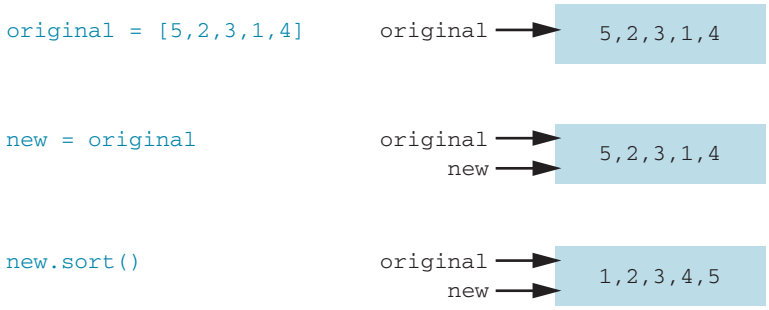
Remember that all the sorting and reversing we just talked about modifies the original list. That means your original order is lost. If you want to preserve the original order and sort a *copy* of the list, you could use slice notation, which we talked about earlier in this chapter, to make a copy—another list equal to the original:

```
>>> original_list = ['Tom', 'James', 'Sarah', 'Fred']
>>> new_list = original_list[:]
>>> new_list.sort()
>>> print original_list
['Tom', 'James', 'Sarah', 'Fred']
>>> print new_list
['Fred', 'James', 'Sarah', 'Tom']
```



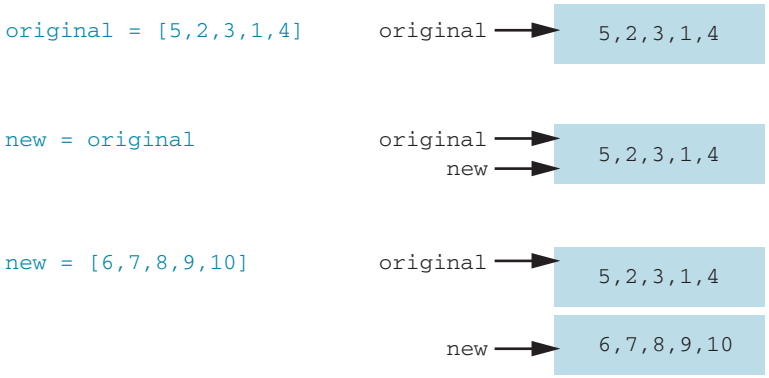
I'm very glad you asked that, Carter. If you remember wayyyyy back when we first talked about names and variables (in chapter 2), we said that when you do something like **`name1 = name2`**, you're just making a new name for the same thing. Remember this picture?

So giving something another name just adds a new tag to the same thing. In Carter's example, **`new_list`** and **`original_list`** both refer to the same list. You can change the list (for example, you can sort it) by using either name. But there is still only *one* list. It looks like this:



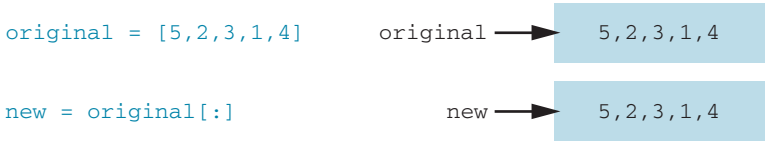
We sorted **`new`**, but **`original`** also got sorted, because **`new`** and **`original`** are two different names for the same list. There are *not* two different lists.

You can, of course, move the **new** tag to a whole new list, like this:



That's the same thing we did with strings and numbers in chapter 2.

This means that, if you really want to make a *copy* of a list, you need to do something different from **new = original**. The easiest way to do this is to use slice notation, like I did above: **new = original[:]**. This means "copy everything in the list, from the first item to the last item." Then you get this:



There are now two separate lists. We made a copy of the original and called it **new**. Now if we sort one list, the other one won't be sorted.

Another way to sort—`sorted()`

There is another way to get a sorted copy of a list without changing the order of the original list. Python has a function called **sorted()** for that purpose. It works like this:

```
>>> original = [5, 2, 3, 1, 4]
>>> newer = sorted(original)
>>> print original
[5, 2, 3, 1, 4]
>>> print newer
[1, 2, 3, 4, 5]
```

The **sorted()** function gives you a *sorted copy* of the original list.

Mutable and immutable

If you remember back to chapter 2, we said that you couldn't actually change a number or string, you could only change what number or string a *name* was assigned to (in other words, move the tag). But lists are one of the types in Python that *can be* changed. As you just saw, lists can have items appended or deleted, and the items can be sorted or reversed.

These two different kinds of variables are called *mutable* and *immutable*. *Mutable* just means "able to be changed" or "changeable." *Immutable* means "not able to be changed" or "unchangeable." In Python, numbers and strings are immutable (cannot be changed), and lists are mutable (can be changed).

Tuple—an immutable list

There are times when you don't want a list to be changeable. So, is there an immutable kind of list in Python? The answer is yes. There is a type called a *tuple*, which is exactly that, an immutable (unchangeable) list. You make one like this:

```
my_tuple = ("red", "green", "blue")
```

You use round brackets, instead of the square ones that lists use.

Because tuples are immutable (unchangeable), you can't do things like sort them or append or delete items. Once you create a tuple with a set of items, it stays that way.

Lists of lists: tables of data

When thinking about how data is stored in a program, it's useful to visualize it.

A variable has a single value.

A list is like a row of values strung together.

Sometimes you need a *table* with rows and columns.

classMarks

	Math	Science	Reading	Spelling
Joe	55	63	77	81
Tom	65	61	67	72
Beth	97	95	92	88

How can you save a table of data? You already know that you can make a list to hold several items. We could put each student's marks in a list, like this:

```
>>> joeMarks = [55, 63, 77, 81]
>>> tomMarks = [65, 61, 67, 72]
>>> bethMarks = [97, 95, 92, 88]
```

Or we could use a list for each subject, like this:

```
>>> mathMarks = [55, 65, 97]
>>> scienceMarks = [63, 61, 95]
>>> readingMarks = [77, 67, 92]
>>> spellingMarks = [81, 72, 88]
```

But you might want to collect all the data together in a single *data structure*.

WORD BOX

A *data structure* is a way of collecting, storing, or representing the data in a program. Data structures can include variables, lists, and some other things we haven't talked about yet. The term *data structure* really refers to the way the data is organized in a program.

To make a single data structure for our class marks, we could do something like this:

```
>>> classMarks = [joeMarks, tomMarks, bethMarks]
>>> print classMarks
[[55, 63, 77, 81], [65, 61, 67, 72], [97, 95, 92, 88]]
```

This gives us a list of items, where each item is itself a list. We have created a *list of lists*. Each of the items in the **classMarks** list is itself a list.

We could also have created **classMarks** directly, without first creating **joeMarks**, **tomMarks**, and **bethMarks**, like this:

```
>>> classMarks = [ [55,63,77,81], [65,61,67,72], [97,95,92,88] ]
>>> print classMarks
[[55, 63, 77, 81], [65, 61, 67, 72], [97, 95, 92, 88]]
```

Now let's try displaying our data structure. **classMarks** has three items, one for each student. So we can just loop through them using **in**:

```
>>> for studentMarks in classMarks:
        print studentMarks

[55, 63, 77, 81]
[65, 61, 67, 72]
[97, 95, 92, 88]
```

Here we looped through the list called **classMarks**. The loop variable is **studentMarks**. Each time through the loop, we print one item in the list. That one item is the marks for a single student, which is itself a list. (We created the student lists above.)

Notice that this looks very similar to the table on the previous page. So we have come up with a data structure to hold all our data in one place.

Getting a single value from the table

How do we get access to values in this table (our list of lists)? We already know that the first student's marks (**joeMarks**) are in a list that is the first item in **classMarks**. Let's check that:

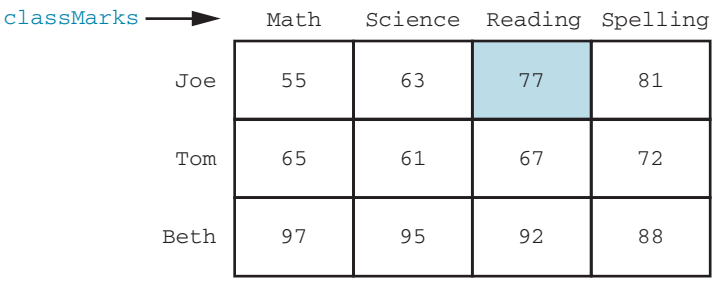
```
>>> print classMarks[0]
[55, 63, 77, 81]
```

classMarks[0] is a list of Joe's marks in the four subjects. Now we want a single value from **classMarks[0]**. How do we do that? We use a second index.

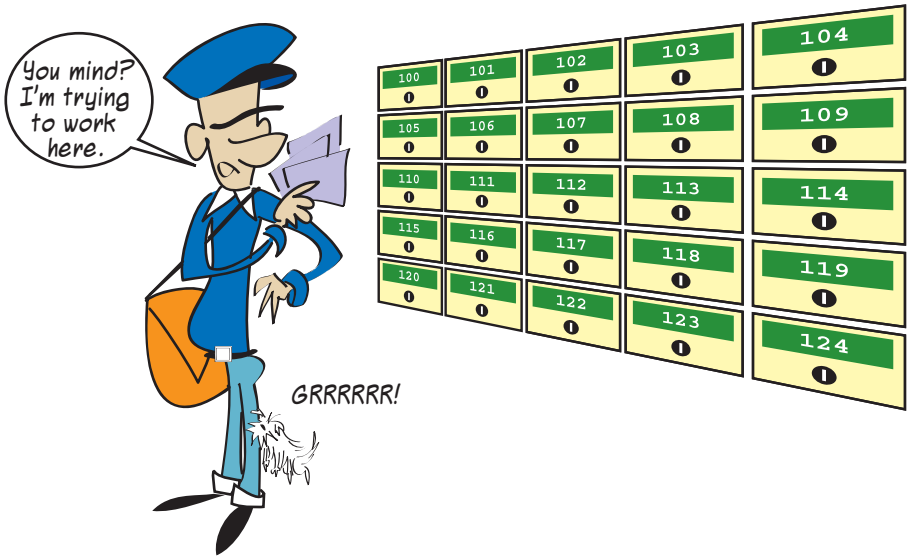
If we want the third of his marks (his Reading mark), which has index 2, we'd do this:

```
>>> print classMarks[0][2]
77
```

This gave us the first item in **classMarks** (index 0), which was the list of Joe's marks, and the third item in that list (index 2), which was his Reading mark. When you see a name with two sets of square brackets, like **classMarks[0][2]**, that is usually referring to a list of lists.



The **classMarks** list doesn't really know about the names Joe, Tom, and Beth, or the subjects Math, Science, Reading, and Spelling. We labeled them that way because we knew what we intended to store in the list. But to Python, they're just numbered places in a list. This is like the numbered mailboxes at a post office. They don't have names on them, just numbers. The postmaster keeps track of what belongs where, and you know which box is yours.



A more accurate way to label the `classMarks` table would be like this:

classMarks →	[0]	[1]	[2]	[3]
classMarks[0]	55	63	77	81
classMarks[1]	65	61	67	72
classMarks[2]	97	95	92	88

Now it's easier to see that the mark 77 is stored in `classMarks[0][2]`.

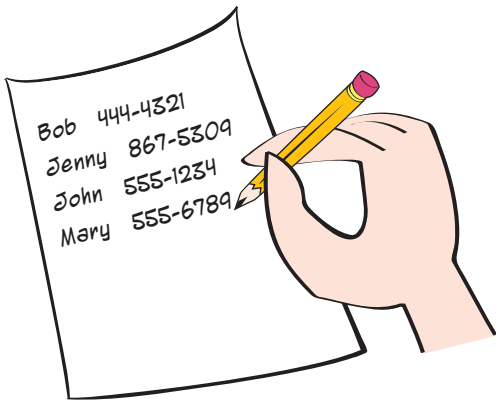
If we were writing a program using `classMarks` to store our data, we'd have to keep track of which data was stored in which row and column. Just like the postmaster, we'd have the job of keeping track of which slot belongs to which piece of data.

Dictionaries

You just saw that a Python list is a way of collecting items together. Quite often in programming you want to collect things together in a way that lets you associate a value with some other value. This is like the way a phone book associates names and phone numbers, or the way a dictionary associates words and their definitions.

A Python *dictionary* is a way of associating two things to each other. These two things are called the *key* and the *value*. Each item or entry in a dictionary has a key and a value. You will hear these referred to as *key-value pairs*. A dictionary is a collection of key-value pairs.

A simple example is a list of phone numbers. Let's say you want to keep a list of your friends' phone numbers. You're going to use their first names to look up the numbers. (Hopefully none of your friends have the same first name.) The name would be the *key* (the thing you'll use to look up the information), and the phone number would be the *value* (the thing you'll look up).



Here's one way to create a Python dictionary to store names and phone numbers. First, let's create the empty dictionary:

```
>>> phoneNumbers = {}
```

This looks very similar to the way you create a list, except you use curly brackets (also called *curly braces* or sometimes just *braces*) instead of the square brackets you use for lists.

Then, let's add an entry:

```
>>> phoneNumbers["John"] = "555-1234"
```

If we then display our dictionary, it looks like this:

```
>>> print phoneNumbers
{'John': '555-1234'}
```

The key is listed first, followed by a colon, and then the value. The quotes are there because both the key and the value happen to be strings in this case (they don't have to be).

Another way to do the same thing is

```
>>> phoneNumbers = {"John": "555-1234"}
```

Let's add some more names. Unlike the `append()` method you use for lists, dictionaries don't have a method for adding new items. You just specify the new key and value:

```
>>> phoneNumbers["Mary"] = "555-6789"
>>> phoneNumbers["Bob"] = "444-4321"
>>> phoneNumbers["Jenny"] = "867-5309"
```

Let's look at the whole dictionary:

```
>>> print phoneNumbers
{'Bob': '444-4321', 'John': '555-1234', 'Mary': '555-6789', 'Jenny': '867-5309'}
```

Now, the whole reason we created a dictionary was so we could look things up. In this case, we want to look something up by name. You do that like this:

```
>>> print phoneNumbers["Mary"]
'555-6789'
```

Notice that you use square brackets to specify which key you want within the dictionary. But the dictionary as a whole is enclosed in curly brackets.

A dictionary is somewhat like a list, but there are a couple of main differences. Both types are *collections*; that is, they are a way of collecting together other types.

Here are some similarities:

- Both lists and dictionaries can hold any type (even lists and dictionaries), so you can have collections of numbers, strings, objects, and even other collections.
- Both lists and dictionaries give you ways to find things in the collection.

And here are some differences:

- Lists are *ordered*. If you put things in a list in a certain order, they stay in that order. And you can sort a list. Dictionaries are *unordered*. If you add things to a dictionary and then display the contents, they may be in a different order than you put them in.
- Items in a list are accessed by their index. Items in a dictionary are accessed by their key:

  ```
  >>> print myList[3]
  'eggs'
  >>> print myDictionary["John"]
  '555-1234'
  ```

As we mentioned before, many things in Python are objects, including lists and dictionaries. Just like lists, dictionaries have some methods you can use to work with them, using the dot notation you saw before.

The `keys()` method gives you a list of all the dictionary keys:

```
>>> phoneNumbers.keys()
['Bob', 'John', 'Mary', 'Jenny']
```

The `values()` method gives you a list of all the values:

```
>>> phoneNumbers.values()
['444-4321', '555-1234', '555-6789', '867-5309']
```

Other languages have things similar to Python dictionaries. They're generally called *associative arrays* (because they *associate* keys and values to each other). Another term you'll hear for them is *hash tables*.

Just like lists, the items in a dictionary can be any type, including simple types (int, float, string) or collections (lists or dictionaries) or compound types (objects).

Yes, you can have dictionaries that contain other dictionaries, just like you can have lists of lists. Actually, that's not entirely true. It is true for the *values* in a dictionary, but the *keys* are more restricted. Earlier we talked about *mutable* versus *immutable* types. Well, dictionary keys can only be immutable types (booleans, integers, floats, strings, and tuples). You can't use a list or a dictionary as a key, because these are mutable types.

I mentioned above that one of the things about dictionaries that's different from lists is that dictionaries are *unordered*. Notice that even though Bob's number was the third one we added to the dictionary, it was the first item when we displayed the contents of the dictionary. Dictionaries have no concept of order, so sorting a dictionary makes no sense. But sometimes you want to display the contents of a dictionary in some kind of order. Remember that lists *can* be sorted, so once you get a list of the keys, you can sort that and then display the dictionary in order of its keys. You can sort the list of keys using the `sorted()` function, like this:

```
>>> for key in sorted(phoneNumbers.keys()):
        print key, phoneNumbers[key]

Bob 444-4321
Jenny 867-5309
John 555-1234
Mary 555-6789
```

That's the same `sorted()` function you saw before for lists. If you think about it, this makes sense, because the collection of a dictionary's keys *is* a list.

What if you want to display the items in order of the values instead of the keys? In our phone numbers example, that would mean sorting by the phone numbers, from lowest number to highest number. Well, a dictionary is really a one-way lookup. It is meant to look up values using the keys, not the other way around. So it's a little more difficult to sort by the values. It's possible—it just takes a bit more work:

```
>>> for value in sorted(phoneNumbers.values()):
        for key in phoneNumbers.keys():
            if phoneNumbers[key] == value:
                print key, phoneNumbers[key]

Bob 444-4321
John 555-1234
Mary 555-6789
Jenny 867-5309
```

Here, once we got the sorted list of values, we took each value and found its key by looping through all the keys until we found the one that was associated to that value.

Here are a few other things you can do with dictionaries:

- Delete an item using **`del`**:

```
>>> del phoneNumbers["John"]
>>> print phoneNumbers
{'Bob': '444-4321', 'Mary': '555-6789', 'Jenny': '867-5309'}
```

- Delete all items (clear the dictionary) using **`clear()`**:

```
>>> phoneNumbers.clear()
>>> print phoneNumbers
{}
```

- Find out if a key exists in the dictionary using **`in`**:

```
>>> phoneNumbers = {'Bob': '444-4321', 'Mary': '555-6789', 'Jenny': '867-5309'}
>>> "Bob" in phoneNumbers
True
>>> "Barb" in phoneNumbers
False
```

Dictionaries are used in lots of Python code.

This certainly isn't a comprehensive overview of Python dictionaries. But it should give you the general idea so you can start using them in your code and recognize them when you see them in other code.

What did you learn?

In this chapter, you learned

- What lists are
- How to add things to a list
- How to delete things from a list
- How to find out if a list contains a certain value
- How to sort a list
- How to make a copy of a list
- About tuples
- About lists of lists
- About Python dictionaries

Test your knowledge

1. What are two ways to add something to a list?
2. What are two ways to remove something from a list?
3. What are two ways to get a sorted copy of a list, without changing the original list?
4. How do you find out whether a certain value is in a list?
5. How do you find out the location of a certain value in a list?
6. What's a tuple?
7. How do you make a list of lists?
8. How do you get a single value from a list of lists?
9. What is a dictionary?
10. How do you add an item to a dictionary?
11. How do you look up an item from its key?

Try it out

1. Write a program to ask the user for five names. The program should store the names in a list and print them all out at the end. It should look something like this:

```
Enter 5 names:
Tony
Paul
Nick
Michel
Kevin
The names are Tony Paul Nick Michel Kevin
```

2. Modify the program from question #1 to print both the original list of names and a sorted list.

3 Modify the program from question #1 to display only the third name the user typed in, like this:

```
The third name you entered is:  Nick
```

4 Modify the program from question #1 to let the user replace one of the names. She should be able to choose which name to replace and then type in the new name. Finally, display the new list like this:

```
Enter 5 names:
Tony
Paul
Nick
Michel
Kevin
The names are Tony Paul Nick Michel Kevin
Replace one name.  Which one? (1-5): 4
New name: Peter
The names are Tony Paul Nick Peter Kevin
```

5 Write a dictionary program that lets users enter certain words and definitions and then look them up later. Make sure you let the user know if their word isn't in the dictionary yet. It should look something like this when it runs:

```
Add or look up a word (a/l)? a
Type the word: computer
Type the definition: A machine that does very fast math
Word added!
Add or look up a word (a/l)? l
Type the word: computer
A machine that does very fast math
Add or look up a word (a/l)? l
Type the word: qwerty
That word isn't in the dictionary yet.
```