CHAPTER 2

Lists and Tuples

This chapter introduces a new concept: *data structures*. A data structure is a collection of data elements (such as numbers or characters, or even other data structures) that is structured in some way, such as by numbering the elements. The most basic data structure in Python is the *sequence*. Each element of a sequence is assigned a number—its position, or *index*. The first index is zero, the second index is one, and so forth.

Note When you count or number things in your daily life, you probably start counting from 1. The numbering scheme used in Python may seem odd, but it is actually quite natural. One of the reasons for this, as you see later in the chapter, is that you can *also* count from the end: the last item of a sequence is numbered –1, the next-to-last –2, and so forth. That means you can count forward *or* backward from the first element, which lies at the beginning, or 0. Trust me, you get used to it.

This chapter begins with an overview of sequences, and then covers some operations that are common to all sequences, including lists and tuples. These operations will also work with strings, which will be used in some of the examples, although for a full treatment of string operations, you have to wait until the next chapter.

After dealing with these basics, we start working with lists and see what's special about them. After lists, we come to tuples, which are very similar to lists, except that you can't change them.

Sequence Overview

Python has six built-in types of sequences. This chapter concentrates on two of the most common ones: *lists* and *tuples*. The other built-in sequence types are strings (which I revisit in the next chapter), Unicode strings, buffer objects, and `xrange` objects.

The main difference between lists and tuples is that you can change a list, but you can't change a tuple. This means a list might be useful if you need to add elements as you go along, while a tuple can be useful if, for some reason, you can't allow the sequence to change. Reasons for the latter are usually rather technical, having to do with how things work internally in Python. That's why you may see built-in functions returning tuples. For your own programs, chances are you can use lists instead of tuples in almost all circumstances. (One notable exception, as described in Chapter 4, is using tuples as dictionary keys. There lists aren't allowed, because you aren't allowed to modify keys.)

Sequences are useful when you want to work with a collection of values. You might have a sequence representing a person in a database, with the first element being their name, and the second their age. Written as a list (the items of a list are separated by commas and enclosed in square brackets), that would look like this:

```
>>> edward = ['Edward Gumby', 42]
```

But sequences can contain other sequences, too, so you could make a list of such persons, which would be your database:

```
>>> edward = ['Edward Gumby', 42]
>>> john = ['John Smith', 50]
>>> database = [edward, john]
>>> database
[['Edward Gumby', 42], ['John Smith', 50]]
```

■ **Note** Python has a basic notion of a kind of data structure called a *container*, which is basically any object that can contain other objects. The two main kinds of containers are sequences (such as lists and tuples) and mappings (such as dictionaries). While the elements of a sequence are numbered, each element in a mapping has a name (also called a key). You learn more about mappings in Chapter 4. For an example of a container type that is neither a sequence nor a mapping, see the discussion of sets in Chapter 10.

Common Sequence Operations

There are certain things you can do with all sequence types. These operations include *indexing*, *slicing*, *adding*, *multiplying*, and checking for *membership*. In addition, Python has built-in functions for finding the length of a sequence, and for finding its largest and smallest elements.

■ **Note** One important operation not covered here is *iteration*. To iterate over a sequence means to perform certain actions repeatedly, once per element in the sequence. To learn more about this, see the section "Loops" in Chapter 5.

Indexing

All elements in a sequence are numbered—from zero and upwards. You can access them individually with a number, like this:

```
>>> greeting = 'Hello'
>>> greeting[0]
'H'
```

■ **Note** A string is just a sequence of characters. The index 0 refers to the first element, in this case the letter *H*.

This is called *indexing*. You use an index to fetch an element. All sequences can be indexed in this way. When you use a negative index, Python counts *from the right*; that is, from the last element. The last element is at position –1 (not –0, as that would be the same as the first element):

```
>>> greeting[-1]
'o'
```

String literals (and other sequence literals, for that matter) may be indexed directly, without using a variable to refer to them. The effect is exactly the same:

```
>>> 'Hello'[1]
'e'
```

If a function call returns a sequence, you can index it directly. For instance, if you are simply interested in the fourth digit in a year entered by the user, you could do something like this:

```
>>> fourth = raw_input('Year: ')[3]
Year: 2005
>>> fourth
'5'
```

Listing 2-1 contains a sample program that asks you for a year, a month (as a number from 1 to 12), and a day (1 to 31), and then prints out the date with the proper month name and so on.

Listing 2-1. *Indexing Example*

```
# Print out a date, given year, month, and day as numbers

months = [
    'January',
    'February',
    'March',
    'April',
    'May',
    'June',
    'July',
    'August',
    'September',
    'October',
    'November',
    'December'
]
```

```
# A list with one ending for each number from 1 to 31
endings = ['st', 'nd', 'rd'] + 17 * ['th'] \
        + ['st', 'nd', 'rd'] +  7 * ['th'] \
        + ['st']

year    = raw_input('Year: ')
month   = raw_input('Month (1-12): ')
day     = raw_input('Day (1-31): ')

month_number = int(month)
day_number = int(day)

# Remember to subtract 1 from month and day to get a correct index
month_name = months[month_number-1]
ordinal = day + endings[day_number-1]

print month_name + ' ' + ordinal + ', ' + year
```

An example of a session with this program might be as follows:

```
Year: 1974
Month (1-12): 8
Day (1-31): 16
August 16th, 1974
```

The last line is the output from the program.

Slicing

Just as you use indexing to access individual elements, you can use *slicing* to access *ranges* of elements. You do this by using *two* indices, separated by a colon:

```
>>> tag = '<a href="http://www.python.org">Python web site</a>'
>>> tag[9:30]
'http://www.python.org'
>>> tag[32:-4]
'Python web site'
```

As you can see, slicing is very useful for extracting parts of a sequence. The numbering here is very important. The *first* index is the number of the first element you want to include. However, the *last* index is the number of the first element *after* your slice. Consider the following:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[3:6]
[4, 5, 6]
>>> numbers[0:1]
[1]
```

In short, you supply two indices as limits for your slice, where the first is *inclusive* and the second is *exclusive*.

A Nifty Shortcut

Let's say you want to access the last three elements of numbers (from the previous example). You could do it explicitly, of course:

```
>>> numbers[7:10]
[8, 9, 10]
```

Now, the index 10 refers to element 11—which does not exist, but is one step after the last element you want. Got it?

This is fine, but what if you want to count from the end?

```
>>> numbers[-3:-1]
[8, 9]
```

It seems you cannot access the last element this way. How about using 0 as the element "one step beyond" the end?

```
>>> numbers[-3:0]
[]
```

Not exactly the desired result. In fact, any time the leftmost index in a slice comes later in the sequence than the second one (in this case, the third-to-last coming later than the first), the result is always an empty sequence. Luckily, you can use a shortcut: if the slice continues to the end of the sequence, you may simply leave out the last index:

```
>>> numbers[-3:]
[8, 9, 10]
```

The same thing works from the beginning:

```
>>> numbers[:3]
[1, 2, 3]
```

In fact, if you want to copy the entire sequence, you may leave out *both* indices:

```
>>> numbers[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Listing 2-2 contains a small program that prompts you for a URL, and (assuming it is of the form `http://www.somedomainname.com`) extracts the domain name.

Listing 2-2. *Slicing Example*

```
# Split up a URL of the form http://www.something.com

url = raw_input('Please enter the URL: ')
domain = url[11:-4]

print "Domain name: " + domain
```

Here is a sample run of the program:

```
Please enter the URL: http://www.python.org
Domain name: python
```

Longer Steps

When slicing, you specify (either explicitly or implicitly) the start and end points of the slice. Another parameter (added to the built-in types in Python 2.3), which normally is left implicit, is the step length. In a regular slice, the step length is one, which means that the slice "moves" from one element to the next, returning all the elements between the start and end:

```
>>> numbers[0:10:1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In this example, you can see that the slice includes another number. This is, as you may have guessed, the step size, made explicit. If the step size is set to a number greater than one, elements will be skipped. For example, a step size of two will include only every other element of the interval between the start and the end:

```
>>> numbers[0:10:2]
[1, 3, 5, 7, 9]
numbers[3:6:3]
[4]
```

You can still use the shortcuts mentioned earlier. For example, if you want every fourth element of a sequence, you need to supply only a step size of four:

```
>>> numbers[::4]
[1, 5, 9]
```

Naturally, the step size can't be zero—that wouldn't get you anywhere—but it *can* be *negative*, which means extracting the elements from right to left:

```
>>> numbers[8:3:-1]
[9, 8, 7, 6, 5]
>>> numbers[10:0:-2]
[10, 8, 6, 4, 2]
>>> numbers[0:10:-2]
[]
>>> numbers[::-2]
[10, 8, 6, 4, 2]
>>> numbers[5::-2]
[6, 4, 2]
>>> numbers[:5:-2]
[10, 8]
```

Getting things right here can involve a bit of thinking. As you can see, the first limit (the leftmost) is still *inclusive*, while the second (the rightmost) is *exclusive*. When using a negative

step size, you need to have a first limit (start index) that is *higher* than the second one. What may be a bit confusing is that when you leave the start and end indices implicit, Python does the "right thing"—for a positive step size, it moves from the beginning toward the end, and for a negative step size, it moves from the end toward the beginning.

Adding Sequences

Sequences can be concatenated with the addition (plus) operator:

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello, ' + 'world!'
'Hello, world!'
>>> [1, 2, 3] + 'world!'
Traceback (innermost last):
  File "<pyshell#2>", line 1, in ?
    [1, 2, 3] + 'world!'
TypeError: can only concatenate list (not "string") to list
```

As you can see from the error message, you can't concatenate a list and a string, although both are sequences. In general, you cannot concatenate sequences of different types.

Multiplication

Multiplying a sequence by a number *x* creates a new sequence where the original sequence is repeated *x* times:

```
>>> 'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

None, Empty Lists, and Initialization

An empty list is simply written as two brackets ([])—there's nothing in it. But what if you want to have a list with room for ten elements but with nothing useful in it? You could use `[42]*10`, as before, or perhaps more realistically `[0]*10`. You now have a list with ten zeros in it. Sometimes, however, you would like a value that somehow means "nothing," as in "we haven't put anything here yet." That's when you use `None`. `None` is a Python value and means exactly that—"nothing here." So if you want to initialize a list of length 10, you could do the following:

```
>>> sequence = [None] * 10
>>> sequence
[None, None, None, None, None, None, None, None, None, None]
```

Listing 2-3 contains a program that prints (to the screen) a "box" made up of characters, which is centered on the screen and adapted to the size of a sentence supplied by the user. The code may look complicated, but it's basically just arithmetic—figuring out how many spaces, dashes, and so on you need in order to place things correctly.

Listing 2-3. *Sequence (String) Multiplication Example*

```
# Prints a sentence in a centered "box" of correct width

# Note that the integer division operator (//) only works in Python
# 2.2 and newer. In earlier versions, simply use plain division (/)


sentence = raw_input("Sentence: ")

screen_width = 80
text_width   = len(sentence)
box_width    = text_width + 6
left_margin  = (screen_width - box_width) // 2

print
print ' ' * left_margin + '+'   + '-' * (box_width-2)  +   '+'
print ' ' * left_margin + '|  ' + ' ' * text_width     + '  |'
print ' ' * left_margin + '|  ' +       sentence       + '  |'
print ' ' * left_margin + '|  ' + ' ' * text_width     + '  |'
print ' ' * left_margin + '+'   + '-' * (box_width-2)  +   '+'
print
```

The following is a sample run:

```
Sentence: He's a very naughty boy!

                    +--------------------------+
                    |                          |
                    | He's a very naughty boy! |
                    |                          |
                    +--------------------------+
```

Membership

To check whether a value can be found in a sequence, you use the `in` operator. This operator is a bit different from the ones discussed so far (such as multiplication or addition). It checks whether something is true and returns a value accordingly: `True` for true and `False` for false. Such operators are called *Boolean operators*, and the truth values are called *Boolean values*. You learn more about Boolean expressions in the section on conditional statements in Chapter 5.

Here are some examples that use the `in` operator:

```
>>> permissions = 'rw'
>>> 'w' in permissions
True
>>> 'x' in permissions
```

```
False
>>> users = ['mlh', 'foo', 'bar']
>>> raw_input('Enter your user name: ') in users
Enter your user name: mlh
True
>>> subject = '$$$ Get rich now!!! $$$'
>>> '$$$' in subject
True
```

The first two examples use the membership test to check whether `'w'` and `'x'`, respectively, are found in the string `permissions`. This could be a script on a UNIX machine checking for writing and execution permissions on a file. The next example checks whether a supplied user name (`mlh`) is found in a list of users. This could be useful if your program enforces some security policy. (In that case, you would probably want to use passwords as well.) The last example checks whether the string `subject` contains the string `'$$$'`. This might be used as part of a spam filter, for example.

■ **Note** The example that checks whether a string contains `'$$$'` is a bit different from the others. In general, the `in` operator checks whether an object is a member (that is, an element) of a sequence (or some other collection). However, the only members or elements of a string are its characters. So, the following makes perfect sense:

```
>>> 'P' in 'Python'
True
```

In fact, in earlier versions of Python this was the only membership check that worked with strings—finding out whether a character is in a string. Trying to check for a longer substring, such as `'$$$'`, would give you an error message (it would raise a `TypeError`), and you'd have to use a string method. You learn more about those in Chapter 3. In Python 2.3 and later, however, you can use the `in` operator to check whether any string is a substring of another.

Listing 2-4 shows a program that reads in a user name and checks the entered PIN code against a database (a list, actually) that contains pairs (more lists) of names and PIN codes. If the name/PIN pair is found in the database, the string `'Access granted'` is printed. (The `if` statement was mentioned in Chapter 1 and will be fully explained in Chapter 5.)

Listing 2-4. *Sequence Membership Example*

```
# Check a user name and PIN code

database = [
    ['albert',  '1234'],
    ['dilbert', '4242'],
    ['smith',   '7524'],
    ['jones',   '9843']
]
```

```
username = raw_input('User name: ')
pin = raw_input('PIN code: ')

if [username, pin] in database: print 'Access granted'
```

Length, Minimum, and Maximum

The built-in functions `len`, `min`, and `max` can be quite useful. The function `len` returns the number of elements a sequence contains. `min` and `max` return the smallest and largest element of the sequence, respectively. (You learn more about comparing objects in Chapter 5, in the section "Comparison Operators.")

```
>>> numbers = [100, 34, 678]
>>> len(numbers)
3
>>> max(numbers)
678
>>> min(numbers)
34
>>> max(2, 3)
3
>>> min(9, 3, 2, 5)
2
```

How this works should be clear from the previous explanation, except possibly the last two expressions. In those, `max` and `min` are not called with a sequence argument; the numbers are supplied directly as arguments.

Lists: Python's Workhorse

In the previous examples, I've used lists quite a bit. You've seen how useful they are, but this section deals with what makes them different from tuples and strings: lists are *mutable*—that is, you can change their contents—and they have many useful specialized *methods*.

The list Function

Because strings can't be modified in the same way as lists, sometimes it can be useful to create a list from a string. You can do this with the `list` function:[1]

```
>>> list('Hello')
['H', 'e', 'l', 'l', 'o']
```

Note that `list` works with all kinds of sequences, not just strings.

1. It's actually a *type*, not a function, but the difference isn't important right now.

■ **Tip** To convert a list of characters such as the preceding code back to a string, you would use the following expression:

```
''.join(somelist)
```

where `somelist` is your list. For an explanation of what this really means, see the section about `join` in Chapter 3.

Basic List Operations

You can perform all the standard sequence operations on lists, such as indexing, slicing, concatenating, and multiplying. But the interesting thing about lists is that they can be modified. In this section, you see some of the ways you can change a list: item assignments, item deletion, slice assignments, and list methods. (Note that not all list methods actually change their list.)

Changing Lists: Item Assignments

Changing a list is easy. You just use ordinary assignment as explained in Chapter 1. However, instead of writing something like `x = 2`, you use the indexing notation to assign to a specific, existing position, such as `x[1] = 2`.

```
>>> x = [1, 1, 1]
>>> x[1] = 2
>>> x
[1, 2, 1]
```

■ **Note** You cannot assign to a position that doesn't exist; if your list is of length 2, you cannot assign a value to index 100. To do that, you would have to make a list of length 101 (or more). See the section "None, Empty Lists, and Initialization," earlier in this chapter.

Deleting Elements

Deleting elements from a list is easy, too. You can simply use the `del` statement:

```
>>> names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
>>> del names[2]
>>> names
['Alice', 'Beth', 'Dee-Dee', 'Earl']
```

Notice how Cecil is completely gone, and the length of the list has shrunk from five to four.

The `del` statement may be used to delete things other than list elements. It can be used with dictionaries (see Chapter 4) or even variables. For more information, see Chapter 5.

Assigning to Slices

Slicing is a very powerful feature, and it is made even more powerful by the fact that you can assign to slices:

```
>>> name = list('Perl')
>>> name
['P', 'e', 'r', 'l']
>>> name[2:] = list('ar')
>>> name
['P', 'e', 'a', 'r']
```

So you can assign to several positions at once. You may wonder what the big deal is. Couldn't you just have assigned to them one at a time? Sure, but when you use slice assignments, you may also replace the slice with a sequence whose length is different from that of the original:

```
>>> name = list('Perl')
>>> name[1:] = list('ython')
>>> name
['P', 'y', 't', 'h', 'o', 'n']
```

Slice assignments can even be used to *insert* elements without replacing any of the original ones:

```
>>> numbers = [1, 5]
>>> numbers[1:1] = [2, 3, 4]
>>> numbers
[1, 2, 3, 4, 5]
```

Here, I basically "replaced" an empty slice, thereby really inserting a sequence. You can do the reverse to delete a slice:

```
>>> numbers
[1, 2, 3, 4, 5]
>>> numbers[1:4] = []
>>> numbers
[1, 5]
```

As you may have guessed, this last example is equivalent to `del numbers[1:4]`. (Now why don't you try a slice assignment with a step size different from 1? Perhaps even a negative one?)

List Methods

You've encountered functions already, but now it's time to meet a close relative: *methods*.

A method is a function that is tightly coupled to some object, be it a list, a number, a string, or whatever. In general, a method is called like this:

```
object.method(arguments)
```

As you can see, a method call looks just like a function call, except that the object is put before the method name, with a dot separating them. (You get a much more detailed explanation of what methods really are in Chapter 7.)

Lists have several methods that allow you to examine or modify their contents.

append

The `append` method is used to append an object to the end of a list:

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

You might wonder why I have chosen such an ugly name as `lst` for my list. Why not call it `list`? I could do that, but as you might remember, `list` is a built-in function.[2] If I use the name for a list instead, I won't be able to call the function anymore. You can generally find better names for a given application. A name such as `lst` really doesn't tell you anything. So if your list is a list of prices, for instance, you probably ought to call it something like `prices`, `prices_of_eggs`, or `pricesOfEggs`.

It's also important to note that `append`, like several similar methods, changes the list *in place*. This means that it does *not* simply return a new, modified list; instead, it modifies the old one directly. This is usually what you want, but it may sometimes cause trouble. I'll return to this discussion when I describe `sort` later in the chapter.

count

The `count` method counts the occurrences of an element in a list:

```
>>> ['to', 'be', 'or', 'not', 'to', 'be'].count('to')
2
>>> x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
>>> x.count(1)
2
>>> x.count([1, 2])
1
```

2. Actually, from version 2.2 of Python, `list` is a type, not a function. (This is the case with `tuple` and `str` as well.) For the full story on this, see the section "Subclassing list, dict, and str" in Chapter 9.

extend

The extend method allows you to append several values at once by supplying a sequence of the values you want to append. In other words, your original list has been extended by the other one:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
```

This may seem similar to concatenation, but the important difference is that the extended sequence (in this case, a) is modified. This is not the case in ordinary concatenation, in which a completely new sequence is returned:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a + b
[1, 2, 3, 4, 5, 6]
>>> a
[1, 2, 3]
```

As you can see, the concatenated list looks exactly the same as the extended one in the previous example, yet a hasn't changed this time. Because ordinary concatenation must make a new list that contains copies of a and b, it isn't quite as efficient as using extend if what you want is something like this:

```
>>> a = a + b
```

Also, this isn't an in-place operation—it won't modify the original.

The effect of extend can be achieved by assigning to slices, as follows:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> a[len(a):] = b
>>> a
[1, 2, 3, 4, 5, 6]
```

While this works, it isn't quite as readable.

index

The index method is used for searching lists to find the index of the first occurrence of a value:

```
>>> knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']
>>> knights.index('who')
4
>>> knights.index('herring')
Traceback (innermost last):
  File "<pyshell#76>", line 1, in ?
    knights.index('herring')
ValueError: list.index(x): x not in list
```

When you search for the word 'who', you find that it's located at index 4:

```
>>> knights[4]
'who'
```

However, when you search for 'herring', you get an exception because the word is not found at all.

insert

The insert method is used to insert an object into a list:

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers.insert(3, 'four')
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

As with extend, you can implement insert with slice assignments:

```
>>> numbers = [1, 2, 3, 5, 6, 7]
>>> numbers[3:3] = ['four']
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

This may be fancy, but it is hardly as readable as using insert.

pop

The pop method removes an element (by default, the last one) from the list and returns it:

```
>>> x = [1, 2, 3]
>>> x.pop()
3
>>> x
[1, 2]
>>> x.pop(0)
1
>>> x
[2]
```

■ **Note** The pop method is the only list method that both modifies the list *and* returns a value (other than None).

Using pop, you can implement a common data structure called a *stack*. A stack like this works just like a stack of plates. You can put plates on top, and you can remove plates from the top. The last one you put into the stack is the first one to be removed. (This principle is called *last-in, first-out*, or LIFO.)

The generally accepted names for the two stack operations (putting things in and taking them out) are *push* and *pop*. Python doesn't have push, but you can use append instead. The pop and append methods reverse each other's results, so if you push (or append) the value you just popped, you end up with the same stack:

```
>>> x = [1, 2, 3]
>>> x.append(x.pop())
>>> x
[1, 2, 3]
```

■ **Tip** If you want a first-in, first-out (FIFO) queue, you can use `insert(0, ...)` instead of append. Alternatively, you could keep using append but substitute `pop(0)` for `pop()`. An even better solution would be to use a deque from the `collections` module. See Chapter 10 for more information.

---

remove

The remove method is used to remove the first occurrence of a value:

```
>>> x = ['to', 'be', 'or', 'not', 'to', 'be']
>>> x.remove('be')
>>> x
['to', 'or', 'not', 'to', 'be']
>>> x.remove('bee')
Traceback (innermost last):
  File "<pyshell#3>", line 1, in ?
    x.remove('bee')
ValueError: list.remove(x): x not in list
```

As you can see, only the first occurrence is removed, and you cannot remove something (in this case, the string `'bee'`) if it isn't in the list to begin with.

It's important to note that this is one of the "nonreturning in-place changing" methods. It modifies the list, but returns nothing (as opposed to pop).

reverse

The reverse method reverses the elements in the list. (Not very surprising, I guess.)

```
>>> x = [1, 2, 3]
>>> x.reverse()
>>> x
[3, 2, 1]
```

Note that reverse changes the list and does not return anything (just like remove and sort, for example).

Tip If you want to iterate over a sequence in reverse, you can use the `reversed` function. This function doesn't return a list, though; it returns an iterator. (You learn more about iterators in Chapter 9.) You can convert the returned object with `list`:

```
>>> x = [1, 2, 3]
>>> list(reversed(x))
[3, 2, 1]
```

sort

The `sort` method is used to sort lists in place.[3] Sorting "in place" means changing the original list so its elements are in sorted order, rather than simply returning a sorted copy of the list:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort()
>>> x
[1, 2, 4, 6, 7, 9]
```

You've encountered several methods already that modify the list without returning anything, and in most cases that behavior is quite natural (as with `append`, for example). But I want to emphasize this behavior in the case of `sort` because so many people seem to be confused by it. The confusion usually occurs when users want a sorted copy of a list while leaving the original alone. An intuitive (but *wrong*) way of doing this is as follows:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = x.sort() # Don't do this!
>>> print y
None
```

Because `sort` modifies `x` but returns nothing, you end up with a sorted `x` and a `y` containing `None`. One correct way of doing this would be to *first* bind `y` to a copy of `x`, and then sort `y`, as follows:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = x[:]
>>> y.sort()
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

Recall that `x[:]` is a slice containing all the elements of `x`, effectively a copy of the entire list. Simply assigning `x` to `y` wouldn't work because both `x` and `y` would refer to the same list:

```
>>> y = x
>>> y.sort()
```

3. In case you're interested: from Python 2.3 on, the `sort` method uses a stable sorting algorithm.

```
>>> x
[1, 2, 4, 6, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

Another way of getting a sorted copy of a list is using the `sorted` function:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> y = sorted(x)
>>> x
[4, 6, 2, 1, 7, 9]
>>> y
[1, 2, 4, 6, 7, 9]
```

This function can actually be used on any sequence, but will always return a list:[4]

```
>>> sorted('Python')
['P', 'h', 'n', 'o', 't', 'y']
```

If you want to sort the elements in reverse order, you can use `sort` (or `sorted`), followed by a call to the `reverse` method, or you could use the `reverse` argument, described in the following section.

Advanced Sorting

If you want to have your elements sorted in a specific manner (other than `sort`'s default behavior, which is to sort elements in ascending order, according to Python's default comparison rules, as explained in Chapter 5), you can define your own *comparison function*, of the form `compare(x,y)`, which returns a negative number when `x < y`, a positive number when `x > y`, and zero when `x == y` (according to your definition). You can then supply this as a parameter to `sort`. The built-in function `cmp` provides the default behavior:

```
>>> cmp(42, 32)
1
>>> cmp(99, 100)
-1
>>> cmp(10, 10)
0
>>> numbers = [5, 2, 9, 7]
>>> numbers.sort(cmp)
>>> numbers
[2, 5, 7, 9]
```

The `sort` method has two other optional arguments: `key` and `reverse`. If you want to use them, you normally specify them by name (so-called *keyword arguments*; you learn more about those in Chapter 6). The `key` argument is similar to the `cmp` argument: you supply a function and it's used in the sorting process. However, instead of being used directly for determining whether

4. The `sorted` function can, in fact, be used on any iterable object. You learn more about iterable objects in Chapter 9.

one element is smaller than another, the function is used to create a *key* for each element, and the elements are sorted according to these keys. So, for example, if you want to sort the elements according to their lengths, you use `len` as the key function:

```
>>> x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']
>>> x.sort(key=len)
>>> x
['add', 'acme', 'aerate', 'abalone', 'aardvark']
```

The other keyword argument, `reverse`, is simply a truth value (`True` or `False`; you learn more about these in Chapter 5) indicating whether the list should be sorted in reverse:

```
>>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort(reverse=True)
>>> x
[9, 7, 6, 4, 2, 1]
```

The `cmp`, `key`, and `reverse` arguments are available in the `sorted` function as well. In many cases, using custom functions for `cmp` or `key` will be useful. You learn how to define your own functions in Chapter 6.

■ **Tip** If you would like to read more about sorting, you may want to check out Andrew Dalke's "Sorting Mini-HOWTO," found at `http://wiki.python.org/moin/HowTo/Sorting`.

Tuples: Immutable Sequences

Tuples are sequences, just like lists. The only difference is that tuples *can't be changed*.[5] (As you may have noticed, this is also true of strings.) The tuple syntax is simple—if you separate some values with commas, you automatically have a tuple:

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

As you can see, tuples may also be (and often are) enclosed in parentheses:

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

The empty tuple is written as two parentheses containing nothing:

```
>>> ()
()
```

5. There are some technical differences in the way tuples and lists work behind the scenes, but you probably won't notice it in any practical way. And tuples don't have methods the way lists do. Don't ask me why.

So, you may wonder how to write a tuple containing a single value. This is a bit peculiar—you have to include a comma, even though there is only one value:

```
>>> 42
42
>>> 42,
(42,)
>>> (42,)
(42,)
```

The last two examples produce tuples of length one, while the first is not a tuple at all. The comma is crucial. Simply adding parentheses won't help: `(42)` is exactly the same as `42`. One lonely comma, however, can change the value of an expression completely:

```
>>> 3*(40+2)
126
>>> 3*(40+2,)
(42, 42, 42)
```

The tuple Function

The `tuple` function works in pretty much the same way as `list`: it takes one sequence argument and converts it to a tuple.[6] If the argument is already a tuple, it is returned unchanged:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

Basic Tuple Operations

As you may have gathered, tuples aren't very complicated—and there isn't really much you can do with them except create them and access their elements, and you do this the same as with other sequences:

```
>>> x = 1, 2, 3
>>> x[1]
2
>>> x[0:2]
(1, 2)
```

As you can see, slices of a tuple are also tuples, just as list slices are themselves lists.

6. Like `list`, `tuple` isn't really a function—it's a type. And, as with `list`, you can safely ignore this for now.

So What's the Point?

By now you are probably wondering why anyone would ever want such a thing as an immutable (unchangeable) sequence. Can't you just stick to lists and leave them alone when you don't want them to change? Basically, yes. However, there are two important reasons why you need to know about tuples:

- They can be used as keys in mappings (and members of sets); lists can't be used this way. You'll learn more mappings in Chapter 4.
- They are returned by some built-in functions and methods, which means that you have to deal with them. As long as you don't try to change them, "dealing" with them most often means treating them just like lists (unless you need methods such as `index` and `count`, which tuples don't have).

In general, lists will probably be adequate for all your sequencing needs.

A Quick Summary

Let's review some of the most important concepts covered in this chapter:

Sequences: A sequence is a data structure in which the elements are numbered (starting with zero). Examples of sequence types are lists, strings, and tuples. Of these, lists are mutable (you can change them), whereas tuples and strings are immutable (once they're created, they're fixed). Parts of a sequence can be accessed through slicing, supplying two indices, indicating the starting and ending position of the slice. To change a list, you assign new values to its positions, or use assignment to overwrite entire slices.

Membership: Whether a value can be found in a sequence (or other container) is checked with the operator `in`. Using `in` with strings is a special case—it will let you look for substrings.

Methods: Some of the built-in types (such as lists and strings, but not tuples) have many useful methods attached to them. These are a bit like functions, except that they are tied closely to a specific value. Methods are an important aspect of object-oriented programming, which we look at in Chapter 7.

New Functions in This Chapter

Function	Description
`cmp(x, y)`	Compares two values
`len(seq)`	Returns the length of a sequence
`list(seq)`	Converts a sequence to a list
`max(args)`	Returns the maximum of a sequence or set of arguments
`min(args)`	Returns the minimum of a sequence or set of arguments
`reversed(seq)`	Lets you iterate over a sequence in reverse
`sorted(seq)`	Returns a sorted list of the elements of `seq`
`tuple(seq)`	Converts a sequence to a tuple

What Now?

Now that you're acquainted with sequences, let's move on to character sequences, also known as *strings*.