Chapter 6

Basic data structures

A *data structure*, sometimes called *data type*, can be thought of as a category of data. *Integer* is a data category which can only contain integers. *String* is a data category holding only strings. A data structure not only defines what elements it may contain, it also supports a set of *operations* on these elements, such as addition or multiplication. Strings and numbers are the core data structures in Python. In this chapter, you'll see a few more, almost as important, data structures. In the next chapter, you'll se how you can design your own, customary data structures.

The concept of a *sequence* is so fundamental to programming that I've had a very hard time avoiding it so far. And as you, alert and perky, have noticed, I actually haven't, since I involuntarily had to introduce sequences in Section 4.4 when talking about the `for` loop. In Python, the word "sequence" covers several phenomena. Strings are sequences of characters, and you heard about those in Chapter 3. In the coming sections, you'll hear about the two other basic types of sequence supported by Python: *Lists* and *tuples*. Later in this chapter, we'll get around to talking about *sets* and *dictionaries*.

Strings and integers represent concrete data objects; a string or a number represents true data in itself.[1] Lists, tuples and dictionaries are designed to *organize* other data, to impose structure upon it; they do not necessarily represent true data in their own right. For this reason, they are also called *abstract* data structures.

6.1 Lists and tuples — mutable vs. immutable

As I mentioned back in Section 4.4, lists and tuples are indexed just like strings: The first item of a sequence of length *l* has index 0, and the last has index *l-1*. You can get the length of a sequence with the built-in function `len`. You may use negative indices (e.g. -1 for the last item in the sequence), and you can slice out a (new) sequence using two or three parameters (e.g., `l[:3]` yields a new sequence containing the first three items of `l`). Go back and re-read Section 3.2 — everything in there applies to lists and tuples as well.

[1]— agreed? Let's not get lost in the obscure realms of philosophy.

Lists are written with square brackets `[1, 2, 3]` while tuples are written with parentheses `(1, 2, 3)`. When you *create* a list of values, you have to use square brackets with the comma-separated values inside; when you create a tuple, the parentheses are optional, and you may just write the values with commas in between. The commas create the tuple, not the parentheses — and square brackets create a list. See some examples, also refreshing the indexing business, below.

```
1   >>> l = ["George", "Paul", "John", "Pete", "Stuart"]
2   >>> l[0]
3   'George'
4   >>> l[3] = "Ringo"
5   >>> "Pete" in l
6   False
7   >>> del l[4]
8   >>> l
9   ['George', 'Paul', 'John', 'Ringo']
10  >>> t = ()
11  >>> t2 = ("A", "C", "G", "T")
12  >>> t3 = 7,
13  >>> t2[1:3]
14  ('C', 'G')
15  >>> t2[3] = "U"
16  Traceback (most recent call last):
17    File "<stdin>", line 1, in ?
18  TypeError: object does not support item assignment
```

In line 1, a list named `l` containing five strings is created. In line 4, the list's fourth element (index 3) is replaced with a new string. In line 5–6, it is checked that the former member element `Pete` is no longer present in the list. Using the `del` command, the list's fifth element (index 4) is deleted in line 7. This command can be used to delete any identifier as well. In line 8–9, the current content of `l` is printed.

In lines 10–12, three tuples are created: An empty tuple, a tuple with four string elements, and a so-called *singleton*, a tuple with only one element. Note that the comma in line 12 ensures that the variable `t3` is assigned a one-element tuple, not just the integer 7.

In line 13, a new tuple is created from a slice of `t2`, but when I attempt to change one of the members of `t2` in line 15, I get an error message: `TypeError: object does not support item assignment`.

This illustrates the one, very important aspect in which lists and tuples are functionally different: Lists are *mutable*, tuples are *immutable*. You may modify a list, but you can't modify tuples. More precisely: Once a list is created, you can replace or delete its elements, or even add new ones in any position. But once a tuple is created, it's take it or leave it. If you need to *pack together* some items that logically belong together and never need to be replaced, you might use a tuple since that way not even erroneous modifications can be made. However, in all other cases, you should use lists. Both lists and tuples may contain data elements of any type, including compound data objects you have created yourself (we'll get to that in Chapter 7).

Adding elements to a list can be done using our old friend the + operator. Recall that that same operator is also used for adding up numbers and concatenating strings, but the overloading of this poor fellow doesn't stop here; you may also "add" two lists using +. Again, this makes perfect sense since the intuitive meaning of, e.g., `[1, 2, 3] + [4, 5]` is clear. You only have to keep in mind that if one of the operands of an addition operation is a list, then so must the other be. Thus, to add just a single element to a list, you have to put that single element in a list first before adding it: `[1, 2, 3] + [4]` works, but `[1, 2, 3] + 4` doesn't.

From 1960 to 1962, The Beatles had five members: John Lennon, Paul McCartney, George Harrison, Pete Best and Stuart Sutcliffe. In 1962, Sutcliffe left the band and Pete Best was replaced with Ringo Starr.

In fact, you can also multiply a list with a number using the well-known multiplication operator `*`. Here of course, at most one of the operands may be a list since it doesn't make sense to multiply a list with a list. Once more, the intuitive meaning is evident, and hence `*` is overladed to also work with a list:

```
>>> [0] * 7
[0, 0, 0, 0, 0, 0, 0]
>>> ['A', 'B'] * 2
['A', 'B', 'A', 'B']
```

Here we have to be a bit careful, though, and to understand why, we have to look into the computer's memory and see what actually goes on when you create a list. Figure 6.1 shows an example where a list with three elements has been created and assigned to a variable `a`. Recall that a variable assignment means that a reference (the horizontal arrow) is created associating some value (a chunk of memory, here the list) to a variable name. Each element in the list (marked by red bullets) *is also a reference* to some chunk of memory which, as mentioned, can represent any kind of data.

Now when you use the multiplication operator on a list, the list's elements are copied to form a new list. But the list elements are *references* to data, not actual data. Thus, no copying of data takes place. Imagine we execute the statement `b = a * 2` following the example of Figure 6.1. The impact on memory would be as shown in Figure 6.2.

The data elements, the yellow, green and blue blobs, are left unchanged and uncopied. The original list, `a`, is left unchanged. The new list, `b`, is a double copy of `a` in the sense that its first three members *reference the same objects* as do the elements of `a`, and so do the last three elements.

Now if the color-blob data objects are immutable, none of this causes problems. But if they are mutable, they may change. And any such changes would

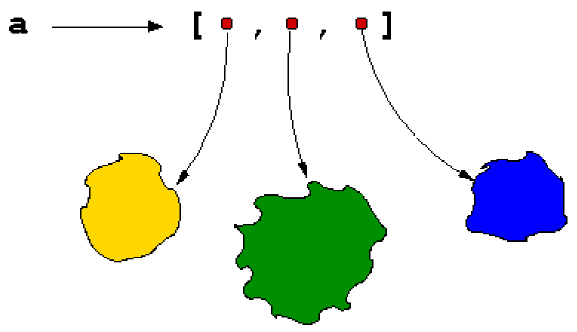

Figure 6.1: Creating a list.

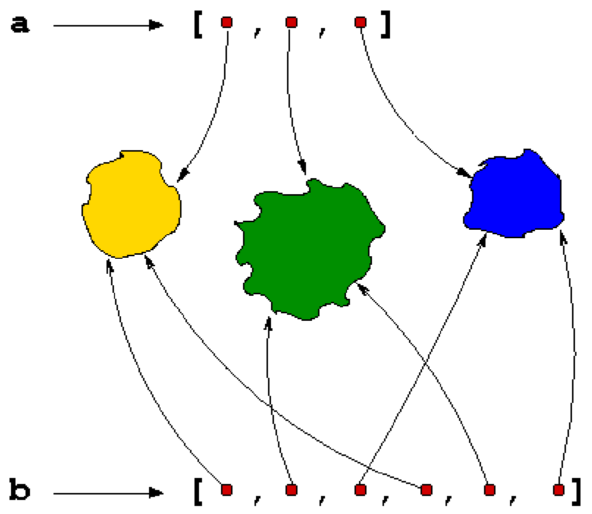

Figure 6.2: Result of executing the statement `b = a * 2` in the situation shown in Figure 6.1.

be visible through both `a` and `b`, since they reference the physically same objects. So far, you've heard mostly about immutable types of data: Numbers are immutable, strings are immutable, tuples are immutable. Thus, if the colored blobs are integers, floats, strings or tuples, they can't be modified and all is well. But we do know one mutable data type — lists ☺. And when we start dealing with classes and objects in Chapter 7, the same considerations apply.

To illustrate what may go wrong, here is a concrete example which corresponds precisely to Figures 6.1 and 6.2:

```
>>> a = [ ['Y'], ['G'], ['B'] ]
>>> b = a * 2
>>> a
[['Y'], ['G'], ['B']]
>>> b
[['Y'], ['G'], ['B'], ['Y'], ['G'], ['B']]
```

```
7  >>> a[0][0] = '**'
8  >>> a
9  [['**'], ['G'], ['B']]
10 >>> b
11 [['**'], ['G'], ['B'], ['**'], ['G'], ['B']]
```

In line 1, I create a list `a` containing three other lists (corresponding to the color-blob data objects of Figure 6.1), namely the one-element lists `['Y']`, `['G']` and `['B']`. Next, I create `b`. Now the situation is exactly as illustrated in Figure 6.2. Printing the contents of `a` and `b` in lines 3–6, one might think that the two are completely independent. However, the object pointed to by `a[0]` is a list (`['Y']`), a list is mutable, and so in line 7 I can change its first (and only) element `a[0][0]` to `'**'`. (Going back to Figure 6.2, you can imagine that the internal state of the yellow blob has changed. It's still there, all the references pointing to it are intact, but its state has changed). It is not surprising that the contents of `a` has changed, as shown in lines 8–9: What is disturbing is that `b` has also changed (lines 10–11) although I haven't touched it!

This is an example of a *side effect*. A side effect is when some programming action has (usually malign) effects other than those intended. In the above example, actually the problem arises from the internal one-element lists, not `a` and `b`. In general, whenever two variables point to the same, mutable object, care must be taken. And when using the multiplication operator on a list, what you get is exactly that: More references to the original data objects pointed to by the first list.

One further example. As mentioned (what, three times now?), lists may contain any kind of objects. If you need a 2-dimensional array, one way to go is to create a list containing other lists. For example, to represent the 3x3 identity matrix,

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

you might do like this:

```
I = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

This way, `I[0]` is the first row of the matrix, `I[1]` is the middle row, `I[2]` is the bottom row, and `I[0][2]` is the last element of the first row. If you want to initialize a 3x3 zero matrix (3 rows and 3 columns of 0's), you might consider using the multiplication operator:

```
1 >>> m = [ [0] * 3 ] * 3
2 >>> m
3 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

```
4  >>> m[0][2] = 7
5  >>> m
6  [[0, 0, 7], [0, 0, 7], [0, 0, 7]]
```

The expression `[0] * 3` creates a list of three 0's. In general, the expression `[ X ] * 3` creates a list of three `X`'s, what ever `X` is. Thus, putting the two together in line 1 creates a list of three lists of three 0's. Print `m` (lines 2–3) and all looks well; at least it looks very similar to the identity matrix above. A lists of three lists, c'est bon, ça!

However, changing the last element of the first row to 7 (line 4) causes mayhem: Unfortunately, the last element of *every* row is changed. To see why, study Figure 6.3. When creating `I`, we explicitly created *three* new lists, one for each row (symbolized by the ugly magenta, light-blue and pink color blobs). When creating `m`, we first created *one* list of three 0's (`[0] * 3`, brown color blob), and then we created a list of three references to that same list.

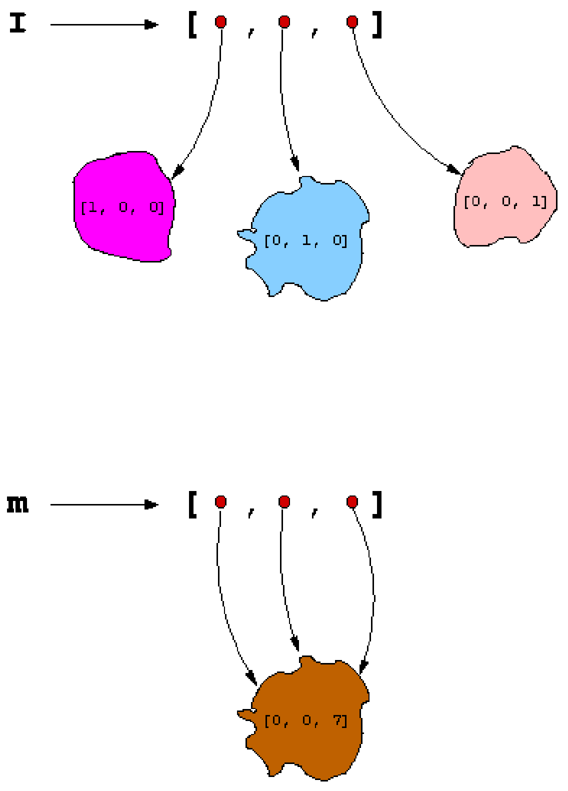

Figure 6.3: One right and one wrong way of creating a 3x3 matrix.

Thus, one way of inadvertently creating more references to the same mutable object is applying the multiplication operator to a list. Another very common one is through functions. As we saw in Section 5.4, when a function is called with some argument, actually a *reference* to that argument is what's passed to the function, not a copy of it. If this argument is a list (call it `arg`), then again we have several references to the same mutable object. This means that any function taking a list as its argument can *modify* the original list.

Sometimes that's what you want — but sometimes it isn't. If not, you might consider using `arg[:]` for the list argument instead of just `arg`. Recall that `arg[:]` creates a slice (i.e., a new sequence) of `arg` which is actually the full sequence. In effect, this trick means shipping off a *copy* of `arg` to the function. There's always a "however", however: `arg[:]` creates a new list, but it doesn't create copies of the list's elements. Thus, the function can safely replace elements in its copy of the argument list without affecting the original list, but if the list's elements are mutable objects, the function can still modify them and thereby affect the original list. See the following demonstration and have a look at the cartoons in Figures 6.4 and 6.5.

```
1  >>> def modify_list(l):
2  ...     l[0] = 'modified'
3  ...
4  >>> def modify_listElement(l):
5  ...     l[0][0] = 'modified'
6  ...
7  >>> arg = [1, 2, 3, 4]
8  >>> modify_list(arg)
9  >>> arg
10 ['modified', 2, 3, 4]
11 >>>
12 >>> arg = [1, 2, 3, 4]
13 >>> modify_list(arg[:])
14 >>> arg
15 [1, 2, 3, 4]
16 >>>
17 >>> arg = [[1, 2], [3, 4]]
18 >>> modify_listElement(arg[:])
19 >>> arg
20 [['modified', 2], [3, 4]]
```

In lines 1–2, the function `modify_list` is defined. When called on a list, it replaces the first element of this list with the string `'modified'`.

Lines 4–5 define a second function, `modify_listElement`, which assumes its argument is a list of lists. When called, it replaces the first element list's first element with the string `'modified'`.

In line 7, a test list `arg` is created. Line 8 calls the first function on `arg` directly, and as lines 9–10 show, the original `arg` has been modified by `modify_list`.

In lines 12–13 we try again, create a test list `arg` and call the function. This time, however, we pass a copy of `arg` using `[:]`, and as lines 14–15 show, now the original `arg` is not modified by `modify_list`.

Finally, in line 17 `arg` is assigned a list of lists. When `modify_listElement` function is called in the same manner as above on a copy of `arg`, it again modifies the original list, as shown in lines 19–20.

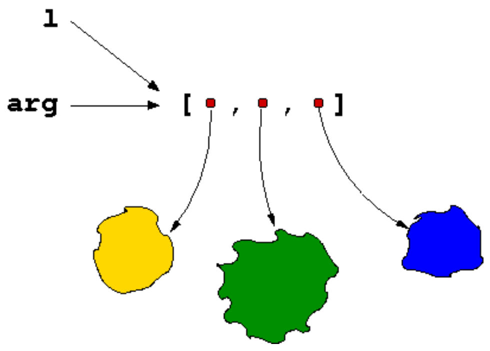

Figure 6.4: Passing an original list `arg` to a function `f(l)`. Any modifications made in `f` to its parameter `l` (e.g. replacing an element) affects the original argument `arg` as well.

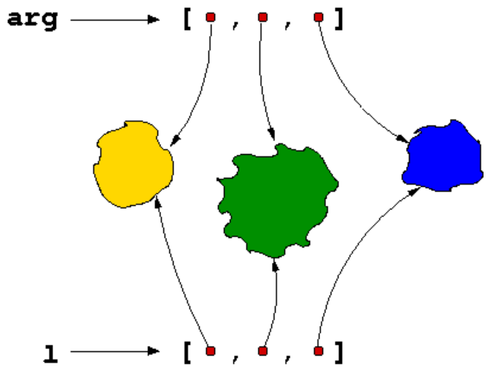

Figure 6.5: Passing a copy of an original list `arg` (e.g. with `arg[:]`) to a function `f(l)`. Any modifications made in `f` to its parameter `l` (e.g. replacing an element) do not affect the original argument `arg`; however, modifications made to the *elements* of `l` affect the original list as well.

Bottomline:

1. Lists are the first mutable objects you have encountered; take care if you've got several variables pointing to the same list.
2. Multiplying a list doesn't copy data, only references to data.
3. In a function call, the arguments passed on to the function are not copies of data, only references to data.

6.2 Creating and accessing lists

Rather than lists you explicitly type in element by element, mostly you'll be working with lists which are somehow automatically generated as the result of some calculation or operation. As you've seen, several string methods return lists. Another very useful way to create sequences is via the built-in function `xrange`[2]. Since Chapter 3, you're familiar with the bracketed way of indicating optional arguments in functions, so here's the official, formal definition of `xrange`:

```
xrange([start,] stop[, step])
```

I know I don't have to explain the notation again, but nevertheless: `xrange` takes a mandatory `stop` argument and optional `start` and `step` arguments (..? Ah, good question! In case you give it two arguments, they'll be interpreted as `start` and `stop`, not `stop` and `step`. (Nice little poem there)). All arguments should be integers.

`xrange` produces a list of integers ranging from (and including) the `start` argument (if not given, 0 is default) up to but not including the `stop` argument. If `step` is given, only every `step`'th integer is included. Watch:

```
>>> xrange(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xrange(7,10)
[7, 8, 9]
>>> xrange(0, 10, 3)
[0, 3, 6, 9]
>>> xrange(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

You can't go backwards with positive steps, and you can't go forward with negative steps, so these commands produce empty lists:

```
>>> xrange(10,1)
[]
>>> xrange(0, 10, -3)
[]
```

It's time we reinvoke randomness. We've touched upon the `random` module in Section 5.3. From this module, you know the `randint` function which takes two arguments a and b and returns an integer N such that $a \leq N \leq b$. There is also a function called `randrange` which perhaps is a better choice: `randrange` returns a random integer from the list a call to `xrange` with the same arguments would generate. In other words: `randrange` and `xrange` have identical definitions (they both take a `stop` and optional `start` and

[2]There is also an equivalent function called `range`; however, `xrange` is more efficient for most purposes.

`step` parameters), and thus if you do `randrange(1, 5)`, you'll get either 1, 2, 3 or 4; *not* 5. This complies with the "up to but not including" convention used throughout Python when dealing with start and end indices, and so the definition of `randrange` is probably easier to remember than that of `randint`, where the end point is included in the list of optional values.

Let's check just how random the `random` module is. We'll do that by asking the user for a maximum number m. Then, over $100 * m$ rounds, we'll pick a random number between 0 and $m - 1$, and we'll keep track of which numbers were picked. On average, each number between 0 and $m - 1$ should be picked 100 times, yes? And conversely, if some number weren't picked at all, that would be suspicious, no? See the program in Figure 6.6.

```
1  from random import randrange

2  m = int(raw_input("Input maximum integer: "))
3  N = [0]*m

4  for i in xrange(100*m):
5      N[randrange(m)] += 1

6  notpicked = []
7  for i in N:
8      if i == 0:
9          notpicked += [i]

10 if notpicked:
11     print "These numbers were never picked: %s"%notpicked
12 else:
13     print "All numbers below %d were picked"%m
```

Figure 6.6: Program `randomness.py`.

In line 3, I create a list `N` of m 0's; `N[i]` will be the number of times i is picked. So far, no numbers have been picked, so `N` is initialized to all zeros. Recall that numbers are immutable, so there's no risk in creating a list of 0's this way.

In line 4, the loop starts; I want $100*m$ rounds, so I use a `for` loop to iterate through a list with precisely this length (generated with `xrange(100*m)`). In each round, I pick a random number between 0 and m with `randrange(m)`, and I use this number as the index into the list `N` and increment this entry by 1.

After the loop, I wish to collect all numbers which haven't been picked at all in a list called `notpicked`. So I go through all number counts in `N` (line 7), and if I see a count which is still 0, I add the index to `notpicked` in line 9 (using the generic shorthand notation `x+=y` for `x=x+y`). Note that in order to add the index `i` to the list, I first have to put in into a small list of its own with `[i]`.

Eventually I check whether any indices have been put into `notpicked`: As with the number 0, an empty list also is interpreted as `False` if used in a condition (non-empty lists are interpreted as `True`). So, if `notpicked` holds any values (line 10), print them. Otherwise, print the obligatory triumphant message. Figure 6.7 shows some test runs.

```
Input maximum integer: 10
All numbers below 10 were picked
Input maximum integer: 100
All numbers below 100 were picked
Input maximum integer: 1000
All numbers below 1000 were picked
Input maximum integer: 10000
All numbers below 10000 were picked
Input maximum integer: 100000
All numbers below 100000 were picked
Input maximum integer: 1000000
All numbers below 1000000 were picked
Input maximum integer: 10000000
All numbers below 10000000 were picked
```

Figure 6.7: Output from program `randomness.py`.

The last run with $m = 10,000,000$ took more than an hour on my computer, so I didn't bother going any further. Anyway, we don't detect any blatant flaws in Python's random number generation, although of course the statistical evidence is monumentally inconclusive in regard to the question about exactly how random the `random` module is. And perhaps you already know that in fact there is no such thing as genuine randomness in any computer[3]. What computers have are *pseudo-random* number generators. You'll most likely never know the difference since these number generators are very good, but it's sensible to keep in mind.

With a sequence of known length, you can *unpack* its contents into a set of named variables. Say that you know that some tuple `student` contains a student's first name, last name and ID number; then you can unpack it into three variables like this:

```
>>> student = ("Albert", "Einstein", "18977987")
>>> first, last, id = student
>>> id
'18977987'
```

[3].. and I'm not convinced that there exists true randomness in the real world either (give or take quantum physics). *Unpredictability* is what it is, but the distinction is purely academic.

This mechanism is actually very convenient. The program in Figure 6.8 demonstrates why. L is a list of early Nobel prize winners represented as tuples. Each tuple contains a name, a field, and a year. This is a typical application of tuples: Bundling together data that belongs together and that isn't necessarily of the same type.

```
1 L = [ ("Wilhelm Conrad Röntgen", "physics", 1901),
2       ("Ivan Pavlov", "medicine", 1904),
3       ("Henryk Sienkiewicz", "literature", 1905),
4       ("Theodore Roosevelt", "peace", 1906) ]

5 formatstring = "%-28s%-13s%s"
6 print formatstring%("Name", "Field", "Year")
7 print "-"*45
8 for name, field, year in L:
9     print formatstring%(name, field, year)
```

Figure 6.8: Program unpacking.py.

The formatstring defines the column widths of the table we're about to print out: First column should be left-centered and 28 characters wide; middle column should be 13 characters wide, and third column has no predefined length (it will hold the year of each Nobel prize award, and since no Nobel prizes were awarded prior to the year 1000, all the entries of this column will have four digits). The format string "expects" three arguments to be inserted, and in line 6 the table header line is printed by giving the column headers as arguments.

A line of suitable length is printed in line 7 (since strings are sequences, you are allowed to multiply them using *, as seen earlier). Finally, a for loop is used to iterate through all the elements in L. Typically, one would use a single variable as the counter (see below), but since we know that the entries are 3-tuples, we might as well use three distinct variables in the for loop, unpacking each tuple directly as we go. Figure 6.9 shows a run.

```
Name                        Field        Year
---------------------------------------------
Wilhelm Conrad Röntgen      physics      1901
Ivan Pavlov                 medicine     1904
Henryk Sienkiewicz          literature   1905
Theodore Roosevelt          peace        1906
```

Figure 6.9: Output from program unpacking.py.

Unpacking can also be used as a quick way of swapping the values of two variables without the need for an explicit temporary variable. Rather than

```
temp = x
x = y
y = temp
```

you can simply do

```
x, y = y, x
```

Neat, eh? Technically, on the right-hand side of the = symbol, you create a tuple containing `y` and `x`, and this tuple is then unpacked into `x` and `y`, respectively.

6.3 List methods

As with strings, once you create a list you have access to a load of list methods that do useful things with it. Figure 6.10 shows them all, and below I'll explain some of them further. But before that, I'll say a few words more about some of the features of sequences that I have already touched upon.

The `del` command can be used to delete an element from a list, but it can also delete a *slice* of a list:

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

It is also possible to *replace* a slice inside a list with some other sequence:

```
1  >>> L = [1, 2, 3, 4, 5]
2  >>> L[1:3] = (9, 9, 9, 9, 9)
3  >>> L
4  [1, 9, 9, 9, 9, 9, 4, 5]
5  >>> L[2:6] = "string"
6  >>> L
7  [1, 9, 's', 't', 'r', 'i', 'n', 'g', 4, 5]
```

Note that the slice in line 2, `L[1:3]`, has length 2 but the sequence whose values are inserted in its place have length 5. Note also that any sequence can be used in a slice replace, including strings which are sequences of characters; hence for strings, the individual characters are inserted in the list and not the string as a whole (see line 5). If you include a stepping parameter (extended slicing) for the slice to be replaced, the sequence to be inserted must have the same length, however:

Method name and example on `L=[1,7,3]`	**Description**

`L.append(item)`
 	
`>>> L.append(27)`
`>>> L`
`[1, 7, 3, 27]`
	Append `item` to `L`. `item` may be any kind of object. NB: Does not return a new list.

`L.extend(x)`
 	
`>>> L.extend([3, 4])`
`>>> L`
`[1, 7, 3, 3, 4]`
	Adds all the individual elements of the sequence `x` to `L`. NB: Does not return a new list.

`L.count(x)`
 	
`>>> [1, 2, 2, 7, 2, 5].count(2)`
`3`
	Return the number of occurrences of `x`.

`L.index(x[, start[, end]])`
 	
`>>> [1, 2, 2, 7, 2, 5].index(2)`
`1`
	Return smallest `i` such that `s[i]==x` and `start<=i<end`. I.e., look for `x` only within the start and stop indices, if given. Yields ValueError if `x` not in list.

`L.insert(i, x)`
 	
`>>> L.insert(1, 8)`
`>>> L`
`[1, 8, 7, 3]`
	Insert `x` at index `i`, keeping all the other elements. NB: Does not return a new list.

`L.pop([i])`
 	
`>>> L.pop()`
`3`
`>>> L`
`[1, 7]`
	Remove element at index `i`, if `i` is given. Otherwise remove last element. Return the removed element.

`L.remove(x)`
 	
`>>> L.remove(7)`
`>>> L`
`[1, 3]`
	Remove first occurrence of `x` from list. Yields ValueError if `x` is not in list. NB: Does not return a new list.

`L.reverse()`
 	
`>>> L.reverse()`
`>>> L`
`[3, 7, 1]`
	Reverse the list in-place (which is more space-efficient); i.e. no new list is returned.

`L.sort([cmp[, key[, reverse]]])`
 	
`>>> L.sort()`
`>>> L`
`[1, 3, 7]`
	Sort the list in-place (for space-efficiency). The optional parameters can specify how to perform the sorting (see text). NB: Does not return a new list.

Figure 6.10: List methods called on some list `L`.

```
>>> L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[2:7:2] = "odd"
>>> L
[1, 2, 'o', 4, 'd', 6, 'd', 8, 9]
>>> L[2:7:2] = "some odd"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: attempt to assign sequence of size 8 to
extended slice of size 3
```

You may also use `del` to delete an extended slice of a list.

Now on to the list methods shown in Figure 6.10; most are pretty straightforward, but still I'll briefly discuss some of them. First and foremost you should note that none of the methods return a new list; since lists are mutable, the list methods all perform their business directly on the given list without producing a new list (except for slicing).[4]

You've seen that one way of appending an element `x` to a list `L` is `L += [x]`. It's more elegant to use the `append` method and simply do `L.append(x)`; this way, you don't have to create a new singleton list first.

If you have two lists `a` and `b` and you want to concatenate them, you can use `a.extend(b)`. This adds all the elements of `b` to the end of `a`, keeping their order. And again: You don't get a new list after calling `a.extend`; the operation is performed directly on `a`. Thus, things like `c = a.extend(b)` are bad style: The list extension is actually performed, but the extended list is not assigned to `c` because `extend` doesn't return a new list.

The `index` method is obviously very useful. The optional parameters may also come in handy: Say that you have a long list of random integers, and that you want the indices of all 7's in the list. This small code snippet does the job (the `try/except` construct is just a way of catching the `ValueError` arising when no more 7's can be found; we'll get back to such *exception handling* later).

```
>>> a = [5,1,3,2,7,2,7,3,4,7,3]
>>> i = -1
>>> try:
...   while 1:
...     i = a.index(7, i+1)
...     print i,
... except:
...   pass
...
4 6 9
```

[4]With strings, the story is different: Strings are immutable, and therefore the generative string methods return new strings. Confer with Section 3.5.

The idea is to use an index pointer `i` as a pointer in the list and repeatedly search for the next 7 starting at index `i+1`. The result as calculated by `index(7, i+1)` is again stored in `i`, ensuring that the next search starts one position to the right of the 7 just found. Eventually, there are no more 7's, and we get the `ValueError`, but this error is caught by the `except` clause which simply executes a `pass` — a command which means "do nothing".

The `sort` method is efficient and versatile. It's also *stable*: The original order of two elements with the same sorting ranking is guaranteed not to change. With no given arguments, the list elements are just sorted in increasing order. Any list containing elements which can be pairwise compared with $<$ and $>$ can be sorted. If you want your list sorted according to some less than obvious criterion, it's possible to tell Python how by passing a *compare* function as an argument to `sort`. This function should take two arguments (list elements) and return -1 if the first should come before the second in the sorted list, 1 if it's the other way around, and 0 if you don't care (i.e. if the elements have the same rank according to your sorting criterion).

For example, look again at the list `L` of Nobel laureates from Figure 6.8. If we simply sort it with `L.sort()`, Python will sort its tuples according to the first element of each tuple. That would mean alphabetically by the prize winners' first name (and note again that `L.sort()` doesn't return a new list):

```
>>> L = [ ("Wilhelm Conrad Röntgen", "physics", 1901),
...       ("Ivan Pavlov", "medicine", 1904),
...       ("Henryk Sienkiewicz", "literature", 1905),
...       ("Theodore Roosevelt", "peace", 1906) ]
>>>
>>> L.sort()
>>> L
[('Henryk Sienkiewicz', 'literature', 1905),
('Ivan Pavlov', 'medicine', 1904),
('Theodore Roosevelt', 'peace', 1906),
('Wilhelm Conrad Röntgen', 'physics', 1901)]
```

Well, it just seems more Nobel prize-like to sort this esteemed group of people by their last name, and in order to do that, we write a special function which compares the last names of two laureates and pass this function to `sort`:

```
>>> def sort_laureate_tuples_by_last_name(a, b):
...   return cmp(a[0].split()[-1], b[0].split()[-1])
...
>>> L.sort(sort_laureate_tuples_by_last_name)
>>> L
[('Ivan Pavlov', 'medicine', 1904),
('Wilhelm Conrad Röntgen', 'physics', 1901),
('Theodore Roosevelt', 'peace', 1906),
('Henryk Sienkiewicz', 'literature', 1905)]
```

Ha-haa! The function receives two tuples `a` and `b`. First element of each (`a[0]` and `b[0]`) is a name. Using the string method `split` we divide each name string into separate words; taking the last of these (index -1) gives us the last name. These last names are then given to the built-in function `cmp` which compares its two arguments using $<$ and $>$ and returns -1, 1 or 0 depending on the relationship of the arguments, just the way we need it. Ideally, if the last names are the same, the function should compare the first names and base the ranking on that (since `sort` is stable, the pre-ordering by last name would not be destroyed by the following first-name sorting) — but I'll leave the details to you. When calling `sort`, you pass the name of your sorting function as an argument with no pair of parentheses after the name; parentheses would *call* the function rather than pass a reference to it. Just pass the function name to `sort` and it will call the function as it needs to.

The second optional parameter to `sort`, `key`, should be a function taking one argument and returning the value which will be subsequentely compared by `sort`. If you use this feature rather than a compare function, you should use `key` as a keyword argument (to prevent Python from attempting to use it as a compare function). See the following example where I sort a list of letters, ignoring the case:

```
>>> l = ['I', 'y', 'b', 'S', 'P', 'o']
>>> l.sort(key=str.lower)
>>> l
['b', 'I', 'o', 'P', 'S', 'y']
```

Here, I do not want to sort the characters the natural Python way; normally, an uppercase **X** would come before a lowercase **a** according to the Unicode character table (see Section 5.1). Thus using `sort` with no parameters would give me the sequence `I P S b o y`. To sort but ignore the case, I have to tell Python how, and in this case I do that by giving it a key function; this function will be called on each element (a character) and should return the key which the character should be ranked by. Here's an opportunity to use the `str` identifier I briefly mentioned at the end of Chapter 3! We need a function which takes one character (string) argument and returns the lowercase version of this string. We can't just use `lowercase`, because this identifier is not freely floating around; it's accessible either through a specific string or, like here, through the `str` thingy (which is actually a *class* — more on classes in the next chapter). So, each time `sort` needs to compare two characters of the list `l`, it uses the given key function, `str.lower`, to obtain the lowercase versions it should actually use for the comparison.

If you need to access a list method directly and not through a specific list, you can do it in exactly the same manner using `list`: `list.pop` is a function which takes one argument, a list, and removes and returns the last element of this list. In general, when accessed through `list` and `str`, all the list and string methods take one extra parameter (expected to be the first in the parameter list), namely the list/string to perform the operation on.

As you see, you can easily pass a function as an argument to another function. Since all parameter passing is really through references, of course you can pass a reference to a function as well. Again, note the difference between `list.pop` and `list.pop()`: The first is a "passive" reference to a function, the latter *calls* the function. Finally, note also that strings are one kind of sequences, lists are another. Thus, you can't use list methods on strings and vice versa.

Due to their very mutability, lists are in many cases more efficient than strings in terms of memory usage. One typical example is when you're building a string little by little, e.g. by adding one character to it in every round of some loop. Each time you add a character to the working string, you'll get a new copy one character longer than the previous one. Say we want a random DNA sequence 100,000 nucleotides long. We might initialize an empty string and add a random nucleotide to it 100,000 times, as shown in Figure 6.11.

```
1 # bad way of building a string step by step
2 import random
3 nuc = ["A", "C", "G", "T"]
4 s = ""
5 for i in xrange(100000):
6     s += nuc[random.randrange(4)]
```

Figure 6.11: Program `buildstring.py`: An inefficient way of building a string step by step.

A random nucleotide is generated by using `randrange` to find a random index in the nucleotide list. This character is then added to `s` (line 6) — but since strings are immutable, in fact a *new* string is created and assigned to `s`. In the end, 99,999 strings have lived and died before the final string is complete. This takes up a lot of memory, although the garbage collector eventually frees this memory again, and it takes a lot of time to create all these strings.

```
import random
nuc = ["A", "C", "G", "T"]
l = []
for i in xrange(100000):
    l.append(nuc[random.randrange(4)])
s = "".join(l)
```

Figure 6.12: Program `buildstring2.py`: It's better to build a list and convert it to a string.

Figure 6.12 solves the same problem in a much more efficient way. It uses a list `l` instead, because since lists are mutable, the *same* list is extended with one nucleotide in each round of the loop. One list is all there ever is, and in the end (line 6), the individual characters of the list are joined (using the empty string as the delimiter) to form the complete DNA string. In other words: Whenever you need to gradually build a string, build a list instead and use the string method `join` in the end to convert the list into a string once and for all.

You may use lists and the list methods `pop` and `append` to build *stacks* and *queues*. A stack is a "last in, first out" data structure which contains data elements in a way such that when you add another element, you add it "on top" of the existing ones. When you remove one, you again pick it from the top. Thus, at any time you only deal with the top of the stack and have no access to elements further down. A stack can be implemented using a list and using `append(x)` to add an element `x` (to the end of the list), and `pop()` to remove an element (from the end of the list).

A queue is a "first in, first out" data structure, where new elements are inserted at the end, but where elements can only be removed from the front. You can implement a queue using a list and using `append(x)` to add an element `x` (to the end of the list), and using `pop(0)` to remove an element (from the front of the list).

6.4 Functions dealing with lists

Since sequences are fundamental to most programs, Python offers several built-in functions working with lists and tuples. As indicated, the typical scheme for accessing the elements of a sequence is the `for` loop, as in

```
for element in sequence:
  print element
```

You can do that for any sequence containing any kind of items: Numbers in a list, characters in a string, rows (lists) in a matrix (list of lists), etc. This way, `element` will in turn take the value of each item in `sequence`. You may also use several counter variables as shown in Figure 6.8 if your sequence contains other sequences.

Sometimes you not only want each item in a sequence but also the *index* of this item in the sequence. For such cases, the above method is insufficient as the index is not explicitly in play, and you're better off with this scheme:

```
for i, element in enumerate(sequence):
  print i, element
```

The expression `enumerate(sequence)` sets it up for you so that you can cycle through the sequence obtaining both the index `i` of each element and the `element` itself.

Next function on stage is `zip` which works as a kind of zipper. You supply a list of sequences, and it returns a list of tuples, where tuple i holds the i'th element of all argument sequences. It's best illustrated through an example (like everything):

```
>>> who = ['H. Sienkiewicz', 'I. Pavlov',
...        'T. Roosevelt', 'W. C. Röntgen']
>>> what = ['literature','medicine','peace','physics']
>>> when = [1905, 1904, 1906, 1901]
>>> for p, f, y in zip(who, what, when):
...   print "%s won the Nobel %s prize in %d"%(p, f, y)
...
H. Sienkiewicz won the Nobel literature prize in 1905
I. Pavlov won the Nobel medicine prize in 1904
T. Roosevelt won the Nobel peace prize in 1906
W. C. Röntgen won the Nobel physics prize in 1901
```

You can give it any number of input sequences (including strings which are then chopped up into characters); the resulting list of tuples will be truncated to have the same length as the shortest given argument sequence.

Another very useful function is `map`. This function requires a function and a sequence as arguments, and it then calls the given function on each element in the sequence and packs the results into a list. Thus, the i'th element of the returned list is the result of applying the given function to the the i'th element of the given sequence. Observe:

```
>>> def addVAT(a):
...   return a*1.25
...
>>> map(addVAT, [100, 220, 360])
[125.0, 275.0, 450.0]
```

The function `addVAT` adds a 25% VAT (value added tax) to the given amount. When this function is passed to `map` together with a list of amounts, the result is a list of the same amounts with added VAT.

If the function you want to map to all elements of some list takes more than one argument, `map` can still help you. You may supply more than one sequence after the function; in this case, the system is as with `zip`: The i'th element of the returned list is the result of the given function called on the i'th arguments of all the given sequences. Here's an example:

```
>>> import math
>>> map(math.pow, [2, 3, 4], [1, 2, 3])
[2.0, 9.0, 64.0]
```

The power function `math.pow` takes two arguments x and y and computes x^y. To use it with `map`, you therefore need to pass two sequences of numbers as well. If we call these sequences X_i and Y_j, `map` will return the list $[{x_0}^{y_0}, {x_1}^{y_1}, \ldots]$. If the sequences don't have the same length, `None` is used for the missing values.

You also have a function called `reduce`. The idea behind it is to reduce all the elements of a sequence into one single element. The nature of the elements and the nature of the type of reduction is up to you, again through a *reduction* function which you supply. `reduce` needs two arguments: The reduction function (which should take two arguments and return one value), and a sequence. `reduce` works by calling the reduction function on the first two elements of the sequence, then on the result returned from the supplied function and the third element, etc., eventually returning one value. A starting value may be passed as an optional third argument — in that case, the first call to the reduction function will be on the starting value and the first element of the sequence.

One obvious example would be to use `reduce` to calculate the sum of all the numbers in a sequence of any length. You would build a function taking two numbers and returning the sum, and then you'd call `reduce` on this function and some list of numbers. `reduce` would then call your sum function on the first two numbers to calculate their sum; then on this sum and the third number to calculate this cumulated sum, etc. (Un)fortunately, however, this functionality already exists in Python through the `sum` function.

So, imagine instead that you are working with n-dimensional vectors. The Euclidian length of a vector (x_i) is by definition $\sqrt{\sum {x_i}^2}$. To calculate the length of any n-dimensional vector, we might use `reduce` the following way:

```
>>> import math
>>> def vector_length(v):
...     def calc(s, n):
...         return s + n*n
...
...     return math.sqrt(reduce(calc, v, 0))
...
>>> vector_length([3, 4])
5.0
>>> vector_length((2, 6, 3))
7.0
>>> vector_length((1, 2, 3, 4, 5, 3))
8.0
```

A helper function `calc` is defined inside the main function `vector_length`. That means that it is only accessible from inside the main function. This nesting of function definitions is not much used and not necessary either; its only claim to fame is that it signals to the world that the internal function is very closely associated to its mother function and only used by it and nowhere else.

`calc` takes two arguments, an "intermediate sum" and the next vector entry to be included in the sum, and it adds this entry squared to the sum and returns the result. The main function `vector_length` calculates the Euclidian length of its argument vector `v` by calling `reduce` on `calc`, `v`, and the initial sum 0, and eventually taking the squareroot. As demonstrated, the function works on vectors of any length (and both lists and tuples). For the list `[2, 6, 3]`, `reduce` first makes the call `calc(0, 2)`. The result, 4, is used in the next call, `calc(4, 6)`, which yields 40. The final call, `calc(40, 3)` returns 49, and in the end, `reduce` returns 49 and the program goes on to return the square root of 49, namely 7.

`reduce` is a very general tool, and with a little imaginary effort, you can think of a variety of creative ways to use it — possibly even for useful things. The intermediate results passed down through the sequence don't necessarily have to be numbers; e.g., the simple, innocent-looking expression

```
reduce(str.join, ['do', 'you', 'get', 'this'])
```

yields this wonderful string:

```
tgydoodoueydoodouthgydoodoueydoodoutigydoodoueydoodouts
```

And now on to something completely different.

With the perhaps slightly more intuitive `filter` function you may filter out those elements from a sequence which do not qualify, according to a *filtering* function you provide which should take one argument and return either `True` or `False`. Only those elements that pass the filtering (i.e. for which the filtering function returns `True`), are put in the returned sequence. If you pass a string or a tuple, you'll get a string/tuple back; otherwise, you'll get a list. E.g., to create a new string with only the capital letters from some other string, you might call `filter`, passing the string method `str.isupper` as the work horse that actually does the filtering; `str.isupper` will be called on each individual character of the argument string, and only the uppercase letters are put in the result string:

```
>>> letter = """Sarah,
... my parents Erica and Elmer have convinced me that
... YOU are not my type of girl. And it
... is Time we don't See each other anymore.
... goodbye for EVEr,
... Norman"""
>>> filter(str.isupper, letter)
'SEEYOUATSEVEN'
```

Both `map`, `reduce` and `filter` are powered by a function you provide. Thus, their potential is bounded only by the helper function you design. You'll probably get the most benefit from using them if you need to do what they do more than once (or if the helper function you need already exists like in some of the above examples); otherwise, you might usually do the same in one single `for` loop (although that won't earn you as much street credit, of course).

List comprehension is a concise way to create lists in a systematic manner. Let's jump right into an example. Earlier, I showed you how *not* to create a 3x3 matrix. Now I can provide you with a clever alternative. Brilliantly pedagogically, I'll first repeat the wrong way:

```
>>> m = [ [0, 0, 0] ] * 3   # don't do this
>>> m[0][0] = 77
>>> m
[[77, 0, 0], [77, 0, 0], [77, 0, 0]]
```

Only one `[0, 0, 0]` list is created which serves both as the upper, middle and bottom rows of the matrix. We need independent lists for the matrix' rows. List comprehension is the word! What we need is to create a new `[0, 0, 0]` three times and add them to a list. Observe:

```
1  >>> m = [ [0, 0, 0] for i in range(3) ]
2  >>> m[0][0] = 77
3  >>> m
4  [[77, 0, 0], [0, 0, 0], [0, 0, 0]]
5  >>>
6  >>> rows = 7; cols = 4
7  >>> m = [ [0] * cols for i in range(rows) ]
```

A list comprehension is written as a pair of brackets containing first an expression (representing each element of the resulting list) and then a `for` loop. In line 1, the "element expression" is `[0, 0, 0]` and the loop is `for i in range(3)`. The result is that in each round in the loop, a *new* list `[0, 0, 0]` is created and appended to `m`. In lines 7–8, a general expression for creating a *rows* x *cols* matrix is shown. You can even add an `if` statement after the `for` loop as a filter, demonstrated here:

```
>>> [ x*x for x in range(10) if x%3==0 or x%4==0]
[0, 9, 16, 36, 64, 81]
```

This code generates an ordered list where each element is a number of the form $x * x$; the x'es are taken from the list $0, 1, \ldots, 9$, but only those which can be divided evenly by 3 or 4 are used. Thus, the `for` loop (and any following `if`'s) generate a list, and from this list, the result list is created according to the element expression at the beginning. In the matrix example above, the element expression didn't reference the `for` loop's counting variable; here, it does.

Finally, there are also a range of functions that work on sequences of any length. `max`, `min`, `sum` calculate the maximum, minimum and sum, respectively, of all the numbers in the given list, regardless of its length. `max` and `min` even work for lists of strings, or just strings, too — as long as the elements in the given sequence can be compared using < and > (for strings and characters, this means a lexicographical comparison), they will work. There is also the function `sorted` which takes a sequence plus the same optional arguments as `list.sort` and returns a sorted *copy* of the given list. E.g., you might use it to loop over a sequence `s` in sorted order while keeping the original list unaltered:

```
for x in sorted(s):
```

6.5 Sets

Python has a data structure for *sets* in the mathematical sense: Unordered collections of items with no duplicates. Between two sets you can do union, intersection, difference and "symmetric difference" (yielding the elements from one but not both of the sets). You create a set from a sequence, and in the process, the sequence is copied and any duplicates are removed. Here are a few examples; first, let's create two sets:

```
1   >>> from random import randrange
2   >>> a = [ randrange(1, 21) for i in range(10) ]
3   >>> b = [ randrange(11, 31) for i in range(10) ]
4   >>> a
5   [4, 3, 8, 4, 15, 14, 14, 10, 19, 10]
6   >>> b
7   [14, 14, 17, 26, 28, 26, 18, 18, 13, 27]
8   >>>
9   >>> setA = set(a)
10  >>> setB = set(b)
11  >>> setA
12  set([3, 4, 8, 10, 14, 15, 19])
13  >>> setB
14  set([13, 14, 17, 18, 26, 27, 28])
```

The two lists `a` and `b` are generated using list comprehension: `a` contains 10 random numbers between 1 and 20 (both included), and `b` contains 10 random numbers between 10 and 30 (both included). From them I create the two sets `setA` and `setB` in lines 9 and 10, and when printing these sets in lines 11 and 13, I can check that the duplicates have indeed been removed. The sets appear to be ordered; in general, this is not guaranteed.

Next, let's perform some set operations. Line 1 below illustrates set difference (elements in `setA` but not in `setB`, i.e. 14 is out). Line 3 shows the union of the two sets, i.e. all elements in the joint set with duplicates removed. Line 5 shows the set intersection, the elements in both sets — here only 14. And finally, line 7 shows the symmetric difference: The elements belonging to one but not both of the sets.

```
1  >>> setA - setB
2  set([3, 4, 8, 10, 15, 19])
3  >>> setA | setB
4  set([3, 4, 8, 10, 13, 14, 15, 17, 18, 19, 26, 27, 28])
5  >>> setA & setB
6  set([14])
7  >>> setA ˆ setB
8  set([3, 4, 8, 10, 13, 15, 17, 18, 19, 26, 27, 28])
```

You can use sets in `for` loops instead of regular sequences; thus, a quick way of iterating through all elements of a list `L` ignoring duplicates would be `for x in set(L)`: The original list `L` would be untouched since the iteration would be performed on a separate set created from `L`.

6.6 Dictionaries

A dictionary in Python is an unordered collection of key/value pairs. The idea is that a dictionary maintains an association, also called a *mapping*, between each key and its corresponding value. Keys must be immutable objects (so that they may not be altered), and within the same dictionary, all the keys must be unique. In the good old days, the obvious simple, introductory example was to create a dictionary representing a phone book. You know, you look up a person and get the person's phone number. In the dictionary, the names would be the keys, and the numbers would be the values. However, these days any person over the age of 5 apparantly has at least 3 phone numbers, so the same direct, one-to-one correspondence has not survived progress; as explained, you can't insert the same name (key) twice with two different phone numbers (values) in a dictionary. So either, we should find another example, or each value should be a *list* of phone numbers instead. I choose A).

In Denmark, everyone has a unique CPR identity number (a Central Person Registry number) composed by the person's birth date, a dash, and a four-digit code. You're given this number moments after you are born. Thus, we can think of the CPR as a giant dictionary where CPR numbers are the keys, and people (in our simple case, names) are the values. We'll use strings for the CPR numbers since they contain a "-". There are basically two ways of initializing a dictionary, both demonstrated in the following example.

```
>>> cpr = { '130371-3121':"Anders Andersen",
...         '050505-0310':"Trylle Espersen",
...          '121123-2626':"Ib Ædeltville-Hegn"}
>>>
>>> cpr = dict([('130371-3121', "Anders Andersen"),
...               ('050505-0310', "Trylle Espersen"),
...               ('121123-2626', "Ib Ædeltville-Hegn")])
```

The first way is using curly braces. You can create an empty dictionary with just `cpr = {}`, or you can put content in it by listing a comma-separated set of `key:value` pairs. The other way is using the built-in method `dict` (which is similar to `str` and `list`); you call it with a list of tuples, each tuple containing a key and its associated value.

You can access (look up) the value of some key the same way you index a list; instead of an index number, you use the key:

```
>>> cpr['050505-0310']
'Trylle Espersen'
```

Attempting to look up the value for a non-existant key is an error, as demonstrated in lines 1–4 below. You may use the same notation to replace an existing value for some key (line 6), and also to add a new key/value pair to the dictionary (line 7). This last feature constitutes a potential pitfall: If you wish to replace the value of some key but then misspell it, you won't get a warning; instead, you will have added a new key/value pair while keeping the one you wanted to change. To remove an item from a dictionary, use `del` (line 8).

```
1   >>> cpr['290872-1200']
2   Traceback (most recent call last):
3     File "<stdin>", line 1, in ?
4   KeyError: '290872-1200'
5   >>>
6   >>> cpr['130371-3121'] = "Anders Ivanhoe Andersen"
7   >>> cpr['250444-7411'] = "Niels Süsslein"
8   >>> del cpr['050505-0310']
9   >>>
10  >>> cpr
11  {'130371-3121': 'Anders Ivanhoe Andersen',
12  '250444-7411': 'Niels Süsslein',
13  '121123-2626': 'Johan Ædeltville-Hegn'}
```

Printing the contents of the dictionary `cpr` in lines 10–13 demonstrates that the items of a dictionary do not appear in any order, just like in sets. The most recently added item appears in the middle of the printed list.

As with strings and lists, a dictionary is pre-equipped with a bunch of ready-made dictionary methods listed in Figure 6.13. I think from the explanation in the figure, most of them are straightforward; below I'll briefly go over a few.

Method name	**Description**
`D.clear()`	Delete all items from `D`.
`D.copy()`	Return *shallow* copy of `D`; i.e. all items in the copy are references to the items in `D`.
`D.get(key [, defaultValue])`	Return the value associated with `key`. If `key` is not in the dictionary, return `None` or the optional `defaultValue` if it is given.
`D.has_key(key)`	Return `True` if `D` contains the given key, `False` otherwise.
`D.items()`	Return the key/value pairs of `D` as a list of tuples.
`D.keys()`	Return a list of all keys of `D`.
`D.pop(key [, defValue])`	Remove `key` from `D` and return its value. If `key` is not found in `D`, return `defValue` if it is given, otherwise raise KeyError.
`D.popitem()`	Return an arbitrary key/value pair of `D` as a tuple.
`D.setdefault(key [, defValue])`	If `D` doesn't already contain `key`, add it with value `None` or `defValue` if it is given. If `key` is already in `D`, do nothing. In all cases, return the value of `D[key]`.
`D.update(D2)`	Add all items of the dictionary `D2` to `D`, overriding the values of those keys in `D` that also exist in `D2`.
`D.values()`	Return a list of all the values in `D` (i.e., not the key/value pairs, just the values).
`D.iterkeys()`	Return an iterator of all the keys of `D`.
`D.iteritems()`	Return an iterator of all the key/value pairs of `D`.
`D.itervalues()`	Return an iterator of all the values of `D`.

Figure 6.13: Dictionary methods called on some dictionary `D`.

The difference between a *shallow* copy and a *deep* copy is something we have already explored to painful depths with lists. If `d` is a dictionary, then `d.copy()` returns a dictionary with the same keys as `d` which all reference the same objects as the keys of `d`. This becomes a problem if some of the values are mutable objects: Modifying a mutable value through `d` would then affect the copied dictionary as well. The same applies to the `update` method: The keys added to `d` through a `d.update(d2)` call point to the same objects as do the keys of the second dictionary `d2`.

Three methods return a so-called *iterator* object. For now, all you need to know about iterators is that they are designed to iterate efficiently through the elements of a sequence. Thus, you can use an iterator in any `for` loop.

In general, there are also several ways to iterate through all the items in a dictionary. If you need both key and value for each item inside the loop, you might use the `iteritems` methods:

```
for c, name in cpr.iteritems():
```

You could also do

```
for c, name in cpr.items():
```

but the difference is that with the latter option, the entire set of dictionary items is copied into a new list (of *references* to the items, that is, but still). With the first option (and with iterators in general), items are not copied before they are accessed, and they are accessed one at a time as the user needs them. For dictionaries, it should hardly ever be a problem, but when we get to discussing reading files, the same principle forcefully applies.

If you just need the keys, you should use this syntax:

```
for key in cpr:
```

instead of the equally legal

```
for key in cpr.keys():
```

The reason is the same: Calling the method `keys` copies the entire set of keys to a new list. The `for key in cpr:` way implicitly retrieves an iterator from the dictionary and uses that to access the keys one by one, avoiding the copying. In fact, this syntax is equivalent to `for key in cpr.iterkeys()`, one of the other methods listed in Figure 6.13.

One other dictionary method which you would use directly through `dict` deserves mention. It is defined as `fromkeys(S [, defaultValue])`, and you can use it to initialize a dictionary with keys taken from a given sequence `S`. All keys are assigned the optional `defaultValue`, or `None` if it isn't given. You may use any sequence; each element of the sequence will become a key in the new dictionary (see lines 1–2 below).

```
1  >>> dict.fromkeys('abcde')
2  {'a': None, 'c': None, 'b': None, 'e': None, 'd': None}
3  >>>
4  >>> d = dict.fromkeys(xrange(1, 6), [])
5  >>> d[1].append('NB')
6  >>> d
7  {1: ['NB'], 2: ['NB'], 3: ['NB'], 4: ['NB'], 5: ['NB']}
```

The one initialization value is really just that one same value used for all keys. I.e., when using a mutable object, once again you have to beware as demonstrated in lines 4–7. Giving an empty list as the default value should ring the alarm bell in the back of your head, and indeed it is *not* a new, fresh, empty list that is assigned to each key: Appending the string `'NB'` to the list pointed to by the key `1` affects all the list of all the other keys as well.

The previous section introduced sets. Python implements sets using dictionaries: Set members are unique, and so are dictionary keys, so the elements of a set are actually represented as the keys of a dictionary (associated with some dummy value). This has the implication that you can't create a set of items which would not be legal dictionary keys. E.g., lists can't serve as dictionary keys because they are mutable, and hence they can't be put in sets either.

As a final remark in this section, let me introduce to you a way of actually performing a "real" copy of some object. All this talk about several references to the same object may be a little puzzling, especially so since there are no problems as long as the object is immutable. If you really need to copy a dictionary or a list such that all the referenced data, and not just each reference, is in fact copied, there's always the `deepcopy` function of the `copy` module. Observe:

```
>>> from copy import deepcopy
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = [[7,8], [9, 10]]
>>> d = {1:a, 2:b, 3:c}
>>> d
{1: [1, 2, 3], 2: [4, 5, 6], 3: [[7, 8], [9, 10]]}
>>>
>>> d2 = deepcopy(d)
>>> d2[2] = 'NB'
>>> d2[3][0][1] = 'NB'
>>> d
{1: [1, 2, 3], 2: [4, 5, 6], 3: [[7, 8], [9, 10]]}
>>> d2
{1: [1, 2, 3], 2: 'NB', 3: [[7, 'NB'], [9, 10]]}
```

First I define three lists `a`, `b` and `c`, the latter of which is a list of lists. Then I create the dictionary `d` with the keys 1, 2 and 3, each pointing to one of the lists. In line 9, I create a *deep copy* `d2` of `d`, i.e. a copy where *all* data pointed to by `d` is copied. The "deep" refers to the copying mechanism which follows all references recursively, i.e. following references to references to references, copying every data object it meets, until it reaches the bottom (or something it has already copied). This is proven by the fact that I can change the second element of the first list in the list pointed to by key 3 (see lines 11 and 15) in the new dictionary `d2` without touching the original `d`.

Just as a small exercise, what would happen if I deleted the last element from the list `a`..? Anyone?

```
>>> del a[-1]
>>> d
{1: [1, 2], 2: [4, 5, 6], 3: [[7, 8], [9, 10]]}
>>> d2
{1: [1, 2, 3], 2: 'NB', 3: [[7, 'NB'], [9, 10]]}
```

Since `d`'s key 1 references `a` (i.e. the same list that `a` references!), `d` is modified when I modify `a`. However, being a deep copy, `d2` is not.

6.7 An issue concerning default arguments

As promised, I now return briefly to the concept of default arguments, introduced in Section 5.6. If one uses a mutable value as the default argument of a function, one might be in trouble. Figure 6.14 has the story.

This program uses a dictionary `d` to record a selection of nations and the number of times they have won the soccer World Cup. The dictionary is initialized in line 7, and in lines 8–11 I call the function `init_team` on `d`, a nation's name, and a list of years this nation has won the Cup — except that for Sweden and Denmark, I don't pass any list since neither Sweden nor Denmark have won the Cup.

Turning to `init_team`, we indeed see that it takes three arguments; a dictionary, a team, and a list which is defaulted to the empty list. The function simply inserts a new entry in the dictionary with the team name being the key and the list being the value.

In lines 12 and 13, I call another function, `win`, to update the dictionary with two more World Cup winners. This function takes a dictionary, a team and a year and simply appends the year to the entry in the dictionary corresponding to the team. Finally, in lines 14–16 I print out the contents of the updated dictionary.

Figure 6.15 shows the output when running the program. Lines 1–6 look okay; they testify to the calls to `init_team` and `win`. When I print the dictionary, it doesn't look okay, though: Apparently, the Swedes claim that they win the 2010 Cup even though I added this year to Denmark's list!

```
def init_team(d, team, winlist=[]):
    print "Adding %s to the table"%team
    d[team] = winlist

def win(d, team, year):
    print "%s wins the world cup in %d"%(team, year)
    d[team].append(year)

d = {}
init_team(d, "Brazil", [1958, 1962, 1970, 1994, 2002] )
init_team(d, "Italy", [1934, 1938, 1982] )
init_team(d, "Sweden")
init_team(d, "Denmark")
win(d, "Italy", 2006)
win(d, "Denmark", 2010)
print
for team, winlist in d.iteritems():
    print team, winlist
```

Figure 6.14: Bad, bad program `worldcup.py`!

```
Adding Brazil to the table
Adding Italy to the table
Adding Sweden to the table
Adding Denmark to the table
Italy wins the world cup in 2006
Denmark wins the world cup in 2010

Brazil [1958, 1962, 1970, 1994, 2002]
Denmark [2010]
Sweden [2010]
Italy [1934, 1938, 1982, 2006]
```

Figure 6.15: Output from program `worldcup.py`.

The explanation again has to do with the mutability of lists. Once Python reads the definition of the `init_team` function and sees that the parameter `winlist` has a default value, it creates this default value, an empty list, and sets up `winlist` to point to it. When the function is called for the first time with no argument passed to `winlist` (in our exampe when initializing Sweden), the empty list pointed to by default by `winlist` is used as the value of Sweden's dictionary item.

But — the local identifier `winlist` lives on in the function's namespace. Thus, after the function terminates, `winlist` still points to the list now used by Sweden. In further calls to `init_team` passing no argument to `winlist`, like the one initializing Denmark, the *same* default list as before is used in the new dictionary items. Sweden's list! Thus, when inserting the year 2010 in Denmark's list, the Swedes see this update and rejoice too. Major bummer!

We learn from this example that a default argument is not assigned anew every time the function is called. It is assigned *once and for all*, and if the default argument is mutable and you modify it — or if you re-assign the defaulted parameter to some other value — the changes will persist in future calls to the function.

Instead of using lists as default arguments, use this work-around:

```
def init_team(d, team, winlist=None):
    if not winlist:
        winlist = []
    print "Adding %s to the table"%team
    d[team] = winlist
```

This way, a *new* empty list is created each time the function is called with no argument passed to `winlist`. I think now you have received decent schooling in the world of references, yes?

6.8 Not-so-small program: Premier League simulation

As this information packed chapter is drawing to a close, I'll throw in a larger example bringing together all the things you've experienced in the last 30 pages.

I like soccer, and back in the eighties, in the heyday of the Commodore 64, I spent a fair amount of time writing simulations of soccer tournaments. Such a simulation might go like this: From a given list of teams, set up a tournament where all teams play each other and the outcome of each match is somehow determined by randomness. Keep track of all statistics (points, goals scored, etc. The works). Output an ordered ranking list in end.

Here, I'll simulate a small version of England's Premier League, and I'll give each team a *strength* such that the program may capture the fact that some teams are in fact better than others. So why not jump right into it? I've split the program in two files, `premierleague.py` and `premierleague_fncs.py`. The first is the main program, the second contains some functions used by the main program. In the following figures, I'll show you the various parts of the files as I discuss them.

Figure 6.16 shows the first part of the main program. I import everything from the functions module in line 1, and next I define the teams of the tournament as a list of tuples and assign it to the variable `teams`. Each tuple contains a team name and a strength (a number between 1 and 10). I'm going to be using 3-letter versions of the full team names as team IDs, so in line 15 I call a function designed to return a list of such short names from the given list of full names. This function is explained below. The list of IDs is assigned to the variable `ids`.

```
1  from premierleague_fncs import *
2  # list of (team, strength) tuples:
3  teams = [("Manchester United", 10),
4           ("Chelsea", 10),
5           ("Bolton", 7),
6           ("Arsenal", 9),
7           ("Everton", 7),
8           ("Aston Villa", 6),
9           ("Manchester City", 5),
10          ("Liverpool", 8),
11          ("Tottenham", 6),
12          ("Wigan", 4)]
13 # build list of 3-letter versions of team names to be
14 # used as unique IDs:
15 ids = createShortNames(teams)
16 # build a dictionary where each key is a short name (ID)
17 # and the corresponding value is a
18 # (full team name, strength) tuple:
19 id2team = dict( zip(ids, teams) )
20 # build a dictionary representing the match board:
21 matchboard = initializeMatchBoard(ids)
22 # build a dictionary such that each team ID is
23 # associated with a tuple of 7 integers representing
24 # played matches, matches won, matches drawn,
25 # matches lost, goals scored, goals received, and points
26 stats = dict([(id, [0]*7) for id in ids])
```

Figure 6.16: Program `premierleague.py`, part 1.

To be able to efficiently look up a team's full name from its short ID, I create the dictionary `id2team` in line 19. I use the `dict(...)` way of initializing it; to do that, `dict` needs a list of (key, value) tuples, and that's exactly what I get from using the function `zip` on the two lists of IDs and teams. `zip` zips together the first ID of the `ids` list with the first team tuple of the `teams` list, and this pair then becomes a key and value, respectively, of the dictionary.

To remember all match results, I'll define a *match board*. It's supposed to simulate a big table with team names on both the x- and y-axis; as the tournament progresses, the result of each game is inserted in the table entry corresponding to the two teams that played the game. Such a match board is created by a function in line 21. I'll explain how in just a minute.

For each team I wish to keep track of the number of games played, games won, games drawn and games lost; also, the number of goals scored and goals received; and, of course the total number of points earned (3 for a win, 1 for a draw, 0 for a loss). I.e., in total, I want to keep seven values updated for each team during the tournament. Again I'll use a dictionary to look up these statistics for each team using its short name ID as key. The statitics themselves I will represent as a list of integers, initialized to seven 0's.

This bit is carried out in line 26 of Figure 6.16. Again, `dict` needs a list of tuples to initialize a dictionary, and this time I use list comprehension to create it. The expression `[(id, [0]*7) for id in ids]` creates a list of pairs where the first item of each pair is an ID from the `ids` list and the second item is a list of seven 0's. Remember that the list comprehension machinery creates a *new* list of 0's for each id.

Figure 6.17 shows the first part of the file `premierleague_fncs.py`, including the two functions we've used already in `premierleague.py`. Let's go over `initializeMatchBoard` first; its job was to initialize a big table for holding the results of all matches. I'll record the matches using a dictionary where the keys are tuples of the form (ID_1, ID_2). Tuples are immutable, so I'm allowed to use tuples as dictionary keys. The value associated with, e.g., the key `(Eve, Wig)` should eventually be the string `'5-0'` if Everton wins their home game against Wigan $5-0$. For the initialization, I'll use an empty string of three spaces (for pretty-printing purposes to be unveiled later). Note that all teams will meet each other both at home and away, so there'll be an `(Eve, Wig)` key as well as a `(Wig, Eve)` key.

To create this dictionary, I again use a list comprehension. The expression `[(a, b) for a in ids for b in ids]` creates a list of all combinations of tuples `(a, b)` where `a` and `b` are from the list of IDs, `ids`. This list is generated in line 35 of Figure 6.17 and assigned a local variable. Next, in line 36 this list is given to the dictionary method `fromkeys` which besides a list of keys also may take an initialization value to assign to all the keys. This is where I give the 3-space string. The dictionary thus created by `fromkeys` is then returned by the function `initializeMatchBoard`.

The job of the `createShortNames` function of Figure 6.17 is to semi-intelligently create a unique three-letter ID string for each of the team names it is given in the parameter list `teams`. As it is supposed to return a list of these IDs, I start by initializing an empty list, `shortnames`, in line 5. In line 6, the loop through the `teams` list begins. Since `teams` is in fact a list of (team name, strength) tuples, I can unpack each list item into two variables `team` and `strength`. Next, in line 7 I use the string method `split` to split the team name into separate words (by default splitting by each whitespace character). If the name contains more than one word (`len(words)>1`, line 8), I want the ID to end with the first letter of the second word. Thus, in line 9 I create a string `s` containing the first two letters of the first word of the team name, `words[0][:2]`, plus the first letter of the second word, `words[1][0]`. Note that in the slicing operation, I don't provide the starting index because it defaults to 0. If there is only one word in the team name, I simply use the first three letters as the short name (line 11).

```
from random import randrange, gauss

def createShortNames(teams):
    """Return new list of unique, 3-letter
    abbreviations of all the names in the given list"""

    shortnames = []

    for team, strength in teams:
        words = team.split()
        if len(words) > 1:
            s = words[0][:2] + words[1][0]
        else:
            s = words[0][:3]

        while s in shortnames:
            s = raw_input("""Enter 3-letter name
            for %s not in %s"""%(team, shortnames))
        shortnames.append(s)

    return shortnames


def longestName(length, teamtuple):
    """Return maximum of given length and given team's
    name, where team is a (team name, strength) tuple"""

    return max(length, len(teamtuple[0]))


def initializeMatchBoard(ids):
    """Return dictionary representing an initialized
    match board holding all results of all matches among
    the teams from the given list of team IDs.
    Each key is a tuple (team1, team2), and the
    corresponding value is the string '   '. Each time a
    match is played between two teams, the corresponding
    entry should be updated with the result. Think of it
    as a large table (here with teams A, B, C):
       A  B  C
    A  x  x  x
    B  x  x  x
    C  x  x  x  (each x is a 3-space string)"""

    # use list comprehension
    matchboard_keys = [(a, b) for a in ids for b in ids]
    return dict.fromkeys( matchboard_keys, '   ' )
```

Figure 6.17: Program `premierleague_fncs.py`, part 1.

These short name IDs will be used as keys in several dictionaries, so they must be unique. Before I append my current short name `s` to the list, I need to make sure it's not already taken by another team. And if it is, I need to suggest another until I find one which isn't. That's what happens in lines 12–14: I have to stay here until my `s` is not in the list `shortnames`, hence the `while` in line 12. If the current `s` is in the list, I ask the user for an ID and assign it to `s`. I loop until `s` is not in the list. In line 15, the unique `s` is appended to the list, and in line 16, the list is returned.

Figure 6.17 also shows a function called `longestName` which takes a number, `length`, and a team tuple (name/strength again). It simply returns the maximum of the given length and the length of the team name `team[0]`. You'll see in a bit what it can be used for.

The next part of `premierleague_fncs.py` is shown in Figure 6.18: The function `playMatch` simulates playing a match between two teams with the given strengths `a` and `b`. One can think of zillions (well, many) ways to do that; to avoid messing with the upcoming, already much-hyped pretty-printing, I don't want any team to score more than 9 goals per game, so when I randomly choose the total number of goals scored in the match to be played (and call it `goals`) in line 42, I use `min` and `max` in suitable ways to force the number to be at most 9 and at least 0. The random number itself is picked through the `gauss` method of the `random` module; it returns a random number from a Gaussian distribution with the given mean and standard deviation (I use 3 and 2.5). In line 43, I initialize two goal counting variables `goals_a` and `goals_b`, and in line 44 I calculate the sum of the given strengths, `totalstrength`.

My idea is to loop `goals` rounds and in each round make a random but appropriately weighted decision as to who gets a goal. With probability $\frac{a}{a+b}$, the team with strength a scores a goal, and analogously, the other team scores with probability $\frac{b}{a+b}$. This is implemented by picking a random integer between 0 and `totalstrength` and comparing it to `a`; if the number is below `a`, team A scores (lines 46–47). Otherwise, team B scores. After the loop, I return a tuple containing the two goal counts.

The other function of Figure 6.18, `sorttuples`, is used to rank two teams according to their accumulated match statistics. It is defined to take two dictinary items `a` and `b`, each being a key/value tuple, as arguments (why this is so becomes clear in a little while). The key of each item is a team ID, the value is this team's 7-entry match statistics list as initialized in line 26 of Figure 6.16. The function is to be used by the function `sorted`, so it should return -1, 0 or 1 depending of the relative order of the given items. For readability, I introduce two variables `A` and `B` to point to the to match statistics lists.

If one team has more points than the other, I immediately return -1 or 1 (lines 61–64). If they have the same number of points, I look at the goal difference which can be calculated from the statistics tuple entries with indices 4 (goals scored) and 5 (goals received). If one of the teams has a better goal difference than the other, I can return -1 or 1, as appropriate (lines 65–68). As a last resort, if the goal differences are also equal, in lines 69–72 I look at which team has scored the most goals. If the two teams agree on this number as well, I return 0, indicating that their ranking can be arbitrary.

```python
def playMatch(a, b):
    """Play a match between two teams with the given
    strengths. Return the tuple (ga, gb) of the number
    of goals scored by a and b, respectively."""

    # between 0 and 9 goals per match:
    goals = int(min(9, max(0, gauss(3, 2.5))))
    goals_a = goals_b = 0
    totalstrength = a + b
    for i in xrange(goals):
        if randrange(totalstrength) < a:
            goals_a += 1
        else:
            goals_b += 1

    return goals_a, goals_b


def sorttuples(a, b):
    """a and b are key/value dictionary items, i.e.
    (id, T), where id is a team ID and T is its match
    statistics tuple.

    Return 1, 0 or -1 depending on whether a should come
    before b when sorted according to the match
    statistics of the two teams."""

    A = a[1]
    B = b[1]

    if A[6] > B[6]:
        return -1

    elif A[6] < B[6]:
        return 1

    elif A[4]-A[5] > B[4]-B[5]:
        return -1

    elif A[4]-A[5] < B[4]-B[5]:
        return 1

    elif A[4] > B[4]:
        return -1

    elif A[4] < B[4]:
        return 1

    else:
        return 0
```

Figure 6.18: Program `premierleague_fncs.py`, part 2.

Moving back to the main program, Figure 6.19 shows the tournament loop. I count the number of matches played in the variable `matches`. The loop itself is implemented as two nested `for` loops, each iterating through all IDs using the counter variables `teamA` and `teamB`, respectively (lines 31 and 32). This way, I'll get all possible pairs twice (both A, B and B, A). Of course, a team shouldn't play itself, so each time `teamA` is the same as `teamB`, I skip the loop body and move on to the next round (lines 33–34).

To call `playMatch`, I first need the strengths of the two teams. I get them through the `id2team` dictionary, using the IDs as keys: `id2team[teamA]` points to the team tuple of team A, the second item of which (index 1) is its strength. I retreive the strengths in lines 35 and 36 and then call `playMatch` in line 37, unpacking the tuple of goals scored in the match in variables `goalsA` and `goalsB`. Next, I create a result string (e.g. `'3-1'`) in line 38 and assign this string to the relevant entry in the match board dictionary (line 39). Recall that the keys in this dictionary are tuples of two team IDs.

```
27 # Play full tournament:
28 # The teams play each other twice.
29 # 3 points for a win, 1 for a draw, 0 for a loss

30 matches = 0
31 for teamA in ids:
32     for teamB in ids:
33         if teamA == teamB:
34             continue
35         strengthA = id2team[teamA][1]
36         strengthB = id2team[teamB][1]
37         goalsA, goalsB = playMatch(strengthA, strengthB)

38         resultstring = "%d-%d"%(goalsA, goalsB)
39         matchboard[(teamA, teamB)] = resultstring

40         if goalsA > goalsB:          # team A won
41             updateWinner(teamA, stats)
42             updateLoser(teamB, stats)

43         elif goalsA < goalsB:        # team B won
44             updateWinner(teamB, stats)
45             updateLoser(teamA, stats)

46         else:                        # draw
47             updateDraw(teamA, teamB, stats)

48         updateMatch(teamA, goalsA, teamB, goalsB, stats)
49         matches += 1
```

Figure 6.19: Program `premierleague.py`, part 2.

The rest of the loop of Figure 6.19 updates the statistics of teams A and B. If team A won the match (`goalsA > goalsB`), I call the functions `updateWinner` and `updateLoser` on `teamA` and `teamB`, respectively, passing also `stats`, the pointer to the match statistics dictionary. If B won the match, I do the opposite. If the match was a draw, I call `updateDraw`. Finally, I call `updateMatch` and increment the `matches` counter. All these functions are listed in Figure 6.20.

```python
def updateWinner(id, stats):
    """The team given by the id has won a match;
    update stats accordingly."""

    stats[id][1] += 1  # matches won
    stats[id][6] += 3  # points


def updateLoser(id, stats):
    """The team given by the id has lost a match;
    update stats accordingly."""

    stats[id][3] += 1  # matches won


def updateDraw(idA, idB, stats):
    """The two given teams have drawn a match,
    update stats accordingly."""

    stats[idA][2] += 1 # matches drawn
    stats[idA][6] += 1 # points
    stats[idB][2] += 1 # matches drawn
    stats[idB][6] += 1 # points


def updateMatch(idA, goalsA, idB, goalsB, stats):
    """Update stats for teams A and B according
    to the match just played in which A scored
    goalsA and B scored goalsB goals."""

    stats[idA][0] += 1      # matches
    stats[idA][4] += goalsA # goals scored by A
    stats[idA][5] += goalsB # goals received by B
    stats[idB][0] += 1      # matches
    stats[idB][4] += goalsB # goals scored by B
    stats[idB][5] += goalsA # goals scored by A
```

Figure 6.20: Program `premierleague_fncs.py`, part 3.

Recall that for a given team ID `id`, `stats[id]` points to a list of 7 values. These values are the number of matches played (index 0), matches won (index 1), matches drawn (index 2), matches lost (index 3), goals scored (index 4), goals received (index 5), and points (index 6). Thus, updating a match winner `id` means incrementing its "won matches" value, `stats[id][1]`, and adding 3 points to its "points total", `stats[id][6]`, as shown in lines 78 and 79 of Figure 6.20. Similarly, to update a loser, I increment its "lost matches" value (index 3, see lines 80–83), and to update two teams after a draw, index 2 ("drawn matches") is incremented for both, as well as their points scores (index 6); see lines 84–90. Finally, the `updateMatch` function updates the "goals scored" and "goals received" totals for both teams, as well as their "games played" value (lines 91–100).

Figure 6.21, then, shows the remainder of the main program which delivers the output. To print things in a pretty fashion, one needs to get down and dirty with counting characters, calculating column widths etc. That part in itself is not pretty and in fact rather tedious, but as you know, underneath a shining surface you often find an ugly truth!

I'm going to need a line of dashes (`----` etc.) of suitable length, so I calculate this length once and for all in line 51 of Figure 6.21. Then I move on to print the match board. I want it printed such that the team IDs appear alphabetically. To that end, I use the list method `sorted` to create a sorted copy of the `ids` list and assign it to the variable `sids` in line 54. I then pass this list to the string method `join`, using a space as the delimiter character. This creates a string with the sorted team IDs separated by one space each. I then print this string, prefixed by another three spaces (line 55) as the top row, the header, of the table.

Next, in alphabetical order, each team should be printed on its own row followed by the results of all the games it played. I again use the sorted list `sids` of IDs to iterate through the team IDs, both in the outer `for` loop in line 56 in which I first print the current team ID (line 57, note the comma yielding a space after the ID instead of a newline), and also in the internal loop (line 58). In the inner-most loop body, I simply print the result of the game played between the two current teams, `teamA` and `teamB`, looking up this result in the `matchboard` dictionary using the tuple `(teamA, teamB)` as key; again, the comma creates a space after the result (ensuring a total column length of four — this is why I didn't want any team to score more than 9 goals in one game..). After each internal loop, I end the row by printing just a newline (line 60).

Lastly, and most importantly, of course I want a table showing the final standings, including all the wonderful statistics. The columns of this table should be full team name and then the headers of each of the entries in the team statistics tuples. To really pretty-print it in a general manner, of course I need to *calculate* the width of the first column rather than just count the length of the longest team name, just in case I later change the list of teams playing the tournament. And strangely — what a coincidence — this high level of ambition leads me to use the `reduce` function introduced earlier.

What I need is the length of the longest name. The full team names are represented as the first items of the tuples in the `teams` list, remember? To go through all these items, keeping track of the maximum length seen so far,

```
50 # pretty-print match board:
51 dashes = (1+len(teams))*4
52 print "\nComplete match board (%d matches):"%matches
53 print "-"*dashes

54 sids = sorted(ids)
55 print "   ", " ".join(sids)
56 for teamA in sids:
57     print teamA,
58     for teamB in sids:
59         print matchboard[teamA, teamB],
60     print

61 # find length of longest team name:
62 namemax = reduce(longestName, teams, 0)

63 # create formatting string:
64 f = "%2d %2d %2d %2d %2d %2d %2d"

65 print "\nFinal standings:"
66 print "-"*dashes

67 print "Team%s  M  W  D  L Gs Gr  P"%(" "*(namemax-4))
68 print "-"*dashes

69 # sort teams by their statistics:
70 ranking = sorted(stats.iteritems(), sorttuples)

71 for id, (m, w, d, l, gs, gr, p) in ranking:
72     name = id2team[id][0]
73     print "%s%s"%(name, " "*(namemax-len(name))),
74     print f%(m, w, d, l, gs, gr, p)
```

Figure 6.21: Program `premierleague.py`, part 3.

I use `reduce`. I just need a tailor-made function which compares this current maximum with the length of the team name in the current team tuple, and that's exactly what the `longestName` function does (see lines 17–20 of Figure 6.17). So, the expression `reduce(longestName, teams, 0)` in line 62 of Figure 6.21 first calls `longestName` with the given starting value 0 and the first item from the `teams` list. Inside `longestName`, 0 is compared to the length of the string in index 0 of the item (which is a tuple), and the maximum of the two is returned. Next, `reduce` moves on to call `longestName` with this new, current maximum and the next item from the `teams` list. And so forth and so on. Eventually, the global maximum over all the list is returned by `reduce` and assigned to the variable `namemax`.

Recall that to print an integer of unknown size in a slot of fixed width, say 2, and print it right-oriented, you use the string conversion specifier `%2d`. In line 64, I create a formatting string `f` in which I can later insert all the seven statistics values in right-oriented columns of width 3. For readability, I use `%2d` plus an explicit space rather than `%3d` for each column.

In lines 67–68 I print the header. For the first column, I need to add a number of spaces after the "Team" header, and I calculate this number as `namemax-4`. Then, in line 70, the decisive sorting of the teams takes place. I again use `sorted`, but this time I pass to it not only a list of items to be sorted but also a function telling it how to sort them. The list I want sorted is the list of key/value items from the `stats` dictionary, where the keys are team IDs and the values are the statistics tuples. I call the dictionary method `iteritems` to get the item list. You saw the sorting function, `sorttuples`, in Figure 6.18; recall that it ranks the dictionary items first by the team's points, then by goal difference, then by goals scored.

Calling `sorted` produces a new, sorted list of the dictionary items from `stats`. I can now iterate through this list in the `for` loop of line 71, unpacking each item into a key value (`id`), and a stats tuple (further unpacked into `(m, w, d, l, gs, gr, p)`). I need the `id` to look up the full name in the `id2team` dictionary in line 72 (that's why I didn't just sort the match stats tuples), and once I have that, I can print it followed by a suitable number of spaces, again calculated from the current name length and the overall maximum name length `namemax`. Next, in line 74 I use the formatting string `f` and insert all the individual values of the statistics tuple for the current team. Since the list is sorted, the tournament winner will appear at the top of the table.

Ah. I do think one might call this a tour de force of lists, tuples and dictionaries, eh compadre? Let's have a look at the output. If your enthusiasms for soccer and/or tables are weaker than mine, you might not find the output fully worth the effort☺. To compensate, I'll print it in a happy green.

```
Complete match board (90 matches):
--------------------------------------------
    Ars AsV Bol Che Eve Liv MaC MaU Tot Wig
Ars     0-0 0-1 2-2 3-1 5-2 1-0 2-0 3-1 2-3
AsV 1-1     2-1 0-0 2-0 2-0 0-0 0-2 1-3 0-0
Bol 0-1 3-2     1-0 1-2 1-1 0-2 3-1 0-3 1-3
Che 0-3 4-0 0-0     3-3 2-3 0-0 5-0 0-0 5-1
Eve 0-0 0-0 1-0 1-2     2-2 2-2 2-2 1-2 1-1
Liv 3-1 0-0 1-3 0-0 2-2     4-3 2-0 4-0 2-2
MaC 0-0 0-2 1-3 0-2 1-2 1-2     0-0 1-0 0-1
MaU 2-1 0-0 0-0 3-2 3-3 0-0 1-0     4-0 1-0
Tot 0-1 0-0 0-1 1-5 2-2 0-0 0-0 0-2     3-1
Wig 0-1 0-2 1-3 1-6 0-0 3-2 2-1 0-1 2-1
```

As a random check, let's calculate the stats of Wigan, reading through the match board: 6 games won, 4 games drawn, 8 games lost, yielding 22 points. 21 goals scored, 32 goals received. We'll check those numbers in the final standings next, but first, note that, e.g. the calculated short version of the two-word name "Manchester United" is "MaU", as expected. Note also the empty diagonal whose entries are strings of 3 spaces each — the default value given when constructing the `matchboard` dictionary in line 36 of Figure 6.17.

Conferring with the final standings, we see that the stats computed by hand for Wigan are correct:

```
Final standings:
--------------------------------------------
Team                M  W  D  L Gs Gr  P
--------------------------------------------
Arsenal            18  9  5  4 27 16 32
Manchester United  18  8  6  4 22 20 30
Chelsea            18  7  7  4 38 19 28
Bolton             18  8  3  7 22 21 27
Liverpool          18  6  8  4 30 27 26
Aston Villa        18  5  9  4 14 14 24
Wigan              18  6  4  8 21 32 22
Everton            18  3 11  4 25 28 20
Tottenham          18  4  5  9 16 28 17
Manchester City    18  2  6 10 12 22 12
```

Note that the width of the first column is indeed exactly the length of the longest team name plus 1. Note also that the ranking more or less mirrors the individual strengths of the teams.

While we're at it, having the whole machine up and running, let's play another tournament (a brownish one):

```
Complete match board (90 matches):
--------------------------------------------
    Ars AsV Bol Che Eve Liv MaC MaU Tot Wig
Ars     0-0 2-1 0-0 4-1 2-4 2-0 3-0 0-2 0-0
AsV 0-3     0-0 0-0 1-1 3-2 1-2 0-4 2-3 0-0
Bol 0-0 3-2     0-2 1-0 0-2 2-5 0-0 0-2 3-1
Che 1-0 0-3 0-0     2-1 1-2 3-1 1-4 0-0 5-1
Eve 0-2 1-0 0-2 3-0     7-2 0-2 0-0 3-1 0-0
Liv 1-2 0-1 2-1 3-2 6-0     0-0 2-0 3-2 3-3
MaC 1-2 0-1 0-0 1-1 3-4 1-2     0-0 1-4 0-0
MaU 0-0 3-0 1-1 1-0 0-0 4-2 2-3     0-0 1-0
Tot 1-2 0-0 2-1 2-5 4-4 0-5 0-0 1-1     0-1
Wig 0-0 2-1 1-1 0-0 0-0 0-0 3-2 1-3 2-3
```

In the standings calculated from this match board, we see the `sorttuples` function fully blossoming. This tournament was won by Liverpool but only at the narrowest margin possible: Arsenal and Liverpool both ended up with 33 points, and they both have a goal difference of $+12$. Thus, the impressive 41 goals scored by Liverpool is what gives them the trophy, as dictated by lines 69–72 of the `sorttuples` function in Figure 6.18, since Arsenal only scored 24. Thus, Liverpool avenge the bitter defeat suffered back in 1989.

```
Final standings:
--------------------------------------------
Team                M  W  D  L Gs Gr  P
--------------------------------------------
Liverpool          18 10  3  5 41 29 33
Arsenal            18  9  6  3 24 12 33
Manchester United  18  7  8  3 24 14 29
Chelsea            18  6  6  6 23 22 24
Tottenham          18  6  6  6 27 30 24
Everton            18  5  6  7 25 30 21
Bolton             18  4  7  7 16 22 19
Wigan              18  3 10  5 15 22 19
Manchester City    18  4  6  8 22 27 18
Aston Villa        18  4  6  8 15 24 18
```

This program has painstakingly exhibited the strengths of sequences and dictionaries and their related functions. It also demonstrates what lists should not be used for: The whole bookkeeping necessary to keep track of the meaning of the seven values in the statistics tuples of each team is really intolerable. That the number of goals scored is located in index 4 is a fact very easily forgotten or confused, and this whole data structure is very obscure indeed. It would be much better if we could reference the "goals scored" value by some meaningful name rather than by some meaningless index. In the next chapter, you'll see how we can do that.

Lastly, note how I through the whole program exploit the fact that arguments passed to functions are references to data, not copies of data. For example, each time I pass the dictionary `stats` to one of the update functions of Figure 6.20, the updates are performed on the *original* `stats`, not some local copy. Thus, they "survive" when the function call terminates, and that is exactly the behavior I need.

The 1988/1989 season match between Liverpool and Arsenal had been postponed until the very end of the season, and in the meantime Liverpool had won the FA Cup, meaning they had the chance of a historic second Double. Before the match, Arsenal were on 73 points with 71 goals scored and 36 received (meaning a goal difference of +35); Liverpool were on 76 points with 65 goals scored and 26 received (a difference of +39). Thus, Arsenal needed to win by at least two goals to take the title: That would level the teams at 73 points and goal difference +37 but with Arsenal having scored more goals than Liverpool. Liverpool had not lost by two goals at home at Anfield for nearly four years.

After a goalless first half, Alan Smith scored soon after the restart, heading in a free kick from Nigel Winterburn. As the clock ticked down, though, it looked as if Arsenal were going to win the battle but lose the war. However, in injury time, in Arsenal's last attack, Michael Thomas surged from midfield, ran onto a Smith flick-on, evaded Steve Nicol and shot low past Bruce Grobbelaar to score Arsenal's second, and win the title, Arsenal's first in eighteen years. (Source: `http://en.wikipedia.org/wiki/Michael_Thomas`).

Please indulge me — here's a final one, sorry. It won't be in the exam.

```
Complete match board (90 matches):
--------------------------------------------
    Ars AsV Bol Che Eve Liv MaC MaU Tot Wig
Ars     2-2 3-3 0-0 4-1 0-0 1-0 0-2 1-1 2-1
AsV 1-3     0-0 0-0 0-0 0-2 0-1 2-0 2-4 1-0
Bol 0-1 0-0     0-0 0-0 0-0 2-1 1-1 2-4 2-0
Che 1-3 6-3 3-0     0-0 3-1 0-0 0-1 4-1 0-0
Eve 0-0 1-0 1-1 1-5     2-2 3-0 1-1 2-0 4-1
Liv 1-2 3-2 0-1 0-0 4-1     0-0 2-1 0-0 3-1
MaC 0-6 0-0 0-0 1-1 0-0 0-0     0-0 3-6 1-0
MaU 0-4 2-1 0-1 1-1 5-0 1-0 2-1     2-1 4-0
Tot 0-0 0-0 0-1 0-0 2-1 1-0 1-0 0-0     1-1
Wig 1-4 1-2 1-1 0-3 2-3 2-4 1-4 1-3 1-2


Final standings:
--------------------------------------------
Team                  M  W  D  L Gs Gr  P
--------------------------------------------
Arsenal              18 10  7  1 36 14 37
Manchester United 18  9  5  4 26 16 32
Chelsea              18  6 10  2 27 12 28
Tottenham            18  7  7  4 24 20 28
Liverpool            18  6  7  5 22 17 25
Bolton               18  5 10  3 15 15 25
Everton              18  5  8  5 21 27 23
Manchester City      18  3  8  7 12 23 17
Aston Villa          18  3  7  8 16 25 16
Wigan                18  0  3 15 14 44  3
```

Randomness truly is wonderful: We are blessed with an extraordinary outcome! Not only does Tottenham's remarkable $6 - 3$ win against Manchester City help them to a healthy 4th place in the standings despite their low strength of only 6; we may also wonder at the poor fate of Wigan winning no game in the entire season, losing all but 3!

That's all for the sports this evening. Thanks for watching.

Now you know about:

- Lists and tuples
- Unpacking
- Mutable vs. immutable
- The `del` command
- The `xrange` function
- List methods
- `enumerate`, `zip`, `map`, `reduce` and `filter`
- List comprehension
- Sets
- Dictionaries
- Dictionary methods
- The `deepcopy` function