6.2 Lists

Lists are the most important type of structured data. They are used in every branch of programming. Lists appear in one form or another in almost all programming laguages. Python in particular makes much use of lists. Python also has some very useful tools for working with lists, which makes this the natural data structure to use in many situations.

Lists and strings share many common features. As with strings, we can use numeric indexes to retrieve the individual elements of a list. Consider, for example, the list

```
L = ["first", "second", "third"]
```

Then L[0] is the initial element of L, the string "first", while L[2] is the string "third". We can use the + operator to concatenate two lists, and the ∗ operator to multiply a list times an integer. As with strings, we can ask if object x is in L:

```
if x in L:
```

and we can find its index with L.index(x).

There is one significant difference between lists and strings: lists are mutable structures and strings are immutable. This means that we can change the contents of a list after it is created; this is not true of strings. Lists have an append() method that adds its argument onto the end of the list. For example, we might say:

```
L = []
L.append( "John" )
L.append( "George" )
L.append( "Paul" )
L.append( "Ringo" )
```

This turns L into the list ["John", "George", "Paul", "Ringo"].

Similarly, we can have a function modify the contents of a list. Think carefully about the next example. We know that the following program with integer variables won't work:

```
def Change(x):
    x = 24

def main():
    num = 4
    Change(num)
    print(num)
```

This prints 4; in spite of the function name, variable num isn't changed by the call Change(num).

Contrast this with the following program, which is similar but uses a list instead of an integer argument for the function call:

```
def ChangeList( L ):
    L.append(24)

def main():
    L = [4]
    ChangeList(L)
    print(L)

main()
```

This time the change function, here called ChangeList( ), does modify something. If we run this program it prints the list [4, 24]. The difference is that function ChangeList( ) changes the contents of list L, it doesn't change the list value, which is the address in memory where the list is stored. This value is what is sent to the ChangeList( ) function. At every point while this program is running the list variables L, both the L in main() and the L in ChangeList(), all contain the locations of the same list.

Here is a third related example. At first glance it looks as though this program will print [24] as the value of L, but this is not correct.

```
def ChangeListBadly( L ):
    L = [ ]
    L.append(24)

def main():
    L = [4]
    ChangeListBadly(L)
    print(L)

main()
```

Remember that each function's variables are visible only to that function. The variable L in function ChangeListBadly( ) and the variable L in main() are two completely different variables that happen to have the same name. When we call ChangeListBadly(L) these variables initially have the same value (i.e., they refer to the same memory location where a list is stored), but as soon as function ChangeListBadly( ) says

```
L = [ ]
```

this is no longer the case. This assignment statement attaches to variable L in ChangeListBadly( ) a completely different list from that in variable L of main( ). After this, the append statement appends onto this new list rather than list L of main( ). This program ends up printing [4], and the call to ChangeListBadly(L) has no effect. When a list is passed to a function, it is seldom useful to have the function re-assign to the list variable; that destroys the connection between the list argument to the function and lists that occur anywhere else in the program.

Here are some of the common operations with lists:

A. Making lists:
L = [] (the empty list, which is the list with no elements)
L = [”abc”, ”de”, ”fghij”, 1, [2, 3]]: this list has 5 elements: three strings, one integer and one list.
L = L1 + L2, whereL1 andL2 are lists. This concatenatesL1 andL2 into a new listL.
L = L1 * 3, whereL1 is a list. This makes a new listL, which is the concatenation ofL1 3 times, as inL1 + L1 + L1.

B. Indexing:
L[0]: the first element in listL
L[1]: the second element in listL
L[2:5]: a *slice* of listL, which is a new list consisting of the elements at positions 2, 3, and 4 (but not 5) ofL.

C. Changing the contents of the list, without changing the list itself:
L[i] = a changes the value of theith entry ofL toa
L.append(x): addsx to the end of the listL
L.extend(L1): whereL1 is a list. This adds all the entries ofL1 ontoL
L.sort(): sorts, or arranges in order, the entries ofL
L.sort(compare): again, this sorts the entries ofL, usingcompare as a function to compare two entries.compare(a, b) should return -1 ifa ¡ b, 0 ifa == b, and 1 ifa ¿ b
L.reverse(): reverses the order of the entries of L
del L[i]: deletes theith element ofL
L[i:j] = [] deletes the indexi throughj slice ofL

D. Other methods
len(L): the length, or number of entries, ofL
for x in L: iterates a loop over all entries ofL
x in L: returnsTrue ifL has an entry whose value isx
L.index(v): returns the index of the first entry ofL that equalsv; generates an *exception* (i.e. crashes) ifL does not containv

Here is an example that illustrates the power and ease of use of Python's list structures. We will write a that allows the user to enter a list of names, edit it, put it in alphabetical order, and then print it out. Each of these steps can be handled by its own function since the list contents can be modified by a function.

The main( ) function for this program is quite easy. For the first version we will dispense with editing and just read and print the list. For this version main( ) is just:

```
def main():
    L = [ ]
    ReadNames(L)
    PrintNames(L)
```

Function ReadNames( ) is our usual input loop for strings; each time we get a non-blank string we append it onto the list:

```
def ReadNames(L):
   done = False
   while not done:
      name=raw_input("Enter a name, or a blank to quit: ")
      if name == "":
         done = True
      else:
         L.append(name)
```

A simple **for**-loop is all that is needed to print the list:

```
def PrintNames(L):
    print "Here is the list: "
    for name in L:
        print(name)
```

Putting main( ), ReadNames( ) and PrintNames( ) together yields the first version of our program. If we add one more line to main( ) we can get the names to print in alphabetical order:

```
def ReadNames(L):
    done = False
    print("Enter names, and a blank to quit.")
    while not done:
        name=input("Enter a name: ")
        if name == "":
            done = True
        else:
            L.append(name)

def PrintNames(L):
    print("Here is the list: ")
    for name in L:
        print(name)

def main():
    L = [ ]
    ReadNames(L)
    L.sort()
    PrintNames(L)
```

Reading and printing names, first version

Here is a typical user interaction with this program; note that the output is alphabetized:

Enter names, and a blank to quit.
Enter a name: `john`
Enter a name: `paul`
Enter a name: `george`
Enter a name: `ringo`
Enter a name:
Here is the list:
george
john
paul
ringo

Editing the list is a little more interesting. While more elaborate edits are possible, we will keep this simple by just allowing the user to delete names. A **for**-loop can run through the list and ask the user if each name should be deleted. However, modifying a list while it is being used to sequence a **for**-loop is a very bad idea that can lead to mysterious bugs that are hard to detect. Rather than this, we will build up a "delete list" of names to be removed, then

in a second loop perform the deletions. Python makes it easy to create and process lists, so this is a common solution to programming problems in Python.

To delete an individual name from a list we need to know the index of this item in the list. Lists have an index( ) method that returns the position of any item in the list. It is important to insure that the argument to this method is actually an element of the list; your program will crash if you ask for the index of something that isn't in the list. Fortunately, we will only ask for the index of names in the delete list, which all come from the original list. Here is all that is needed to delete the entry stored in variable name from list L:

```
i = L.index(name)
del L[i]
```

Putting these ideas together, here is the DeleteNames( ) function:

```
def DeleteNames(L):
    print "Which of these should be deleted?"
    deleteList = []
    for name in L:
        print(name)
        print("Delete?")
        response=input("Enter 'y' to delete: ")
        if response == 'y':
            deleteList.append(name)
    for name in deleteList:
        i = L.index(name)
        del L[i]
```

Here is the complete code for the final version of our program:

```
# This programs allows the user to read,
# a list is names, edit that list, alphabetize it,
# and then print it.
def ReadNames(L):
    # This reads names and stores them in a list
    done = False
    print("Enter names, and a blank to quit.")
    while not done:
        name=input("Enter a name: ")
        if name == "":
            done = True
        else:
            L.append(name)

def DeleteNames(L):
    # This runs through L and asks if each
    # entry should be deleted.
    print("Which of these should be deleted?")
    deleteList = []
    for name in L:
        print(name)
        print("Delete?")
        response=input("Enter 'y' to delete: ")
        if response == 'y':
           deleteList.append(name)
    for name in deleteList:
        i = L.index(name)
        del L[i]

def PrintNames(L):
    # This prints all the entries of L
    print "Here is the list: "
    for name in L:
        print(name)

def main():
    L = [ ]
    ReadNames(L)
    DeleteNames(L)
    PrintNames(L)

main()
```

Program 6.2.1: Reading and printing names, final version