Chapter 6

Strings

Strings are a data type in Python for dealing with text. Python has a number of powerful features for manipulating strings.

6.1 Basics

Creating a string A string is created by enclosing text in quotes. You can use either single quotes, `'`, or double quotes, `"`. A triple-quote can be used for multi-line strings. Here are some examples:

```
s = 'Hello'
t = "Hello"
m = """This is a long string that is
spread across two lines."""
```

Input Recall from Chapter 1 that when getting numerical input we use an **`eval`** statement with the **`input`** statement, but when getting text, we do not use **`eval`**. The difference is illustrated below:

```
num = eval(input('Enter a number: '))
string = input('Enter a string: ')
```

Empty string The empty string `''` is the string equivalent of the number 0. It is a string with nothing in it. We have seen it before, in the print statement's optional argument, `sep=''`.

Length To get the length of a string (how many characters it has), use the built-in function **`len`**. For example, **`len`**`('Hello')` is 5.

6.2 Concatenation and repetition

The operators + and * can be used on strings. The + operator combines two strings. This operation is called *concatenation*. The * repeats a string a certain number of times. Here are some examples.

Expression	Result
`'AB'+'cd'`	`'ABcd'`
`'A'+'7'+'B'`	`'A7B'`
`'Hi'*4`	`'HiHiHiHi'`

Example 1 If we want to print a long row of dashes, we can do the following

```
print('-'*75)
```

Example 2 The + operator can be used to build up a string, piece by piece, analogously to the way we built up counts and sums in Sections 5.1 and 5.2. Here is an example that repeatedly asks the user to enter a letter and builds up a string consisting of only the vowels that the user entered.

```
s = ''
for i in range(10):
    t = input('Enter a letter: ')
    if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
        s = s + t
print(s)
```

This technique is very useful.

6.3 The `in` operator

The `in` operator is used to tell if a string contains something. For example:

```
if 'a' in string:
    print('Your string contains the letter a.')
```

You can combine `in` with the `not` operator to tell if a string does not contain something:

```
if ';' not in string:
    print('Your string does not contain any semicolons.')
```

Example In the previous section we had the long if condition

```
if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
```

Using the `in` operator, we can replace that statement with the following:

```
if t in 'aeiou':
```

6.4 Indexing

We will often want to pick out individual characters from a string. Python uses square brackets to do this. The table below gives some examples of indexing the string `s='Python'`.

Statement	Result	Description
`s[0]`	`P`	first character of `s`
`s[1]`	`y`	second character of `s`
`s[-1]`	`n`	last character of `s`
`s[-2]`	`o`	second-to-last character of `s`

- The first character of `s` is `s[0]`, not `s[1]`. Remember that in programming, counting usually starts at 0, not 1.
- Negative indices count backwards from the end of the string.

A common error Suppose `s='Python'` and we try to do `s[12]`. There are only six characters in the string and Python will raise the following error message:

```
IndexError: string index out of range
```

You *will* see this message again. Remember that it happens when you try to read past the end of a string.

6.5 Slices

A *slice* is used to pick out part of a string. It behaves like a combination of indexing and the **`range`** function. Below we have some examples with the string `s='abcdefghij'`.

```
index:     0 1 2 3 4 5 6 7 8 9
letters:   a b c d e f g h i j
```

Code	Result	Description
`s[2:5]`	`cde`	characters at indices 2, 3, 4
`s[ :5]`	`abcde`	first five characters
`s[5: ]`	`fghij`	characters from index 5 to the end
`s[-2:]`	`ij`	last two characters
`s[ : ]`	`abcdefghij`	entire string
`s[1:7:2]`	`bdf`	characters from index 1 to 6, by twos
`s[ : :-1]`	`jihgfedcba`	a negative step reverses the string

- The basic structure is

  string name[*starting location* : *ending location+1*]

  Slices have the same quirk as the **range** function in that they do not include the ending location. For instance, in the example above, `s[2:5]` gives the characters in indices 2, 3, and 4, but not the character in index 5.

- We can leave either the starting or ending locations blank. If we leave the starting location blank, it defaults to the start of the string. So `s[:5]` gives the first five characters of `s`. If we leave the ending location blank, it defaults to the end of the string. So `s[5:]` will give all the characters from index 5 to the end. If we use negative indices, we can get the ending characters of the string. For instance, `s[-2:]` gives the last two characters.

- There is an optional third argument, just like in the **range** statement, that can specify the step. For example, `s[1:7:2]` steps through the string by twos, selecting the characters at indices 1, 3, and 5 (but not 7, because of the aforementioned quirk). The most useful step is -1, which steps backwards through the string, reversing the order of the characters.

6.6 Changing individual characters of a string

Suppose we have a string called `s` and we want to change the character at index 4 of `s` to `'X'`. It is tempting to try `s[4]='X'`, but that unfortunately will not work. Python strings are *immutable*, which means we can't modify any part of them. There is more on why this is in Section 19.1. If we want to change a character of `s`, we have to instead build a new string from `s` and reassign it to `s`. Here is code that will change the character at index 4 to `'X'`:

```
s = s[:4] + 'X' + s[5:]
```

The idea of this is we take all the characters up to index 4, then `X`, and then all of the characters after index 4.

6.7 Looping

Very often we will want to scan through a string one character at a time. A for loop like the one below can be used to do that. It loops through a string called `s`, printing the string, character by character, each on a separate line:

```
for i in range(len(s)):
    print (s[i])
```

In the **range** statement we have **len**`(s)` that returns how long `s` is. So, if `s` were 5 characters long, this would be like having **range**`(5)` and the loop variable `i` would run from 0 to 4. This means that `s[i]` will run through the characters of `s`. This way of looping is useful if we need to keep track of our location in the string during the loop.

If we don't need to keep track of our location, then there is a simpler type of loop we can use:

```
for c in s:
    print(c)
```

This loop will step through `s`, character by character, with `c` holding the current character. You can almost read this like an English sentence, "For every character `c` in `s`, print that character."

6.8 String methods

Strings come with a ton of *methods*, functions that return information about the string or return a new string that is a modified version of the original. Here are some of the most useful ones:

Method	Description
`lower()`	returns a string with every letter of the original in lowercase
`upper()`	returns a string with every letter of the original in uppercase
`replace(x,y)`	returns a string with every occurrence of `x` replaced by `y`
`count(x)`	counts the number of occurrences of `x` in the string
`index(x)`	returns the location of the first occurrence of `x`
`isalpha()`	returns **`True`** if every character of the string is a letter

Important note One very important note about `lower`, `upper`, and `replace` is that they do not change the original string. If you want to change a string, `s`, to all lowercase, it is not enough to just use `s.lower()`. You need to do the following:

```
s = s.lower()
```

Short examples Here are some examples of string methods in action:

Statement	Description
`print(s.count(' '))`	prints the number of spaces in the string
`s = s.upper()`	changes the string to all caps
`s = s.replace('Hi','Hello')`	replaces each `'Hi'` in `s` with `'Hello'`
`print(s.index('a'))`	prints location of the first `'a'` in `s`

`isalpha` The `isalpha` method is used to tell if a character is a letter or not. It returns **`True`** if the character is a letter and **`False`** otherwise. When used with an entire string, it will only return **`True`** if every character of the string is a letter. The values **`True`** and **`False`** are called booleans and are covered in Section 10.2. For now, though, just remember that you can use `isalpha` in if conditions. Here is a simple example:

```
s = input('Enter a string')
```

```
if s[0].isalpha():
    print('Your string starts with a letter')

if not s.isalpha():
    print('Your string contains a non-letter.')
```

A note about `index` If you try to find the index of something that is not in a string, Python will raise an error. For instance, if `s='abc'` and you try `s.index('z')`, you will get an error. One way around this is to check first, like below:

```
if 'z' in s:
    location = s.index('z')
```

Other string methods There are many more string methods. For instance, there are methods `isdigit` and `isalnum`, which are analogous to `isalpha`. Some other useful methods we will learn about later are `join` and `split`. To see a list of all the string methods, type `dir(str)` into the Python shell. If you do this, you will see a bunch of names that start with ___. You can ignore them. To read Python's documentation for one of the methods, say the `isdigit` method, type `help(str.isdigit)`.

6.9 Escape characters

The backslash, `\`, is used to get certain special characters, called escape characters, into your string. There are a variety of escape characters, and here are the most useful ones:

- `\n` the *newline character*. It is used to advance to the next line. Here is an example:

  ```
  print('Hi\n\nthere!')
  ```

  ```
  Hi

  There!
  ```

- `\'` for inserting apostrophes into strings. Say you have the following string:

  ```
  s = 'I can't go'
  ```

  This will produce an error because the apostrophe will actually end the string. You can use `\'` to get around this:

  ```
  s = 'I can\'t go'
  ```

  Another option is to use double quotes for the string:

  ```
  "s = I can't go"
  ```

- `\"` analogous to `\'`.

- `\\` This is used to get the backslash itself. For example:

```
filename = 'c:\\programs\\file.py'
```

- `\t` the tab character

6.10 Examples

Example 1 An easy way to print a blank line is `print()`. However, if we want to print ten blank lines, a quick way to do that is the following:

```
print('\n'*9)
```

Note that we get one of the ten lines from the `print` function itself.

Example 2 Write a program that asks the user for a string and prints out the location of each `'a'` in the string.

```
s = input('Enter some text: ')
for i in range(len(s)):
    if s[i]=='a':
        print(i)
```

We use a loop to scan through the string one character at a time. The loop variable `i` keeps track of our location in the string, and `s[i]` gives the character at that location. Thus, the third line checks each character to see if it is an `'a'`, and if so, it will print out `i`, the location of that `'a'`.

Example 3 Write a program that asks the user for a string and creates a new string that doubles each character of the original string. For instance, if the user enters `Hello`, the output should be `HHeellllloo`.

```
s = input('Enter some text: ')
doubled_s = ''
for c in s:
    doubled_s = doubled_s + c*2
```

Here we can use the second type of loop from Section 6.7. The variable `c` will run through the characters of `s`. We use the repetition operator, `*`, to double each character. We build up the string `s` in the way described at the end of Section 6.2.

Example 4 Write a program that asks a user for their name and prints it in the following funny pattern:

```
E El Elv Elvi Elvis
```

We will require a loop because we have to repeatedly print sections of the string, and to print the sections of the string, we will use a slice:

```
name = input('Enter your name: ')
for i in range(len(name)):
    print(name[:i+1], end=' ')
```

The one trick is to use the loop variable `i` in the slice. Since the number of characters we need to print is changing, we need a variable amount in the slice. This is reminiscent of the triangle program from Section 2.4. We want to print one character of the name the first time through the loop, two characters the second time, etc. The loop variable, `i`, starts at 0 the first time through the loop, then increases to 1 the second time through the loop, etc. Thus we use `name[:i+1]` to print the first `i+1` characters of the name. Finally, to get all the slices to print on the same line, we use the **`print`** function's optional argument `end=''`.

Example 5 Write a program that removes all capitalization and common punctuation from a string `s`.

```
s = s.lower()
for c in ',.;:-?!()\'"':
    s = s.replace(c, '')
```

The way this works is for every character in the string of punctuation, we replace every occurrence of it in `s` with the empty string, `''`. One technical note here: We need the `'` character in a string. As described in the previous section, we get it into the string by using the escape character `\'`.

Example 6 Write a program that, given a string that contains a decimal number, prints out the decimal part of the number. For instance, if given `3.14159`, the program should print out `.14159`.

```
s = input('Enter your decimal number: ')
print(s[s.index('.'):])
```

The key here is the `index` method will find where the decimal point is. The decimal part of the number starts there and runs to the end of the string, so we use a slice that starts at `s.index('.')`.

Here is another, more mathematical way, to do this:

```
from math import floor
num = eval(input('Enter your decimal number: ')
print(num - floor(num))
```

One difference between the two methods is the first produces a string, whereas the second produces a number.

Example 7 A simple and very old method of sending secret messages is the substitution cipher. Basically, each letter of the alphabet gets replaced by another letter of the alphabet, say every `a` gets replaced with an `x`, and every `b` gets replaced by a `z`, etc. Write a program to implement this.

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
key = 'xznlwebgjhqdyvtkfuompciasr'

secret_message = input('Enter your message: ')
secret_message = secret_message.lower()

for c in secret_message:
    if c.isalpha():
        print(key[alphabet.index(c)],end='')
    else:
        print(c, end='')
```

The string `key` is a random reordering of the alphabet.

The only tricky part of the program is the for loop. What it does is go through the message one character at a time, and, for every letter it finds, it replaces it with the corresponding letter from the key. This is accomplished by using the `index` method to find the position in the alphabet of the current letter and replacing that letter with the letter from the key at that position. All non-letter characters are copied as is. The program uses the `isalpha` method to tell whether the current character is a letter or not.

The code to decipher a message is nearly the same. Just change `key[alphabet.index(c)]` to `alphabet[key.index(c)]`. Section 19.11 provides a different approach to the substitution cipher.

6.11 Exercises

1. Write a program that asks the user to enter a string. The program should then print the following:
   (a) The total number of characters in the string
   (b) The string repeated 10 times
   (c) The first character of the string (remember that string indices start at 0)
   (d) The first three characters of the string
   (e) The last three characters of the string
   (f) The string backwards
   (g) The seventh character of the string if the string is long enough and a message otherwise
   (h) The string with its first and last characters removed
   (i) The string in all caps
   (j) The string with every *a* replaced with an *e*

(k) The string with every letter replaced by a space

2. A simple way to estimate the number of words in a string is to count the number of spaces in the string. Write a program that asks the user for a string and returns an estimate of how many words are in the string.

3. People often forget closing parentheses when entering formulas. Write a program that asks the user to enter a formula and prints out whether the formula has the same number of opening and closing parentheses.

4. Write a program that asks the user to enter a word and prints out whether that word contains any vowels.

5. Write a program that asks the user to enter a string. The program should create a new string called `new_string` from the user's string such that the second character is changed to an asterisk and three exclamation points are attached to the end of the string. Finally, print `new_string`. Typical output is shown below:

```
Enter your string: Qbert
Q*ert!!!
```

6. Write a program that asks the user to enter a string `s` and then converts `s` to lowercase, removes all the periods and commas from `s`, and prints the resulting string.

7. Write a program that asks the user to enter a word and determines whether the word is a palindrome or not. A palindrome is a word that reads the same backwards as forwards.

8. At a certain school, student email addresses end with `@student.college.edu`, while professor email addresses end with `@prof.college.edu`. Write a program that first asks the user how many email addresses they will be entering, and then has the user enter those addresses. After all the email addresses are entered, the program should print out a message indicating either that all the addresses are student addresses or that there were some professor addresses entered.

9. Ask the user for a number and then print the following, where the pattern ends at the number that the user enters.

```
1
 2
  3
   4
```

10. Write a program that asks the user to enter a string, then prints out each letter of the string doubled and on a separate line. For instance, if the user entered `HEY`, the output would be

```
HH
EE
YY
```

11. Write a program that asks the user to enter a word that contains the letter *a*. The program should then print the following two lines: On the first line should be the part of the string up to and including the first *a*, and on the second line should be the rest of the string. Sample output is shown below:

```
Enter a word: buffalo
buffa
lo
```

12. Write a program that asks the user to enter a word and then capitalizes every other letter of that word. So if the user enters `rhinoceros`, the program should print `rHiNoCeRoS`.

13. Write a program that asks the user to enter two strings of the same length. The program should then check to see if the strings are of the same length. If they are not, the program should print an appropriate message and exit. If they are of the same length, the program should alternate the characters of the two strings. For example, if the user enters `abcde` and `ABCDE` the program should print out `AaBbCcDdEe`.

14. Write a program that asks the user to enter their name in lowercase and then capitalizes the first letter of each word of their name.

15. When I was a kid, we used to play this game called *Mad Libs*. The way it worked was a friend would ask me for some words and then insert those words into a story at specific places and read the story. The story would often turn out to be pretty funny with the words I had given since I had no idea what the story was about. The words were usually from a specific category, like a place, an animal, etc.

    For this problem you will write a *Mad Libs* program. First, you should make up a story and leave out some words of the story. Your program should ask the user to enter some words and tell them what types of words to enter. Then print the full story along with the inserted words. Here is a small example, but you should use your own (longer) example:

```
Enter a college class: CALCULUS
Enter an adjective: HAPPY
Enter an activity: PLAY BASKETBALL

CALCULUS class was really HAPPY today.  We learned how to
PLAY BASKETBALL today in class.  I can't wait for tomorrow's
class!
```

16. Companies often try to personalize their offers to make them more attractive. One simple way to do this is just to insert the person's name at various places in the offer. Of course, companies don't manually type in every person's name; everything is computer-generated. Write a program that asks the user for their name and then generates an offer like the one below. For simplicity's sake, you may assume that the person's first and last names are one word each.

```
Enter name: George Washington
```

```
Dear George Washington,

I am pleased to offer you our new Platinum Plus Rewards card
at a special introductory APR of 47.99%.  George, an offer
like this does not come along every day, so I urge you to call
now toll-free at 1-800-314-1592. We cannot offer such a low
rate for long, George, so call right away.
```

17. Write a program that generates the 26-line block of letters partially shown below. Use a loop containing one or two print statements.

```
abcdefghijklmnopqrstuvwxyz
bcdefghijklmnopqrstuvwxyza
cdefghijklmnopqrstuvwxyzab
...
yzabcdefghijklmnopqrstuvwx
zabcdefghijklmnopqrstuvwxy
```

18. The goal of this exercise is to see if you can mimic the behavior of the `in` operator and the `count` and `index` methods using only variables, for loops, and if statements.

    (a) Without using the `in` operator, write a program that asks the user for a string and a letter and prints out whether or not the letter appears in the string.

    (b) Without using the `count` method, write a program that asks the user for a string and a letter and counts how many occurrences there are of the letter in the string.

    (c) Without using the `index` method, write a program that asks the user for a string and a letter and prints out the index of the first occurrence of the letter in the string. If the letter is not in the string, the program should say so.

19. Write a program that asks the user for a large integer and inserts commas into it according to the standard American convention for commas in large numbers. For instance, if the user enters `1000000`, the output should be `1,000,000`.

20. Write a program that converts a time from one time zone to another. The user enters the time in the usual American way, such as `3:48pm` or `11:26am`. The first time zone the user enters is that of the original time and the second is the desired time zone. The possible time zones are `Eastern`, `Central`, `Mountain`, or `Pacific`.

```
Time: 11:48pm
Starting zone: Pacific
Ending zone: Eastern
2:48am
```

21. An anagram of a word is a word that is created by rearranging the letters of the original. For instance, two anagrams of *idle* are *deli* and *lied*. Finding anagrams that are real words is beyond our reach until Chapter 12. Instead, write a program that asks the user for a string and returns a random anagram of the string—in other words, a random rearrangement of the letters of that string.

22. A simple way of encrypting a message is to rearrange its characters. One way to rearrange the characters is to pick out the characters at even indices, put them first in the encrypted string, and follow them by the odd characters. For example, the string *message* would be encrypted as *msaeesg* because the even characters are *m*, *s*, *a*, *e* (at indices 0, 2, 4, and 6) and the odd characters are *e*, *s*, *g* (at indices 1, 3, and 5).

    (a) Write a program that asks the user for a string and uses this method to encrypt the string.

    (b) Write a program that decrypts a string that was encrypted with this method.

23. A more general version of the above technique is the *rail fence cipher*, where instead of breaking things into evens and odds, they are broken up by threes, fours or something larger. For instance, in the case of threes, the string *secret message* would be broken into three groups. The first group is *sr sg*, the characters at indices 0, 3, 6, 9 and 12. The second group is *eemse*, the characters at indices 1, 4, 7, 10, and 13. The last group is *ctea*, the characters at indices 2, 5, 8, and 11. The encrypted message is *sr sgeemsectea*.

    (a) Write a program the asks the user for a string and uses the rail fence cipher in the threes case to encrypt the string.

    (b) Write a decryption program for the threes case.

    (c) Write a program that asks the user for a string, and an integer determining whether to break things up by threes, fours, or whatever. Encrypt the string using the rail-fence cipher.

    (d) Write a decryption program for the general case.

24. In calculus, the derivative of x^4 is $4x^3$. The derivative of x^5 is $5x^4$. The derivative of x^6 is $6x^5$. This pattern continues. Write a program that asks the user for input like `x^3` or `x^25` and prints the derivative. For example, if the user enters `x^3`, the program should print out `3x^2`.

25. In algebraic expressions, the symbol for multiplication is often left out, as in $3x+4y$ or $3(x+5)$. Computers prefer those expressions to include the multiplication symbol, like `3*x+4*y` or `3*(x+5)`. Write a program that asks the user for an algebraic expression and then inserts multiplication symbols where appropriate.

Chapter 7

Lists

Say we need to get thirty test scores from a user and do something with them, like put them in order. We could create thirty variables, `score1`, `score2`, ..., `score30`, but that would be very tedious. To then put the scores in order would be extremely difficult. The solution is to use lists.

7.1 Basics

Creating lists Here is a simple list:

```
L = [1,2,3]
```

Use square brackets to indicate the start and end of the list, and separate the items by commas.

The empty list The empty list is `[]`. It is the list equivalent of 0 or `''`.

Long lists If you have a long list to enter, you can split it across several lines, like below:

```
nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38, 39, 40]
```

Input We can use **`eval`**`(`**`input`**`())` to allow the user to enter a list. Here is an example:

```
L = eval(input('Enter a list: '))
print('The first element is ', L[0])
```

```
Enter a list: [5,7,9]
The first element is 5
```

Printing lists You can use the `print` function to print the entire contents of a list.

```
L = [1,2,3]
print(L)
```

```
[1, 2, 3]
```

Data types Lists can contain all kinds of things, even other lists. For example, the following is a valid list:

```
[1, 2.718, 'abc', [5,6,7]]
```

7.2 Similarities to strings

There are a number of things which work the same way for lists as for strings.

- **`len`** — The number of items in `L` is given by **`len`**`(L)`.
- `in` — The `in` operator tells you if a list contains something. Here are some examples:

```
if 2 in L:
    print('Your list contains the number 2.')
if 0 not in L:
    print('Your list has no zeroes.')
```

- Indexing and slicing — These work exactly as with strings. For example, `L[0]` is the first item of the list `L` and `L[:3]` gives the first three items.
- `index` and `count` — These methods work the same as they do for strings.
- `+` and `*` — The `+` operator adds one list to the end of another. The `*` operator repeats a list. Here are some examples:

Expression	Result
`[7,8]+[3,4,5]`	`[7,8,3,4,5]`
`[7,8]*3`	`[7,8,7,8,7,8]`
`[0]*5`	`[0,0,0,0,0]`

  The last example is particularly useful for quickly creating a list of zeroes.

- Looping — The same two types of loops that work for strings also work for lists. Both of the following examples print out the items of a list, one-by-one, on separate lines.

```
for i in range(len(L)):
    print(L[i])
```

```
for item in L:
    print(item)
```

  The left loop is useful for problems where you need to use the loop variable `i` to keep track of where you are in the loop. If that is not needed, then use the right loop, as it is a little simpler.

7.3 Built-in functions

There are several built-in functions that operate on lists. Here are some useful ones:

Function	Description
`len`	returns the number of items in the list
`sum`	returns the sum of the items in the list
`min`	returns the minimum of the items in the list
`max`	returns the maximum of the items in the list

For example, the following computes the average of the values in a list `L`:

```
average = sum(L)/len(L)
```

7.4 List methods

Here are some list methods:

Method	Description
`append(x)`	adds `x` to the end of the list
`sort()`	sorts the list
`count(x)`	returns the number of times `x` occurs in the list
`index(x)`	returns the location of the first occurrence of `x`
`reverse()`	reverses the list
`remove(x)`	removes first occurrence of `x` from the list
`pop(p)`	removes the item at index `p` and returns its value
`insert(p,x)`	inserts `x` at index `p` of the list

Important note There is a big difference between list methods and string methods: String methods do not change the original string, but list methods do change the original list. To sort a list `L`, just use `L.sort()` and not `L=L.sort()`. In fact, the latter will not work at all.

wrong	*right*
`s.replace('X','x')`	`s = s.replace('X','x')`
`L = L.sort()`	`L.sort()`

Other list methods There are a few others list methods. Type **`help(list)`** in the Python shell to see some documentation for them.

7.5 Miscellaneous

Making copies of lists Making copies of lists is a little tricky due to the way Python handles lists. Say we have a list `L` and we want to make a copy of the list and call it `M`. The expression `M=L` will not work for reasons covered in Section 19.1. For now, do the following in place of `M=L`:

```
M = L[:]
```

Changing lists Changing a specific item in a list is easier than with strings. To change the value in location 2 of `L` to 100, we simply say `L[2]=100`. If we want to insert the value 100 into location 2 without overwriting what is currently there, we can use the `insert` method. To delete an entry from a list, we can use the `del` operator. Some examples are shown below. Assume `L=[6,7,8]` for each operation.

Operation	New `L`	Description
`L[1]=9`	`[6,9,8]`	replace item at index 1 with 9
`L.insert(1,9)`	`[6,9,7,8]`	insert a 9 at index 1 without replacing
`del L[1]`	`[6,8]`	delete second item
`del L[:2]`	`[8]`	delete first two items

7.6 Examples

Example 1 Write a program that generates a list `L` of 50 random numbers between 1 and 100.

```
from random import randint
L = []
for i in range(50):
    L.append(randint(1,100))
```

We use the `append` method to build up the list one item at a time starting with the empty list, `[]`. An alternative to `append` is to use the following:

```
L = L + [randint(1,100)]
```

Example 2 Replace each element in a list `L` with its square.

```
for i in range(len(L)):
    L[i] = L[i]**2
```

Example 3 Count how many items in a list `L` are greater than 50.

```
count = 0
for item in L:
    if item>50:
        count=count+1
```

Example 4 Given a list `L` that contains numbers between 1 and 100, create a new list whose first element is how many ones are in `L`, whose second element is how many twos are in `L`, etc.

```
frequencies = []
for i in range(1,101):
    frequences.append(L.count(i))
```

The key is the list method `count` that tells how many times a something occurs in a list.

Example 5 Write a program that prints out the two largest and two smallest elements of a list called `scores`.

```
scores.sort()
print('Two smallest: ', scores[0], scores[1])
print('Two largest: ', scores[-1], scores[-2])
```

Once we sort the list, the smallest values are at the beginning and the largest are at the end.

Example 6 Here is a program to play a simple quiz game.

```
num_right = 0

# Question 1
print('What is the capital of France?', end=' ')
guess = input()
if guess.lower()=='paris':
    print('Correct!')
    num_right+=1
else:
    print('Wrong.  The answer is Paris.')
print('You have', num_right, 'out of 1 right')

#Question 2
print('Which state has only one neighbor?', end=' ')
guess = input()
if guess.lower()=='maine':
    print('Correct!')
    num_right+=1
else:
    print('Wrong.  The answer is Maine.')
print('You have', num_right, 'out of 2 right,')
```

The code works, but it is very tedious. If we want to add more questions, we have to copy and paste one of these blocks of code and then change a bunch of things. If we decide to change one of the questions or the order of the questions, then there is a fair amount of rewriting involved. If we decide to change the design of the game, like not telling the user the correct answer, then every single block of code has to be rewritten. Tedious code like this can often be greatly simplified with lists and loops:

```
questions = ['What is the capital of France?',
             'Which state has only one neighbor?']
answers = ['Paris','Maine']

num_right = 0
for i in range(len(questions)):
    guess = input(questions[i])
    if guess.lower()==answers[i].lower():
        print('Correct')
        num_right=num_right+1
    else:
        print('Wrong.  The answer is', answers[i])
    print('You have', num_right, 'out of', i+1, 'right.')
```

If you look carefully at this code, you will see that the code in the loop is the nearly the same as the code of one of the blocks in the previous program, except that in the statements where we print the questions and answers, we use `questions[i]` and `answers[i]` in place of the actual text of the questions themselves.

This illustrates the general technique: If you find yourself repeating the same code over and over, try lists and a for loop. The few parts of your repetitious code that are varying are where the list code will go.

The benefits of this are that to change a question, add a question, or change the order, only the `questions` and `answers` lists need to be changed. Also, if you want to make a change to the program, like not telling the user the correct answer, then all you have to do is modify a single line, instead of twenty copies of that line spread throughout the program.

7.7 Exercises

1. Write a program that asks the user to enter a list of integers. Do the following:
   (a) Print the total number of items in the list.
   (b) Print the last item in the list.
   (c) Print the list in reverse order.
   (d) Print `Yes` if the list contains a 5 and `No` otherwise.
   (e) Print the number of fives in the list.
   (f) Remove the first and last items from the list, sort the remaining items, and print the result.

(g) Print how many integers in the list are less than 5.

2. Write a program that generates a list of 20 random numbers between 1 and 100.

   (a) Print the list.

   (b) Print the average of the elements in the list.

   (c) Print the largest and smallest values in the list.

   (d) Print the second largest and second smallest entries in the list

   (e) Print how many even numbers are in the list.

3. Start with the list `[8,9,10]`. Do the following:

   (a) Set the second entry (index 1) to 17

   (b) Add 4, 5, and 6 to the end of the list

   (c) Remove the first entry from the list

   (d) Sort the list

   (e) Double the list

   (f) Insert 25 at index 3

   The final list should equal `[4,5,6,25,10,17,4,5,6,10,17]`

4. Ask the user to enter a list containing numbers between 1 and 12. Then replace all of the entries in the list that are greater than 10 with 10.

5. Ask the user to enter a list of strings. Create a new list that consists of those strings with their first characters removed.

6. Create the following lists using a for loop.

   (a) A list consisting of the integers 0 through 49

   (b) A list containing the squares of the integers 1 through 50.

   (c) The list `['a','bb','ccc','dddd', ...]` that ends with 26 copies of the letter `z`.

7. Write a program that takes any two lists `L` and `M` of the same size and adds their elements together to form a new list `N` whose elements are sums of the corresponding elements in `L` and `M`. For instance, if `L=[3,1,4]` and `M=[1,5,9]`, then `N` should equal `[4,6,13]`.

8. Write a program that asks the user for an integer and creates a list that consists of the factors of that integer.

9. When playing games where you have to roll two dice, it is nice to know the odds of each roll. For instance, the odds of rolling a 12 are about 3%, and the odds of rolling a 7 are about 17%. You can compute these mathematically, but if you don't know the math, you can write a program to do it. To do this, your program should simulate rolling two dice about 10,000 times and compute and print out the percentage of rolls that come out to be 2, 3, 4, ..., 12.

10. Write a program that rotates the elements of a list so that the element at the first index moves to the second index, the element in the second index moves to the third index, etc., and the element in the last index moves to the first index.

11. Using a for loop, create the list below, which consists of ones separated by increasingly many zeroes. The last two ones in the list should be separated by ten zeroes.

```
[1,1,0,1,0,0,1,0,0,0,1,0,0,0,0,1,....]
```

12. Write a program that generates 100 random integers that are either 0 or 1. Then find the longest *run* of zeros, the largest number of zeros in a row. For instance, the longest run of zeros in `[1,0,1,1,0,0,0,0,1,0,0]` is 4.

13. Write a program that removes any repeated items from a list so that each item appears at most once. For instance, the list `[1,1,2,3,4,3,0,0]` would become `[1,2,3,4,0]`.

14. Write a program that asks the user to enter a length in feet. The program should then give the user the option to convert from feet into inches, yards, miles, millimeters, centimeters, meters, or kilometers. Say if the user enters a `1`, then the program converts to inches, if they enter a `2`, then the program converts to yards, etc. While this can be done with if statements, it is much shorter with lists and it is also easier to add new conversions if you use lists.

15. There is a provably unbreakable cipher called a one-time pad. The way it works is you shift each character of the message by a random amount between 1 and 26 characters, wrapping around the alphabet if necessary. For instance, if the current character is *y* and the shift is 5, then the new character is *d*. Each character gets its own shift, so there needs to be as many random shifts as there are characters in the message. As an example, suppose the user enters *secret*. The program should generate a random shift between 1 and 26 for each character. Suppose the randomly generated shifts are 1, 3, 2, 10, 8, and 2. The encrypted message would be *thebmv*.

    (a) Write a program that asks the user for a message and encrypts the message using the one-time pad. First convert the string to lowercase. Any spaces and punctuation in the string should be left unchanged. For example, `Secret!!!` becomes `thebmv!!!` using the shifts above.

    (b) Write a program to decrypt a string encrypted as above.

    The reason it is called a one-time-pad is that the list of random shifts should only be used once. It becomes easily breakable if the same random shifts are used for more than one message. Moreover, it is only provably unbreakable if the random numbers are truly random, and the numbers generated by `randint` are not truly random. For this problem, just use `randint`, but for cryptographically safe random numbers, see Section 22.8.