Repetitive Tasks

3

When my children were very little I played with them and read books to them. If they were particularly entertained I would get the, "Do it again!", command from them. And, of course, I did it or read it again. Who can say no to a three year-old when they are being so cute. They never seemed to grow tired of repetition when they found something entertaining. Eventually, I grew tired of it myself and would give them the, "One more time . . .", warning.

Computers are very good at doing repetitive tasks, often called iteration in Computer Science lingo. Computers don't get tired and they don't get bored. Usually, when a task is repeated, it is repeated for the same type of data over and over again. For instance, sending out paychecks is a repetitive job since each employee's deductions must be computed and then a paycheck must be printed or electronically deposited. For large companies, this job would require many people since each person would only be able to compute the withholdings for a relatively small number of people. In fact, before the advent of electronic computers, the word *Computer* referred to people whose job it was to carry out these kinds of calculations. That certainly must have been a mundane and repetitive job. Electronic computers on the other hand don't get tired, can work around the clock, and can work at lightning speed.Repeating a task in a programming language is often called iteration or a loop. In this chapter you learn about loops in Python. You learn how to write various kinds of loops and more importantly, you learn *when* to write various kinds of loops.

When doing a task over and over again it is probably the case that the data that the computer needs to do its job is located in some sort of list or sequence. Python has built-in support for lists. In addition, Python also supports strings, which are sequences of characters. Since so much of what computers do are repetitive tasks, it is important to know how to repeat code and how to manipulate strings and lists. This chapter explores the use of strings and lists. You learn that strings and lists are types of objects and discover what you can do with these objects. In Computer Science sequences and iteration go hand in hand.

So, what is a string? In the first chapter a *string literal* was defined as any sequence of characters surrounded by either single or double quotes. A string literal is used to

K.D. Lee, *Python Programming Fundamentals*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-4471-6642-9_3

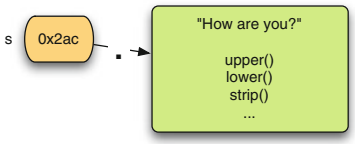

Fig. 3.1 A string object

represent a specific *string object* in Python. So a string literal is written in a Python program when you have a specific string object that you want to use in your program.

So what is an object? Every value in Python is an object. Types of objects include integers, floats, and strings. An object is a value along with methods that can either change the value of the object or give us more information about its value.

Example 3.1 Consider the string literal "How are you?". The letters in quotes are written to construct a string object. The string object has both a value, the string itself, and methods that may operate on that value. If we write the code below we get the reference called *s* pointing to the string object containing "How are you?" as shown in Fig. 3.1.

```
s = "How are you?"
```

We can interact with an object by sending messages to the object. We send a message by writing the object reference or variable name, followed by a dot (i.e. a period), followed by the method we want to call on the object. In parentheses we may pass some information to the method. The additional information are called arguments. So, calling a *method* on an *object* that is pointed to by a *reference* with zero or more *arguments* looks like this:

reference.method(*arguments*)

Sometimes it helps us to think about this interaction as sending messages to the object and getting the object to respond to these messages. So sending a message to an object or calling a method on the object are the same thing. Whatever we decide to call it, the result is the same. The object's method does something for us.

Methods can either retrieve some information about an object or they can alter the object in some way. The *lower* and *upper* methods of the string class return a new copy of the string with the characters converted to lower or upper case. The *strip* method returns a copy of a string with leading and trailing blanks removed. All the methods on strings are provided in Chap. 10.

Example 3.2 When the following code is executed, *t* refers to a new string "how are you?". Notice the first letter of the string that *t* refers to is now lower case. To call the method called *lower*() on *s* you write *s.lower*().

```
s ="How are you?"
t = s.lower()
print(t)
```

Practice 3.1 Write a short program that asks the user to enter a sentence. Then print the sentence back to the screen with all lower case letters capitalized and all upper case letters in lower case.

Types in Python are sometimes called *classes*. The term *class* is just another name for *type* in Object-Oriented Programming languages. In Object-Oriented Programming (i.e. OOP) terminology a *type* is a *class* and a *value* is an *object*. These are just different names for the same thing in Python because every type is also a class and every value is an object.

Strings have many methods that can be called on them. To find out what methods you can call on a string you can use the internet and search for *python string class* or you can go to the Python Shell Window in the Wing IDE or some other IDE and type *help*(*str*). Remember that *str* is the name of the string class in Python. Chapter 10 contains a table of most of the available string operators and methods as well.

Practice 3.2 Use Chap. 10 to help you write a program that asks the user to enter "yes" or "no". If they enter a string with any capital letters the program should print a message that says, "Next time please use all lower case letters".

3.1 Operators

If you take a look at Chap. 10 to peruse the string methods you will notice there are two kinds of methods described there. At the beginning of the appendix there are operators like <=. These operators are just special methods in Python. They describe methods that are not written using the *reference.method*(*arguments*) format. Instead, the <= method describes an infix operation that can be performed between two string objects to see if one string is less than or equal to another string object.

Example 3.3 Consider the following code.

```
1  s = input("Please enter a your name:")
2  t = input("Please enter your mom's name:")
3  if s <= t:
4      print("Your name comes before your mom's name.")
5  else:
6      print("Your mom's name comes before your name.")
```

The code in Example 3.3 asks the user to enter two strings and compares the two strings. If your name would appear first alphabetically it prints the first message, otherwise it prints the second message. The comparison of $s <= t$ on the third line of code is possible because of the existence of the *__le__* method for strings. This is a special method that you will see if you type *help*(*str*).

When reading Chap. 10 most of the operators are really methods that aren't called in the usual way. These methods are sometimes called *hooks*, *syntactic sugar*, or just *operators*. A hook in Python is just a special way of calling a method. Most methods are called in the usual way by writing *reference.method*(*arguments*). In fact, even the special hook methods can be called in the usual way. So, comparing two strings, *s* and *t*, to see if one is less than or equal to the other could be written *s.__le__*(*t*). Of course, it is more convenient and descriptive to use the operator format and write $s <= t$ when comparing two strings. This is why it is called *syntactic sugar*. It is much nicer to write the comparison operator $s <= t$ than to write *s.__le__*(*t*). Syntactic sugar refers to the ability to write a part of a program in a pleasing way as opposed to having to always stick to writing code using the same rules.

Operators are methods that are not called using the *reference.method*(*arguments*) format. Figure 3.2 has examples of calling several of the string operators and some of the string methods. All the string methods can be found in Chap. 10. Chapters 8 and 9 describe operators on integers and floats that are similar to the string operators and are called in a similar fashion.

Practice 3.3 Use Fig. 3.2 and Chap. 10 to help you write a program that asks the user to enter "yes" or "no". If they enter "yes" then you should print "You entered yes". and likewise if they enter "no". However, make sure you accept "Yes", "yEs", or any other combination of upper and lower case letters for "yes" and for "no". Identify the syntactically sugared methods that you are calling on the string class in your answer.

Operator	Returns	Result	Comments
str(90)	str	"90"	for most argument types
chr(90)	str	"Z"	ASCII character equivalent of int
ord("Z")	int	90	ASCII int equivalent of character
s+t "how"+"are"+"you"	str	"hithere" "howareyou"	same as s.__add__(t)
s in t 'he' in "there"	bool	False True	same as s.__in__(t)
s==t s=='hi'	bool	False True	same as s.__eq__(t)
s> =t	bool	False	same as s.__ge__(t)
s< =t	bool	True	same as s.__le__(t)
s> t	bool	False	same as s.__gt__(t)
s< t	bool	True	same as s.__lt__(t)
len(s)	int	2	same as s.__len__()
t[1:4] t[:3] t[1:]	str	"her" "the" "here"	same as t.__getslice__(1,4) same as t.__getslice__(0,3) same as t.__getslice__(1,len(t))
s.upper()	str	"HI"	does not change s
s.strip()	str	"hi"	removes surrounding whitespace
u.split()	list	["how","are","you"]	splits on whitespace
All examples assume s = "hi", t = "there", and u = " how are you "			

Fig. 3.2 String operators and common methods

3.2 Iterating Over a Sequence

In Python, a string is sometimes thought of as a sequence of characters. Sequences have special status in Python. You can iterate over sequences. *Iteration* refers to repeating the same thing over and over again. In the case of string sequences, you can write code that will be executed for each character in the string. The same code is executed for each character in a string. However, the result of executing the code might depend on the current character in the string. To iterate over each element of a sequence you may write a *for* loop. A *for* loop looks like this:

```
<statements before for loop>

for <variable> in <sequence>:
   <body of for loop>

<statements after for loop>
```

In this code the *<variable>* is any variable name you choose. The variable will be assigned to the first element of the sequence and then the statements in the body of the for loop will be executed. Then, the variable is assigned to the second element of the sequence and the body of the for loop is repeated. This continues until no elements are left in the sequence.

If you write a for loop and try to execute it on an empty sequence, the body of the for loop is not executed even once. The for loop means just what is says: for each element of a sequence. If the sequence is zero in length then it won't execute the body at all. If there is one element in the sequence, then the body is executed once, and so on.

For loops are useful when you need to do something for every element of a sequence. Since computers are useful when dealing with large amounts of similar data, for loops are often at the center of the programs we write.

Example 3.4 Consider the following program.

```
1  s = input("Please type some characters and press enter:")
2  for c in s:
3      print(c)
4  print("Done")
```

If the user enters *how are you?* the output is:

```
h
o
w

a
r
e

y
o
u
?
Done
```

Figure 3.3 depicts what happens when executing the code of Example 3.4. Each character of the sequence is printed on a separate line. Notice that there are blank lines, or what appear to be blank lines, between the words. This is because there are space characters between each of the words in the original string and the for loop is executed once for every character of the string including the space characters. Each of these blank lines really contains one space character.

Practice 3.4 Type in the code in Example 3.4. Set a break point on the *print*(*c*) line. Run it with the debugger and watch it as it runs. Then answer these questions:

1. Does the string *s* change as the code is executed?
2. What happens if the user just presses enter when prompted instead of typing any characters?

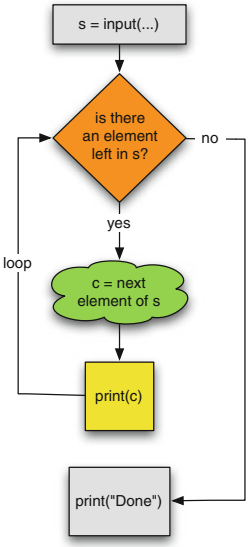

Fig. 3.3 A For Loop

Practice 3.5 Modify the code in Example 3.4 to print the characters to the screen as capital letters whether the user enters capital letters or not. For instance, it would print "HOW ARE YOU?" to the screen, with one letter on each line if "how are you?" were entered at the keyboard.

3.3 Lists

A list in Python is any sequence of values surrounded by square brackets (i.e. [ ]). So for instance [*0, 1, 2, 3*] is a list. So is [*'a', 1,'b', 4.2*]. Lists are any sequence of values inside square brackets. The items of the list can be of different types, although it is quite common for all values in a list to be of the same type. The list type is called *list* in Python as you might expect.

A list is a sequence too. A list can be iterated over using a for loop just like a string. Each element of the list is used to execute the body of the for loop once. Chapter 11 contains a table that outlines the methods and operators that apply to lists. There are several operations on sequences that are useful. For instance, *len*(s) returns the length of a sequence (the number of elements in the sequence). We can concatenate two sequences using $+$. So writing $s + t$ returns a new string which is the juxtaposition of the strings referenced by s and t. We can get part of a sequence by slicing it. A slice

is one or more contiguous elements of a sequence. It is created by using brackets and a colon. For instance, if *s* refers to the string "how are you?", then *s*[*0:3*] is the string "how" and *s*[*4:7*] is the string "are". You can even get a slice starting at the end of a sequence. So, *s*[−*4:*] gives you the last four items of a sequence, the string "you?" in this case. You can learn more about slicing in Chaps. 10 or 11. The length function, concatenation operator, and slicing apply to either strings or lists since they apply to all types of sequences in Python.

Practice 3.6 Write a for loop that prints the following output.

```
0
1
2
3
4
```

Python 2 ⇝ 3

In Python 2 the range function returned a list of integers. Because this was deemed inefficient for large lists of integers, Python 3's range function returns a generator which generates the list of integers as needed. This is called lazy evaluation and is more efficient since each new value is generated only when it is needed. To see the list that range(n) generates in Python 3 you can write list(range(n)) which will convert the generator to a list that you can inspect.

The list of integers starting from 0 and going to n − 1 is so useful there is a function in Python that we can use to generate such a list. It is called *range*. The range function can be called on an integer, n, and it will generate a list of integers from 0 to n − 1. For instance, *range*(*5*) generates the list [*0, 1, 2, 3, 4*].

The range function can be used to generate other ranges of integers, too. In general the range function is called by writing *range*([*start,*]*stop*[*,increment*]). For example, *range*(*10, 110, 10*) generates the list [*10, 20, 30, 40, 50, 60, 70, 80, 90, 100*] and *range*(*10, 0, −1*] generates the list [*10, 9, 8, 7, 6, 5, 4, 3, 2, 1*]. In Sect. 1.13 we learned that writing *s*[*0*] referred to the first character in the string *s*. *s*[*1*] refers to the second character. Writing *s*[−1] returns the last element of *s*. The indexing operations apply to all sequences, not just strings. Using indexing and a for loop together we can write some interesting code.

Example 3.5 This example uses indexing to print each of the characters in a string on separate lines. The output from this program is exactly the same as the output from Example 3.4. Contrast this code to the code that appeared in Example 3.4.

```
s = input("Please type some characters and press enter:")
for i in range(len(s)):
    print(s[i])
print("Done")
```

Notice the use of the *len* function inside the call to the *range* function. When we wish to go through all the elements of a list and we need an index into that list, the *len* function can be used along with *range* to generate the proper list of integers for the indices of the list.

Practice 3.7 Write a program that prints out the characters of a string in reverse order. So, if "hello" is entered, the program prints:

```
o
l
l
e
h
```

To accomplish this, you must use a for loop over the indices of the list since you cannot directly go backwards through a sequence with a for loop. However, you can generate a list with the indices going from the last to first index.

Python includes a few methods that make it much easier to process strings in your programs. One of these methods is called *split*. The split method splits a string into words. Each word is defined as a sequence of characters separated by whitespace in your string. Whitespace are blanks, tabs, and newline characters in your strings. The split method splits a string into a list of strings.

Example 3.6 Contrast the code found here with the code in Example 3.4. Notice that the for loop contains *s.split*() instead of just *s*.

```
s = input("Please type some characters and press enter:")
for word in s.split():
    print(word)
print("Done")
```

If the user enters "how are you?" the output is:

```
how
are
you?
Done
```

Practice 3.8 You can see what the split method does by setting some variable to the result of *s.split*(). For instance, the second line could be:

```
splitWords = s.split()
```

Modify the code to add this line and use *splitWords* in the *for loop*. Run the code in Example 3.6 using the debugger. Step into and over the code and watch the *word* and *splitWords* variables. Run the program several times with different input and make note of what splitWords ends up containing.

What is the type of the value that *s.split*() returns? What does the for loop iterate over?

Another useful operator on sequences is the *in* operator. This operator makes it possible to check to see if an item is in a sequence. For a string, this means you can ask, "Is a character in this string?". For a list it means you can ask if an item is in a list.

Example 3.7 Consider this code that determines if you like something similar to Sophus Lie. The *in* operator let's you find an item in a list and returns True if it does and False otherwise.

```
1 activity = input("What do you like to do?")
2 liesActivities = ["math", "hike", "walk", "gymnastics"]
3 if activity in liesActivities:
4     print("Sopus Lie like to do that, too!")
5 else:
6     print("Good for you!")
```

3.4 The Guess and Check Pattern for Lists

While the *in* operator works well to test for membership in a sequence, it won't work in all situations. Sometimes we need to know if a value with some property other than equality is in a sequence. In these circumstances, the guess and check pattern may be appropriate. The guess and check pattern that we learned about in the last chapter can be applied to sequences, too. You still make a guess at the beginning of the pattern, but then you fix your guess while executing a loop over each element in the sequence you are working with. An example will make things clear.

Example 3.8 Assume we want to know if the user enters an even number in a list of numbers. Here is some code that will decide if one of those numbers is even.

```
s = input("Please enter a list of integers:")
lst = s.split() # Now lst is a list of strings.

# make a guess first
containsEven = False

# the iterate over the list
for element in lst:
    x = int(element)
    # check your guess in the loop
    # and fix it if needed
    if x % 2 == 0:
        containsEven = True

# after the loop you know whether
# your guess was correct or not.
if containsEven:
    print("The list contained an even number")
else:
    print("The list did not contain an even number")
```

The code shown in Example 3.8 works by making a guess and then running through the list of possible counter-examples to fix the guess if needed. Notice the *if containsEven* appears after the *for* loop. It is not indented under the *for* loop. This is very important because other wise you would be checking if the property held for the entire list before you have even looked at the entire list.

Practice 3.9 Type this code and run it using step into and over. Make sure you get the expected output. What would happen in Example 3.8 if the *if containsEven* statement were indented under the for loop?

Practice 3.10 Imagine you work at a rehabilitation center for those that suffer from obsessive-compulsive disorders. You have to write a program that monitors your patients by looking for key words in their daily blogs that they are required to keep. The words are *orderly*, *shopping*, *repeat*, *again*, *gamble*, and *bid*. If any of these words appear in their blog entry then you should print "You really need to talk to someone about this". Otherwise you can print, "Thanks for updating your blog". Here is one possible interaction with this program.

```
Please make your blog entry for today: I am going to eat
    breakfast, then I'll make a bid on some items that I'm
    shopping for.
You really need to talk to someone about this.
```

Write this program using the guess and check pattern to see if any of the sensored words appear in their blog entry. Your blog entry will appear on the first line only. It was wrapped around to fit on the page here.

3.5 Mutability of Lists

Section 1.11 on p. 20 introduced you to variables as references to objects. The mental picture of variables pointing at objects was not really all that important at the time. Now, it becomes more crucial that you have this mental picture formed in your mind. Up until this moment, the objects we've looked at were immutable. This means that once an object was created, it could not be modified. For instance, if $x = 6$ is written in a Python program, you cannot modify the 6 later on. You *can* modify the reference x to point to a new integer, but the 6 itself cannot be modified. Integers are immutable in Python. So are float, bool, and string objects. They are all immutable. Lists, however, are not immutable. A list object can be changed. This is because of the way list objects are constructed.

Example 3.9 Consider the code given here. The code builds a list called *question*. The question object is pictured in Fig. 3.4.

```
question = ['are','you','awake','for','this']
```

What we learned on p. 20 says that *question* is a reference to an object. However, all the elements of the list are also objects. The way a list is formed, the elements of a list are actually references that point to the individual items of the list. A list is really a list of references. Unlike strings, individual references within a list can be made to point to new objects using indexed assignment. It is valid to write:

```
<list reference>[<index>] = <value>
```

Writing this changes a reference within the list object to point to a new object. This mutates the list object. A list object is mutable because of indexed assignment. It should be noted that indexed assignment is not valid on strings. Strings in Python are immutable and therefore attempting to use indexed assignment on a string will result in an error.

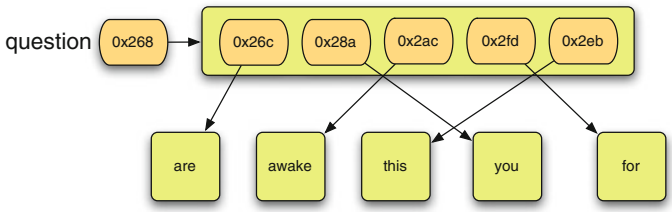

Fig. 3.4 A list object

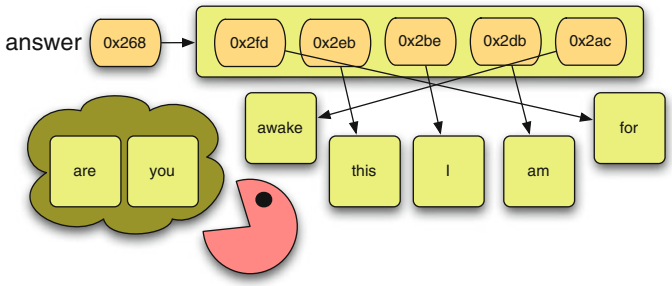

Fig. 3.5 A mutated list object

Example 3.10 Assume we want to change the sentence contained in the list from "are you awake for this" to "for this I am awake". But, we want to avoid creating any more string objects than necessary. The code below does this and prints [*'for', 'this', 'I', 'am', 'awake'*] since *answer* is a list. Figure 3.5 depicts what *answer* looks like in memory after the code below has been executed.

```
answer = question
answer[0] = answer[3]
answer[1] = answer[4]
answer[4] = answer[2]
answer[2] = 'I'
answer[3] = 'am'
print(answer)
```

Practice 3.11 Given what you now know about references, what would print if the *question* variable were printed after executing the code in Example 3.10? Run this code with the debugger.

In Example 3.10 the *answer* list started out with [*'are', 'you', 'awake', 'for', 'this'*] and ended up containing [*'for', 'this', 'I', 'am', 'awake'*]. It's not a new list. The existing list was updated. In addition, as you just discovered, the variable *question* was also mutated because both *question* and *answer* refer to the same list. This can be seen in Fig. 3.6, which shows the code in Example 3.10 while it is being executed and just before *answer* [*4*] is assigned its new value. In Wing, and in many IDEs, it looks as if there are two separate lists, the *answer* and the *question* lists. However, if you look carefully, both lists have the same reference. They are both located at 0x644bc0. If you were to type in this code and execute it you would see that the two lists truly update in synchronization with each other. When one is updated, the other simultaneously updates.

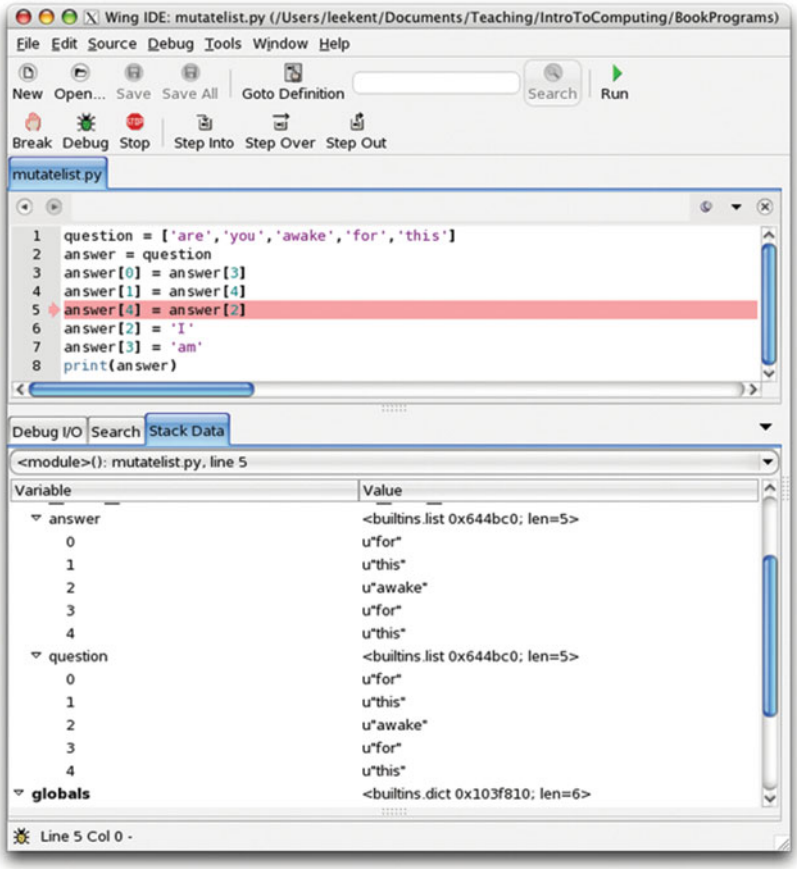

Fig. 3.6 Using wing to inspect a list

Also worth noting is that sometimes you can see the reference value when using a debugger and other times you may not. For instance, in Fig. 3.6 you can see the two references to the *question* and *answer* list. However, you cannot see the references to any of the strings contained in the list. The creators of the Wing IDE chose not to show references for strings for two reasons: Including all the references would clutter up the debugger and make it harder to use and in the case of strings, references are not really necessary since strings are immutable. Nevertheless, it does not mean that the list does not contain references to the individual items. It does; the Wing designers have just chosen not to show them in this case.

The idea that variables are really references to objects is important when objects are mutable, like lists. Understanding how the code works depends on you having the correct mental picture. Lists are the only objects we've seen so far that are mutable.

Objects of type integer, floats, booleans, and strings are not mutable. There are other types of objects that are mutable in Python including dictionaries.

3.6 The Accumulator Pattern

Iterating over sequences can be useful when we want to count something. Counting is a common occurrence in computer programs. We may want to count the number of people who are taking an Introduction to Computer Science, we may want to add up the amount of money made from ticket sales to a concert. The applications of counting could go on and on. To count we can use what is called the *Accumulator Pattern*. This pattern works by initializing a variable that keeps track of how much we have counted so far. Then we can write a for loop to go through a list of elements in a sequence and add each element's value to the accumulator. The pattern looks like this:

```
<accumulator> = <identity>
for <element> in <sequence>:
   <accumulator> = <accumulator> <operator> <element>
```

This pattern is pretty abstract. With an example it should make some more sense.

Example 3.11 Here is a program that counts the number of elements in a list. Of course, we could use the *len*(*lst*) function to give us the number of elements in the list, but this illustrates the accumulator pattern for us. This code counts the number of integers in a list. Actually, it counts the number of whitespace separated strings in the list since the code never converts the strings to integers.

```
1  s = input("Please enter a list of integers:")
2  lst = s.split() # Now lst is a list of strings.
3  count = 0 # Here is the beginning of the accumulator pattern
4  for e in lst:
5     count = count + 1
6
7  print("There were", count,"integers in the list.")
```

The Accumulator pattern can be used in a multitude of ways. It can be used to count by adding one each time through the loop, it can be used to count the number of items that satisfy some constraint. It can be used to add some number of items in the list together. It can be used to compute a product if needed.

Practice 3.12 Modify the code in Example 3.11 to count the number of even integers entered by the user.

Practice 3.13 Write a program that asks the user to enter an integer and computes the factorial of that integer, usually written $n!$ in mathematics. The definition of factorial says that $0! = 1$ and for $n > 0$, $n! = 1 * 2 * 3 \ldots * n$. You can write this program by using the range function and the accumulator pattern to multiply all the numbers from 1 to n together. If you need to review how to use the range function you can refer to p. 69.

In the previous exercise it is worth mentioning that if written correctly not only will it compute $n!$ when $n > 0$, but it will also compute 0! correctly. When 0! is computed, the body of the for loop is not executed at all. Take a look at your code or at the solution to the practice exercise to confirm this. This sometimes happens when writing code and is called a *boundary condition*. A boundary condition happens when there is a special case that causes the program control to take a slightly different path. In this case, computing 0! is a boundary condition and the body of the for loop is not executed. When testing code you have written it is important that you consider your boundary conditions and that you test them to be sure that your program handles them correctly.

3.7 Reading from and Writing to a File

A file is a grouping of related data that can be read by a computer program. Files may be stored in many different places including the hard drive, a thumb drive, on a CD, at a network location, really any place where a program could have access to it. While files occur in many forms and sizes, a *text file* is a bunch of text written using an editor and usually stored on a hard drive. Files can be read and written from Python programs. Files are another type of sequence as far as Python programs are concerned and we can iterate over them just as we would any sequence. Files are sequences of strings, one string for each line of the file. To read from a file we open it and then iterate over the lines of the file.

Example 3.12 A commonly used command in the Linux operating system is called *cat* which stands for catalog but actually prints the contents of a file to the screen. We can write a similar program in Python. Here is the code. For this to work, you must enter the name of a file in the same directory or folder as the program that you are running.

```
1 filename = input("Please enter the name of a file:")
2 catfile = open(filename,"r")
3 for line in catfile:
4     print(line)
5 catfile.close()
```

Practice 3.14 If you run the program in Example 3.12 you will notice an extra blank line between the lines of the file. This is because there is a '\n' newline character at the end of each line read from the file. You can't see the newline character, but it is there. The *print* statement prints another newline at the end of each line. Modify the code in Example 3.12 to eliminate the extra line. Look at Chap. 10 for a method that will help you eliminate the extra newline character at the end of each line.

The program in Example 3.12 reads one line at a time from the file. The second line of the example opens the file for *reading*. To write a file it may be opened for *writing* by using a "w" instead of a "r". You can also open a file with "a" for append to add to the end of an existing file.

Example 3.13 The program below writes to a file named by the user. The file is opened and it is closed. Closing is important when writing a file so you know when the file as been completely written. Otherwise, in some situations, the data may still be in memory and waiting to be written out. Closing the output file insures that the data has actually made it to the file.

```
1  filename = input("Please enter the name of a file:")
2  yourName = input("What is your name? ")
3  age = int(input("How old are you? "))
4  outfile = open(filename,"w")
5  outfile.write("Hello "+ yourName +". How are you?\n")
6  outfile.write("Next year you will be "+str(age+1) \
7          +" years old\n")
8  outfile.close()
```

When writing to a file you use the *file.write* method. Unlike the *print* function, you cannot write multiple items by separating them with commas. The write method takes only one argument, the string to write. To write multiple items to a line of a file, you must use string concatenation (i.e. the + operator) to concatenate the items together as was done in Example 3.13. When comma separated items in a print statement are printed, a space character is automatically added between comma separated items. This is not true of string concatenation. If you want a space in the concatenated strings, you must add it yourself.

If you have non-string items to write to a file, they must be converted to strings using the *str* function. Otherwise, you'll get a run-time error when Python tries to concatenate a string to a non-string item. In Example 3.13 the *age* variable is an integer because of the *int* conversion on the third line. In the sixth line, one is added to the age and then the sum *age* + *1* is converted to a string so it can be concatenated to the string literals and then written to the file.

3.8 Reading Records from a File

It is frequently the case that a file contains more than one line that relate to each other in some way. For example, consider an address book program. Each entry in your address book may contain last name, first name, street, city, zip code, home phone number, and mobile number. Typically, each of these pieces of information would be stored on a separate line in a file. A program that reads such a file would need to read all these lines together and a *for loop* will not suffice. In this case it can be done if we use a *while loop*. A *while loop* looks like this:

```
<statements before while loop>
while <condition>:
   <body of while loop>
<statements after the while loop>
```

The condition of the while loop is evaluated first. If the condition evaluates to true, then the body of the while loop is executed. The condition is evaluated again and if the condition evaluates to true, the body of the while loop is performed again. The body of the while loop is repeated until the condition evaluates to false. It is possible the body of the while loop will never be executed if the condition evaluates to false the first time as graphically depicted in Fig. 3.7.

A *while loop is* used to read records from a file that are composed of multiple lines. A *for loop* will not suffice because a *for loop* only reads one line per iteration. Since multiple lines must be read, a *while loop* gives you the extra control you need. To read a multi-line record from a file we can use this pattern:

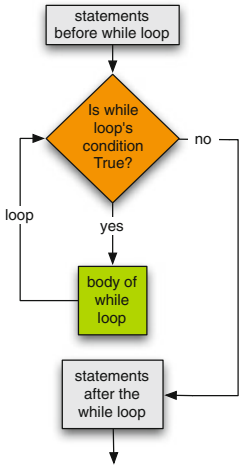

Fig. 3.7 A While Loop

```
<read first line from first record>
while <line> !="":
   <read the rest of the record>
   <process the record>

   <read the first line of the next record>
<close the file>
```

This pattern can be illustrated by looking at part of an address book application where each address book record resides on 6 lines of a file.

Example 3.14 Here is a program that counts the number of entries in your phonebook. This assumes that the file looks something like the following:

```
Lie
Sophus
2234 Valdres Rd
Decorah, IA 52101
777-555-1234
777-554-4765
Lee
Kent D.
700 College Drive
Decorah, IA 52101
777-555-1212
777-554-0789
...
```

To read this file and count the entries the code would look like this:

```
1  phonebook = open("addressbook.txt","r")
2  numEntries = 0
3  # reads the first line of the first record
4  lastName = phonebook.readline().rstrip()
5  while lastName !="":
6     # when the file is completely read the lastName string
7     # will be empty. Since the lastName wasn't an empty
8     # string, read the rest of the record.
9     firstName = phonebook.readline().rstrip()
10    street = phonebook.readline().rstrip()
11    citystatezip = phonebook.readline().rstrip()
12    homephone = phonebook.readline().rstrip()
13    mobilephone = phonebook.readline().rstrip()
14
15    # Process the record by adding to the accumulator
16    numEntries = numEntries + 1
17
18    # Read the first line of the next record
19    lastName = phonebook.readline().rstrip()
20
21 print("You have", numEntries,"entries in your address book.")
```

The code in Example 3.14 reads the first line of a record, or at least it tries to. Every opened file has a current position that is set to the beginning of the file when the file is opened. As lines are read from the file, the current position advances through the file. When the current position is at the end of the file, the program in Example 3.14 will

attempt to read one more line on either line 4 or line 19, depending on whether the file is empty or not. When the current position is at the end and it attempts to read a line, the *lastName* variable will be a reference to an empty string. This is the indication in Python that the current position is at the *end of file* sometimes abbreviated *EOF*. When this happens the code exits the while loop and prints the output on line 21. If the *lastName* variable is not empty, then the code assumes that because one line was present, all six lines will be present in the file. The code depends on each record being a six line record in the input file called *addressbook.txt*.

When you read a line from a file using the *readline* method you not only get the data on that line, but you also get the newline character at the end of the line in the file. The use of the *rstrip* method on the string read by *readline* strips away any white space from the right end of the string. If you need to look at the data at all you probably don't want the newline character on the end of each line of the record.

Whether you are writing code in Python or some other language, this *Reading Records From a File* pattern comes up over and over again. It is sometimes called the loop and a half problem. The idea is that you must attempt to read a line from the file before you know whether you are at the end of file or not. This can also be done if a boolean variable is introduced to help with the while loop. This boolean variable is the condition that gets you out of the while loop and the first time through it must be set to get your code to execute the while loop at least one.

Example 3.15 As with nearly every program, there is more than one way to do the same thing. The loop and a half code can be written differently as well. Here is another variation that while slightly different, accomplishes the same thing as Example 3.14.

```
1  phonebook = open("addressbook.txt","r")
2  numEntries = 0
3  eof = False
4  while not eof:
5     # when the file is completely read the lastName string
6     # will be empty. So will the other lines, but if the
7     # lastName is empty then we know not to process the record.
8     lastName = phonebook.readline().rstrip()
9     firstName = phonebook.readline().rstrip()
10    street = phonebook.readline().rstrip()
11    citystatezip = phonebook.readline().rstrip()
12    homephone = phonebook.readline().rstrip()
13    mobilephone = phonebook.readline().rstrip()
14
15    # if lastName is empty then we didn't really read a record.
16    if lastName !="":
17       # Process the record by adding to the accumulator
18       numEntries = numEntries + 1
19    else:
20       eof = True
21 print("You have", numEntries,"entries in your address book.")
```

Examples 3.14 and 3.15 do exactly the same thing. They each perform a loop and a half. The half part is one half of the body of the loop. In Example 3.14 this was

reading the *lastName* variable before the loop started. In Example 3.15 this was the first half of the body of the while loop. Some may feel one is easier to memorize than the other. Some experienced programmers may even prefer another way of writing the loop and a half. The important thing is that one of these patterns should be memorized. You can use it any time you need to read multi-line records from a file.

William Edward Deming was a mathematician and consultant who is widely recognized as an important contributor to the rebuilding of Japan after the second world war [15]. One of his principles emphasized that you should not repeat the same process in more than one location. In Computer Science this translates to "You should avoid writing the same code in more than one location in your program". If you write code more than once and have to make a change later, you have to remember to change it in every location. If you've only written the code once, you only have to remember to change it in that one location. Copying code within your program increases the risk of there being a bug introduced by changing only some of the locations and not all of them when new function is being added or when a bug is being fixed. This guiding principle should be followed whenever possible. Example 3.14 appears to violate this principle with one line of repeated code. That's the tradeoff for not having to include an extra *if* statement in the body of the while loop as was done in Example 3.15.

3.9 Review Questions

1. Where did the term *computer* originate?
2. What is a sequence in Python? Give an example.
3. How do you call a method on an object? What is the general form? Give an example that's not in the book.
4. What is a *class* in Python?
5. What is a *type* in Python?
6. *Definite iteration* is when the number of iterations is known before the loop starts. What construct in Python is used for definite iteration?
7. *Indefinite iteration* is what happens when the exact number of iterations is not known before the loop begins (but still may be calculable if you know the input). What construct in Python is used for indefinite iteration?
8. How can you get at the last element of a list? Give two examples of expressions that return the last element of a list.
9. If you wanted to print all the items of a list in reverse order using a while loop, how would you do it? Write some example code that demonstrates how this might be accomplished. Remember, you must use a while loop in your answer.
10. How would you use the *Guess and Check* pattern to find a name in a phonebook? Write some code that searches a list of names for someone's name. Is there a more efficient way of finding a name in a phonebook?

11. Lists and strings are similar in many ways. One major difference is that lists are mutable and strings are not. What does that mean? Give an example of an operation that lists support but strings do not.
12. Why does mutable data sometimes lead to confusion when programming?
13. What is the *accumulator pattern*? Give an example of how it might be used.
14. There are two ways to read from a file that are presented in the text. Describe both of them. When is one more appropriate than the other?

3.10 Exercises

1. Write a program that prints all the prime numbers less than 1,000. You can write this program by creating a list of prime numbers. To begin, the list is empty. Then you write two nested for loops. The outer for loop runs through all the numbers from 2 to 999. The inner for loop runs through the list of prime numbers. If the next number in the outer for loop is not divisible by any of the prime numbers, then it is prime and can be printed as a prime and added to the list of primes. To add an *element*, e, to a *list*, lst, you can write *lst.append*(*e*). This program uses both the guess and check pattern and the accumulator pattern to build the list of prime numbers.
2. Write a menu driven program that works with an address book file as described in Example 3.14. You may want to consult Example 2.6 to see how to print a menu to the user and get input from them. Your program should have three menu items, look up a name, add a contact, and quit. Interacting with your program should look something like this:

```
1) Look up a person by last name
2) Add a person to the address book.
3) Quit

Enter your choice: 1
Please enter the last name to look up: Lie

Sophus Lie
2234 Valdres Rd
Decorah, IA 52101
home: 777-555-1234
mobile: 777-554-4765

1) Look up a person by last name
2) Add a person to the address book.
3) Quit

Enter your choice: 3
Done
```

   You will want to create your own address book file for this problem. Call the file "addressbook.txt". You can create it by selecting *New* in your IDE and then saving it in the same directory as your program. You should call the file "addressbook.txt". Don't add a ".py" to the end of this text file. Be sure when

you write to the file that you put a newline character at the end of each line. If you create your own file there should be a newline character at the end of each line. If you don't do this then when you try to write another record to the file it may not end up formatted correctly. You can always open the text file with Wing to take a look at it and see if it looks like the format presented in Example 3.14.

3. Write a program that asks the user to enter a list of numbers and then prints the count of the numbers in the list and the average of the numbers in the list. Do not use the *len* function to find the length of the list. Use the accumulator pattern instead. The program would print this when run.

```
Please enter a list of numbers: 1.0 10 3.5 4.2 10.6
There were 5 numbers in the list.
The average of the numbers was 5.86
```

4. Write a program that asks the user to enter a list of numbers. The program should take the list of numbers and add only those numbers between 0 and 100 to a new list. It should then print the contents of the new list. Running the program should look something like this:

```
Please enter a list of numbers: 10.5 -8 105 76 83.2 206
The numbers between 0 and 100 are: 10.5 76.0 83.2
```

5. Write a program that asks the user to enter a list and then builds a new list which is the reverse of the original list.
6. Draw a picture of the variable references and values that result from running the code in Exercise 5.
7. Write a program that asks the user to enter a list and then reverses the list in place so that after reversing, the original list has been reversed instead of creating a new list.
8. Draw a picture of the variable references and values that result from running the code in Exercise 7.
9. Write a program that asks the user to enter a list of integers one at a time. It should allow the user to terminate the list by entering a -1. Running the program would look something like this.

```
Enter a list of integers terminated by a -1.
Please enter the first integer and press enter: 5
Please enter another integer: 4
Please enter another integer: 3
Please enter another integer: 8
Please enter another integer: -1
The list of integers is 5 4 3 8
```

10. Write a program that computes a user's GPA on a 4 point scale. Each grade on a 4 point scale is multiplied by the number of credits for that class. The sum of all the credit, grade products is divided by the total number of credits earned. Assume the 4 point scale assigns values of 4.0 for an A, 3.7 for an A−, 3.3 for a B+, 3.0 for a B, 2.7 for a B−, 2.3 for a C+, 2.0 for a C, 1.7 for a C−, 1.3 for a D+, 1.0 for a D, 0.7 for a D−, and 0 for an F. Ask the user to enter their credit grade pairs using the following format until the enter 0 for the number of credits.

```
This program computes your GPA.
Please enter your completed courses.
Terminate your entry by entering 0 credits.
Credits? 4
Grade? A
Credits? 3
Grade? B+
Credits? 4
Grade? B-
Credits? 2
Grade? C
Credits? 0
Your GPA is 3.13
```

11. Example 1.1 on p. 11 presented a nice algorithm for converting a base 10 integer to binary. It turns out that this algorithm works for both positive and negative integers. Write this algorithm one more time. This time, use a loop to avoid duplicating any code. Write the algorithm so it will convert any 32-bit signed integer to its binary equivalent. Thirty-two bit signed integers are integers in the range of -2^{31} to $2^{31} - 1$. That would be integers in the range $-2,147,483,648$ to $2,147,483,647$. Be sure to eliminate any leading 0s from the result before it is printed. Your loop should terminate when the number you are converting has reached zero (according to the algorithm) or when you've reached the requisite 32 bits for your number.

3.11 Solutions to Practice Problems

These are solutions to the practice problems in this chapter. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

3.11.1 Solutions to Practice Problem 3.1

```
sentence = input("Please enter a sentence: ")
print("Here is the sentence with the case swapped.")
print(sentence.swapcase())
```

3.11.2 Solutions to Practice Problem 3.2

```
answer = input("Please answer yes or no: ")
if not answer.islower():
  print("Next time please user all lower case letters.")
```

3.11.3 Solutions to Practice Problem 3.3

The else would be optional for this exercise.

```
1  answer = input("Please answer yes or no: ")
```

```
2   if answer.lower()=="yes":
3     print("You entered yes.")
4   elif answer.lower()=="no":
5     print("You answered no.")
6   else:
7     print("You answered neither yes or no.")
```

3.11.4 Solutions to Practice Problem 3.4

1. Does the string *s* change as the code is executed?
   No it does not.
2. What happens if the user just presses enter when prompted instead of typing any characters?
   The body of the for loop is not executed at all.

3.11.5 Solutions to Practice Problem 3.5

```
s = input("Please type some characters and press enter:")
for c in s:
    print(c.upper())
```

3.11.6 Solutions to Practice Problem 3.6

```
for i in range(5):
    print(i)
```

3.11.7 Solutions to Practice Problem 3.7

```
s = input("Please type some characters and press enter:")
for i in range(len(s)-1,-1,-1):
    print(s[i])
```

3.11.8 Solutions to Practice Problem 3.8

The split method returns a list of strings. The for loop iterates over the list. Each time through the loop the word variable is referencing the next string in the list.

3.11.9 Solutions to Practice Problem 3.9

If the containsEven *if* statement were indented, then the for loop would check to see if *containsEven* were true or false each time through the loop. The program would print that the list did not contain an even number (even though it might) over and over again until an even number was found. Then it would print it did contain an even number over and over again. It would print one line for each element of the list.

3.11.10 Solutions to Practice Problem 3.10

```
entry = input("Please make your blog entry for today: ")
found = False
for word in entry.split():
  if word in ['orderly','shopping','repeat','again', \
              'gamble','bid']:
    found = True

if found:
  print("You really need to talk to somone about this.")
else:
  print("Thanks for you entry.")
```

3.11.11 Solutions to Practice Problem 3.11

If the *question* variable were printed it would be the same as if the *answer* variable were printed. Both *question* and *answer* refer to the same list.

3.11.12 Solutions to Practice Problem 3.12

```
s = input("Please enter a list of integers:")

lst = s.split() # Now lst is a list of strings.

count = 0 # Here is the beginning of the accumulator pattern

for e in lst:
    if int(e) % 2 == 0:
        count = count + 1

print("There were", count,"even integers in the list.")
```

3.11.13 Solutions to Practice Problem 3.13

```
n = int(input("Please enter a non-negative integer: "))

factorial = 1
for i in range(1,n+1):
  factorial = factorial * i

print(str(n)+"! =",factorial)
```

3.11.14 Solutions to Practice Problem 3.14

```
filename = input("Please enter the name of a file:")
catfile = open(filename,"r")
for line in catfile:
    print(line.rstrip())
catfile.close()
```