UNIT-II
Control Statement

Topic 1: Definite iteration: for Loop

We begin our study of control statements with repetition statements, also known as loops, which repeat an action. Each repetition of the action is known as a pass or an iteration.

There are two types of loops—those that repeat an action a predefined number of times (definite iteration) and those that perform the action until the program determines that it needs to stop (indefinite iteration).
In this section, we examine Python's for loop, the control statement that most easily supports definite iteration.

1. Executing a Statement a Given Number of Times:

Here is a for loop that runs the same output statement four times:

```
>>> for eachPass in range(4):
          print("It's alive!", end = " ")


It's alive! It's alive! It's alive! It's alive!
```

This loop repeatedly calls one function—the print function. The constant 4 on the first line tells the loop how many times to call this function. If we want to print 10 or 100 exclamations, we just change the 4 to 10 or to 100. The form of this type of for loop is

```
for <variable> in range(<an integer expression>):
 <statement-1>
 .
 .
 <statement-n>
```

2. Count-Controlled Loops:

When Python executes the type of for loop just discussed, it counts from 0 to the value of the header's integer expression minus 1. On each pass through the loop, the header's variable is bound to the current value of this count. The next code segment demonstrates this fact:

```
>>> for count in range(4):
 print(count, end = " ")
```

0 1 2 3

Loops that count through a range of numbers are also called count-controlled loops.

3.Augmented Assignment

Expressions such as x = x + 1 or x = x + 2 occur so frequently in loops that Python includes abbreviated forms for them. The assignment symbol can be combined with the arithmetic and concatenation operators to provide augmented assignment operations.

Following are several examples:

```
a = 17
s = "hi"
a += 3 # Equivalent to a = a + 3
a -= 3 # Equivalent to a = a - 3
a *= 3 # Equivalent to a = a * 3
a /= 3 # Equivalent to a = a / 3
a %= 3 # Equivalent to a = a % 3
s += " there" # Equivalent to s = s + " there
```

4.Loop Errors: Off-by-One Error

The for loop is not only easy to write but also fairly easy to write correctly. Once we get the syntax correct, we need to be concerned about only one other possible error: The loop fails to perform the expected number of iterations. Because this number is typically off by one, the error is called an off-by-one error.

```
# Count from 1 through 4, we think
>>> for count in range(1,4):
 print(count)
1
2
3
```

5.Traversing the Contents of a Data Sequence:

Although we have been using the for loop as a simple count-controlled loop, the loop itself visits each number in a sequence of numbers generated by the range function. The next code segment shows what these sequences look like:

```
>>> list(range(4))
[0, 1, 2, 3]
```

```
>>> list(range(l, 5))
[1, 2, 3, 4]
```

6. Specifying the Steps in the Range:

A variant of Python's range function expects a third argument that allows you to nicely skip some numbers. The third argument specifies a step value, or the interval between the numbers used in the range, as shown in the

examples that follow:

```
>>> list(range(1, 6, 1)) # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2)) # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3)) # Use every third number
[1, 4]
```

Topic2:Formatting Text for output

Many data-processing applications require output that has a tabular format, like that used in spreadsheets or tables of numeric data. In this format, numbers and other information are aligned in columns that can be either left-justified or right-justified.

A column of data is left-justified if its values are vertically aligned beginning with their leftmost characters. A column of data is right-justified if its values are vertically aligned beginning with their rightmost characters.

To maintain the margins between columns of data, left-justification requires the addition of spaces to the right of the datum, where as right-justification requires adding spaces to the left of the datum. A column of data is centered if there are an equal number of spaces on either side of the data within that column.

The next example, which displays the exponents 7 through 10 and the values of 107 through 1010, shows the format of two columns produced by the print function:

```
>>> for exponent in range(7, 11):
 print(exponent, 10 ** exponent)
7 10000000
```

```
8 100000000
9 1000000000
10 10000000000
```

Python includes a general formatting mechanism that allows the programmer to specify field widths for different types of data. The next session shows how to right-justify and left-justify the string "four" within a field width of 6:

```
>>> "%6s" % "four"              # Right justify
' four'
>>> "%-6s" % "four"             # Left justify
'four
```

The first line of code right-justifies the string by padding it with two spaces to its left. The next line of code left-justifies by placing two spaces to the string’s right.
The simplest form of this operation is the following:

<format string> % <datum>

Topic3: Selection if and if else Statement

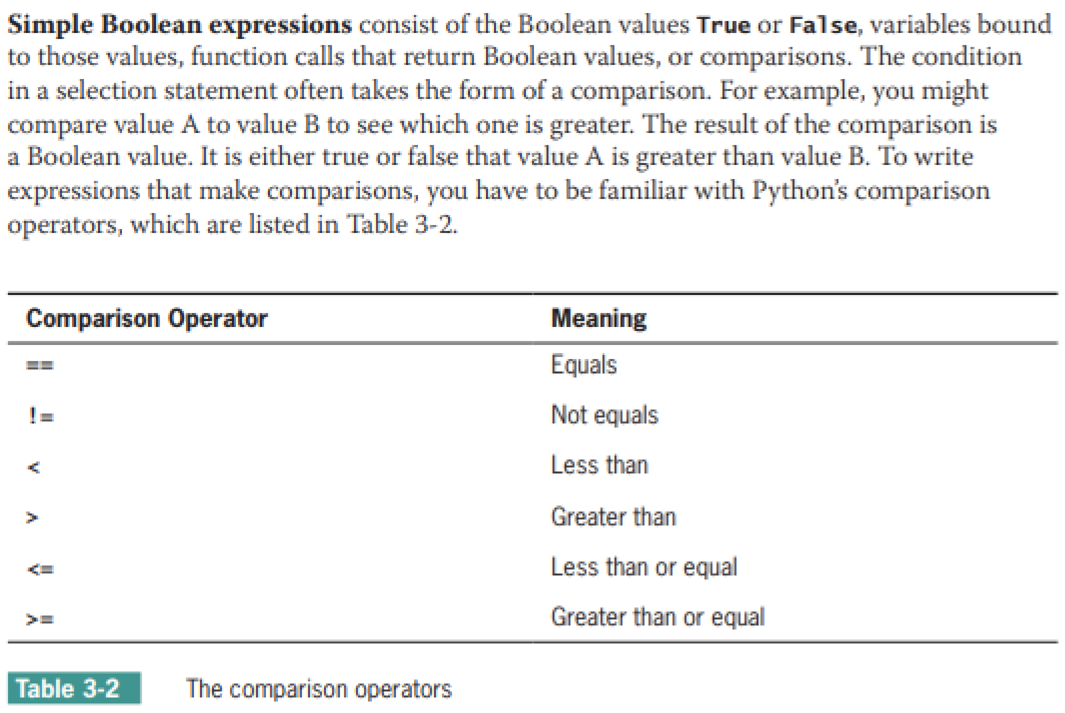
The computer must pause to examine or test a condition, which expresses a hypothesis about the state of its world at that point in time. If the condition is true, the computer executes the first alternative action and skips the second alternative.

If the condition is false, the computer skips the first alternative action and executes the second alternative.In other words, instead of moving blindly ahead, the computer exercises some intelligence by responding to conditions in its environment.

In this section, we explore several types of selection statements, or control statements, that allow a computer to make choices.

1.The Boolean Type, Comparisons, and Boolean Expressions:

Before you can test conditions in a Python program, you need to understand the Boolean data type, which is named for the nineteenth century British mathematician George Boole. The Boolean data type consists of only two data values—true and false.



Simple Boolean expressions consist of the Boolean values `True` or `False`, variables bound to those values, function calls that return Boolean values, or comparisons. The condition in a selection statement often takes the form of a comparison. For example, you might compare value A to value B to see which one is greater. The result of the comparison is a Boolean value. It is either true or false that value A is greater than value B. To write expressions that make comparisons, you have to be familiar with Python's comparison operators, which are listed in Table 3-2.

Comparison Operator	Meaning
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Table 3-2 The comparison operators

The following session shows some example comparisons and their values:

```
>>> 4 == 4
True
>>> 4 != 4
False
>>> 4 < 5
True
>>> 4 >= 3
True
>>> "A" < "B"
True
```

2.if-else Statements:

The if-else statement is the most common type of selection statement. It is also called a two-way selection statement, because it directs the computer to make a choice between two alternative courses of action.

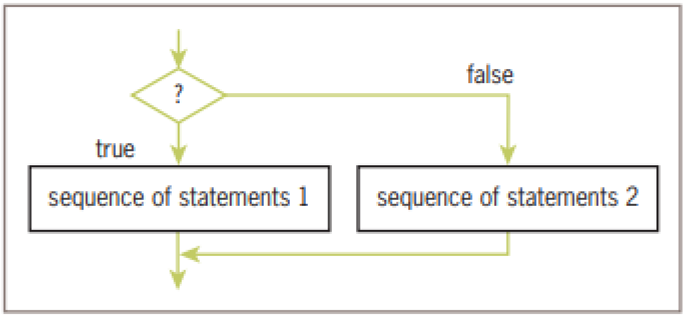


Figure 3-2 The semantics of the **if-else** statement

3.One-Way Selection Statement:

The simplest form of selection is the if statement. This type of control statement is also called a one-way selection statement, because it consists of a condition and just a single sequence of statements.

Figure 3-3 shows a flow diagram of the semantics of the **if** statement.

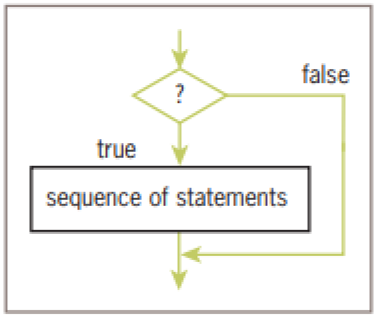


Figure 3-3 The semantics of the **if** statement

4.Multi-Way if Statements

Occasionally, a program is faced with testing several conditions that entail more than two alternative courses of action. For example, consider the problem of converting numeric grades to letter grades.

Letter Grade	Range of Numeric Grades
A	All grades above 89
B	All grades above 79 and below 90
C	All grades above 69 and below 80
F	All grades below 70

Table 3-3 A simple grading scheme

5.Logical Operators and Compound Boolean Expressions

Often a course of action must be taken if either of two conditions is true. For example, valid inputs to a program often lie within a given range of values. Any input above this range should be rejected with an error message, and any input below this range should be dealt with in a similar fashion.

EX:

```
number = int(input("Enter the numeric grade: "))
if number > 100:
 print("Error: grade must be between 100 and 0")
elif number < 0:
 print("Error: grade must be between 100 and 0")
else:
 # The code to compute and print the result goes here
```

6.Short-Circuit Evaluation

The Python virtual machine sometimes knows the value of a Boolean expression before it has evaluated all of its operands. For instance, in the expression A and B, if A is false, then so is the expression, and there is no need to evaluate B.

Likewise, in the expression A or B, if A is true, then so is the expression, and again there is no need to evaluate B. This approach, in which evaluation stops as soon as possible, is called short-circuit evaluation.

```
count = int(input("Enter the count: "))
theSum = int(input("Enter the sum: "))
if count > 0 and theSum // count > 10:
 print("average > 10")
else:
 print("count = 0 or average <= 10")
```

Topic4: Conditional Iteration :The While Loop

The program's input loop accepts these values until the user enters a special value or sentinel that terminates the input. This type of process is called conditional iteration, meaning that the process continues to repeat as long as a condition remains true.

In this section, we explore the use of the while loop to describe conditional iteration.

1.The Structure and Behavior of a while Loop:

- Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Such a condition is called the loop's continuation condition.
- If the continuation condition is false, the loop ends. If the continuation condition is true, the statements within the loop are executed again.

The while loop is tailor-made for this type of control logic. Here is its syntax:

```
while <condition>:
    <sequence of statements>
```

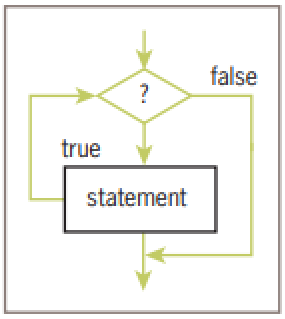


Figure 3-6 The semantics of a **while** loop

2.Count Control with a while Loop:

You can also use a while loop for a count-controlled loop. The next two code segments show the same summations with a for loop and a while loop, respectively.

```
# Summation with a for loop
theSum = 0
for count in range(1, 100001):
 theSum += count
print(theSum)
```

```
# Summation with a while loop
theSum = 0
count = 1
while count <= 100000:
 theSum += count
 count += 1
print(theSum)
```

3.The while True Loop and the break Statement

Although the while loop can be complicated to write correctly, it is possible to simplify its structure and thus improve its readability.Python includes a break statement that will allow us to make this change in the program. Here is the modified script:

```
theSum = 0.0
while True:
 data = input("Enter a number or just enter to quit: ")
 if data == "":
 break
 number = float(data)
 theSum += number
print("The sum is", theSum)
```

3. Random Numbers

Python's random module supports several ways to do this, but the easiest is to call the function random.randint with two integer arguments. The function **random.randint** returns a random number from among the numbers between the two arguments and including those numbers. The next session simulates the roll of a die 10 times:

```
>>> import random
>>> for roll in range(10):
 print(random.randint(1, 6), end = " ")

2 4 6 4 3 2 3 6 2 2
```

4.Loop Logic, Errors, and Testing

You have seen that the while loop is typically a condition-controlled loop, meaning that its continuation depends on the truth or falsity of a given condition. Because while loops can be the most complex control statements, to ensure their correct behavior, careful design and testing are needed.

Strings and Text Files

Topic:5 Accessing Character and Substring in Strings

1.The Structure of Strings

Unlike an integer, which cannot be decomposed into more primitive parts, a string is a data structure.

A data structure is a compound unit that consists of several other pieces of data. A string is a sequence of zero or more characters.

Recall that you can mention a Python string using either single quote marks or double quote marks. Here are some examples:

```
>>> "Hi there!"
'Hi there!'
>>> ""
''
>>> 'R'
'R'
```

Python's len function returns this value when it is passed a string, as shown in the following session:

```
>>> len("Hi there!")
9
>>> len("")
0
```

The positions of a string's characters are numbered from 0, on the left, to the length of the string minus 1, on the right. Figure 4-1 illustrates the sequence of characters and their positions in the string "Hi there!". Note that the ninth and last character, '!', is at position

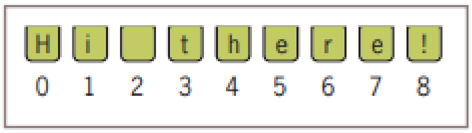


Figure 4-1 Characters and their positions in a string

2.The Subscript Operator

Although a simple for loop can access any of the characters in a string, sometimes you just want to inspect one character at a given position without visiting them all. The subscript operator [] makes this possible. The simplest form of the subscript operation is the following:

<a string>[<an integer expression>]

The first part of this operation is the string you want to inspect. The integer expression in brackets indicates the position of a particular character in that string.

3.Slicing for Substrings

Some applications extract portions of strings called substrings. For example, an application that sorts filenames according to type might use the last three characters in a filename, called its extension, to determine the file’s type (exceptions to this rule, such as the extensions ".py" and ".html", will be considered later in this chapter).

On a Windows file system, a filename ending in ".txt" denotes a human-readable text file, whereas a filename ending in ".exe" denotes an executable file of machine code.

You can use Python’s subscript operator to obtain a substring through a process called slicing.

```
>>> name = "myfile.txt" # The entire string
>>> name[0:]
'myfile.txt'
>>> name[0:1] # The first character
'm'
>>> name[0:2] # The first two characters
'my'
```

```
>>> name[:len(name)] # The entire string
'myfile.txt'
>>> name[-3:] # The last three characters
'txt'
>>> name[2:6] # Drill to extract 'file'
'file'
```

Topic:6 Data Encryption

As you might imagine, data traveling on the Internet is vulnerable to spies and potential thieves. It is easy to observe data crossing a network, particularly now that more and more communications involve wireless transmissions.

For example, a person can sit in a car in the parking lot outside any major hotel and pick up transmissions between almost any two computers if that person runs the right **sniffing software.**

For this reason, most applications now use **data encryption** to protect information transmitted on networks

Encryption techniques are as old as the practice of sending and receiving messages. The sender encrypts a message by translating it to a secret code, called a **cipher text.** At the other end, the receiver decrypts the cipher text back to its original **plaintext** form.

A very simple encryption method that has been in use for thousands of years is called a **Caesar cipher**. Recall that the character set for text is ordered as a sequence of distinct values.

Note the wraparound effect for the last three plaintext characters, whose cipher text characters start at the beginning of the alphabet.

For example, the plaintext character ‘x’ with ASCII 120 maps to the cipher character ‘a’ with ASCII 97, because ASCII 120 is less than 3 characters from the end of the plaintext sequence.

The next two Python scripts implement Caesar cipher methods for any strings that contain the lowercase letters of the alphabet and for any distance values between 0 and 2.

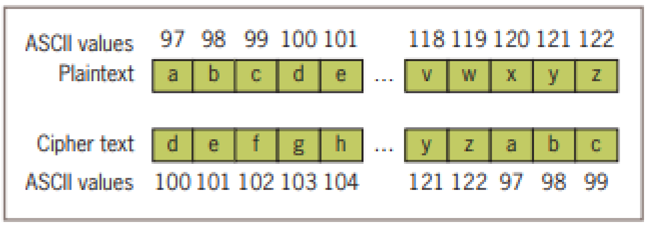

ASCII values 97 98 99 100 101 ... 118 119 120 121 122
Plaintext a b c d e ... v w x y z
Cipher text d e f g h ... y z a b c
ASCII values 100 101 102 103 104 ... 121 122 97 98 99

Figure 4-2 A Caesar cipher with distance +3 for the lowercase alphabet

A more sophisticated encryption scheme is called a **block cipher.** A block cipher uses plaintext characters to compute two or more encrypted characters. This is accomplished by using a mathematical structure known as an **invertible matrix** to determine the values of the encrypted characters.

Topic:7 Strings and Number Systems

When you perform arithmetic operations, you use the decimal number system. This system, also called the base ten number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits.

The binary number system is used to represent all information in a digital computer. The two digits in this base two number system are 0 and 1. Because binary numbers can be long strings of 0s and 1s, computer scientists often use other number systems, such as octal (base eight) and hexadecimal (base 16) as shorthand for these numbers.

To identify the system being used, you attach the base as a subscript to the number. For example, the following numbers represent the quantity 41510 in the binary, octal, decimal, and hexadecimal systems:

415 in binary notation	1100111112
415 in octal notation	6378
415 in decimal notation	41510
415 in hexadecimal notation	19F16

The digits used in each system are counted from 0 to n – 1, where n is the system's base.

1.The Positional System for Representing Numbers

All of the number systems we have examined use positional notation—that is, the value of each digit in a number is determined by the digit's position in the number. In other words, each digit has a positional value.

$415_{10} =$

$4 * 10^2 + 1 * 10^1 + 5 * 10^0 =$

$4 * 100 + 1 * 10 + 5 * 1 =$

$400 + 10 + 5 = 415$

```
Positional values   100   10   1
Positions             2    1   0
```

Figure 4-3 The first three positional values in the base-10 number system

2. Converting Binary to Decimal:

Like the decimal system, the binary system also uses positional notation. However, each digit or bit in a binary number has a positional value that is a power of 2. In the discussion that follows, we occasionally refer to a binary number as a string of bits or a **bit string**. You determine the integer quantity that a string of bits represents in the usual manner: Multiply the value of each bit (0 or 1) by its positional value and add the results. Let's do that for the number 1100111_2:

$1100111_2 =$

$1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 =$

$1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1 =$

$64 + 32 + 4 + 2 + 1 = 103$

3.Converting Decimal to Binary

How are integers converted from decimal to binary? One algorithm uses division and subtraction instead of multiplication and addition. This algorithm repeatedly divides the decimal number by 2.

```
Enter a decimal integer: 34
Quotient Remainder Binary
   17        0         0
    8        1        10
    4        0       010
    2        0      0010
    1        0     00010
    0        1    100010
The binary representation is 100010
```

4.Octal and Hexadecimal Numbers

The octal system uses a base of eight and the digits 0 . . . 7. Conversions of octal to decimal and decimal to octal use algorithms similar to those discussed thus far (using powers of 8 and multiplying or dividing by 8, instead of 2). But the real benefit of the octal system is the ease of converting octal numbers to and from binary.

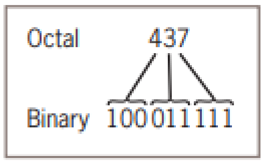


Figure 4-4 The conversion of octal to binary

The conversion between numbers in the two systems works as follows. Each digit in the hexadecimal number is equivalent to four digits in the binary number. Thus, to convert from hexadecimal to binary, you replace each hexadecimal digit with the corresponding 4-bit binary number. To convert from binary to hexadecimal, you factor the bits into groups of four and look up the corresponding hex digits. (This is the kind of stuff that hackers memorize). Figure 4-5 shows a mapping of hexadecimal digits to binary digits.

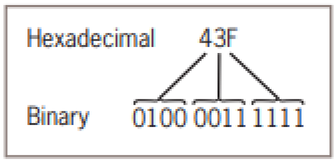


Figure 4-5 The conversion of hexadecimal to binary

Topic: 8 String Methods

Let’s start with counting the words in a single sentence and finding the average word length. This task requires locating the words in a string. Fortunately, Python includes a set of string operations called methods that make tasks like this one easy. In the next session, we use the string method split to obtain a list of the words contained in an input string.

```
>>> sentence = input("Enter a sentence: ")
Enter a sentence: This sentence has no long words.
>>> listOfWords = sentence.split()
>>> print("There are", len(listOfWords), "words.")
There are 6 words.
>>> sum = 0
>>> for word in listOfWords:
 sum += len(word)
>>> print("The average word length is", sum / len(listOfWords))
The average word length is 4.5
```

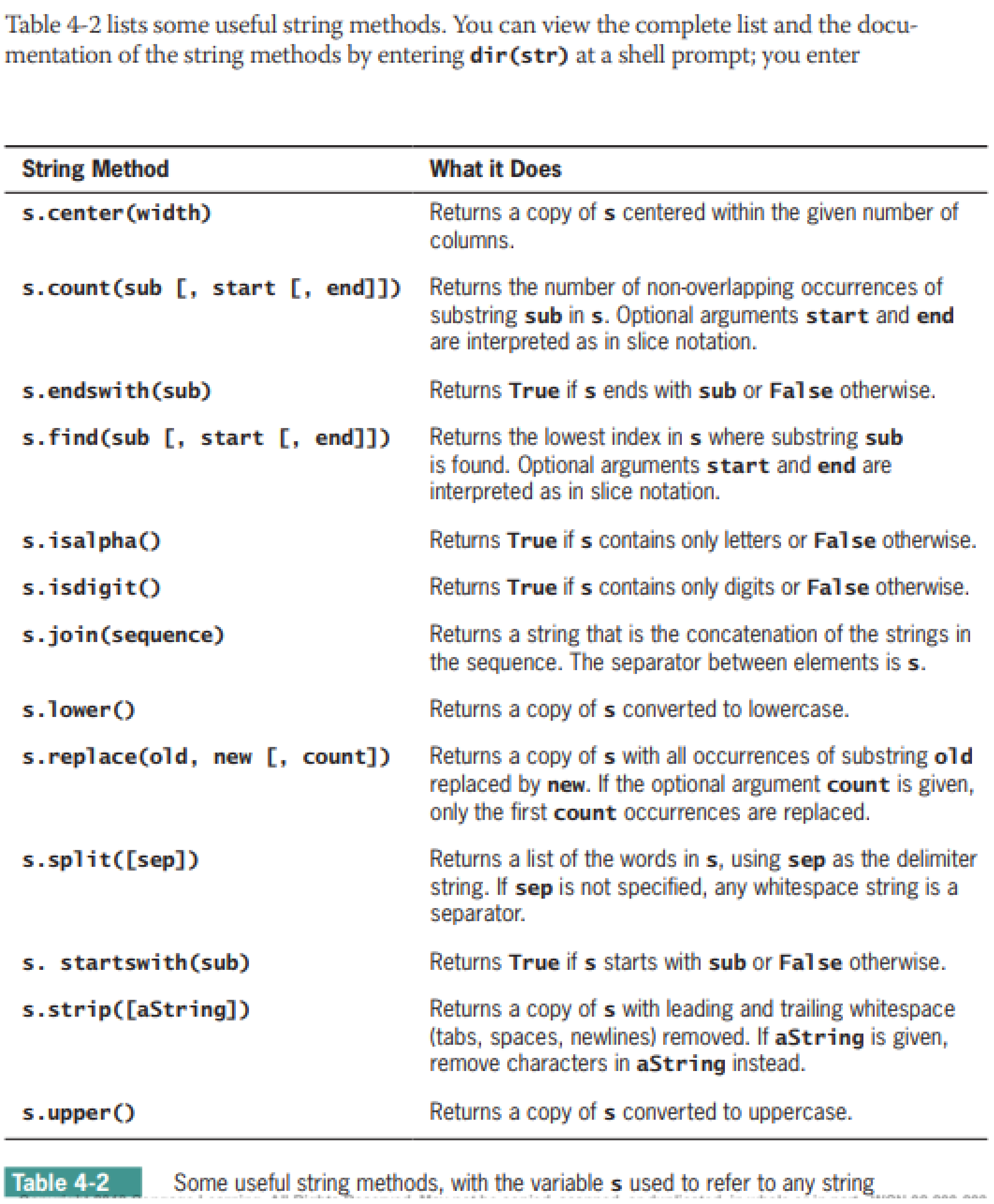


Table 4-2 lists some useful string methods. You can view the complete list and the documentation of the string methods by entering `dir(str)` at a shell prompt; you enter

String Method	What it Does
`s.center(width)`	Returns a copy of **s** centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring **sub** in **s**. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.endswith(sub)`	Returns **True** if **s** ends with **sub** or **False** otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in **s** where substring **sub** is found. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.isalpha()`	Returns **True** if **s** contains only letters or **False** otherwise.
`s.isdigit()`	Returns **True** if **s** contains only digits or **False** otherwise.
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is **s**.
`s.lower()`	Returns a copy of **s** converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of **s** with all occurrences of substring **old** replaced by **new**. If the optional argument **count** is given, only the first **count** occurrences are replaced.
`s.split([sep])`	Returns a list of the words in **s**, using **sep** as the delimiter string. If **sep** is not specified, any whitespace string is a separator.
`s. startswith(sub)`	Returns **True** if **s** starts with **sub** or **False** otherwise.
`s.strip([aString])`	Returns a copy of **s** with leading and trailing whitespace (tabs, spaces, newlines) removed. If **aString** is given, remove characters in **aString** instead.
`s.upper()`	Returns a copy of **s** converted to uppercase.

Table 4-2 Some useful string methods, with the variable **s** used to refer to any string

Ex:

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
```

```
>>> s.startswith("Hi")
True
>>> s.find("the")
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> " ".join(words)
'Hithere!'
>>> " ". join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
```

Topic: 9 Text Files

Thus far in this book, we have seen examples of programs that have taken input data from users at the keyboard. Most of these programs can receive their input from text files as well.

A text file is a software object that stores data on a permanent medium such as a disk, CD, or flash memory. When compared to keyboard input from a human user, the main advantages of taking input data from a file are the following:

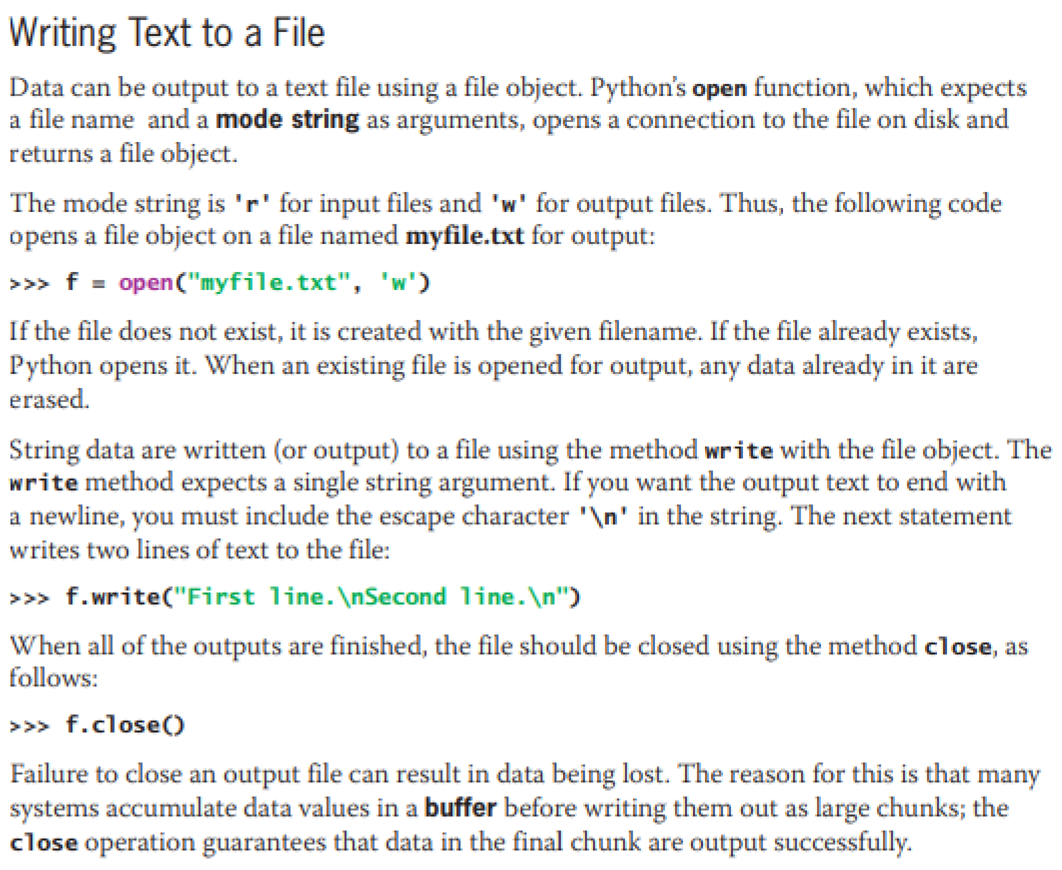
• The data set can be much larger.

• The data can be input much more quickly and with less chance of error.

• The data can be used repeatedly with the same program or with different programs

1.Text Files and Their Format

Using a text editor such as Notepad or TextEdit, you can create, view, and save data in a text file (but be careful: some text editors use RTF as a default format for text, so you should make sure to change this to "Plain text" in your editor's preferences if that is the case).



Writing Text to a File

Data can be output to a text file using a file object. Python's **open** function, which expects a file name and a **mode string** as arguments, opens a connection to the file on disk and returns a file object.

The mode string is **'r'** for input files and **'w'** for output files. Thus, the following code opens a file object on a file named **myfile.txt** for output:

```
>>> f = open("myfile.txt", 'w')
```

If the file does not exist, it is created with the given filename. If the file already exists, Python opens it. When an existing file is opened for output, any data already in it are erased.

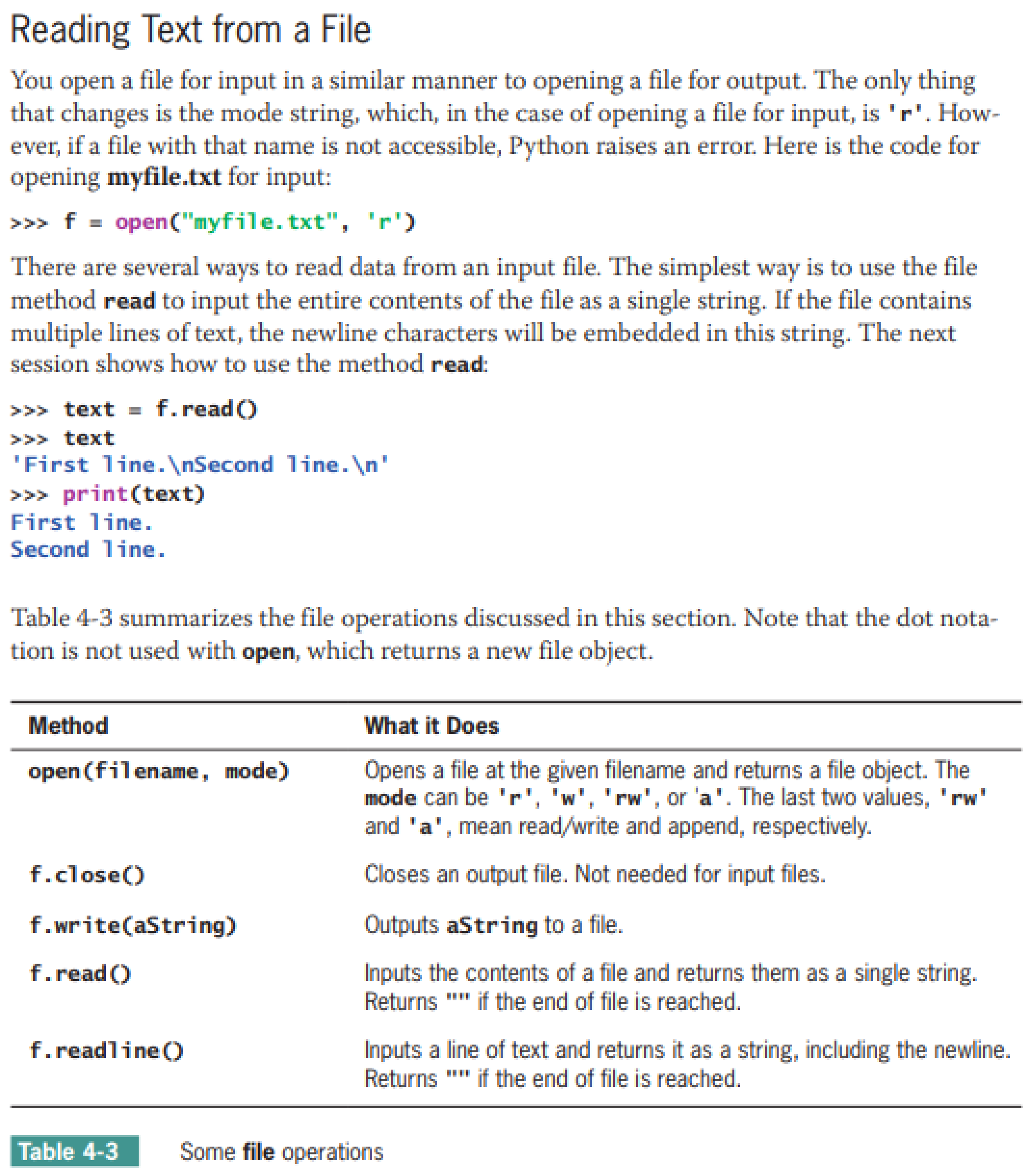
String data are written (or output) to a file using the method **write** with the file object. The **write** method expects a single string argument. If you want the output text to end with a newline, you must include the escape character **'\n'** in the string. The next statement writes two lines of text to the file:

```
>>> f.write("First line.\nSecond line.\n")
```

When all of the outputs are finished, the file should be closed using the method **close**, as follows:

```
>>> f.close()
```

Failure to close an output file can result in data being lost. The reason for this is that many systems accumulate data values in a **buffer** before writing them out as large chunks; the **close** operation guarantees that data in the final chunk are output successfully.



Reading Text from a File

You open a file for input in a similar manner to opening a file for output. The only thing that changes is the mode string, which, in the case of opening a file for input, is **'r'**. However, if a file with that name is not accessible, Python raises an error. Here is the code for opening **myfile.txt** for input:

```
>>> f = open("myfile.txt", 'r')
```

There are several ways to read data from an input file. The simplest way is to use the file method **read** to input the entire contents of the file as a single string. If the file contains multiple lines of text, the newline characters will be embedded in this string. The next session shows how to use the method **read**:

```
>>> text = f.read()
>>> text
'First line.\nSecond line.\n'
>>> print(text)
First line.
Second line.
```

Table 4-3 summarizes the file operations discussed in this section. Note that the dot notation is not used with **open**, which returns a new file object.

Method	What it Does
`open(filename, mode)`	Opens a file at the given filename and returns a file object. The **mode** can be **'r'**, **'w'**, **'rw'**, or **'a'**. The last two values, **'rw'** and **'a'**, mean read/write and append, respectively.
`f.close()`	Closes an output file. Not needed for input files.
`f.write(aString)`	Outputs **aString** to a file.
`f.read()`	Inputs the contents of a file and returns them as a single string. Returns "" if the end of file is reached.
`f.readline()`	Inputs a line of text and returns it as a string, including the newline. Returns "" if the end of file is reached.

Table 4-3 Some **file** operations

Accessing and Manipulating Files and Directories on Disk

As you probably know, the file system of a computer allows you to create folders or directories, within which you can organize files and other directories.

The complete set of directories and files forms a tree-like structure, with a single root directory at the top and branches down to nested files and subdirectories.

Figure 4-6 shows a portion of a file system,with directories named lambertk, parent, current, sibling, and child. Each of the last four directories contains a distinct file named myfile.txt.

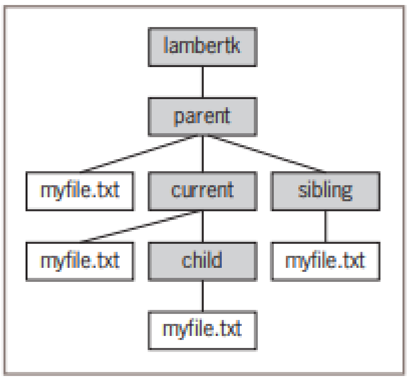


Figure 4-6 A portion of a file system