Accessing Characters and Substrings in Strings: Strings can be combined via concatenation to form new strings.

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'
```

Note that the shell prints a string using single quotes, even when you enter it using double quotes. In this, we use single quotes with single-character strings and double quotes with the empty string or with multi-character strings.

When working with strings, the programmer sometimes must be aware of a string's length and the positions of the individual characters within the string. A string's length is the number of characters it contains. 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 1 illustrates the sequence of characters and their positions in the string "Hi there!". Note that the ninth and last character, '!', is at position 8.

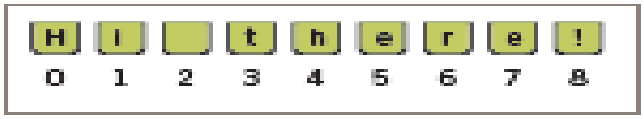

Figure: Characters and their positions in a string

The string is an immutable data structure. This means that its internal data elements, the characters, can be accessed, but cannot be replaced, inserted, or removed.

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. The integer expression is also called an index. In the following examples, the subscript operator is used to access characters in the string "Alan Turing":

```
>>> name = "Alan Turing"
>>> name[0] # Examine the first character
'A'
>>> name[3] # Examine the fourth character
'n'
>>> name[len(name)] # Oops! An index error!
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> name[len(name) - 1] # Examine the last character
'g'
>>> name[-l] # Shorthand for the last character
'g'
>>> name[-2] # Shorthand for next to last character
'n'
```

Note that attempting to access a character using a position that equals the string's length results in an error. The positions usually range from 0 to the length minus 1. However, Python allows negative subscript values to access characters at or near the end of a string. The programmer counts backward from –1 to access characters from the right end of the string.

The subscript operator is also useful in loops where you want to use the positions as well as the characters in a string. The next code segment uses a count-controlled loop to display the characters and their positions:

```
>>> data = "Hi there!"
>>> for index in range(len(data)):
        print(index, data[index])
0 H
1 i
2
3 t
```

```
h
5 e
6 r
7 e
8 !
```

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. 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.

To extract a substring, the programmer places a colon (:) in the subscript. An integer value can appear on either side of the colon. Here are some examples that show how slicing is used:

```
>>> 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'
```

Generally, when two integer positions are included in the slice, the range of characters in the substring extends from the first position up to but not including the second position. When the integer is omitted on either side of the colon, all of the characters extending to the end or the beginning are included in the substring.

Testing for a Substring with the in Operator: Another problem involves picking out strings that contain known substrings. For example, you might want to pick out filenames with a .txt extension. A slice would work for this, but using Python's in operator is much simpler. When used with strings, the left operand of in is a target substring, and the right operand is the string to be searched. The operator in returns True if the target string is somewhere in the search string, or

False otherwise. The next code segment traverses a list of filenames and prints just the filenames that have a .txt extension:

```
>>> fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
>>> for fileName in fileList:
        if ".txt" in fileName:
        print(fileName)
myfile.txt
yourfile.txt
```

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. Some application protocols include secure versions that use data encryption. Examples of such versions are FTPS and HTTPS, which are secure versions of FTP and HTTP for file transfer and Web page transfer, respectively.

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. Both parties to this transaction must have at their disposal one or more keys that allow them to encrypt and decrypt messages. To give you a taste of this process, let us examine an encryption strategy in detail.

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. This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence. For positive distances, the method wraps around to the beginning of the sequence to locate the replacement characters for those characters near its end. For example, if the distance value of a Caesar cipher equals three characters, the string "invaders" would be encrypted as "lqydghuv". To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its

replacement. This decryption method wraps around to the end of the sequence to find a replacement character for one near its beginning. Figure 2 shows the first five and the last five plaintext characters of the lowercase alphabet and the corresponding cipher text characters, for a Caesar cipher with a distance of 13. The numeric ASCII values are listed above and below the characters.

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 26.

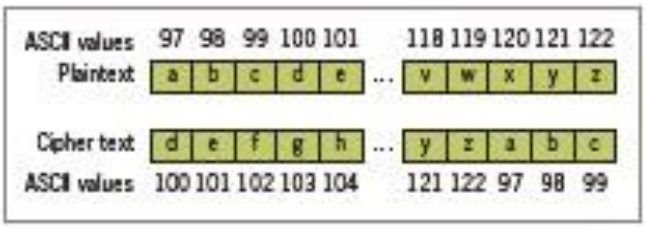

Fig : A Caesar cipher with distance 13 for the lowercase alphabet

Recall that the ord function returns the ordinal position of a character value in the ASCII sequence, whereas chr is the inverse function.

```
"""
File: encrypt.py Encrypts an input string of lowercase letters and prints the result. The other input is the distance value.
"""
plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance
    if cipherValue > ord('z'):
        cipherValue = ord('a') + distance - (ord('z') - ordvalue + 1)
```

```
        code += chr(cipherValue)
print(code)

"""
 File: decrypt.py Decrypts an input string of lowercase letters and prints the result. The other input is the distance value.
"""
code = input("Enter the coded text: ")
distance = int(input("Enter the distance value: "))
plainText = ""
for ch in code:
        ordvalue = ord(ch)
        cipherValue = ordvalue - distance
        if cipherValue < ord('a'):
                cipherValue = ord('z') - (distance - (ord('a') - ordvalue - 1))
        plainText += chr(cipherValue)
print(plainText)
```

Here are some executions of the two scripts in the IDLE shell:

```
Enter a one-word, lowercase message: invaders
Enter the distance value: 3
Lqydghuv
Enter the coded text: lqydghuv
Enter the distance value: 3
invaders
```

These scripts could easily be extended to cover all of the characters, including spaces and punctuation marks.

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 of binary 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.

Using 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 $(415)_{10}$ in the binary, octal, decimal, and hexadecimal systems:

415 in binary notation	110011111_2
415 in octal notation	637_8
415 in decimal notation	415_{10}
415 in hexadecimal notation	$19F_{16}$

The digits used in each system are counted from 0 to n – 1, where n is the system's base. Thus, the digits 8 and 9 do not appear in the octal system. To represent digits with values larger than 910, systems such as base 16 use letters.

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. The positional value of a digit is determined by raising the base of the system to the power specified by the position (baseposition). For an n-digit number, the positions (and exponents) are numbered from n – 1 down to 0, starting with the leftmost digit and moving to the right. For example, as Figure illustrates, the positional values of the three-digit. number $(415)_{10}$ are 100 (10^2), 10 (10^1), and 1 (10^0), moving from left to right in the number.

To determine the quantity represented by a number in any system from base 2 through base 10, you multiply each digit (as a decimal number) by its positional value and add the results. The following example shows how this is done for a three-digit number in base 10:

$$
\begin{aligned}
&415_{10} = \\
&4 * 10^2 + 1 * 10^1 + 5 * 10^0 = \\
&4 * 100 + 1 * 10 + 5 * 1 = \\
&400 + 10 + 5 = 415
\end{aligned}
$$

Positional values	100	10	1
Positions	2	1	0

Fig: The first three positional values in the base-10 number system

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. 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:

$$1100111_1 =$$

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

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

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

String Methods: Text processing involves many different operations on strings. For example, consider the problem of analyzing someone's writing style. Short sentences containing short words are generally considered more readable than long sentences containing long words. A program to compute a text's average sentence length and the average word length might provide a rough analysis of style.

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. We use the string method *split* to obtain a list of the words contained in an input string. We then print the length of the list, which equals the number of words, and compute and print the average of the lengths of the words in the list.

```
>>> 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
```

In short, methods are as useful as functions, but you need to get used to the dot notation, which you have already seen when using a function associated with a module. In Python, all data values are in fact objects, and every data type includes a set of methods to use with objects of that type.

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.

Text Files: 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.

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. The data in a text file can be viewed as characters, words, numbers, or lines of text, depending on the text file's format and on the purposes for which the data are used. When the data are numbers (either integers or floats), they must be separated by whitespace characters— spaces, tabs, and newlines—in the file. For example, a text file containing six floating-point numbers might look like

34.6 22.33 66.75

77.12 21.44 99.01

When examined with a text editor. Note that this format includes a space or a newline as a separator of items in the text. All data output to or input from a text file must be strings. Thus, numbers must be converted to strings before output, and these strings must be converted back to numbers after input.

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.

Writing Numbers to a File: The file method write expects a string as an argument. Therefore, other types of data, such as integers or floating-point numbers, must first be converted to strings before being written to an output file. In Python, the values of most data types can be converted to strings by using the **str** function. The resulting strings are then written to a file with a space or a newline as a separator character.

The next code segment illustrates the output of integers to a text file. Five hundred random integers between 1 and 500 are generated and written to a text file named **integers.txt**. The newline character is the separator.

```
import random
f = open("integers.txt", 'w')
for count in range(500):
        number = random.randint(1, 500)
        f.write(str(number) + '\n')
f.close()
```

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.

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

After input is finished, another call to read would return an empty string, to indicate that the end of the file has been reached. To repeat an input, the file must be reopened, in order to “rewind” it for another input process. It is not necessary to close the file. Alternatively, an application might

read and process the text one line at a time. A for loop accomplishes this nicely. The for loop views a file object as a sequence of lines of text. On each pass through the loop, the loop variable is bound to the next line of text in the sequence. Here is a session that reopens our example file and visits the lines of text in it:

```
>>> f = open("myfile.txt", 'r')
>>> for line in f:
        print(line)
First line.
Second line.
```

Note that print appears to output an extra newline. This is because each line of text input from the file retains its newline character.

In cases where you might want to read a specified number of lines from a file (say, the first line only), you can use the file method readline. The readline method consumes a line of input and returns this string, including the newline. If readline encounters the end of the file, it returns the empty string. The next code segment uses the while True loop, to input all of the lines of text with readline:

```
>>> f = open("myfile.txt", 'r')
>>> while True:
        line = f.readline()
        if line == "":
                break
        print(line)
First line.
Second line.
```

Reading Numbers from a File: All of the file input operations return data to the program as strings. If these strings represent other types of data, such as integers or floating-point numbers, the programmer must convert them to the appropriate types before manipulating them further. In Python, the string representations of integers and floating-point numbers can be converted to the numbers themselves by using the functions int and float, respectively.

When reading data from a file, another important consideration is the format of the data items in the file. During input, these data can be read with a simple for loop.

This loop accesses a line of text on each pass. To convert this line to the integer contained in it, the Programmer runs the string method strip to remove the newline and then runs the int function to obtain the integer value.

The next code segment illustrates this technique. It opens the file of random integers written earlier, reads them, and prints their sum.

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
        line = line.strip()
        number = int(line)
        theSum += number
print("The sum is", theSum)
```

Obtaining numbers from a text file in which they are separated by spaces is a bit trickier. One method proceeds by reading lines in a for loop, as before. But each line now can contain several integers separated by spaces. You can use the string method split to obtain a list of the strings representing these integers, and then process each string in this list with another for loop.

The next code segment modifies the previous one to handle integers separated by spaces and/or newlines.

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
        wordlist = line.split()
        for word in wordlist:
        number = int(word)
        theSum += number
print("The sum is", theSum)
```

Note that the line does not have to be stripped of the newline, because split takes care of that automatically.