MANIPULATING STRINGS

Working with Strings

Python lets you write, print, and access strings in your code.

String Literals

Typing string values in Python code is fairly straightforward: They begin and end with a single quote. But then how can you use a quote inside a string? Typing **'That is Alice's cat.'** won't work, because Python thinks the string ends after Alice, and the rest (s cat.') is invalid Python code.

Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it.

```
>>> spam = "That is Alice's cat."
```

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character you want to add to the string.

For example,

The escape character for a single quote is \'. You can use this inside a string that begins and ends with single quotes. To see how escape characters work,

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in Bob\'s has a backslash, it is not a single quote meant to end the string value. The escape characters \' and \" let you put single quotes and double quotes inside your strings, respectively.

Table: lists the escape characters you can use.

Table 6-1: Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

Example:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Raw Strings:

You can place an r before the beginning quotation mark of a string to make it a raw string. A *raw string* completely ignores all escape characters and prints any backslash that appears in the string.

For example:

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Remove raw String

```
>>> print('That is Carol\'s cat.')
That is Carol's cat.
>>> print(r'That is Carol\n cat')
That is Carol\n cat
>>> print('That is Carol\n cat')
That is Carol
 cat
```

- Python considers the backslash as part of the string and not as the start of an escape character.
- Raw strings are helpful if you are typing string values that contain many backslashes,

Multiline Strings with Triple Quotes

While you can use the \n escape character to put a newline into a string, it is often easier to use multiline strings.

A multiline string in Python begins and ends with either three single quotes or three double quotes.

Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks do not apply to lines inside a multiline string.

Example:

```
>>> print('''Dear Alice,
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
Sincerely,
Bob''')
```

OUTPUT

```
Dear Alice,
Eve's cat has been arrested for catnapping, cat burglary, and extortion.
Sincerely,
Bob
```

Notice that the single quote character in Eve's does not need to be escaped. Escaping single and double quotes is optional in multiline strings.

The following print() call would print identical text but doesn't use a multiline string:

Example:

```
>>> print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat burglary, and extortion.\n\nSincerely,\nBob')
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines.

The following is perfectly valid Python code:

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

Indexing and Slicing Strings

Strings use indexes and slices the same way lists do. You can think of the string 'Hello world!' as a list and each character in the string as an item with a corresponding index.

H	e	l	l	o		w	o	r	l	d	!	'
0	1	2	3	4	5	6	7	8	9	10	11	

The space and exclamation point are included in the character count, so 'Hello world!' is 12 characters long, from H at index 0 to ! at index 11.

Example:

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[4]
'o'
>>> spam[-1]
'!'
>>> spam[0:5]
'Hello'
>>> spam[:5]
'Hello'
>>> spam[6:]
'world!'
```

- If you specify an index, you'll get the character at that position in the string.
- If you specify a range from one index to another, the starting index is included and the ending index is not.
- That's why, if spam is 'Hello world!', spam[0:5] is 'Hello'.
- The substring you get from spam[0:5] will include everything from spam[0] to spam[4], leaving out the space at index 5.
- Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable.

Example:

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

The in and not in Operators with Strings

The in and not in operators can be used with strings just like with list values. An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

Example:

```
>>> 'Hello' in 'Hello World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello World'
False
>>>
>>> 'cats' not in 'cats and dogs'
False
```

These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

Useful String Methods

Several string methods analyze strings or create transformed string values.

The upper(), lower(), isupper(), and islower() String Methods

The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively. Non-letter characters in the string remain unchanged.

Example:

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
>>> spam = spam.lower()
>>> spam
'hello world!'
```

Note that these methods do not change the string itself but return new string values.

- If you want to change the original string, you have to call upper() or lower() on the string and then assign the new string to the variable where the original was stored.
- This is why you must use spam = spam.upper() to change the string in spam instead of simply spam.upper().

The upper() and lower() methods are helpful if you need to make a case-insensitive comparison.

- The strings 'great' and 'GREat' are not equal to each other. But in the following small program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to lowercase.

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

- When you run this program, the question is displayed, and entering a variation on great, such as GREat, will still give the output I feel great too.
- Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail.

```
How are you?
GREat
I feel great too.
```

The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns False.

Example:

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

- Since the upper() and lower() string methods themselves return strings, you can call string methods on *those* returned string values as well.

Expressions that do this will look like a chain of method calls.

Example:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

The isX String Methods.

- Along with islower() and isupper(), there are several string methods that have names beginning with the word *is*.
- These methods return a Boolean value that describes the nature of the string.

Here are some common is*X* string methods:

1. isalpha() returns True if the string consists only of letters and is not blank.
2. isalnum() returns True if the string consists only of letters and numbers and is not blank.
3. isdecimal() returns True if the string consists only of numeric characters and is not blank.
4. isspace() returns True if the string consists only of spaces, tabs, and newlines and is not blank.
5. istitle() returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Example:

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>>
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> ' '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
>>> 'This Is Title Case 123'.istitle()
True
>>> 'This Is not Title Case'.istitle()
False
>>> 'This Is NOT Title Case Either'.istitle()
False
```

- The is*X* string methods are helpful when you need to validate user input.

The startswith() and endswith() String Methods

The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return False.

Example:

```
>>> 'Hello world!'.startswith('Hello')
True
>>> 'Hello world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello world!'.startswith('Hello world!')
True
>>> 'Hello world!'.endswith('Hello world!')
True
```

- These methods are useful alternatives to the == equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The join() and split() String Methods

- The join() method is useful when you have a list of strings that need to be joined together into a single string value.
- The join() method is called on a string, gets passed a list of strings, and returns a string.
- The returned string is the concatenation of each string in the passed-in list.

Example:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string join() calls on is inserted between each string of the list argument.

For example, when join(['cats', 'rats', 'bats']) is called on the ', ' string, the returned string is 'cats, rats, bats'.

Remember that join() is called on a string value and is passed a list value.

The split() method does the opposite: It's called on a string value and returns a list of strings.

Example:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the string 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found.

These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the split() method to specify a different string to split upon.

Example:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of split() is to split a multiline string along the newline characters.

```
>>> spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment".
Please do not drink it.
Sincerely,
Bob'''
>>> spam.split('\n')
['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the fridge', 'that is labeled "Milk Experiment".', 'Please do not drink it.', 'Sincerely,', 'Bob']
```

Passing split() the argument '\n' lets us split the multiline string stored in spam along the newlines and return a list in which each item corresponds to one line of the string.

Justifying Text with rjust(), ljust(), and center()

- The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text.
- The first argument to both methods is an integer length for the justified string.

Example:

```
>>> 'Hello'.rjust(10)
'     Hello'
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello World'.rjust(20)
'         Hello World'
>>> 'Hello'.ljust(10)
'Hello     '
```

- 'Hello'.rjust(10) says that we want to right-justify 'Hello' in a string of total length 10. 'Hello' is five characters, so five spaces will be added to its left, giving us a string of 10 characters with 'Hello' justified right.

- An optional second argument to rjust() and ljust() will specify a fill character other than a space character.

Example:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

Example:

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

These methods are especially useful when you need to print tabular data that has the correct spacing.

Removing Whitespace with strip(), rstrip(), and lstrip

- Sometimes you may want to strip off whitespace characters (Like space, tab, and newline) from the left side, right side, or both sides of a string.
- The strip() string method will return a new string without any whitespace characters at the beginning or end.
- The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively.

Example:

```
>>> spam = '    Hello World     '
>>> spam.strip()
'Hello World'
>>> spam.lstrip()
'Hello World     '
>>> spam.rstrip()
'    Hello World'
```

- Optionally, a string argument will specify which characters on the ends should be stripped.

Example:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

- Passing strip() the argument 'ampS' will tell it to strip occurences of a, m, p, and capital S from the ends of the string stored in spam.
- The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').

Copying and Pasting Strings with the pyperclip Module

- The pyperclip module has **copy() and paste() functions** that can send text to and receive text from your computer's clipboard.
- Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software.

Note: Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing the pyperclip module.

Example:

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')
>>> pyperclip.paste()
'Hello world!'
```

----------**END**----------

Reading and Writing Files

Files and File Paths

A file has two key properties: a *filename* (usually written as one word) and a *path*.

The path specifies the location of a file on the computer.

For example,

There is a file on my Windows 7 laptop with the filename *projects.docx* in the path *C:\Users\asweigart\Documents*.

The part of the filename after the last period is called the file's *extension* and tells you a file's type. *project.docx* is a Word document, and *Users*, *asweigart*, and *Documents* all refer to *folders* (also called *directories*).

Folders can contain files and other folders.

For example, *project.docx* is in the *Documents* folder, which is inside the *asweigart* folder, which is inside the *Users* folder.

Figure 3-1 shows this folder organization.

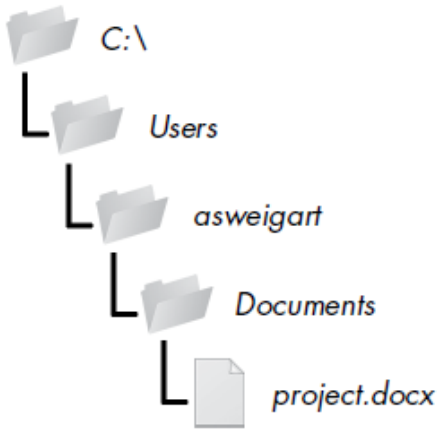

Figure 3-1: A file in a hierarchy of folders

The *C:* part of the path is the *root folder*, which contains all other folders.

On Windows, the root folder is named *C:* and is also called the *C: drive*. On OS X and Linux, the root folder is */*.

I'll be using the Windows-style root folder, *C:*. If you are entering the interactive shell examples on OS X or Linux, enter / instead.

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (\) as the separator between folder names. OS X and Linux, however, use the forward slash (/) as their path separator.

This is simple to do with the os.path.join() function. If you pass it the string values of individual file and folder names in your path, os.path.join() will return a string with a file path using the correct path separators.

```
>>> import re
>>> import os
>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

I'm running these examples on Windows, so os.path.join('usr', 'bin', 'spam') returned 'usr\\bin\\spam'. If I had called this function on OS X or Linux, the string would have been 'usr/bin/spam'.

The os.path.join() function is helpful if you need to create strings for filenames.

For example,

The following example joins names from a list of filenames to the end of a folder's name:

```
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:


        print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

The Current Working Directory

Every program that runs on your computer has a *current working directory*, or *cwd*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. You can get the current working directory as a string value with the os.getcwd() function and change it with os.chdir().

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Avinash\\AppData\\Local\\Programs\\Python\\Python38'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Here, the current working directory is set to

'C:\\Users\\Avinash\\AppData\\Local\\Programs\\Python\\Python38'*project,*

so the filename *project.docx* refers to..

'C:\\Users\\Avinash\\AppData\\Local\\Programs\\Python\\Python38'*project.docx*.

is interpreted as *C:\ Windows\project.docx*.

Python will display an error if you try to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified: 'C:\\ThisFolderDoesNo
tExist'
```

Absolute vs. Relative Paths

There are two ways to specify a file path.

1. An *absolute path*, which always begins with the root folder.
2. A *relative path,* which is relative to the program's current working directory

There are also the *dot* (.) and *dot-dot* (..) folders. These are not real folders but special names that can be used in a path.

A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."

Figure 3-2 is an example of some folders and files. When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

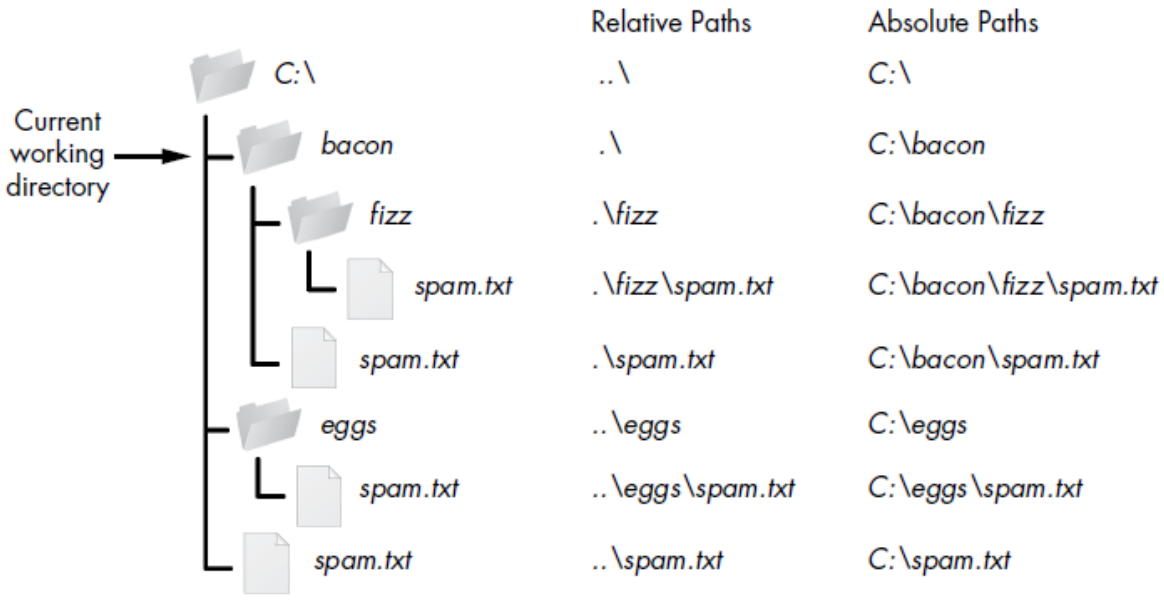

Figure 3-2: *The relative paths for folders and files in the working directory* C:\bacon

The .\ at the start of a relative path is optional.

For example, .*spam.txt* and *spam.txt* refer to the same file.

Creating New Folders with os.makedirs()

Your programs can create new folders (directories) with the os.makedirs() function.

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

This will create not just the *C:\delicious* folder but also a *walnut* folder inside *C:\delicious* and a *waffles* folder inside *C:\delicious\walnut*.

That is, os.makedirs() will create any necessary intermediate folders in order to ensure that the full path exists.

Figure 8-3 shows this hierarchy of folders.

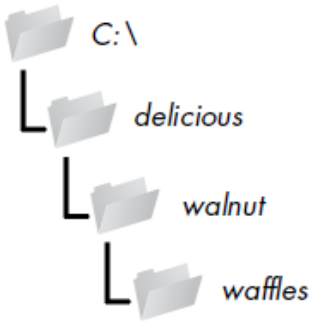

Figure 3-3: The result of os.makedirs('C:\\delicious \\walnut\\waffles')

The os.path Module

The os.path module contains many helpful functions related to filenames and file paths.

For instance, you've already used os.path.join() to build paths in a way that will work on any operating system.

Since os.path is a module inside the os module, you can import it by simply running importos.

Whenever your programs need to work with files, folders, or file paths, you can refer to the short examples in this section.

The full documentation for the os.path module is on the Python website at *http://docs.python.org/3/library/os.path.html*.

Handling Absolute and Relative Paths

The os.path module provides functions for returning the absolute path of a relative path and for checking whether a given path is an absolute path.

1. Calling os.path.abspath(*path*) will return a string of the absolute path of the argument. This is an easy way to convert a relative path into an absolute one.
2. Calling os.path.isabs(*path*) will return True if the argument is an absolute path and False if it is a relative path.
3. Calling os.path.relpath(*path*, *start*) will return a string of a relative path from the *start* path to *path*. If *start* is not provided, the current working directory is used as the start path.

Example:

```
>>> os.path.abspath('.')
'C:\\Python34'
>>> os.path.abspath('.\\Scripts')
'C:\\Python34\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Since *C:\Python34* was the working directory when os.path.abspath() was called, the "single-dot" folder represents the absolute path 'C:\\Python34'.

Another example:

```
>>> os.path.abspath('.')
'C:\\Users\\Avinash\\AppData\\Local\\Programs\\Python\\Python38'
>>> os.path.abspath('.\\Scripts')
'C:\\Users\\Avinash\\AppData\\Local\\Programs\\Python\\Python38\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Enter the following calls to os.path.relpath()

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
>>> os.getcwd()
'C:\\Python34'
```

Calling os.path.dirname(*path*) will return a string of everything that comes before the last slash in the path argument. Calling os.path.basename(*path*) will return a string of everything that comes after the last slash in the path argument.

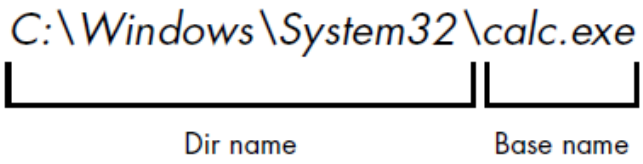

Figure: 3-4: *The base name follows the last slash in a path and is the same as the filename. The dir.name is everything before the last slash.*

For example

```
>>> path = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(path)
'calc.exe'
>>> os.path.dirname(path)
'C:\\Windows\\System32'
```

Finding File Sizes and Folder Contents

The os.path module provides functions for finding the size of a file in bytes and the files and folders inside a given folder.

- Calling os.path.getsize(*path*) will return the size in bytes of the file in the *path* argument.
- Calling os.listdir(*path*) will return a list of filename strings for each file in the *path* argument. (Note that this function is in the os module, not os.path.)

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

OR

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
922112
>>> os.listdir('C:\\Windows\\System32')
```

Squeezed text (962 lines).

As you can see, the *calc.exe* program on my computer is 776,192 bytes in size, and I have a lot of files in *C:\Windows\system32*. If I want to find the total size of all the files in this directory, I can use os.path.getsize() and os.listdir() together.

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
      totalSize = totalSize + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
1117846456
```

- As you loop over each filename in the *C:\Windows\System32* folder, the totalSize variable is incremented by the size of each file.
- Notice how when I call os.path.getsize(), I use os.path.join() to join the folder name with the current filename.
- The integer that os.path.getsize() returns is added to the value of total Size.
- After looping through all the files, you **print total Size** to see the total size of the *C:\Windows\System32* folder.

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist.

The os.path module provides functions to check whether a given path exists and whether it is a file or folder.

1. Calling os.path.exists(*path*) will return True if the file or folder referred to in the argument exists and will return False if it does not exist.
2. Calling os.path.isfile(*path*) will return True if the path argument exists and is a file and will return False otherwise.
3. Calling os.path.isdir(*path*) will return True if the path argument exists and is a folder and will return False otherwise.

```
>>> os.path.exists('C:\\Windows')
True
>>> os.path.exists('C:\\some_made_up_folder')
False
>>> os.path.isdir('C:\\Windows\\System32')
True
>>> os.path.isfile('C:\\Windows\\System32')
False
>>> os.path.isdir('C:\\Windows\\System32\\calc.exe')
False
>>> os.path.isfile('C:\\Windows\\System32\\calc.exe')
True
```

The File Reading/Writing Process

Plaintext files contain only basic text characters and do not include font, size, or color information.

Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files.

These can be opened with Windows's Notepad or OS X's TextEdit application. Your programs can easily read the contents of plaintext files and treat them as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense

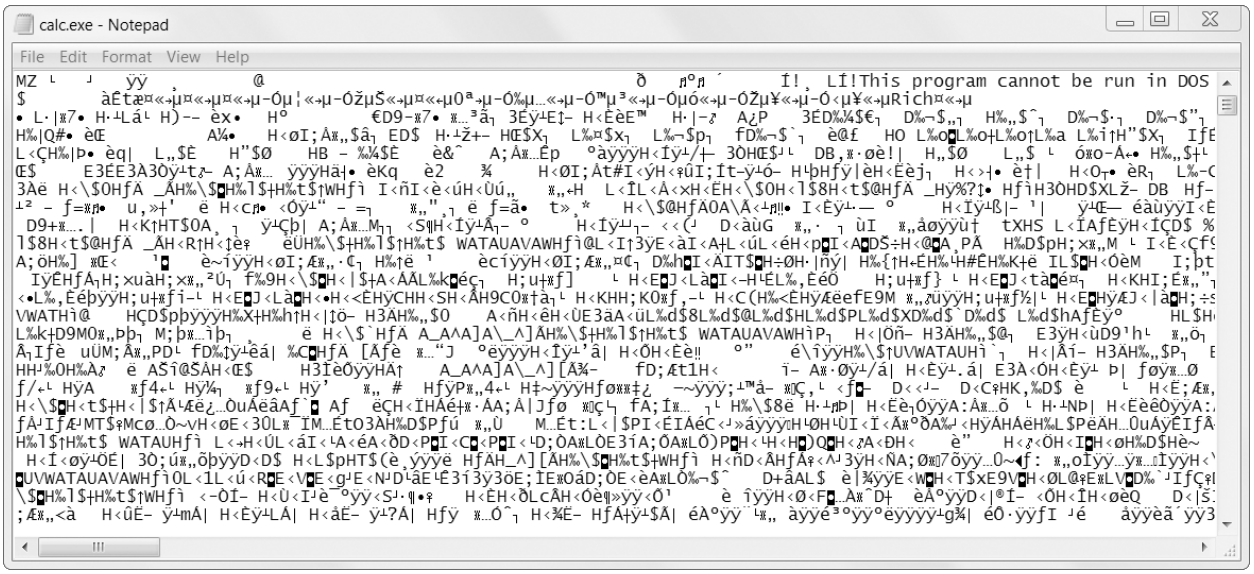

Figure 3-5: The Windows calc.exe program opened in Notepad

Since every different type of binary file must be handled in its own way, this book will not go into reading and writing raw binary files directly.

Fortunately, many modules make working with binary files easier—you will explore one of them.

There are three steps to reading or writing files in Python.

1. Call the open() function to return a File object.
2. Call the read() or write() method on the File object.
3. Close the file by calling the close() method on the File object.

Opening Files with the open() Function.

To open a file with the open() function, you pass it a string path indicating the file you want to open; it can be either an absolute or relative path. The open() function returns a File object.

Try it by creating a text file named *hello.txt* using Notepad or TextEdit. Type **Hello world!** as the content of this text file and save it in your user home folder.

```
>>> helloFile = open('C:\\Users\\your_home_folder\\hello.txt')
```

If you're using OS X, enter the following into the interactive shell instead:

```
>>> helloFile = open('/Users/your_home_folder/hello.txt')
```

- Both these commands will open the file in "reading plaintext" mode, or *read mode* for short. When a file is opened in read mode, Python lets you only read data from the file; you can't write or modify it in any way.
- Read mode is the default mode for files you open in Python. But if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value 'r' as a second argument to open().
- So open('/Users/asweigart/ hello.txt', 'r') and open('/Users/asweigart/hello.txt') do the same thing.

The call to open() returns a File object. A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with.

Reading the Contents of Files

Now that you have a File object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the File object's read() method. Let's continue with the *hello.txt* File object you stored in helloFile .

```
>>> helloContent = helloFile.read()
>>> helloContent
'Hello world!'
```

If you think of the contents of a file as a single large string value, the read() method returns the string that is stored in the file.

Alternatively, you can use the readlines() method to get a *list* of string values from the file, one string for each line of text.

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Make sure to separate the four lines with line breaks.

```
>>> sonnetFile = open('sonnet29.txt')
>>> sonnetFile.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']
```

Note that each of the string values ends with a newline character, \n , except for the last line of the file. A list of strings is often easier to work with than a single large string value.

Writing to Files

Python allows you to write content to a file in a way similar to how the print() function "writes" strings to the screen.

- Write mode will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value. Pass 'w' as the second argument to open() to open the file in write mode.
- Append mode, on the other hand, will append text to the end of the existing file. Pass 'a' as the second argument to open() to open the file in append mode.
- If the filename passed to open() does not exist, both write and append mode will create a new, blank file. After reading or writing a file, call the close() method before opening the file again.

Example:

```
>>> baconFile = open('bacon.txt', 'w')
>>> baconFile.write('Hello world!\n')
13
>>> baconFile.close()
>>> baconFile = open('bacon.txt', 'a')
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> baconFile = open('bacon.txt')
>>> content = baconFile.read()
>>> baconFile.close()
>>> print(content)
Hello world!
Bacon is not a vegetable.
```

First, we open *bacon.txt* in write mode. Since there isn't a *bacon.txt* yet, Python creates one.

Calling write() on the opened file and passing write() the string argument 'Hello world! /n' writes the string to the file and returns the number of characters written, including the newline. Then we close the file.

- To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode.
- We write 'Bacon is not a vegetable.' to the file and close it.

- Finally, to print the file contents to the screen, we open the file in its default read mode, call read(), store the resulting File object in content, close the file, and print content.

Note that the write() method does not automatically add a newline character to the end of the string like the print() function does. You will have to add this character yourself.

Saving Variables with the shelve Module

You can save variables in your Python programs to binary shelf files using the shelve module.

This way, your program can restore data to variables from the hard drive. The shelve module will let you add Save and Open features to your program.

For example,

if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load them the next time it is run.

```
>>> import shelve
>>> shelfFile = shelve.open('mydata')
>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> shelfFile['cats'] = cats
>>> shelfFile.close()
```

To read and write data using the shelve module, you first import shelve. Call shelve.open() and pass it a filename, and then store the returned shelf value in a variable.

- You can make changes to the shelf value as if it were a dictionary. When you're done, call close() on the shelf value. Here, our shelf value is stored in shelfFile.
- We create a list cats and write shelfFile['cats'] = cats to store the list in shelfFile as a value associated with the key 'cats' (like in a dictionary).
- Then we call close() on shelfFile.

- After running the previous code on Windows, you will see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On OS X, only a single *mydata.db* file will be created. These binary files contain the data you stored in your shelf.
- Programmer use the shelve module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode—they can do both once opened.

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
<class 'shelve.DbfilenameShelf'>
>>> shelfFile['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelfFile.close()
```

Here, we open the shelf files to check that our data was stored correctly. Entering shelfFile['cats'] returns the same list that we stored earlier, so we know that the list is correctly stored, and we call close().

- Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form.

```
>>> shelfFile = shelve.open('mydata')
>>> list(shelfFile.keys())
['cats']
>>> list(shelfFile.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelfFile.close()
```

- Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the shelve module.

Saving Variables with the pprint.pformat() Function

Recall from "Pretty Printing" on previous chapter that the pprint.pprint() function will "pretty print" the contents of a list or dictionary to the screen, while the pprint.pformat() function will return this same text as a string instead of printing it.

Not only is this string formatted to be easy to read, but it is also syntactically correct Python code. Say you have a dictionary stored in a variable and you want to save this variable and its contents for future use.

Using pprint.pformat() will give you a string that you can write to *.py* file. This file will be your very own module that you can import whenever you want to use the variable stored in it.

```
>>> import pprint
>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]
>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"
>>> fileObj = open('myCats.py', 'w')
>>> fileObj.write('cats = ' + pprint.pformat(cats) + '\n')
83
>>> fileObj.close()
```

We import pprint to let us use pprint.pformat(). We have a list of dictionaries, stored in a variable cats. To keep the list in cats available even after we close the shell, we use pprint.pformat() to return it as a string.

The modules that an import statement imports are themselves just Python scripts When the string from pprint.pformat() is saved to a *.py* file, the Python programs can even generate other Python programs. we can then import these files into scripts.

Example:

```
>>> import myCats
>>> myCats.cats
[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]
>>> myCats.cats[0]
{'desc': 'chubby', 'name': 'Zophie'}
>>> myCats.cats[0]['name']
'Zophie'
>>>
>>> myCats.cats[0]['desc']
'chubby'
>>> myCats.cats[1]
{'desc': 'fluffy', 'name': 'Pooka'}
>>> myCats.cats[1]['name']
'Pooka'
>>> myCats.cats[1]['desc']
'fluffy'
```

The benefit of creating a *.py* file (as opposed to saving variables with the shelve module) is that because it is a text file, the contents of the file can be read and modified by anyone with a simple text editor.

----------END----------