Reading and Writing Files

A file has two key properties: a f**ilename** and **a path**. The path specifies the location of a file on the computer.

Example:

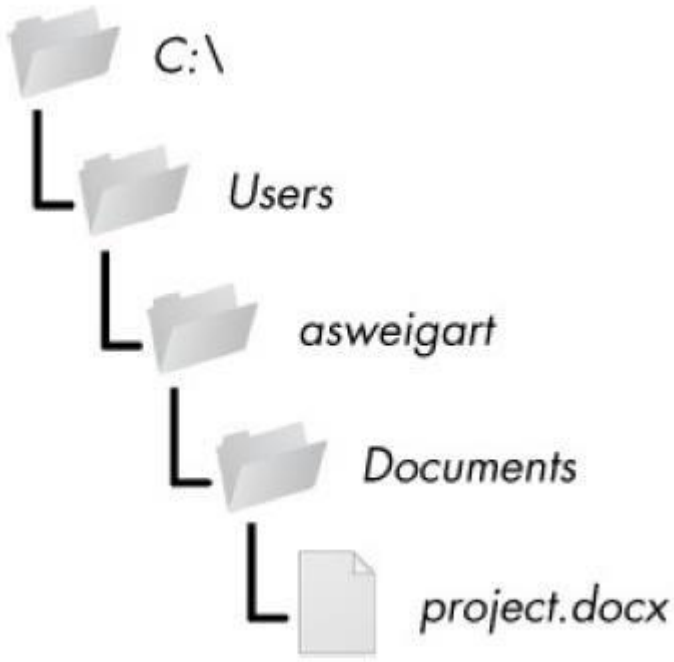
A file on my Windows 7 laptop with the filename projects.docx in the path C:\Users\asweigart\Documents.

The part of the filename after the last period is called the file‟s extension and indicates 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.

Example:

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



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 /.

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, uses the forward slash (/) as their path separator. Programs to work on all operating systems, should write Python scripts to handle both cases and can be done 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.

Example

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

Running these interactive shell examples on Windows, so os.path.join('usr', 'bin', 'spam') returned 'usr\\bin\\spam'. (Notice that the backslashes are doubled because each backslash needs to be escaped by another backslash character.)

Calling this function on OS X or Linux, the string would have been 'usr/bin/spam'. The os.path.join() function is helpful to create strings for filenames.

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 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().

Example

```
>>> import os
>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')
>>> os.getcwd()
'C:\\Windows\\System32'
```

Here, the current working directory is set to C:\Python34, so the filename project.docx refers to C:\Python34\project.docx. When we change the current working directory to C:\Windows, project.docx is interpreted as C:\ Windows\project.docx.

Python will display an error, trying to change to a directory that does not exist.

```
>>> os.chdir('C:\\ThisFolderDoesNotExist')
Traceback (most recent call last):
        File "", line 1, in
```

```
os.chdir('C:\\ThisFolderDoesNotExist')
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:\\ThisFolderDoesNotExist'
```

Absolute vs. Relative Paths

There are two ways to specify a file path.

An absolute path, which always begins with the root folder.

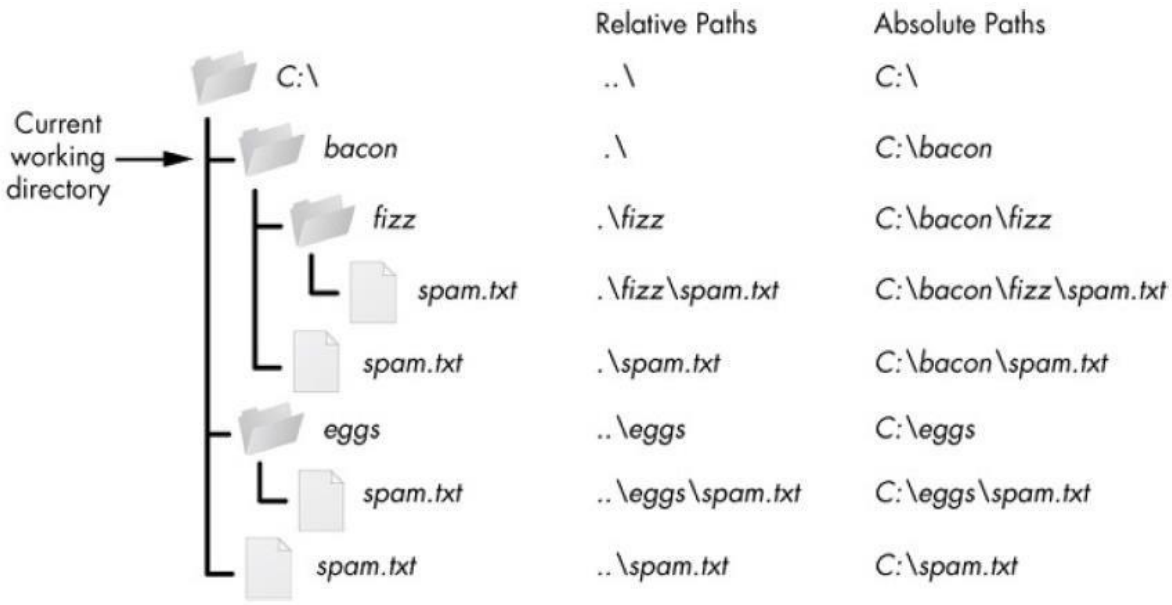
A relative path, which is relative to the program‟s current working directory.

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.

The .\ at the start of a relative path is optional. For example, .\spam.txt and spam.txt refer to the same file.



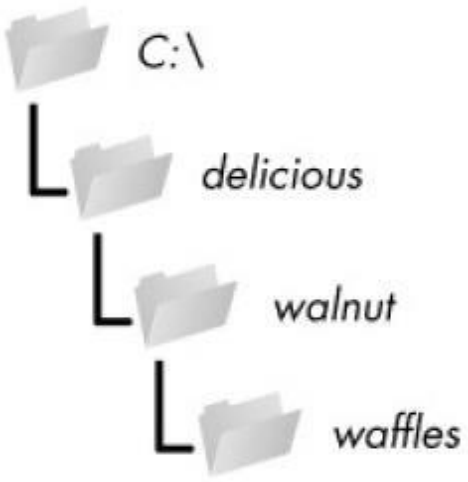
Creating New Folders with os.makedirs()

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

Example

```
>>> 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.



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.
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.
Calling os.path.isabs(path) will return True if the argument is an absolute path and False if it is a relative path.
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
```

Example:

```
>>> 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.

Example:

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

A path‟s dir name and base name together, can call os.path.split() to get a tuple value with these two strings:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
 ('C:\\Windows\\System32', 'calc.exe')
```

The same tuple can be created by calling os.path.dirname() and os.path.basename() and placing their return values in a tuple.

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

But os.path.split() is a nice if both values are required.

Note:

os.path.split() does not take a file path and return a list of strings of each folder.
By using split() string method and split on the string in os.sep.
The os.sep variable is set to the correct folder-separating slash for the computer running the program.

Example:

```
>>> calcFilePath.split(os.path.sep)
 ['C:', 'Windows', 'System32', 'calc.exe']
```

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.

Example:

```
>>> 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']
```

The calc.exe program on my computer is 776,192 bytes in size, and having a lot of files in C:\Windows\system32.
To find the total size of all the files in this directory, 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
```

Looping over each filename in the C:\Windows\System32 folder, the totalSize variable is incremented by the size of each file.

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.

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

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, pass it a string path indicating the file name 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.

Example:

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

In OS X,

Example:

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

Make sure to replace your_home_folder with your computer username.

Example, my username is asweigart, so I‟d enter 'C:\\Users\\asweigart\\ hello.txt' on Windows.

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 allows only reading data from the file; but can‟t writing or modifying 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, 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.

Reading the Contents of Files

To read the entire contents of a file as a string value, use the File object‟s read() method.
Example:

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

The read() method returns the string that is stored in the file, as a single large string value.
Alternatively, you can use the readlines() method to get a list of string values from the file, one string for each line of text.

Example:

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,

```
>>> 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,']

Writing to Files

Python allows to write content to a file in a way similar to how the print() function “writes” strings to the screen. It can‟t write to a file opened in read mode, though. Instead, need to open it in “write plaintext” mode or “append plaintext” mode, or write mode and append mode for short. Write mode will overwrite the existing file and start from scratch. 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, opened 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

To save variables in Python programs to binary shelf files using the shelve module. Hence, **program can restore data to variables from the hard drive.**

Example:

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

To read and write data using the shelve module, first import shelve. Call shelve.open() and pass it a filename, and then store the returned shelf value in a variable. Shelf values can also be changed as if it were a dictionary. Finally, 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, 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 stored in shelf. The format of these binary files is not important; only need to know what the shelve module does, not how it does it. Programs can 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.

Example:

```
>>> shelfFile = shelve.open('mydata')
>>> type(shelfFile)
>>> 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 listmlike values of the keys and values in the shelf. Since these methods return list-like values instead of true lists.

Example:

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

Plaintext is useful for creating files that will read in a text editor such as Notepad or TextEdit, but to save data from Python programs, use the shelve module.

Saving Variables with the pprint.pformat()

Function from Pretty Printing 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.
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.
Example:

```
>>> 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()
```

Here, we import pprint to 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. Once we have the data in cats as a string, it‟s easy to write the string to a file, which we‟ll call myCats.py. 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 file is a module that can be imported just like any other. And since Python scripts are themselves just text files with the .py file extension, your Python programs can even generate other Python programs.

Example:

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

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. For most applications, however, saving data using the shelve module is the preferred way to save variables to a file. Only basic data types such as integers, floats, strings, lists, and

dictionaries can be written to a file as simple text. File objects, for example, cannot be encoded as text.

Organizing Files

With OS X and Linux, file browser most likely shows extensions automatically. With Windows, file extensions may be hidden by default. To show extensions, go to Start ▸ Control Panel ▸ Appearance and Personalization ▸ Folder Options. On the View tab, under Advanced Settings, uncheck the Hide extensions for known file types checkbox.

The shutil Module

The shutil (or shell utilities) module has functions to copy, move, rename, and delete files in Python programs.
To use the shutil functions, first need to use import shutil.

Copying Files and Folders

The shutil module provides functions for copying files, as well as entire folders. Calling shutil.copy(source, destination) will copy the file at the path source to the folder at the path destination. (Both source and destination are strings.) If destination is a filename, it will be used as the new name of the copied file. This function returns a string of the path of the copied file.
Example:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
 'C:\\delicious\\spam.txt'
 >>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
 'C:\\delicious\\eggs2.txt'

>>> import shutil,os
>>> os.chdir('D:\\TPS\\PYTHON')
>>> shutil.copy('D:\\TPS\\PYTHON\\files\\read_files.txt','D:\\TPS\\BIT\\ADP')
```

The first shutil.copy() call copies the file at C:\spam.txt to the folder C:\delicious. The return value is the path of the newly copied file.

Note: A folder was specified as the destination, the original spam.txt filename is used for the new, copied file‟s filename.

The second shutil.copy() call also copies the file at C:\eggs.txt to the folder C:\delicious but gives the copied file the name eggs2.txt.

While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it. Calling shutil.copytree(source, destination) will copy the folder at the path source, along with all of its files and subfolders, to the folder at the path destination. The source and destination parameters are both strings. The function returns a string of the path of the copied folder.

Example:

```
>>> import shutil, os
>>> os.chdir('C:\\')
>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

```
import shutil,os
>>> os.chdir('D:\\TPS\\PYTHON')
>>> shutil.copytree('D:\\TPS\\PYTHON\\files','D:\\TPS\\BIT\\ADP\\FILES1')
```

The shutil.copytree() call creates a new folder named bacon_backup with the same content as the original bacon folder.

Moving and Renaming Files and Folders

Calling shutil.move(source, destination) will move the file or folder at the path source to the path destination and will return a string of the absolute path of the new location. If destination points to a folder, the source file gets moved into destination and keeps its current filename.

Example:

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

Assuming a folder named eggs already exists in the C:\ directory, this shutil.move() calls says, “Move C:\bacon.txt into the folder C:\eggs.” If there had been a bacon.txt file already in C:\eggs, it would have been overwritten. Since it‟s easy to overwrite files. The destination path can also specify a filename.

Example:

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
 'C:\\eggs\\new_bacon.txt'
```

This line says, “Move C:\bacon.txt into the folder C:\eggs, rename that bacon.txt file to new_bacon.txt.” Both of the previous examples worked under the assumption that there was a folder eggs in the C:\ directory. But if there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
 'C:\\eggs'
```

Here, move() can‟t find a folder named eggs in the C:\ directory and so assumes that destination must be specifying a filename, not a folder. So the bacon.txt text file is renamed to eggs.
Finally, the folders that make up the destination must already exist, or else Python will throw an exception.

Example:

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')

Traceback (most recent call last):
        File "C:\Python34\lib\shutil.py", line 521, in move
os.rename(src, real_dst)
FileNotFoundError: [WinError 3] The system cannot find the path specified:
'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
        File "", line 1, in
                shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
        File "C:\Python34\lib\shutil.py", line 533, in move
                copy2(src, real_dst)
        File "C:\Python34\lib\shutil.py", line 244, in copy2
                copyfile(src, dst, follow_symlinks=follow_symlinks)
        File "C:\Python34\lib\shutil.py", line 108, in
                copyfile with open(dst, 'wb') as fdst:
FileNotFoundError: [Errno 2] No such file or directory: 'c:\\does_not_exist\\ eggs\\ham'
```

Python looks for eggs and ham inside the directory does_not_exist. It doesn‟t find the nonexistent directory, so it can‟t move spam.txt to the path you specified.

Permanently Deleting Files and Folders

Deleting a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, can use the shutil module.
Calling os.unlink(path) will delete the file at path.
Calling os.rmdir(path) will delete the folder at path. This folder must be empty of any files or folders.
Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted.

Note:
Be careful when using these functions in your programs! It‟s often a good idea to first run your program with these calls commented out and with print() calls added to show the files that would be deleted. Here is a Python program that was intended to delete files that have the .txt file extension but has a typo (highlighted in bold) that causes it to delete .rxt files instead:

```
import os for filename in os.listdir():
        if filename.endswith('.rxt'):
                os.unlink(filename)
```

Any important files ending with .rxt, they would have been accidentally, permanently deleted. Instead, first run the program:

```
import os
for filename in os.listdir():
        if filename.endswith('.rxt'):
                #os.unlink(filename)
                print(filename)
```

Now the os.unlink() call is commented, so Python ignores it. Instead, print the filename of the file that would have been deleted. Running this version of the program first will show that you‟ve accidentally told the program to delete .rxt files instead of .txt files. Once you are certain the program works as intended, delete the print(filename) line and uncomment the os.unlink(filename) line. Then run the program again to actually delete the files.

Safe Deletes with the send2trash Module

Python‟s built-in shutil.rmtree() function irreversibly deletes files and folders, it can be dangerous to use. A much better way to delete files and folders is with the thirdparty send2trash module.

Install this module by running pip install send2trash from a Terminal window.
Using send2trash is much safer than Python‟s regular delete functions, because it will send folders and files to your computer‟s trash or recycle bin instead of permanently deleting them. If a bug in your program deletes something with send2trash you didn‟t intend to delete, you can later restore it from the recycle bin.

Example:

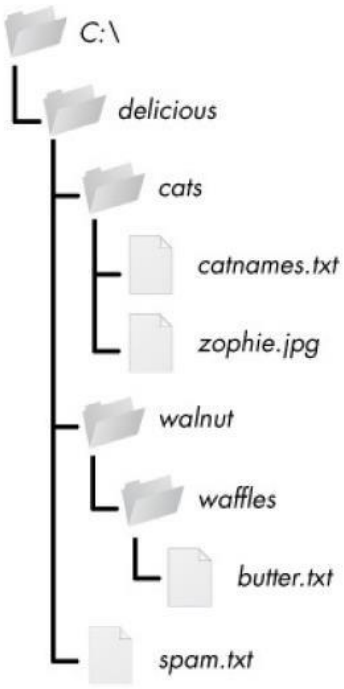
```
>>> import send2trash
>>> baconFile = open('bacon.txt', 'a') # creates the file
>>> baconFile.write('Bacon is not a vegetable.')
25
>>> baconFile.close()
>>> send2trash.send2trash('bacon.txt')
```

Use the send2trash.send2trash() function to delete files and folders. But while sending files to the recycle bin helps to recover them later, it will not free up disk space like permanently deleting them does. To free up disk space, use the os and shutil functions for deleting files and folders.

Note:
The send2trash() function can only send files to the recycle bin; it cannot pull files out of it.

Walking a Directory Tree

To rename every file in some folder and also every file in every subfolder of that folder. That is, to navigate through the directory tree, touching each file in the path. Python provides a function to handle this process for you. Let‟s look at the C:\delicious folder with its contents, shown in Figure




```
import os
for folderName, subfolders, filenames in os.walk('C:\\delicious'):
    print('The current folder is ' + folderName)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
    for filename in filenames:
        print('FILE INSIDE ' + folderName + ': '+ filename)

    print('')

    import os
    >>> for folderName, subfolders, filenames in os.walk('D:\\TPS\\PYTHON'):
        print('The current folder is ' + folderName)
```

The os.walk() function is passed a single string value: the path of a folder. os.walk() function is used in a for loop statement to walk a directory tree, much like how you can use the range() function to walk over a range of numbers. Unlike range(), the os.walk() function will return three values on each iteration through the loop:

1. A string of the current folder‟s name
2. A list of strings of the folders in the current folder

3. A list of strings of the files in the current folder (By current folder, I mean the folder for the current iteration of the for loop. The current working directory of the program is not changed by os.walk().)

Since , it returns three values hence make use of three variables with the names foldername, subfolders, and filenames.

Output:

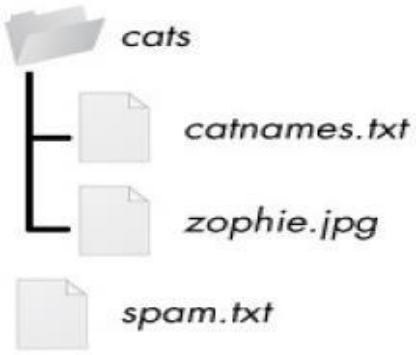
```
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt.
```

Since os.walk() returns lists of strings for the subfolder and filename variables, you can use these lists in their own for loops.

Compressing Files with the zipfile Module

ZIP files (with the .zip file extension), which can hold the compressed contents of many other files.

Compressing a file reduces its size, which is useful when transferring it over the Internet. And since a ZIP file can also contain multiple files and subfolders, it‟s a handy way to package several files into one. This single file, called an archive file, can then be, attached to an email. Python programs can both create and open (or extract) ZIP files using functions in the **zipfile module.**



Reading ZIP Files

To read the contents of a ZIP file, first create a **ZipFile object** (note the capital letters Z and F). ZipFile objects are conceptually similar to the File objects. They are values through which the program interacts with the file. To create a ZipFile object, call the zipfile.ZipFile() function, passing it a string of the .zip file‟s filename.

zipfile is the name of the **Python module** and **ZipFile()** is the name of the **function.**

Example

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.namelist()
['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
>>> spamInfo = exampleZip.getinfo('spam.txt')
>>> spamInfo.file_size
13908
>>> spamInfo.compress_size
3828
>>> 'Compressed file is %sx smaller!' % (round(spamInfo.file_size / spamInfo .compress_size,
2))
'Compressed file is 3.63x smaller!'
>>> exampleZip.close()
```

A ZipFile object has a namelist() method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the getinfo() ZipFile method to return a ZipInfo object about that particular file. ZipInfo objects have their own attributes, such as file_size and compress_size in bytes, which hold integers of the original file size and compressed file size, respectively. While a ZipFile object represents an entire archive file, a ZipInfo object holds useful information about a single file in the archive.

The command calculates how efficiently example.zip is compressed by dividing the original file size by the compressed file size and prints this information using a string formatted with %s.

Extracting from ZIP Files

The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.

Example:

```
>>> import zipfile, os
>>> os.chdir('C:\\') # move to the folder with example.zip
>>> exampleZip = zipfile.ZipFile('example.zip')
>>> exampleZip.extractall()
>>> exampleZip.close()
```

After running this code, the contents of example.zip will be extracted to C:\. Optionally, can pass a folder name to extractall() to have it extract the files into a folder other than the current working directory. If the folder passed to the extractall() method does not exist, it will be created. For instance, replaced the call with exampleZip.extractall('C:\\ delicious'), the code would extract the files from example.zip into a newly created C:\delicious folder. The extract() method for ZipFile objects will extract a single file from the ZIP file.

Example:

```
>>> exampleZip.extract('spam.txt')
'C:\\spam.txt'
>>> exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\spam.txt'
>>> exampleZip.close()
```

The string pass to extract() must match one of the strings in the list returned by namelist(). Optionally, can pass a second argument to extract() to extract the file into a folder other than the current working directory. If this second argument is a folder that doesn‟t yet exist, Python will create the folder. The value that extract() returns is the absolute path to which the file was extracted.

Creating and Adding to ZIP Files

To create compressed ZIP files, must open the ZipFile object in write mode by passing 'w' as the second argument.

By passing a path to the write() method of a ZipFile object, Python will compress the file at that path and add it into the ZIP file. The write() method‟s first argument is a string of the filename to add. The second argument is the compression type parameter, which tells the computer what algorithm it should use to compress the files.

you can always just set this value to zipfile.ZIP_DEFLATED. (This specifies the deflate compression algorithm, which works well on all types of data.)

Example:

```
>>> import zipfile
>>> newZip = zipfile.ZipFile('new.zip', 'w')
>>> newZip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
>>> newZip.close()
```

This code will create a new ZIP file named new.zip that has the compressed contents of spam.txt. Writing to files in write mode will erase all existing contents of a ZIP file. To add files to an existing ZIP file, pass 'a' as the second argument to zipfile.ZipFile() to open the ZIP file in append mode.

Debugging

Writting more complicated programs, may start finding not-so-simple to find bugs in them. There are some tools and techniques for finding the root cause of bugs in program and help to fix bugs faster and with less effort.

There are a few tools and techniques to identify what exactly code is doing and where it‟s going wrong.

First, will look at **logging** and **assertions,** are two features that can helps to detect bugs early.

The earlier catch bugs it will be easier to fix.

Second, will look at how to use the **debugger.** The debugger is a feature of IDLE that executes a program one instruction at a time, giving a chance to inspect the values in variables while code runs, and track how the values change over the course of your program. This is much slower than running the program at full speed, but it is helpful to see the actual values in a program while it runs, rather than deducing what the values might be from the source code.

Raising Exceptions

Python raises an exception whenever it tries to execute invalid code.

To handle Python‟s exceptions will use **try and except** statements to recover from exceptions.

But can also **raise own exceptions** in code by raising a exception.

Raising an exception is a way of saying, “Stop running the code in this function and move the program execution to the except statement.” Exceptions are raised with a raise statement.

In code, a raise statement consists of the following:

The raise keyword

A call to the Exception() function

A string with a helpful error message passed to the Exception() function

Example:

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
```

```
    File "", line 1, in
        raise Exception('This is the error message.')
Exception: This is the error message.
```

If there are no try and except statements covering the raise statement that raised the exception, the program simply crashes and displays the exception‟s error message.

Often it‟s the code that calls the function, not the fuction itself, that knows how to handle an exception. So you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.

Example:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
        raise Exception('Symbol must be a single character string.')
    if width <= 2:
        raise Exception('Width must be greater than 2.')
    if height <= 2:
        raise Exception('Height must be greater than 2.')

    print(symbol * width)

    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
        try:
            boxPrint(sym, w, h)
        except Exception as err:
            print('An exception happened: ' + str(err))
```

A boxPrint() function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height. This box shape is printed to the console.
The character to be a single character, and the width and height to be greater than 2.
Adding if statements to raise exceptions if these requirements aren‟t satisfied.
By calling boxPrint() with various arguments, our try/except will handle invalid arguments.

This program uses the except Exception as err form of the except statement. If an Exception object is returned from boxPrint(), this except statement will store it in a variable named err. The

Exception object can then be converted to a string by passing it to str() to produce a user-friendly error message.

Output:

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOO
O                 O
O                 O
O                 O
OOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Using the try and except statements, you can handle errors more gracefully instead of letting the entire program crash.

Getting the Traceback as a String

When Python encounters an error, it produces a error information called the traceback. The traceback includes the error message, the line number of the line that caused the error, and the sequence of the function calls that led to the error. This sequence of calls is called the call stack.

Example:

```
def spam():
    bacon()
def bacon():
    raise Exception('This is the error message.')

spam()
```

Output:

```
Traceback (most recent call last):
    File "errorExample.py", line 7, in
        spam()
    File "errorExample.py", line 2, in spam
        bacon()
    File "errorExample.py", line 5, in bacon
        raise Exception('This is the error message.')
Exception: This is the error message.
```

From the traceback, can see that the error happened on line 5, in the bacon() function. This particular call to bacon() came from line 2, in the spam() function, which in turn was called on line 7.

In programs where functions can be called from multiple places, the call stack can help to determine which call led to the error. The traceback is displayed by Python whenever a raised exception goes unhandled. But can also obtain it as a string by calling traceback.format_exc(). This function is useful if the information from an exception‟s traceback is needed and also want an except statement to gracefully handle the exception.

Instead of crashing your program right when an exception occurs, can write the traceback information to a log file and keep your program running. Later can look at the log file, to debug your program.

Example:

```
>>> import traceback
>>> try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')
```

Output:

```
116
The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the write() method, since 116 characters were written to the file. The traceback text was written to errorInfo.txt.

```
Traceback (most recent call last):
        File "<pyshell#28>", line 2, in
Exception: This is the error message.
```