Organizing Fi les

The shutil Module

- The shutil (or shell utilities) module has functions to let you copy, move, rename, and delete files in your Python programs. To use the shutil functions, you will 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 *destinations 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.

Enter the following into the interactive shell to see how shutil.copy() works:

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

- 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 that since a folder was specified as the destination **1,** the original *spam.txt* filename is used for the new, copied file's filename.
- The second shutil.copy() call **2,** 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'
```

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.

For 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 accidentally overwrite files in this way, you should take some care when using move().
- ✓ The *destination* path can also specify a filename.

In the following example, the *source* file is moved *and* renamed.

```
>>> 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,* and while you're at it, 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* (a text file without the *.txt* file extension).

- This can be a tough-to-spot bug in your programs since the move() call can happily do something that might be quite different from what you were expecting. This is yet another reason to be careful when using move().

- Finally, the folders that make up the destination must already exist, or else Python will throw an exception.

Enter the following into the interactive shell:

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

```
>>> shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
Traceback (most recent call last):
  File "C:\Users\AGM\AppData\Local\Programs\Python\Python38-32\lib\shutil.py", line 788, in move
    os.rename(src, real_dst)
FileNotFoundError: [WinError 2] The system cannot find the file specified: 'spam.txt' -> 'c:\\does_not_exist\\eggs\\ham'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    shutil.move('spam.txt', 'c:\\does_not_exist\\eggs\\ham')
  File "C:\Users\AGM\AppData\Local\Programs\Python\Python38-32\lib\shutil.py", line 802, in move
    copy_function(src, real_dst)
  File "C:\Users\AGM\AppData\Local\Programs\Python\Python38-32\lib\shutil.py", line 432, in copy2
    copyfile(src, dst, follow_symlinks=follow_symlinks)
  File "C:\Users\AGM\AppData\Local\Programs\Python\Python38-32\lib\shutil.py", line 261, in copyfile
    with open(src, 'rb') as fsrc, open(dst, 'wb') as fdst:
FileNotFoundError: [Errno 2] No such file or directory: 'spam.txt'
```

✓ 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

You can delete a single file or a single empty folder with functions in the os module, whereas to delete a folder and all of its contents, you use the shutil module.

1. Calling os.unlink(*path*) will delete the file at *path*.
2. Calling os.rmdir(*path*) will delete the folder at *path*. This folder must be empty of any files or folders.
3. Calling shutil.rmtree(*path*) will remove the folder at *path*, and all files and folders it contains will also be deleted.

   Be careful when using these functions in your programs!

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)
```

- ✓ If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
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, you will print the filename of the file that would have been deleted.

Running this version of the program first will show you that you've accidentally told the program to delete *.rxt* files instead of *.txt* files.

Safe Deletes with the send2trash Module

Since 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 third-party send2trash module.

You can 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.

After you have installed send2trash

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')
```

In general, you should always use the send2trash.send2trash() function to delete files and folders.

- But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does.
- If you want your program to free up disk space, use the os and shutil functions for deleting files and folders. Note that the send2trash() function can only send files to the recycle bin; it cannot pull files out of it.

WALKING A DIRECTORY TREE

Roughly you want to rename every file in some folder and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, touching each file as you go.

Writing a program to do this could get tricky; fortunately, Python provides a function to handle this process for you.

Let's look at the *C:\delicious* folder with its contents, shown in Figure 3.4

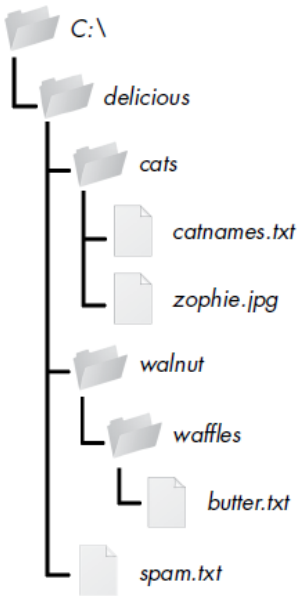

Figure 3.4: An example folder that contains three folders and four files

```
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('')
```

- The os.walk() function is passed a single string value: the path of a folder.

You can use os.walk() 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

Note: 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().)

- Just like you can choose the variable name i in the code for i in range(10):, you can also choose the variable names for the three values listed earlier. I usually use the names folder name, subfolders, and filenames.

When you run this program, it will output the following:

```
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. Replace the print() function calls with your own custom code. (Or if you don't need one or both of them, remove the for loops.)

COMPRESSING FILES WITH THE ZIPFILE MODULE

You may be familiar with 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, say, attached to an email.

Your Python programs can both create and open (or *extract*) ZIP files using functions in the zipfile module. Say you have a ZIP file named *example.zip* that has the contents shown in Figure 3.2.

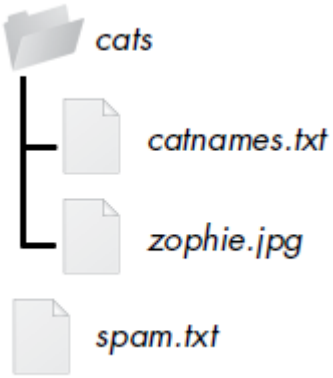

Figure 3.2: *The contents of* example.zip

Reading ZIP Files

To read the contents of a ZIP file, first you must create a ZipFile object (note the capital letters *Z* and *F*). ZipFile objects are conceptually similar to the File objects.

To create a ZipFile object, call the zipfile.ZipFile() function, passing it a string of the *.zip* file's filename. Note that zipfile is the name of the Python module, and ZipFile() is the name of the function.

```
>>> 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
❶ >>> '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 at 1 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.

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

After running this code, the contents of *example.zip* will be extracted to *C:*. Optionally, you 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, if you replaced the call at 1 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.

```
>>> 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 you pass to extract() must match one of the strings in the list returned by namelist(). Optionally, you 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 your own compressed ZIP files, you must open the ZipFile object in *write mode* by passing 'w' as the second argument.

When you pass 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.

```
>>> 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*.

Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to zipfile.ZipFile() to open the ZIP file in *append mode*.