Module3
Ch.10
ORGANIZING FILES

In the previous chapter, you learned how to create and write to new files in Python. Your programs can also organize preexisting files on the hard drive. Maybe you've had the experience of going through a folder full of dozens, hundreds, or even thousands of files and copying, renaming, moving, or compressing them all by hand. Or consider tasks such as these:

- Making copies of all PDF files (and *only* the PDF files) in every subfolder of a folder
- Removing the leading zeros in the filenames for every file in a folder of hundreds of files named *spam001.txt*, *spam002.txt*, *spam003.txt*, and so on
- Compressing the contents of several folders into one ZIP file (which could be a simple backup system)

All this boring stuff is just begging to be automated in Python. By programming your computer to do these tasks, you can transform it into a quick-working file clerk who never makes mistakes.

As you begin working with files, you may find it helpful to be able to quickly see what the extension (*.txt*, *.pdf*, *.jpg*, and so on) of a file is. With macOS and Linux, your 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 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 destination can be strings or Path objects.) If destination is a filename, it will be used as the new name of the copied file. This function returns a string or Path object of the copied file.

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

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
❶ >>> shutil.copy(p / 'spam.txt', p / 'some_folder')
'C:\\Users\\Al\\some_folder\\spam.txt'
❷ >>> shutil.copy(p / 'eggs.txt', p / 'some_folder/eggs2.txt')
WindowsPath('C:/Users/Al/some_folder/eggs2.txt')
```

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

that since a folder was specified as the destination ❶, the original *spam.txt* filename is used for the new, copied file's filename. The second shutil.copy() call ❷ also copies the file at *C:\Users\Al\eggs.txt* to the folder *C:\Users\Al\some_folder* 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.
Enter the following into the interactive shell:

```
>>> import shutil, os
>>> from pathlib import Path
>>> p = Path.home()
>>> shutil.copytree(p / 'spam', p / 'spam_backup')
WindowsPath('C:/Users/Al/spam_backup')
```

The shutil.copytree() call creates a new folder named *spam_backup* with the same content as the original *spam* folder. You have now safely backed up your precious, precious spam.

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, enter the following into the interactive shell:

```
>>> 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() call 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)—probably not what you wanted!

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')
Traceback (most recent call last):
  --snip--
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

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.

- 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.

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
from pathlib import Path
for filename in Path.home().glob('*.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
from pathlib import Path
for filename in Path.home().glob('*.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.

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

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 --user send2trash from a Terminal window. (See Appendix A for a more in-depth explanation of how to install third-party modules.)
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, enter the following into the interactive shell:

```
>>> 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

Say 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 10-1.

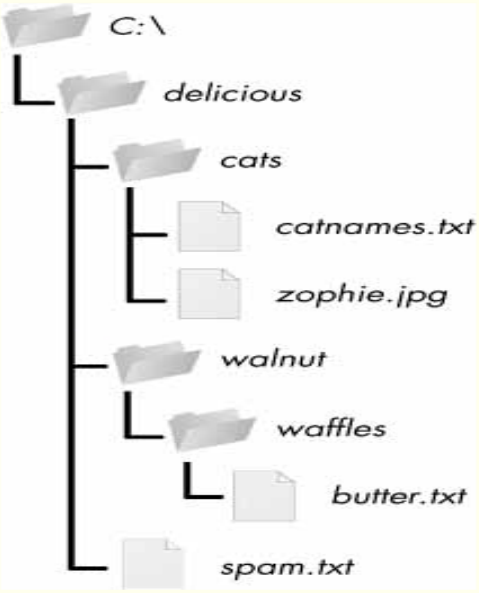


Figure 10-1: An example folder that contains three folders and four files

Here is an example program that uses the os.walk() function on the directory tree from Figure 10-1:

```
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:

- A string of the current folder's name
- A list of strings of the folders in the current folder
- 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().)
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 foldername, subfolders, and filenames.

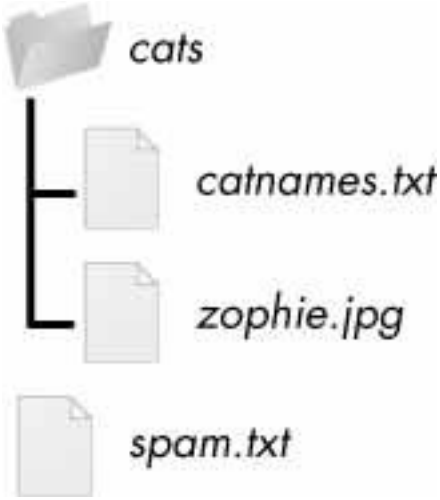
When you run this program, it will output the following:

```
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 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 10-2.



You can download this ZIP file from https://nostarch.com/automatestuff2/ or just follow along using a ZIP file already on your computer.

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 you saw returned by the open() function in the previous chapter: 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. Note that zipfile is the name of the Python module, and ZipFile() is the name of the function.
For example, enter the following into the interactive shell:

```
>>> import zipfile, os

>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / '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 >>> f'Compressed file is {round(spamInfo.file_size / spamInfo
.compress_size, 2)}x smaller!'
)
'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 ❶ calculates how efficiently *example.zip* is compressed by dividing the original file size by the compressed file size and prints this information.

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
>>> from pathlib import Path
>>> p = Path.home()
>>> exampleZip = zipfile.ZipFile(p / '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 ❶ 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. Continue the interactive shell 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 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. (This is similar to opening a text file in write mode by passing 'w' to the open() function.)
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. (This specifies the *deflate* compression algorithm, which works well on all types of data.) Enter the following into the interactive shell:

```
>>> 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*.

PROJECT: RENAMING FILES WITH AMERICAN-STYLE DATES TO EUROPEAN-STYLE DATES

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Let's write a program to do it instead.

Here's what the program does:

1. It searches all the filenames in the current working directory for American-style dates.
2. When one is found, it renames the file with the month and day swapped to make it European-style.

This means the code will need to do the following:

1. Create a regex that can identify the text pattern of American-style dates.
2. Call os.listdir() to find all the files in the working directory.
3. Loop over each filename, using the regex to check whether it has a date.
4. If it has a date, rename the file with shutil.move().

For this project, open a new file editor window and save your code as *renameDates.py*.

Step 1: Create a Regex for American-Style Dates

The first part of the program will need to import the necessary modules and create a regex that can identify MM-DD-YYYY dates. The to-do comments will remind you what's left to write in this program. Typing them as TODO makes them easy to find using Mu editor's CTRL-F find feature. Make your code look like the following:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

❶ import shutil, os, re

# Create a regex that matches files with the American date format.
❷ datePattern = re.compile(r"""^(.*?) # all text before the date
    ((0|1)?\d)-                 # one or two digits for the month
    ((0|1|2|3)?\d)-             # one or two digits for the day
    ((19|20)\d\d)               # four digits for the year
    (.*?)$                      # all text after the date
    """, re.VERBOSE❸)

# TODO: Loop over the files in the working directory.

# TODO: Skip files without a date.

# TODO: Get the different parts of the filename.

# TODO: Form the European-style filename.

# TODO: Get the full, absolute file paths.
```

```
# TODO: Rename the files.
```

From this chapter, you know the shutil.move() function can be used to rename files: its arguments are the name of the file to rename and the new filename. Because this function exists in the shutil module, you must import that module ❶.

But before renaming the files, you need to identify which files you want to rename. Filenames with dates such as *spam4-4-1984.txt* and *01-03-2014eggs.zip* should be renamed, while filenames without dates such as *littlebrother.epub* can be ignored.

You can use a regular expression to identify this pattern. After importing the re module at the top, call re.compile() to create a Regex object ❷. Passing re.VERBOSE for the second argument ❸ will allow whitespace and comments in the regex string to make it more readable.

The regular expression string begins with ^(.*?) to match any text at the beginning of the filename that might come before the date. The ((0|1)?\d) group matches the month. The first digit can be either 0 or 1, so the regex matches 12 for December but also 02 for February. This digit is also optional so that the month can be 04 or 4 for April. The group for the day is ((0|1|2|3)?\d) and follows similar logic; 3, 03, and 31 are all valid numbers for days. (Yes, this regex will accept some invalid dates such as 4-31-2014, 2-29-2013, and 0-15-2014. Dates have a lot of thorny special cases that can be easy to miss. But for simplicity, the regex in this program works well enough.)

While 1885 is a valid year, you can just look for years in the 20th or 21st century. This will keep your program from accidentally matching nondate filenames with a date-like format, such as *10-10-1000.txt*.

The (.*?)$ part of the regex will match any text that comes after the date.

Step 2: Identify the Date Parts from the Filenames

Next, the program will have to loop over the list of filename strings returned from os.listdir() and match them against the regex. Any files that do not have a date in them should be skipped. For filenames that have a date, the matched text will be stored in several variables. Fill in the first three TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format
# to European DD-MM-YYYY.

--snip--

# Loop over the files in the working directory.
for amerFilename in os.listdir('.'):
    mo = datePattern.search(amerFilename)

    # Skip files without a date.
❶   if mo == None:
❷       continue

❸   # Get the different parts of the filename.
```

```
    beforePart = mo.group(1)
    monthPart  = mo.group(2)
    dayPart    = mo.group(4)
    yearPart   = mo.group(6)
    afterPart  = mo.group(8)

--snip--
```

If the Match object returned from the search() method is None ❶, then the filename in amerFilename does not match the regular expression. The continue statement ❷ will skip the rest of the loop and move on to the next filename.

Otherwise, the various strings matched in the regular expression groups are stored in variables named beforePart, monthPart, dayPart, yearPart, and afterPart ❸. The strings in these variables will be used to form the European-style filename in the next step.

To keep the group numbers straight, try reading the regex from the beginning, and count up each time you encounter an opening parenthesis. Without thinking about the code, just write an outline of the regular expression. This can help you visualize the groups. Here's an example:

```
datePattern = re.compile(r"""^(1) # all text before the date
    (2 (3) )-                 # one or two digits for the month
    (4 (5) )-                 # one or two digits for the day
    (6 (7) )                  # four digits for the year
    (8)$                      # all text after the date
    """, re.VERBOSE)
```

Here, the numbers **1** through **8** represent the groups in the regular expression you wrote. Making an outline of the regular expression, with just the parentheses and group numbers, can give you a clearer understanding of your regex before you move on with the rest of the program.

Step 3: Form the New Filename and Rename the Files

As the final step, concatenate the strings in the variables made in the previous step with the European-style date: the date comes before the month. Fill in the three remaining TODOs in your program with the following code:

```
#! python3
# renameDates.py - Renames filenames with American MM-DD-YYYY date format # to
European DD-MM-YYYY.

--snip--

    # Form the European-style filename.
❶   euroFilename = beforePart + dayPart + '-' + monthPart + '-' + yearPart +
                   afterPart

    # Get the full, absolute file paths.
    absWorkingDir = os.path.abspath('.')
    amerFilename = os.path.join(absWorkingDir, amerFilename)
    euroFilename = os.path.join(absWorkingDir, euroFilename)
```

```
    # Rename the files.
❷ print(f'Renaming "{amerFilename}" to "{euroFilename}"...')
❸ #shutil.move(amerFilename, euroFilename)  # uncomment after testing
```

Store the concatenated string in a variable named euroFilename ❶. Then, pass the original filename in amerFilename and the new euroFilename variable to the shutil.move() function to rename the file ❸.

This program has the shutil.move() call commented out and instead prints the filenames that will be renamed ❷. Running the program like this first can let you double-check that the files are renamed correctly. Then you can uncomment the shutil.move() call and run the program again to actually rename the files.

Ideas for Similar Programs

There are many other reasons you might want to rename a large number of files.

- To add a prefix to the start of the filename, such as adding *spam_* to rename *eggs.txt* to *spam_eggs.txt*
- To change filenames with European-style dates to American-style dates
- To remove the zeros from files such as *spam0042.txt*

PROJECT: BACKING UP A FOLDER INTO A ZIP FILE

Say you're working on a project whose files you keep in a folder named *C:\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd like to keep different versions, so you want the ZIP file's filename to increment each time it is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*, *AlsPythonBook_3.zip*, and so on. You could do this by hand, but it is rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backupToZip.py*.

Step 1: Figure Out the ZIP File's Name

The code for this program will be placed into a function named backupToZip(). This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

❶ import zipfile, os

def backupToZip(folder):
    # Back up the entire contents of "folder" into a ZIP file.
```

```
    folder = os.path.abspath(folder)   # make sure folder is absolute

    # Figure out the filename this code should use based on
    # what files already exist.
❷   number = 1
❸   while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

❹   # TODO: Create the ZIP file.

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Do the basics first: add the shebang (#!) line, describe what the program does, and import the zipfile and os modules ❶.

Define a backupToZip() function that takes just one parameter, folder. This parameter is a string path to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create; then the function will create the file, walk the folder folder, and add each of the subfolders and files to the ZIP file. Write TODO comments for these steps in the source code to remind yourself to do them later ❹.

The first part, naming the ZIP file, uses the base name of the absolute path of folder. If the folder being backed up is *C:\delicious*, the ZIP file's name should be *delicious_N.zip*, where *N* = 1 is the first time you run the program, *N* = 2 is the second time, and so on.

You can determine what *N* should be by checking whether *delicious_1.zip* already exists, then checking whether *delicious_2.zip* already exists, and so on. Use a variable named number for *N* ❷, and keep incrementing it inside the loop that calls os.path.exists() to check whether the file exists ❸. The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new zip.

Step 2: Create the New ZIP File

Next let's create the ZIP file. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--
    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1
```

```
    # Create the ZIP file.
    print(f'Creating {zipFilename}...')
❶   backupZip = zipfile.ZipFile(zipFilename, 'w')

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backupToZip('C:\\delicious')
```

Now that the new ZIP file's name is stored in the zipFilename variable, you can call zipfile.ZipFile() to actually create the ZIP file ❶. Be sure to pass 'w' as the second argument so that the ZIP file is opened in write mode.

Step 3: Walk the Directory Tree and Add to the ZIP File

Now you need to use the os.walk() function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
#! python3
# backupToZip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments.

--snip--

    # Walk the entire folder tree and compress the files in each folder.
❶   for foldername, subfolders, filenames in os.walk(folder):
        print(f'Adding files in {foldername}...')
        # Add the current folder to the ZIP file.
❷       backupZip.write(foldername)

        # Add all the files in this folder to the ZIP file.
❸       for filename in filenames:
            newBase = os.path.basename(folder) + '_'
            if filename.startswith(newBase) and filename.endswith('.zip'):
                continue   # don't back up the backup ZIP files
            backupZip.write(os.path.join(foldername, filename))
    backupZip.close()
    print('Done.')

backupToZip('C:\\delicious')
```

You can use os.walk() in a for loop ❶, and on each iteration it will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder.

In the for loop, the folder is added to the ZIP file ❷. The nested for loop can go through each filename in the filenames list ❸. Each of these is added to the ZIP file, except for previously made backup ZIPs.

When you run this program, it will produce output that will look something like this:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```

The second time you run it, it will put all the files in *C:\delicious* into a ZIP file named *delicious_2.zip*, and so on.

Ideas for Similar Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you can write programs that do the following:

- Walk a directory tree and archive just files with certain extensions, such as *.txt* or *.py*, and nothing else.
- Walk a directory tree and archive every file except the *.txt* and *.py* ones.
- Find the folder in a directory tree that has the greatest number of files or the folder that uses the most disk space.

SUMMARY

Even if you are an experienced computer user, you probably handle files manually with the mouse and keyboard. Modern file explorers make it easy to work with a few files. But sometimes you'll need to perform a task that would take hours using your computer's file explorer.

The os and shutil modules offer functions for copying, moving, renaming, and deleting files. When deleting files, you might want to use the send2trash module to move files to the recycle bin or trash rather than permanently deleting them. And when writing programs that handle files, it's a good idea to comment out the code that does the actual copy/move/rename/delete and add a print() call instead so you can run the program and verify exactly what it will do.

Often you will need to perform these operations not only on files in one folder but also on every folder in that folder, every folder in those folders, and so on. The os.walk() function handles this trek across the folders for you so that you can concentrate on what your program needs to do with the files in them.

The zipfile module gives you a way of compressing and extracting files in *.ZIP* archives through Python. Combined with the file-handling functions of os and shutil, zipfile makes it easy to package up several files from anywhere on your hard drive. These *.ZIP* files are much easier to upload to websites or send as email attachments than many separate files.

Previous chapters of this book have provided source code for you to copy. But when you write your own programs, they probably won't come out perfectly the first time. The next chapter focuses on some Python modules that will help you analyze and debug your programs so that you can quickly get them working correctly.