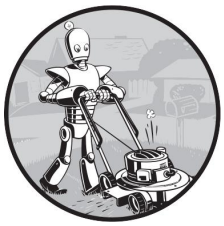
11

ORGANIZING FILES



In addition to creating and writing to new files, your programs can 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 serve as 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 that never makes mistakes.

While Windows uses backslashes (\) to separate folders in a filepath, the Python code in this chapter will use forward slashes (/) instead, as they work on all operating systems.

The shutil Module

The `shutil` module has functions to let you copy, move, rename, and delete files in your Python programs. (The module's name is short for shell utilities, where *shell* is another term for a terminal command line.) To use the `shutil` functions, you'll first need to run `import shutil`.

To create an example file and folder to work with, run the following code before the interactive shell examples in this chapter:

```
>>> from pathlib import Path
>>> h = Path.home()
>>> (h / 'spam').mkdir(exist_ok=True)
>>> with open(h / 'spam/file1.txt', 'w',
encoding='utf-8') as file:
...     file.write('Hello')
...
```

This will create a folder named *spam* with a text file named *file1.txt*. The examples in this chapter will copy, move, rename, and delete this file and folder. All `shutil` functions can take filepath arguments that are either strings or `Path` objects.

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. If *`destination`* is a folder, the file will be copied to that folder with its original name. This function returns the path of the copied file.

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

```
>>> import shutil
>>> from pathlib import Path
>>> h = Path.home()
❶ >>> shutil.copy(h / 'spam/file1.txt', h)
'C:\\Users\\Al\\file1.txt'
❷ >>> shutil.copy(h / 'spam/file1.txt', h /
'spam/file2.txt')
WindowsPath('C:/Users/Al/spam/file2.txt')
```

The first `shutil.copy()` call copies the file at *C:\Users\Al\spam\file1.txt* to the home folder *C:\Users\Al*. The return value is the path of the newly copied file. Note that since we specified a

folder as the destination ❶, the new, copied file will have the same filename as the original *file1.txt* file. The second `shutil.copy()` call ❷ copies the file at *C:\Users\Al\spam\file1.txt* to the *C:\Users\Al\spam* folder but gives the copied file the name *file2.txt*.

While `shutil.copy()` will copy a single file, 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 function returns the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil
>>> from pathlib import Path
>>> h = Path.home()
>>> shutil.copytree(h / 'spam', h /
'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 return a string of the new location's absolute path.

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
>>> from pathlib import Path
>>> h = Path.home()
>>> (h / 'spam2').mkdir()
>>> shutil.move(h / 'spam/file1.txt', h /
'spam2')
'C:\\Users\\Al\\spam2\\file1.txt'
```

After creating the *spam2* folder in the home folder, this `shutil.move()` call says, "Move *C:\Users\Al\spam\file1.txt* into the folder *C:\Users\Al\spam2*." If there had been a *file1.txt* file already in *C:\Users\Al\spam2*, Python would have overwritten it.

If the *`destination`* path is not an existing folder, `shutil.move()` will use this path to rename the file. In the following example, the *`source`* file is moved *and* renamed:

```
>>> shutil.move(h / 'spam/file1.txt', h /
'spam2/new_name.txt')
'C:\\Users\\Al\\spam2\\new_name.txt'
```

This line says, "Move *C:\Users\Al\spam\file1.txt* into the folder *C:\Users\Al\spam2*, and while you're at it, rename that *file1.txt* file to *new_name.txt*."

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 `shutil.rmtree(`*`path`*`)` will delete (that is, remove) the entire folder tree at *`path`*, including all the files and subfolders it contains.
- Calling `os.unlink(`*`path`*`)` will delete the single file at *`path`*.
- Calling `os.rmdir(`*`path`*`)` will delete the folder at *`path`*. This folder must be empty.

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 `print()` calls added to show the files that would be deleted. This is called a *dry run*. Here is a Python program that was intended to delete files with the *.txt* file extension, but it has a typo (shown 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('Deleting', filename)
```

The `os.unlink()` call is now commented, so Python ignores it. Instead, you'll 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.

You should also do dry runs for programs that copy, rename, or move files. Lastly, it may be a good idea to create a backup copy of the entire folder of any files your program touches, just in case you need to completely restore the original files. Once you're 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.

Deleting to the Recycle Bin

Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders. This makes the function dangerous to use, because a bug could delete files you didn't intend. A much better way to delete files and folders is with the third-party `send2trash` module. (See Appendix A for a more in-depth explanation of how to install third-party packages.)

Using the `send2trash` module's `send2trash()` function is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycling bin instead of permanently deleting them. If a bug in your program deletes something with `send2trash` that 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 to send the file *file1.txt* to the recycle bin:

```
>>> import send2trash
>>> send2trash.send2trash('file1.txt')
```

In general, you should 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. Note that the `send2trash()` function can only send files to the recycle bin; it cannot pull files out of it.

Walking a Directory Tree

If you want to list all the files and subfolders in a folder, call the `os.listdir()` function and pass it a folder name:

```
>>> import os
>>> os.listdir(r'C:\Users\Al')
['.anaconda', '.android', '.cache',
'.dotnet', '.eclipse', '.gitconfig',
--snip--
'__pycache__']
```

You can also get a list of `Path` objects in a folder by calling the `iterdir()` method:

```
>>> from pathlib import Path
>>> home = Path.home()
>>> list(home.iterdir())
[WindowsPath('C:/Users/Al/.anaconda'),
WindowsPath('C:/Users/Al/.android'),
WindowsPath('C:/Users/Al/.cache'),
--snip--
WindowsPath('C:/Users/Al/__pycache__')]
```

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, accessing each file as you go. Writing a program to do this could get tricky; fortunately, Python provides the `os.walk()` function to handle this process for you.

Let's create a series of folders and files by running the following code in the interactive shell:

```
>>> from pathlib import Path
>>> h = Path.home()
>>> (h / 'spam').mkdir(exist_ok=True)
>>> (h / 'spam/eggs').mkdir(exist_ok=True)
>>> (h / 'spam/eggs2').mkdir(exist_ok=True)
>>> (h / 'spam/eggs/
bacon').mkdir(exist_ok=True)
>>> for f in ['spam/file1.txt', 'spam/eggs/
file2.txt', 'spam/eggs/file3.txt',
'spam/eggs/bacon/file4.txt']:
...     with open(h / f, 'w',
encoding='utf-8') as file:
...         file.write('Hello')
...
>>> # At this point, the folders and files
now exist.
```

This code will create the following folders and files in your home folder:

- The *spam* folder
- The *spam/file1.txt* file
- The *spam/eggs* folder
- The *spam/eggs/file2.txt* file
- The *spam/eggs/file3.txt* file
- The *spam/eggs2* folder
- The *spam/eggs/bacon* folder
- The *spam/eggs/bacon/file4.txt* file

Here is an example program that uses the `os.walk()` function on this tree of folders and renames each file to uppercase letters:

```
import os, shutil
from pathlib import Path
h = Path.home()

for folder_name, subfolders, filenames in
os.walk(h / 'spam'):
    print('The current folder is ' +
folder_name)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folder_name +
': ' + subfolder)

    for filename in filenames:
        print('FILE INSIDE ' + folder_name +
': '+ filename)
        # Rename file to uppercase:
        p = Path(folder_name)
        shutil.move(p / filename, p /
filename.upper())

    print('')
```

The `os.walk()` function gets passed a single string value: the path of a folder. You can use `os.walk()` in a `for` loop 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 subfolders in the current folder
- A list of strings of the files in the current folder

The *current folder* here refers to the folder accessed in the current iteration of the `for` loop. The `os.walk()` function doesn't change the current working directory of the program. Just as 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 always use the descriptive names `folder_name`, `subfolders`, and `filenames`.

When I ran this program on my computer, it gave the following output:

```
The current folder is C:\Users\Al\spam
SUBFOLDER OF C:\Users\Al\spam: eggs
SUBFOLDER OF C:\Users\Al\spam: eggs2
FILE INSIDE C:\Users\Al\spam: file1.txt

The current folder is C:\Users\Al\spam\eggs
SUBFOLDER OF C:\Users\Al\spam\eggs: bacon
FILE INSIDE C:\Users\Al\spam\eggs: file2.txt
FILE INSIDE C:\Users\Al\spam\eggs: file3.txt

The current folder is C:
\Users\Al\spam\eggs\bacon
FILE INSIDE C:\Users\Al\spam\eggs\bacon:
file4.txt

The current folder is C:\Users\Al\spam\eggs2
```

Because `os.walk()` returns lists of strings for the `subfolder` and `filename` variables, you can use the return values in their own `for` loops. For instance, you can pass the folder and filename to functions like `shutil.move()`, as in the example.

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 or extract from ZIP files using functions in the `zipfile` module.

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. (Note the capital letters *Z* and *F* in the object name, which differs from the `zipfile` module name.) This process is similar to opening a text file in write mode by passing `'w'` to the `open()` function. For the filename, you can pass either a string or a `Path` object.

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 set this value to `zipfile.ZIP_DEFLATED` to specify the *deflate* compression algorithm, which works well on all types of data. If you don't pass this value, the `write()` method adds the file to the ZIP file with its regular, uncompressed size. Enter the following into the interactive shell:

```
>>> import zipfile
>>> with open('file1.txt', 'w',
encoding='utf-8') as file_obj:
...     file_obj.write('Hello' * 10000)
...
>>> with zipfile.ZipFile('example.zip', 'w')
as example_zip:
...     example_zip.write('file1.txt',
compress_type=zipfile.ZIP_DEFLATED,
        compresslevel=9)
```

This code creates a text file named *file1.txt* and writes to it the 50,000- character string `'Hello' * 10000` (about 49KB). Then, it creates a new ZIP file named *example.zip* that has the compressed contents of *file1.txt* (about 213 bytes; highly repetitive data is also highly compressible). The `compresslevel` keyword argument (added in Python 3.7 and later) can be set to any value from `0` to `9`, with

9 being the slowest but most compressed level. If you don't specify this keyword argument, the default is 6.

The zipfile.ZipFile() function opens a ZIP file in a with statement, in a manner similar to how the open() function opens files. This ensures that the close() method is automatically called when the execution leaves the with statement's block.

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*.

Reading ZIP Files

To read the contents of a ZIP file, you must first create a ZipFile object by calling the zipfile.ZipFile() function and passing 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

>>> example_zip =
zipfile.ZipFile('example.zip')
>>> example_zip.namelist()
['file1.txt']
>>> file1_info =
example_zip.getinfo('file1.txt')
>>> file1_info.file_size
50000
>>> file1_info.compress_size
97
❶ >>> f'Compressed file is
{round(file1_info.file_size / file1_info
    .compress_size, 2)}x smaller!'

'Compressed file is 515.46x smaller!'
>>> example_zip.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`, which hold integers representing the original file size and compressed file size, respectively, in bytes. 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, then 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. Create a ZIP file named *example.zip* by following the instructions in "Creating and Adding to ZIP Files" on page 249, and then enter the following into the interactive shell:

```
>>> import zipfile
>>> example_zip =
zipfile.ZipFile('example.zip')
❶ >>> example_zip.extractall()
>>> example_zip.close()
```

After running this code, Python will extract the contents of *example.zip* to the current working directory. 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 doesn't exist, Python will create it. For instance, if you replaced the call at ❶ with `example_zip.extractall('C:\\spam)`, the code would extract the files from *example.zip* into a newly created *C:\spam* folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example by entering the following:

```
>>> example_zip.extract('file1.txt')
'C:\\Users\\Al\\Desktop\\file1.txt'
```

```
>>> example_zip.extract('file1.txt', 'C:\
\some\\new\\folders')
'C:\\some\\new\\folders\\file1.txt'
>>> example_zip.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.

Project 5: Back Up a Folder into a ZIP File

Say you're working on a project whose files you keep in a folder named *C:\Users\Al\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd also like to keep different versions of these snapshots, so you want the ZIP file's filename to increment each time a new version is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*, *AlsPythonBook_3.zip*, and so on. You could do this by hand, but that would be 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 *backup_to_zip.py*.

Step 1: Figure Out the ZIP File's Name

We'll place the code for this program into a function named `backup_to_zip()`. 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:

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

import zipfile, os
```

```
from pathlib import Path

def backup_to_zip(folder):
    # Back up the entire contents of "folder"
into a ZIP file.
    folder = Path(folder)  # Make sure folder
is a Path object, not string.

    # Figure out the ZIP filename this code
should use, based on
    # what files already exist.
  ❶ number = 1
  ❷ while True:

        zip_filename = Path(folder.parts[-1]
+ '_' + str(number) + '.zip')
        if not zip_filename.exists():
            break
        number = number + 1

  ❸ # TODO: Create the ZIP file.

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

backup_to_zip(Path.home() / 'spam')
```

First, import the `zipfile` and `os` modules. Next, define a `backup_to_zip()` function that takes just one parameter, `folder`. This parameter is a string or `Path` object 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, it 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 ❸.

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

You can determine what *N* should be by checking whether *spam_1.zip* already exists, then checking whether *spam_2.zip* already exists, and so on. Use a variable named `number` for *N* ❶, and keep incrementing it inside the loop that calls `exists()` to check whether the file exists ❷. 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:

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

--snip--

    # Create the ZIP file.
    print(f'Creating {zip_filename}...')
    backup_zip =
zipfile.ZipFile(zip_filename, 'w')

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

backup_to_zip(Path.home() / 'spam')
```

Now that the new ZIP file's name is stored in the `zip_filename` variable, you can call `zipfile.ZipFile()` to actually create the ZIP file. Be sure to pass `'w'` as the second argument to open the ZIP

file in write mode. We'll also remove the TODO from the comment, as we've finished writing the code for this section.

Step 3: Walk the Directory Tree

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:

```
# backup_to_zip.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 folder_name, subfolders, filenames in
os.walk(folder):
        folder_name = Path(folder_name)
        print(f'Adding files in folder
{folder_name}...')


# Add all the files in this folder to the ZIP
file.
      ❷ for filename in filenames:
            print(f'Adding file
{filename}...')
            backup_zip.write(folder_name /
filename)
    backup_zip.close()
    print('Done.')
backup_to_zip(Path.home() / 'spam')
```

Use `os.walk()` in a `for` loop ❶. On each iteration, the function will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder. The nested `for` loop can go through each filename in the `filenames` list ❷. Each of these is added to the ZIP file.

When you run this program, it should produce output that looks something like this:

```
Creating spam_1.zip...
Adding files in spam...
Adding file file1.txt...
Done.
```

The second time you run it, it will put all the files in the *spam* folder into a ZIP file named *spam_2.zip*, and so on.

Ideas for Other Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you could 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.
- Only archive the folders in a directory tree that use the most disk space or have been modified since the previous archive.

Summary

Even if you're 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 the 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 do a dry run; comment out the code that does the actual copy, move, rename, or delete, and replace it with a `print()` call. This way, you can run the program and verify exactly what it will do.

Often, you'll need to perform these operations not only on files in one folder, but also on every subfolder in that folder, every subfolder in those subfolders, 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 to compress and extract 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.

Practice Questions

1. What is the difference between `shutil.copy()` and `shutil.copytree()`?
2. What function is used to rename files?
3. What is the difference between the delete functions in the `send2trash` and `shutil` modules?
4. `ZipFile` objects have a `close()` method just like `File` objects' `close()` method. What `ZipFile` method is equivalent to `File` objects' `open()` method?

Practice Programs

For practice, write programs to do the following tasks.

Selectively Copying

Write a program that walks through a folder tree and searches for files with a certain file extension (such as *.pdf* or *.jpg*). Copy these files from their current location to a new folder.

Deleting Unneeded Files

It's not uncommon for a few unneeded but humongous files or folders to take up the bulk of the space on your hard drive. If you're trying to free up room on your computer, it's more effective to identify the largest unneeded files first.

Write a program that walks through a folder tree and searches for exceptionally large files or folders—say, ones that have a file size of more than 100MB. (Remember that, to get a file's size, you can use `os.path.getsize()` from the `os` module.) Print these files with their absolute path to the screen.

Renumbering Files

Write a program that finds all files with a given prefix, such as *spam001.txt*, *spam002.txt*, and so on, in a single folder and locates any gaps in the numbering (such as if there is a *spam001.txt* and a *spam003.txt* but no *spam002.txt*). Have the program rename all the later files to close this gap.

To create these example files (skipping *spam042.txt*, *spam086.txt*, and *spam103.txt*), run the following code:

```
>>> for i in range(1, 121):
...     if i not in (42, 86, 103):
...         with
open(f'spam{str(i).zfill(3)}.txt', 'w') as
file:
...             pass
...
```

As an added challenge, write another program that can insert gaps into numbered files (and bump up the numbers in the filenames after the gap) so that a new file can be inserted.

Converting Dates from American- to European-Style

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! Instead, write a program that does the following:

1. Searches all filenames in the current working directory and all subdirectories for American-style dates. Use the `os.walk()` function to go through the subfolders.
2. Uses regular expressions to identify filenames with the MM-DD-YYYY pattern in them—for example, *spam12-31-1900.txt*. Assume the months and days always use two digits, and that files with non-date matches don't exist. (You won't find files named something like *99-99-9999.txt*.)
3. When a filename is found, renames the file with the month and day swapped to make it European-style. Use the `shutil.move()` function to do the renaming.