Using the filesystem

This lab covers

- [Managing paths and pathnames]
- [Getting information about files]
- [Performing filesystem operations]
- [Processing all files in a directory subtree]

Working with files involves one of two things: basic I/O (described in [chapter 13], *"*Reading and writing files") and working with the filesystem (for example, naming, creating, moving, or referring to files), which is a bit tricky, because different operating systems have different filesystem conventions.

It would be easy enough to learn how to perform basic file I/O without learning all the features Python has provided to simplify cross-platform filesystem interaction---but I wouldn't recommend it. Instead, read at least the first part of this lab, which gives you the tools you need to refer to files in a manner that doesn't depend on your particular operating system. Then, when you use the basic I/O operations, you can open the relevant files in this manner.

os and os.path vs. pathlib

The traditional way that file paths and filesystem operations have been handled in Python is by using functions included in the [os] and [os.path] modules. These functions have worked well enough but often resulted in more verbose code than necessary. Since Python 3.5, a new library, [pathlib], has been added; it offers a more object-oriented and more unified way of doing the same operations. Because a lot of code out there still uses the older style, I've retained those examples and their explanations. On the other hand, [pathlib] has a lot going for it and is likely to become the new standard, so after each example of the old method, I include an example (and brief explanation, where necessary) of how the same thing would be done with [pathlib].

Paths and pathnames

All operating systems refer to files and directories with strings naming a given file or directory. Strings used in this manner are usually called *pathnames* (or sometimes just *paths*), which is the word I'll use for them. The fact that pathnames are strings introduces possible complications into working with them. Python does a good job of providing functions that help avert these complications; but to use these Python functions effectively, you need to understand the underlying problems. This section discusses these details.

Pathname semantics across operating systems are very similar because the filesystem on almost all operating systems is modeled as a tree structure, with a disk being the root and folders, subfolders, and so on being branches, subbranches, and so on. This means that most operating systems refer to a specific file in fundamentally the same manner: with a pathname that specifies the path to follow from the root of the filesystem tree (the disk) to the file in question. (This characterization of the root corresponding to a hard disk is an oversimplification, but it's close enough to the truth to serve for this chapter.) This pathname consists of a series of folders to descend into to get to the desired file.

Different operating systems have different conventions regarding the precise syntax of pathnames. The character used to separate sequential file or directory names in a Linux/UNIX pathname is /, whereas the character used to separate file or directory names in a Windows pathname is \. In addition, the UNIX filesystem has a single root (which is referred to by having a / character as the first character in a pathname), whereas the Windows filesystem has a separate root for each drive, labeled [A:\], [B:\], [C:\], and so forth (with C: usually being the main drive). Because of these differences, files have different pathname representations on different operating systems. A file called [C:\data\myfile] in MS Windows might be called [/data/myfile] on UNIX and on the Mac OS. Python provides functions and constants that allow you to perform common pathname manipulations without worrying about such syntactic details. With a little care, you can write your Python programs in such a manner that they'll run correctly no matter what the underlying filesystem happens to be.

Absolute and relative paths

These operating systems allow two types of pathnames:

- [*Absolute* pathnames specify the exact location of a file in a filesystem without any ambiguity; they do this by listing the entire path to that file, starting from the root of the filesystem.]
- [*Relative* pathnames specify the position of a file relative to some other point in the filesystem, and that other point isn't specified in the relative pathname itself; instead, the absolute starting point for relative pathnames is provided by the context in which they're used.]

As examples, here are two Windows absolute pathnames:

```
C:\Program Files\Doom
D:\backup\June
```

and here are two Linux absolute pathnames and a Mac absolute pathname:

```
/bin/Doom
/floppy/backup/June
/Applications/Utilities
```

and here are two Windows relative pathnames:

```
mydata\project1\readme.txt
games\tetris
```

and these are Linux/UNIX/Mac relative pathnames:

```
mydata/project1/readme.txt
games/tetris
Utilities/Java
```

Relative paths need context to anchor them. This context is typically provided in one of two ways.

The simpler way is to append the relative path to an existing absolute path, producing a new absolute path. You might have a relative Windows path, [Start Menu\Programs\Startup], and an absolute path, [C:\Users\Administrator]. By appending the two, you have a new absolute path: [C:\Users\Administrator\Start Menu\Programs\Startup], which refers to a specific location in the filesystem. By appending the same relative path to a different absolute path (say, [C:\Users\myuser]), you produce a path that refers to the Startup folder in a different user's (myuser's) Profiles directory.

The second way in which relative paths may obtain a context is via an implicit reference to the *current working directory*, which is the particular directory where a Python program considers itself to be at any point during its execution. Python commands may implicitly make use of the current working directory when they're given a relative path as an argument. If you use the [os.listdir(path)] command with a relative path argument, for example, the anchor for that relative path is the current working directory, and the result of the command is a list of the filenames in the directory whose path is formed by appending the current working directory with the relative path argument.

The current working directory

Whenever you edit a document on a computer, you have a concept of where you are in that computer's file structure because you're in the same directory (folder) as the file you're working on. Similarly, whenever Python is running, it has a concept of where in the directory structure it is at any moment. This fact is important because the program may ask for a list of files stored in the current directory. The directory that a Python program is in is called the *current working directory* for that program. This directory may be different from the directory the program resides in.

To see this in action, start Python and use the [os.getcwd] (get current working directory) command to find Python's initial current working directory:

```
>>> import os
>>> os.getcwd()
```

Note that [os.getcwd] is used as a zero-argument function call, to emphasize the fact that the value it returns isn't a constant but will change as you issue commands that alter the value of the current working directory. (That directory probably will be either the directory the Python program itself resides in or the directory you were in when you started Python. On a Linux machine, the result is /home/myuser, which is the home directory.) On Windows machines, you'll see extra backslashes inserted into the path because Windows uses \ as its path separator, and in Python strings (as discussed in [section 6.3.1]), \ has a special meaning unless it's itself backslashed.

Now type

```
>>> os.listdir(os.curdir)
```

The constant [os.curdir] returns whatever string your system happens to use as the same directory indicator. On both UNIX and Windows, the current directory is represented as a single dot, but to keep your programs portable, you should always use [os.curdir] instead of typing just the dot. This string is a relative path, meaning that [os.listdir] appends it to the path for the current working directory, giving the same path. This command returns a list of all the files or folders inside the current working directory. Choose some folder name, and type

```
>>> os.chdir(folder name)
>>> os.getcwd()
```

As you can see, Python moves into the folder specified as an argument of the [os.chdir] function. Another call to [os.listdir(os.curdir)] would return a list of files in [folder], because [os.curdir] would then be taken relative to the new current working directory. Many Python filesystem operations use the current working directory in this manner.

Accessing directories with pathlib

To get the current directory with [pathlib], you could do the following:

```
>>> import pathlib
>>> cur_path = pathlib.Path()
>>> cur_path.cwd()
PosixPath('/home/naomi')
```

There's no way for [pathlib] to change the current directory in the way that [os.chdir()] does (see the preceding section), but you could work with a new folder by creating a new path object, as discussed in [section 12.2.5], " [Manipulating pathnames with pathlib]."

Manipulating pathnames

Now that you have the background to understand file and directory pathnames, it's time to look at the facilities Python provides for manipulating these pathnames. These facilities consist of several functions and constants in the [os.path] submodule, which you can use to manipulate paths without explicitly using any operating-system-specific syntax. Paths are still represented as strings, but you need never think of them or manipulate them as such.

To start, construct a few pathnames on different operating systems, using the [os.path.join] function. Note that importing [os] is sufficient to bring in the [os.path] submodule also; there's no need for an explicit [import os.path] statement.

First, start Python under Windows:

```
>>> import os
>>> print(os.path.join('bin', 'utils', 'disktools'))
bin\utils\disktools
```

The [os.path.join] function interprets its arguments as a series of directory names or filenames, which are to be joined to form a single string understandable as a relative path by the underlying operating system. In a Windows system, that means path component names should be joined with backslashes, which is what was produced.

Now try the same thing in UNIX:

```
>>> import os
>>> print(os.path.join('bin', 'utils', 'disktools'))
bin/utils/disktools
```

The result is the same path, but using the Linux/UNIX convention of forward slash separators rather than the Windows convention of backslash separators. In other words, [os.path.join] lets you form file paths from a sequence of directory or filenames without any worry about the conventions of the underlying operating system. [os.path.join] is the fundamental way by which file paths may be built in a manner that doesn't constrain the operating systems on which your program will run.

The arguments to [os.path.join] need not be a single directory or filename; they may also be subpaths that are then joined to make a longer pathname. The following example illustrates this in the Windows environment and is also a case in which you'd find it necessary to use double backslashes in your strings. Note that you could enter the pathname with forward slashes (/) as well, because Python converts them before accessing the Windows operating system:

```
>>> import os
>>> print(os.path.join('mydir\\bin', 'utils\\disktools\\chkdisk'))
mydir\bin\utils\disktools\chkdisk
```

If you always use [os.path.join] to build up your paths, of course, you'll rarely need to worry about this situation. To write this example in a portable manner, you should enter

```
>>> path1 = os.path.join('mydir', 'bin');
>>> path2 = os.path.join('utils', 'disktools', 'chkdisk')
>>> print(os.path.join(path1, path2))
mydir\bin\utils\disktools\chkdisk
```

The [os.path.join] command also has some understanding of absolute versus relative pathnames. In Linux/UNIX, an *absolute* path always begins with a / (because a single slash denotes the topmost directory of the entire system, which contains everything else, including the various floppy and CD drives that might be available). A *relative* path in UNIX is any legal path that does *not* begin with a slash. Under any of the Windows operating systems, the situation is more complicated because the way in which Windows handles relative and absolute paths is messier. Rather than go into all of the details, I'll just say that the best way to handle this situation is to work with the following simplified rules for Windows paths:

- [A pathname beginning with a drive letter followed by a colon and a backslash and then a path is an absolute path: C:\Program Files\Doom. (Note that C: by itself, without a trailing backslash, can't reliably be used to refer to the top-level directory on the C: drive. You must use C:\ to refer to the top-level directory on C:. This requirement is a result of DOS conventions, not Python design.)]
- [A pathname beginning with neither a drive letter nor a backslash is a relative path: [mydirectory\letters\business].]
- [A pathname beginning with \\ followed by the name of a server is the path to a network resource.]

- [Anything else can be considered to be an invalid pathname.^[[1]]^] Microsoft Windows allows some other constructs, but it's probably best to stick to the given definitions.

Regardless of the operating system used, the [os.path.join] command doesn't perform sanity checks on the names it's constructing. It's possible to construct pathnames containing characters that, according to your OS, are forbidden in pathnames. If such checks are a requirement, probably the best solution is to write a small path-validity-checker function yourself.

The [os.path.split] command returns a two-element tuple splitting the basename of a path (the single file or directory name at the end of the path) from the rest of the path. You might use this example on a Windows system:

```
>>> import os
>>> print(os.path.split(os.path.join('some', 'directory', 'path')))
('some\\directory', 'path')
```

The [os.path.basename] function returns only the basename of the path, and the [os.path.dirname] function returns the path up to but not including the last name, as in this example:

```
>>> import os
>>> os.path.basename(os.path.join('some', 'directory', 'path.jpg'))
'path.jpg'
>>> os.path.dirname(os.path.join('some', 'directory', 'path.jpg'))
'some\\directory'
```

To handle the dotted extension notation used by most filesystems to indicate file type (the Macintosh is a notable exception), Python provides [os.path.splitext]:

```
>>> os.path.splitext(os.path.join('some', 'directory', 'path.jpg'))
('some/directory/path', '.jpg')
```

The last element of the returned tuple contains the dotted extension of the indicated file (if there was a dotted extension). The first element of the returned tuple contains everything from the original argument except the dotted extension.

You can also use more specialized functions to manipulate pathnames. [os.path.commonprefix(path1, path2, ...)] finds the common prefix (if any) for a set of paths. This technique is useful if you want to find the lowest-level directory that contains every file in a set of files. [os.path.expanduser] expands username shortcuts in paths, such as for UNIX. Similarly, [os.path.expandvars] does the same for environment variables. Here's an example on a Windows 10 system:

```
>>> import os
>>> os.path.expandvars('$HOME\\temp')
'C:\\Users\\administrator\\personal\\temp'
```

Manipulating pathnames with pathlib

Just as you did in the preceding section, start by constructing a few pathnames on different operating systems, using the path object's methods.

First, start Python under Windows:

```
>>> from pathlib import Path
>>> cur_path = Path()
>>> print(cur_path.joinpath('bin', 'utils', 'disktools'))
bin\utils\disktools
```

The same result can be achieved by using the slash operator:

```
>>> cur_path / 'bin' / 'utils' / 'disktools'
WindowsPath('bin/utils/disktools')
```

Note that in the representation of the path object, forward slashes are always used, but Windows Path objects have the forward slashes converted to backslashes as required by the OS. So if you try the same thing in UNIX:

```
>>> cur_path = Path()
>>> print(cur_path.joinpath('bin', 'utils', 'disktools'))
bin/utils/disktools
```

The [parts] property returns a tuple of all the components of a path. You might use this example on a Windows system:

```
>>> a_path = WindowsPath('bin/utils/disktools')
>>> print(a_path.parts)
('bin', 'utils', 'disktools')
```

The [name] property returns only the basename of the path, the [parent] property returns the path up to but not including the last name, and the [suffix] property handles the dotted extension notation used by most filesystems to indicate file type (but the Macintosh is a notable exception). Here's an example:

```
>>> a_path = Path('some', 'directory', 'path.jpg')
>>> a_path.name
'path.jpg'
>>> print(a_path.parent)
some\directory
>>> a_path.suffix
'.jpg'
```

Several other methods associated with [Path] objects allow flexible manipulation of both pathnames and files themselves, so you should review the documentation of the [pathlib] module. It's likely that the [pathlib] module will make your life easier and your file-handling code more concise.

Useful constants and functions

You can access several useful path-related constants and functions to make your Python code more system-independent than it otherwise would be. The most basic of these constants are [os.curdir] and [os.pardir], which respectively define the symbol used by the operating system for the directory and parent directory path indicators. In Windows as well as Linux/UNIX and macOS, these indicators are [.] and [..] respectively, and they can be used as normal path elements. This example

```
os.path.isdir(os.path.join(path, os.pardir, os.curdir))
```

asks whether the parent of the parent of [path] is a directory. [os.curdir] is particularly useful for requesting commands on the current working directory. This example

```
os.listdir(os.curdir)
```

returns a list of filenames in the current working directory (because [os.curdir] is a relative path, and [os.listdir] always takes relative paths as being relative to the current working directory).

The [os.name] constant returns the name of the Python module imported to handle the operating system--specific details. Here's an example on my Windows XP system:

```
>>> import os
>>> os.name
'nt'
```

Note that [os.name] returns ['nt'] even though the actual version of Windows could be Windows 10. Most versions of Windows, except for Windows CE, are identified as ['nt'].

On a Mac running OS X and on Linux/UNIX, the response is [posix]. You can use this response to perform special operations, depending on the platform you're working on:

```
import os
if os.name == 'posix':
    root_dir = "/"
elif os.name == 'nt':
    root_dir = "C:\\"
else:
    print("Don't understand this operating system!")
```

You may also see programs use [sys.platform], which gives more exact information. On Windows 10, [sys.platform] is set to [win32]---even if the machine is running the 64-bit version of the operating system. On Linux, you may see [linux2], whereas on Solaris, it may be set to [sunos5] depending on the version you're running.

All your environment variables and the values associated with them are available in a dictionary called [os.environ]. On most operating systems, this directory includes variables related to paths---typically, search paths for binaries and so forth. If what you're doing requires this directory, you know where to find it now.

At this point, you've received an introduction to the major aspects of working with pathnames in Python. If your immediate need is to open files for reading or writing, you can jump directly to the next chapter. Continue reading for further information about pathnames, testing what they point to, useful constants, and so forth.

Quick Check: Manipulating paths

How would you use the [os] module's functions to take a path to a file called [test.log] and create a new file path in the same directory for a file called [test.log.old]? How would you do the same thing using the [pathlib] module?

What path would you get if you created a pathlib [Path] object from [os .pardir]? Try it and find out.

Getting information about files

File paths are supposed to indicate actual files and directories on your hard drive. You're probably passing a path around, of course, because you want to know something about what it points to. Various Python functions are available for this purpose.

The most commonly used Python path-information functions are [os.path.exists], [os.path.isfile], and [os.path.isdir], all of which take a single path as an argument. [os.path.exists] returns [True] if its argument is a path corresponding to something that exists in the filesystem. [os.path.isfile] returns [True] if and only if the path it's given indicates a normal data file of some sort (executables fall under this heading), and it returns [False] otherwise, including the possibility that the path argument doesn't indicate anything in the filesystem. [os.path.isdir] returns [True] if and only if its path argument indicates a directory; it returns [False] otherwise. These examples are valid on my system. You may need to use different paths on yours to investigate the behavior of these functions:

```
>>> import os
>>> os.path.exists('C:\\Users\\myuser\\My Documents')
True
>>> os.path.exists('C:\\Users\\myuser\\My Documents\\Letter.doc')
True
```

```
>>> os.path.exists('C:\\Users\\myuser\\\My Documents\\ljsljkflkjs')
False
>>> os.path.isdir('C:\\Users\\myuser\\My Documents')
True
>>>   os.path.isfile('C:\\Users\\ myuser\\My Documents')
False
>>> os.path.isdir('C:\\Users\\ myuser\\My Documents
\\Letter.doc')
False
>>> os.path.isfile('C:\\Users\\ myuser\\My Documents\\Letter.doc')
True
```

Several similar functions provide more specialized queries. [os.path.islink] and [os.path.ismount] are useful in the context of Linux and other UNIX operating systems that provide file links and mount points; they return [True] if, respectively, a path indicates a file that's a link or a mount point. [os.path.islink] does *not* return [True] on Windows shortcuts files (files ending with .lnk), for the simple reason that such files aren't true links. However, [os.path.islink] returns [True] on Windows systems for true symbolic links created with the [mklink()] command. The OS doesn't assign them a special status, and programs can't transparently use them as though they were the actual file. [os.path.samefile(path1, path2)] returns [True] if and only if the two path arguments point to the same file. [os.path.isabs(path)] returns [True] if its argument is an absolute path; it returns [False] otherwise. [os.path.getsize(path)], [os.path.getmtime(path)], and [os.path.getatime(path)] return the size, last modify time, and last access time of a pathname, respectively.

Getting information about files with scandir

In addition to the [os.path] functions listed, you can get more complete information about the files in a directory by using [os.scandir], which returns an iterator of [os.DirEntry] objects. [os.DirEntry] objects expose the file attributes of a directory entry, so using [os.scandir] can be faster and more efficient than combining [os.listdir] (discussed in the next section) with the [os.path] operations. If, for example, you need to know whether the entry refers to a file or directory, [os.scandir]'s ability to access more directory information than just the name will be a plus. [os.DirEntry] objects have methods that correspond to the [os.path] functions mentioned in the previous section, including [exists], [is_dir], [is_file], [is_socket], and [is_symlink].

[os.scandir] also supports a context manager using [with], and using one is recommended to ensure resources are properly disposed of. This example code iterates over all of the entries in a directory and prints both the name of the entry and whether it's a file:

```
>>> with os.scandir(".") as my_dir:
...     for entry in my_dir:
...         print(entry.name, entry.is_file())
...
pip-selfcheck.json True
pyvenv.cfg True
include False
test.py True
lib False
lib64 False
bin False
```

More filesystem operations

In addition to obtaining information about files, Python lets you perform certain filesystem operations directly through a set of basic but highly useful commands in the [os] module.

I describe only those true cross-platform operations in this section. Many operating systems have access to more advanced filesystem functions, and you need to check the main Python library documentation for the details.

You've already seen that to obtain a list of files in a directory, you use [os.listdir]:

```
>>> os.chdir(os.path.join('C:', 'my documents', 'tmp'))
>>> os.listdir(os.curdir)
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp']
```

Note that unlike the list-directory command in many other languages or shells, Python does *not* include the [os.curdir] and [os.pardir] indicators in the list returned by [os.listdir.]

The [glob] function from the [glob] module (named after an old UNIX function that did pattern matching) expands Linux/UNIX shell-style wildcard characters and character sequences in a pathname, returning the files in the current working directory that match. A [*] matches any sequence of characters. A [?] matches any single character. A character sequence ([[h,H]] or [[0-9]]) matches any single character in that sequence:

```
>>> import glob
>>> glob.glob("*")
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp']
>>> glob.glob("*bkp")
['registry.bkp']
>>> glob.glob("?.tmp")
['a.tmp', '1.tmp', '7.tmp', '9.tmp']
>>> glob.glob("[0-9].tmp")
['1.tmp', '7.tmp', '9.tmp']
```

To rename (move) a file or directory, use [os.rename]:

```
>>> os.rename('registry.bkp', 'registry.bkp.old')
>>> os.listdir(os.curdir)
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

You can use this command to move files across directories as well as within directories.

Remove or delete a data file with [os.remove]:

```
>>> os.remove('book1.doc.tmp')
>>> os.listdir(os.curdir)
['a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

Note that you can't use [os.remove] to delete directories. This restriction is a safety feature, to ensure that you don't accidentally delete an entire directory substructure.

Files can be created by writing to them, as discussed in [chapter 11]. To create a directory, use [os.makedirs] or [os.mkdir]. The difference between them is that [os.mkdir] doesn't create any necessary intermediate directories, but [os.makedirs] does:

```
>>> os.makedirs('mydir')
>>> os.listdir(os.curdir)
['mydir', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
>>> os.path.isdir('mydir')
True
```

To remove a directory, use [os.rmdir]. This function removes only empty directories. Attempting to use it on a nonempty directory raises an exception:

```
>>> os.rmdir('mydir')
>>> os.listdir(os.curdir)
['a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

To remove nonempty directories, use the [shutil.rmtree] function. It recursively removes all files in a directory tree. See the Python standard library documentation for details on its use.

More filesystem operations with pathlib

Path objects have most of the same methods mentioned earlier. Some differences exist, however. The [iterdir] method is similar to the [os.path.listdir] function except that it returns an iterator of paths rather than a list of strings:

```
>>> new_path = cur_path.joinpath('C:', 'my documents', 'tmp'))
>>> list(new_path.iterdir())
[WindowsPath('book1.doc.tmp'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'), WindowsPath('registry.bkp')]
```

Note that in a Windows environment, the paths returned are [WindowsPath] objects, whereas on Mac OS or Linux, they're [PosixPath] objects.

[pathlib] path objects also have a [glob] method built in, which again returns not a list of strings but an iterator of path objects. Otherwise, this function behaves very much like the [glob.glob] function demonstrated above:

```
>>> list(cur_path.glob("*"))
[WindowsPath('book1.doc.tmp'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'), WindowsPath('registry.bkp')]
>>> list(cur_path.glob("*bkp"))
[WindowsPath('registry.bkp')]
>>> list(cur_path.glob("?.tmp"))
[WindowsPath('a.tmp'), WindowsPath('1.tmp'), WindowsPath('7.tmp'),
     WindowsPath('9.tmp')]
>>> list(cur_path.glob("[0-9].tmp"))
[WindowsPath('1.tmp'), WindowsPath('7.tmp'), WindowsPath('9.tmp')]
```

To rename (move) a file or directory, use the path object's [rename] method:

```
>>> old_path = Path('registry.bkp')
>>> new_path = Path('registry.bkp.old')
>>> old_path.rename(new_path)
>>> list(cur_path.iterdir())
[WindowsPath('book1.doc.tmp'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'),
     WindowsPath('registry.bkp.old')]
```

You can use this command to move files across directories as well as within directories.

Remove or delete a data file with [unlink]:

```
>>> new_path = Path('book1.doc.tmp')
>>> new_path.unlink()
>>> list(cur_path.iterdir())
```

```
[WindowsPath('a.tmp'), WindowsPath('1.tmp'), WindowsPath('7.tmp'),
     WindowsPath('9.tmp'), WindowsPath('registry.bkp.old')]
```

Note that as with [os.remove], you can't use the [unlink] method to delete directories. This restriction is a safety feature, to ensure that you don't accidentally delete an entire directory substructure.

To create a directory by using a path object, use the path object's [mkdir] method. If you give the [mkdir] method a [parents=True] parameter, it creates any necessary intermediate directories; otherwise, it raises a [FileNotFoundError] if an intermediate directory isn't there:

```
>>> new_path = Path ('mydir')
>>> new_path.mkdir(parents=True)
>>> list(cur_path.iterdir())
[WindowsPath('mydir'), WindowsPath('a.tmp'), WindowsPath('1.tmp'),
     WindowsPath('7.tmp'), WindowsPath('9.tmp'),
     WindowsPath('registry.bkp.old')]]
>>> new_path.is_dir('mydir')
True
```

To remove a directory, use the [rmdir] method. This method removes only empty directories. Attempting to use it on a nonempty directory raises an exception:

```
>>> new_path = Path('mydir')
>>> new_path.rmdir()
>>> list(cur_path.iterdir())
[WindowsPath('a.tmp'), WindowsPath('1.tmp'), WindowsPath('7.tmp'),
     WindowsPath('9.tmp'), WindowsPath('registry.bkp.old']
```

Lab 12: More file operations

How might you calculate the total size of all files ending with .txt that aren't symlinks in a directory? If your first answer was using [os.path], also try it with [pathlib], and vice versa.

Write some code that builds off your solution to move the same .txt files in the lab question to a new subdirectory called backup in the same directory.

Processing all files in a directory subtree

Finally, a highly useful function for traversing recursive directory structures is the [os.walk] function. You can use it to walk through an entire directory tree, returning three things for each directory it traverses: the root, or path, of that directory; a list of its subdirectories; and a list of its files.

[os.walk] is called with the path of the starting, or top, directory and can have three optional arguments: [os.walk(directory], [topdown=True], [onerror=None], [followlinks= False)]. [directory] is a starting directory path; if [topdown] is [True] or not present, the files in each directory are processed *before* its subdirectories, resulting in a listing that starts at the top and goes down; whereas if [topdown] is [False], the subdirectories of each directory are processed *first*, giving a bottom-up traversal of the tree. The [onerror] parameter can be set to a function to handle any errors that result from calls to [os.listdir], which are ignored by default. [os.walk] by default doesn't walk down into folders that are symbolic links unless you give it the [followlinks=True] parameter.

When called, [os.walk] creates an iterator that recursively applies itself to all the directories contained in the [top] parameter. In other words, for each subdirectory [subdir] in [names], [os.walk] recursively invokes a call to itself, of the form [os.walk(subdir, ...)]. Note that if [topdown] is [True] or not given, the list of subdirectories may be modified (using any of the list-modification operators or methods) before its items are used for the next level of recursion; you can use this to control into which---if any---subdirectories [os.walk] will descend.

To get a feel for [os.walk], I recommend iterating over the tree and printing out the values returned for each directory. As an example of the power of [os.walk], list the current working directory and all of its subdirectories along with a count of the number of entries in each of them, excluding any .git directories:

```
import os
for root, dirs, files in os.walk(os.curdir):
    print("{0} has {1} files".format(root, len(files)))
    if ".git" in dirs:
        dirs.remove(".git")
```

This example is complex, and if you want to use [os.walk] to its fullest extent, you should probably play around with it quite a bit to understand the details of what's going on.

The [copytree] function of the [shutil] module recursively makes copies of all the files in a directory and all of its subdirectories, preserving permission mode and stat (that is, access/modify times) information. [shutil] also has the already-mentioned [rmtree] function for removing a directory and all of its subdirectories, as well as several functions for making copies of individual files. See the standard library documentation for details.

Summary

- [Python provides a group of functions and constants that handle filesystem references (pathnames) and filesystem operations in a manner independent of the underlying operating system.]
- [For more advanced and specialized filesystem operations that typically are tied to a certain operating system or systems, look at the main Python documentation for the [os, pathlib,] and [posix] modules.]