Chapter 14. Files and Directories

> *I have files, I have computer files and, you know, files on paper. But most of it is really in my head. So God help me if anything ever happens to my head!*
>
> —George R. R. Martin

When you first start programming, you hear some words over and over but aren't sure whether they have a specific technical meaning or are just handwaving. The terms *file* and *directory* are such words, and they do have actual technical meanings. A *file* is a sequence of bytes, stored in some *filesystem*, and accessed by a *filename*. A *directory* is a collection of files, and possibly other directories. The term *folder* is a synonym for directory. It turned up when computers gained graphical user interfaces, and mimicked office concepts to make things seem more familiar.

Many filesystems are hierarchical, and often referred to as being like a tree. Real offices don't tend to have trees in them, and the folder analogy only works if you visualize subfolders all the way down.

File Input and Output

The simplest kind of persistence is a plain old file, sometimes called a *flat file*. You *read* from a file into memory and *write* from memory to a file. Python makes these jobs easy. As with many languages, its file operations were largely modeled on the familiar and popular Unix equivalents.

Create or Open with open()

You need to call the `open` function before you do the following:

- Read an existing file
- Write to a new file
- Append to an existing file
- Overwrite an existing file

```
fileobj = open( filename, mode )
```

Here's a brief explanation of the pieces of this call:

- *fileobj* is the file object returned by `open()`
- *filename* is the string name of the file
- *mode* is a string indicating the file's type and what you want to do with it

The first letter of *mode* indicates the *operation*:

- `r` means read.
- `w` means write. If the file doesn't exist, it's created. If the file does exist, it's overwritten.
- `x` means write, but only if the file does *not* already exist.
- `a` means append (write after the end) if the file exists.

The second letter of *mode* is the file's *type*:

- `t` (or nothing) means text.
- `b` means binary.

After opening the file, you call functions to read or write data; these will be shown in the examples that follow.

Last, you need to *close* the file to ensure that any writes complete, and that memory is freed. Later, you'll see how to use `with` to automate this for you.

This program opens a file called *oops.txt* and closes it without writing anything. This would create an empty file:

```
>>> fout = open('oops.txt', 'wt')
>>> fout.close()
```

Write a Text File with print()

Let's re-create *oops.txt*, but now write a line to it and then close it:

```
>>> fout = open('oops.txt', 'wt')
```

```
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

We created an empty *oops.txt* file in the previous section, so this just overwrites it.

We used the `file` argument to `print`. Without it, `print` writes to *standard output*, which is your terminal (unless you've told your shell program to redirect output to a file with `>` or piped it to another program with `|`).

Write a Text File with write()

We just used `print` to write a line to a file. We can also use `write`.

For our multiline data source, let's use this limerick about special relativity:[1]

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

The following code writes the entire poem to the file `'relativity'` in one call:

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

The `write` function returns the number of bytes written. It does not add any spaces or newlines, as `print` does. As before, you can also `print` a multiline string to a text file:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
```

```
>>> fout.close()
```

So, should you use `write` or `print`? As you've seen, by default `print` adds a space after each argument and a newline at the end. In the previous example, it appended a newline to the `relativity` file. To make `print` work like `write`, pass it the following two arguments:

- `sep` (separator, which defaults to a space, `' '`)
- `end` (end string, which defaults to a newline, `'\n'`)

We'll use empty strings to replace these defaults:

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

If you have a large source string, you can also write chunks (using slices) until the source is done:

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(poem[offset:offset+chunk])
...     offset += chunk
...
100
50
>>> fout.close()
```

This wrote 100 characters on the first try and the last 50 characters on the next. Slices allow you to "go over the end" without raising an exception.

If the `relativity` file is precious to us, let's see whether using mode `x` really protects us from overwriting it:

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

You can use this with an exception handler:

```
>>> try:
...     fout = open('relativity', 'xt')]
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close one.')
...
relativity already exists!. That was a close one.
```

Read a Text File with read(), readline(), or readlines()

You can call `read()` with no arguments to slurp up the entire file at once, as shown in the example that follows (be careful when doing this with large files; a gigabyte file will consume a gigabyte of memory):

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

You can provide a maximum character count to limit how much `read()` returns at one time. Let's read 100 characters at a time and append each chunk to a `poem` string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
```

```
150
```

After you've read all the way to the end, further calls to `read()` will return an empty string (`''`), which is treated as `False` in `if not fragment`. This breaks out of the `while True` loop.

You can also read the file a line at a time by using `readline()`. In this next example, we append each line to the `poem` string to rebuild the original:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...         break
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

For a text file, even a blank line has a length of one (the newline character), and is evaluated as `True`. When the file has been read, `readline()` (like `read()`) also returns an empty string, which is also evaluated as `False`.

The easiest way to read a text file is by using an *iterator*. This returns one line at a time. It's similar to the previous example but with less code:

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

All of the preceding examples eventually built the single string `poem`. The `readlines()` call reads a line at a time, and returns a list of one-line

strings:

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...     print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

We told `print()` to suppress the automatic newlines because the first four lines already had them. The last line did not, causing the interactive prompt `>>>` to occur right after the last line.

Write a Binary File with write()

If you include a `'b'` in the *mode* string, the file is opened in binary mode. In this case, you read and write `bytes` instead of a string.

We don't have a binary poem lying around, so we'll just generate the 256 byte values from 0 to 255:

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

Open the file for writing in binary mode and write all the data at once:

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

Again, `write()` returns the number of bytes written.

As with text, you can write binary data in chunks:

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...         break
...     fout.write(bdata[offset:offset+chunk])
...     offset += chunk
...
100
100
56
>>> fout.close()
```

Read a Binary File with read()

This one is simple; all you need to do is just open with `'rb'`:

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> len(bdata)
256
>>> fin.close()
```

Close Files Automatically by Using with

If you forget to close a file that you've opened, it will be closed by Python after it's no longer referenced. This means that if you open a file within a function and don't close it explicitly, it will be closed automatically when the function ends. But you might have opened the file in a long-running function or the main section of the program. The file should be closed to force any remaining writes to be completed.

Python has *context managers* to clean up things such as open files. You use the form `with` *`expression`* `as` *`variable`*:

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
```

```
...
```

That's it. After the block of code under the context manager (in this case, one line) completes (normally *or* by a raised exception), the file is closed automatically.

Change Position with seek()

As you read and write, Python keeps track of where you are in the file. The `tell()` function returns your current offset from the beginning of the file, in bytes. The `seek()` function lets you jump to another byte offset in the file. This means that you don't have to read every byte in a file to read the last one; you can `seek()` to the last one and just read one byte.

For this example, use the 256-byte binary file `'bfile'` that you wrote earlier:

```
>>> fin = open('bfile', 'rb')
>>> fin.tell()
0
```

Use `seek()` to jump to one byte before the end of the file:

```
>>> fin.seek(255)
255
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

`seek()` also returns the current offset.

You can call `seek()` with a second argument: `seek(` *`offset`* `,` *`origin`* `)`:

- If `origin` is `0` (the default), go *`offset`* bytes from the start

- If `origin` is `1`, go *`offset`* bytes from the current position
- If `origin` is `2`, go *`offset`* bytes relative to the end

These values are also defined in the standard `os` module:

```
>>> import os
>>> os.SEEK_SET
0
>>> os.SEEK_CUR
1
>>> os.SEEK_END
2
```

So, we could have read the last byte in different ways:

```
>>> fin = open('bfile', 'rb')
```

One byte before the end of the file:

```
>>> fin.seek(-1, 2)
255
>>> fin.tell()
255
```

Read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

NOTE

You don't need to call `tell()` for `seek()` to work. I just wanted to show that they both report the same offset.

Here's an example of seeking from the current position in the file:

```
>>> fin = open('bfile', 'rb')
```

This next example ends up two bytes before the end of the file:

```
>>> fin.seek(254, 0)
254
>>> fin.tell()
254
```

Now go forward one byte:

```
>>> fin.seek(1, 1)
255
>>> fin.tell()
255
```

Finally, read until the end of the file:

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

These functions are most useful for binary files. You can use them with text files, but unless the file is ASCII (one byte per character), you would have a hard time calculating offsets. These would depend on the text encoding, and the most popular encoding (UTF-8) uses varying numbers of bytes per character.

Memory Mapping

An alternative to reading and writing a file is to *memory-map* it with the standard `mmap` module. This makes the contents of a file look like a `bytearray` in memory. See the documentation and some examples for more details.

File Operations

Python, like many other languages, patterned its file operations after Unix. Some functions, such as `chown()` and `chmod()`, have the same names, but there are a few new ones.

I'll first show how Python handles these tasks with functions from the `os.path` module and then with the newer `pathlib` module.

Check Existence with exists()

To verify whether the file or directory is really there or you just imagined it, you can provide `exists()`, with a relative or absolute pathname, as demonstrated here:

```
>>> import os
>>> os.path.exists('oops.txt')
True
>>> os.path.exists('./oops.txt')
True
>>> os.path.exists('waffles')
False
>>> os.path.exists('.')
True
>>> os.path.exists('..')
True
```

Check Type with isfile()

The functions in this section check whether a name refers to a file, directory, or symbolic link (see the examples that follow for a discussion of links).

The first function we'll look at, `isfile`, asks a simple question: is it a plain old law-abiding file?

```
>>> name = 'oops.txt'
>>> os.path.isfile(name)
True
```

Here's how you determine a directory:

```
>>> os.path.isdir(name)
```

```
False
```

A single dot (`.`) is shorthand for the current directory, and two dots (`..`) stands for the parent directory. These always exist, so a statement such as the following will always report `True` :

```
>>> os.path.isdir('.')
True
```

The `os` module contains many functions dealing with *pathnames* (fully qualified filenames, starting with `/` and including all parents). One such function, `isabs()` , determines whether its argument is an absolute pathname. The argument doesn't need to be the name of a real file:

```
>>> os.path.isabs(name)
False
>>> os.path.isabs('/big/fake/name')
True
>>> os.path.isabs('big/fake/name/without/a/leading/slash')
False
```

Copy with copy()

The `copy()` function comes from another module, `shutil` . This example copies the file *oops.txt* to the file *ohno.txt*:

```
>>> import shutil
>>> shutil.copy('oops.txt', 'ohno.txt')
```

The `shutil.move()` function copies a file and then removes the original.

Change Name with rename()

This function does exactly what it says. In the example here, it renames *ohno.txt* to *ohwell.txt*:

```
>>> import os
>>> os.rename('ohno.txt', 'ohwell.txt')
```

Link with link() or symlink()

In Unix, a file exists in one place, but it can have multiple names, called *links*. In low-level *hard links*, it's not easy to find all the names for a given file. A *symbolic link* is an alternative method that stores the new name as its own file, making it possible for you to get both the original and new names at once. The `link()` call creates a hard link, and `symlink()` makes a symbolic link. The `islink()` function checks whether the file is a symbolic link.

Here's how to make a hard link to the existing file *oops.txt* from the new file *yikes.txt*:

```
>>> os.link('oops.txt', 'yikes.txt')
>>> os.path.isfile('yikes.txt')
True
>>> os.path.islink('yikes.txt')
False
```

To create a symbolic link to the existing file *oops.txt* from the new file *jeepers.txt*, use the following:

```
>>> os.symlink('oops.txt', 'jeepers.txt')
>>> os.path.islink('jeepers.txt')
True
```

Change Permissions with chmod()

On a Unix system, `chmod()` changes file permissions. There are read, write, and execute permissions for the user (that's usually you, if you created the file), the main group that the user is in, and the rest of the world. The command takes an intensely compressed octal (base 8) value that combines user, group, and other permissions. For instance, to make *oops.txt* readable only by its owner, type the following:

```
>>> os.chmod('oops.txt', 0o400)
```

If you don't want to deal with cryptic octal values and would rather deal with (slightly less) obscure cryptic symbols, you can import some con-

stants from the `stat` module and use a statement such as the following:

```
>>> import stat
>>> os.chmod('oops.txt', stat.S_IRUSR)
```

Change Ownership with chown()

This function is also Unix/Linux/Mac–specific. You can change the owner and/or group ownership of a file by specifying the numeric user ID (*uid*) and group ID (*gid*):

```
>>> uid = 5
>>> gid = 22
>>> os.chown('oops', uid, gid)
```

Delete a File with remove()

In this snippet, we use the `remove()` function and say farewell to *oops.txt*:

```
>>> os.remove('oops.txt')
>>> os.path.exists('oops.txt')
False
```

Directory Operations

In most operating systems, files exist in a hierarchy of *directories* (often called *folders*). The container of all of these files and directories is a *filesystem* (sometimes called a *volume*). The standard `os` module deals with operating specifics such as these and provides the following functions with which you can manipulate them.

Create with mkdir()

This example shows how to create a directory called `poems` to store that precious verse:

```
>>> os.mkdir('poems')
>>> os.path.exists('poems')
True
```

Delete with rmdir()

Upon second thought,[2] you decide you don't need that directory after all. Here's how to delete it:

```
>>> os.rmdir('poems')
>>> os.path.exists('poems')
False
```

List Contents with listdir()

OK, take two; let's make `poems` again, with some contents:

```
>>> os.mkdir('poems')
```

Now get a list of its contents (none so far):

```
>>> os.listdir('poems')
[]
```

Next, make a subdirectory:

```
>>> os.mkdir('poems/mcintyre')
>>> os.listdir('poems')
['mcintyre']
```

Create a file in this subdirectory (don't type all these lines unless you really feel poetic; just make sure you begin and end with matching quotes, either single or tripled):

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
>>> fout.write('''Cheerful and happy was his mood,
... He to the poor was kind and good,
... And he oft' times did find them food,
```

```
... Also supplies of coal and wood,
... He never spake a word was rude,
... And cheer'd those did o'er sorrows brood,
... He passed away not understood,
... Because no poet in his lays
... Had penned a sonnet in his praise,
... 'Tis sad, but such is world's ways.
... ''')
344
>>> fout.close()
```

Finally, let's see what we have. It had better be there:

```
>>> os.listdir('poems/mcintyre')
['the_good_man']
```

Change Current Directory with chdir()

With this function, you can go from one directory to another. Let's leave the current directory and spend a little time in `poems`:

```
>>> import os
>>> os.chdir('poems')
>>> os.listdir('.')
['mcintyre']
```

List Matching Files with glob()

The `glob()` function matches file or directory names by using Unix shell rules rather than the more complete regular expression syntax. Here are those rules:

- `*` matches everything (`re` would expect `.*`)
- `?` matches a single character
- `[abc]` matches character `a`, `b`, or `c`
- `[!abc]` matches any character *except* `a`, `b`, or `c`

Try getting all files or directories that begin with `m`:

```
>>> import glob
```

```
>>> glob.glob('m*')
['mcintyre']
```

How about any two-letter files or directories?

```
>>> glob.glob('??')
[]
```

I'm thinking of an eight-letter word that begins with `m` and ends with `e`:

```
>>> glob.glob('m??????e')
['mcintyre']
```

What about anything that begins with a `k`, `l`, or `m`, and ends with `e`?

```
>>> glob.glob('[klm]*e')
['mcintyre']
```

Pathnames

Almost all of our computers use hierarchical filesystems, with directories ("folders") containing files and other directories, down to various levels. When you want to refer to a specific file or directory, you need its *pathname*: the sequence of directories needed to get there, either *absolute* from the top (the *root*), or *relative* to your current directory.

You'll often hear people confusing a forward *slash* (`'/'`, not the Guns N' Roses guy) and *backslash* (`'\'`).[3] Unix and Macs (and web URLs) use slash as the *path separator*, and Windows uses backslash.[4]

Python lets you use slash as the path separator when you're specifying names. On Windows, you can use backslash, but you know that backslash is a ubiquitous escape character in Python, so you have to double it everywhere, or use Python's raw strings:

```
>>> win_file = 'eek\\urk\\snort.txt'
>>> win_file2 = r'eek\urk\snort.txt'
>>> win_file
```

```
'eek\\urk\\snort.txt'
>>> win_file2
'eek\\urk\\snort.txt'
```

When you're building a pathname, you can do the following:

- Use the appropriate path separation character (`'/'` or `'\'`)
- Build a pathname (see "Build a Pathname with os.path.join()")
- Use pathlib (see "Use pathlib")

Get a Pathname with abspath()

This function expands a relative name to an absolute one. If your current directory is */usr/gaberlunzie* and the file *oops.txt* is there, you can type the following:

```
>>> os.path.abspath('oops.txt')
'/usr/gaberlunzie/oops.txt'
```

Get a symlink Pathname with realpath()

In one of the earlier sections, we made a symbolic link to *oops.txt* from the new file *jeepers.txt*. In circumstances such as this, you can get the name of *oops.txt* from *jeepers.txt* by using the `realpath()` function, as shown here:

```
>>> os.path.realpath('jeepers.txt')
'/usr/gaberlunzie/oops.txt'
```

Build a Pathname with os.path.join()

When you're constructing a multipart pathname, you can call `os.path.join()` to combine them pairwise with the proper path separation character for your operating system:

```
>>> import os
>>> win_file = os.path.join("eek", "urk")
>>> win_file = os.path.join(win_file, "snort.txt")
```

If I run this on a Mac or Linux box, I get this:

```
>>> win_file
'eek/urk/snort.txt'
```

Running on Windows would produce this:

```
>>> win_file
'eek\\urk\\snort.txt'
```

But if the same code produces different result depending on where it's run, that could be a problem. The new `pathlib` module is a portable solution to this.

Use pathlib

Python added the `pathlib` module in version 3.4. It's an alternative to the `os.path` modules that I just described. But why do we need another module?

Rather that treating filesystem pathnames as strings, it introduces the `Path` object to treat them at a little higher level. Create a `Path` with the `Path()` class, and then knit your path together with bare slashes (not `'/'` characters):

```
>>> from pathlib import Path
>>> file_path = Path('eek') / 'urk' / 'snort.txt'
>>> file_path
PosixPath('eek/urk/snort.txt')
>>> print(file_path)
eek/urk/snort.txt
```

This slash trick took advantage of Python's "Magic Methods". A `Path` can tell you a bit about itself:

```
>>> file_path.name
'snort.txt'
>>> file_path.suffix
'.txt'
>>> file_path.stem
```

```
'snort'
```

You can feed `file_path` to `open()` as you would any filename or pathname string.

You can also see what would happen if you ran this program on another system or if you needed to generate foreign pathnames on your computer:

```
>>> from pathlib import PureWindowsPath
>>> PureWindowsPath(file_path)
PureWindowsPath('eek/urk/snort.txt')
>>> print(PureWindowsPath(file_path))
eek\urk\snort.txt
```

See the docs for all the details.

BytesIO and StringIO

You've seen how to modify data in memory and how to get data in and out of files. What do you do if you have in-memory data, but want to call a function that expects a file (or the reverse)? You'd want to modify the data and pass those bytes or characters around, without reading and writing temporary files.

You can use `io.BytesIO` for binary data (`bytes`) and `io.StringIO` for text data (`str`). Using either of these wraps data as a *file-like object*, suitable to use with all the file functions you've seen in this chapter.

One use case for this is data format conversion. Let's apply this to the PIL library (details coming in "PIL and Pillow"), which reads and writes image data. The first argument to its `Image` object's `open()` and `save()` methods is a filename *or a file-like object*. The code in Example 14-1 uses `BytesIO` to read *and* write in-memory data. It reads one or more image files from the command line, converts its image data to three different formats, and prints the length and first 10 bytes of these outputs.

Example 14-1. convert_image.py

```
from io import BytesIO
from PIL import Image
import sys

def data_to_img(data):
    """Return PIL Image object, with data from in-memory <data>"""
    fp = BytesIO(data)
    return Image.open(fp)    # reads from memory

def img_to_data(img, fmt=None):
    """Return image data from PIL Image <img>, in <fmt> format"""
    fp = BytesIO()
    if not fmt:
        fmt = img.format     # keeps the original format
    img.save(fp, fmt)        # writes to memory
    return fp.getvalue()

def convert_image(data, fmt=None):
    """Convert image <data> to PIL <fmt> image data"""
    img = data_to_img(data)
    return img_to_data(img, fmt)

def get_file_data(name):
    """Return PIL Image object for image file <name>"""
    img = Image.open(name)
    print("img", img, img.format)
    return img_to_data(img)

if __name__ == "__main__":
    for name in sys.argv[1:]:
        data = get_file_data(name)
        print("in", len(data), data[:10])
        for fmt in ("gif", "png", "jpeg"):
            out_data = convert_image(data, fmt)
            print("out", len(out_data), out_data[:10])
```

NOTE

Because it acts like a file, you can `seek()`, `read()`, and `write()` a `BytesIO` object just like a normal file; if you did a `seek()` followed by a `read()`, you would get only the bytes from that seek position to the end. That `getvalue()` returns all the bytes in the `BytesIO` object.

Here's the output, using an input image file that you'll see in Chapter 20:

```
$ python convert_image.py ch20_critter.png
img <PIL.PngImagePlugin.PngImageFile image mode=RGB size=154x141 at 0x10340CF28> |
in 24941 b'\\x89PNG\\r\\n\\x1a\\n\\x00\\x00'
out 14751 b'GIF87a\\x9a\\x00\\x8d\\x00'
out 24941 b'\\x89PNG\\r\\n\\x1a\\n\\x00\\x00'
out 5914 b'\\xff\xd8\\xff\\xe0\\x00\\x10JFIF'
```

Coming Up

The next chapter is a bit more complex. It deals with *concurrency* (ways of doing multiple things at about the same time) and *processes* (running programs).

Things to Do

14.1 List the files in your current directory.

14.2 List the files in your parent directory.

14.3 Assign the string `'This is a test of the emergency text system'` to the variable `test1`, and write `test1` to a file called *test.txt*.

14.4 Open the file *test.txt* and read its contents into the string `test2`. Are `test1` and `test2` the same?

[1] In the first manuscript of this book, I said *general* relativity, and was kindly corrected by a physicist reviewer.

[2] Why is it never first?

[3] One way to remember: forward slash tilts *forward*, backslash tilts *back*.

[4] QDOS was the operating system that Bill Gates bought for $50,000 to have "MS-DOS" when IBM came calling about their first PC. It mimicked CP/M, which used slashes for command-line arguments. When MS-DOS later added folders, it had to use backslashes.