Files and Databases

Contents

Most of the programs we have seen so far are **ephemeral** in the sense that they run for a short time and produce output, but when they end, their data disappears. Each time you run an ephemeral program, it starts with a clean slate.

Other programs are **persistent**: they run for a long time (or all the time); they keep at least some of their data in long-term storage; and if they shut down and restart, they pick up where they left off.

A simple way for programs to maintain their data is by reading and writing text files. A more versatile alternative is to store data in a database. Databases are specialized files that can be read and written more efficiently than text files, and they provide additional capabilities.

In this chapter, we'll write programs that read and write text files and databases, and as an exercise you'll write a program that searches a collection of photos for duplicates. But before you can work with a file, you have to find it, so we'll start with file names, paths, and directories.

13.1. Filenames and paths

Files are organized into **directories**, also called "folders". Every running program has a **current working directory**, which is the default directory for most operations. For example, when you open a file, Python looks for it in the current working directory.

The `os` module provides functions for working with files and directories ("os" stands for "operating system"). It provides a function called `getcwd` that gets the name of the current working directory.

```
import os

os.getcwd()
```

```
'/home/dinsdale'
```

The result in this example is the home directory of a user named `dinsdale`. A string like `'/home/dinsdale'` that identifies a file or directory is called a **path**.

A simple filename like `'memo.txt'` is also considered a path, but it is a **relative path** because it specifies a file name relative to the current directory. In this example, the current directory is `/home/dinsdale`, so `'memo.txt'` is equivalent to the complete path `'/home/dinsdale/memo.txt'`.

A path that begins with `/` does not depend on the current directory – it is called an **absolute path**. To find the absolute path to a file, you can use `abspath`.

```
os.path.abspath('memo.txt')
```

```
'/home/dinsdale/memo.txt'
```

The `os` module provides other functions for working with filenames and paths. `listdir` returns a list of the contents of the given directory, including files and other directories. Here's an example that lists the contents of a directory named `photos`.

```
os.listdir('photos')
```

```
['digests.dat',
 'digests.dir',
 'notes.txt',
 'new_notes.txt',
 'mar-2023',
 'digests.bak',
 'jan-2023',
 'feb-2023']
```

This directory contains a text file named `notes.txt` and three directories. The directories contain image files in the JPEG format.

```
os.listdir('photos/jan-2023')
```

```
['photo3.jpg', 'photo2.jpg', 'photo1.jpg']
```

To check whether a file or directory exists, we can use `os.path.exists`.

```
os.path.exists('photos')
```

```
True
```

```
os.path.exists('photos/apr-2023')
```

```
False
```

To check whether a path refers to a file or directory, we can use `isdir`, which return `True` if a path refers to a directory.

```
os.path.isdir('photos')
```

```
True
```

And `isfile` which returns `True` if a path refers to a file.

```
os.path.isfile('photos/notes.txt')
```

```
True
```

One challenge of working with paths is that they look different on different operating systems. On macOS and UNIX systems like Linux, the directory and file names in a path are separated by a forward slash, `/`. Windows uses a backward slash, `\`. So, if you you run these examples on Windows, you will see backward slashes in the paths, and you'll have to replace the forward slashes in the examples.

Or, to write code that works on both systems, you can use `os.path.join`, which joins directory and filenames into a path using a forward or backward slash, depending on which operating system you are using.

```
os.path.join('photos', 'jan-2023', 'photo1.jpg')
```

```
'photos/jan-2023/photo1.jpg'
```

Later in this chapter we'll use these functions to search a set of directories and find all of the image files.

13.2. f-strings

One way for programs to store data is to write it to a text file. For example, suppose you are a camel spotter, and you want to record the number of camels you have seen during a period of observation. And suppose that

in one and a half years, you have spotted `23` camels. The data in your camel-spotting book might look like this.

```
num_years = 1.5
num_camels = 23
```

To write this data to a file, you can use the `write` method, which we saw in Chapter 8. The argument of `write` has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with the built-in function `str`.

Here's what that looks like:

```
writer = open('camel-spotting-book.txt', 'w')
writer.write(str(num_years))
writer.write(str(num_camels))
writer.close()
```

That works, but `write` doesn't add a space or newline unless you include it explicitly. If we read back the file, we see that the two numbers are run together.

```
open('camel-spotting-book.txt').read()
```

```
'1.523'
```

At the very least, we should add whitespace between the numbers. And while we're at it, let's add some explanatory text.

To write a combination of strings and other values, we can use an **f-string**, which is a string that has the letter `f` before the opening quotation mark, and contains one or more Python expressions in curly braces. The following f-string contains one expression, which is a variable name.

```
f'I have spotted {num_camels} camels'
```

```
'I have spotted 23 camels'
```

The result is a string where the expression has been evaluated and replaced with the result. There can be more than one expression.

```
f'In {num_years} years I have spotted {num_camels} camels'
```

```
'In 1.5 years I have spotted 23 camels'
```

And the expressions can contain operators and function calls.

```
line = f'In {round(num_years * 12)} months I have spotted {num_camels} camels'
line
```

```
'In 18 months I have spotted 23 camels'
```

So we could write the data to a text file like this.

```
writer = open('camel-spotting-book.txt', 'w')
writer.write(f'Years of observation: {num_years}\n')
writer.write(f'Camels spotted: {num_camels}\n')
writer.close()
```

Both f-strings end with the sequence `\n`, which adds a newline character.

We can read the file back like this:

```
data = open('camel-spotting-book.txt').read()
print(data)
```

```
Years of observation: 1.5
Camels spotted: 23
```

In an f-string, an expression in curly brace is converted to a string, so you can include lists, dictionaries, and other types.

```
t = [1, 2, 3]
d = {'one': 1}
f'Here is a list {t} and a dictionary {d}'
```

```
"Here is a list [1, 2, 3] and a dictionary {'one': 1}"
```

13.3. YAML

One of the reasons programs read and write files is to store **configuration data**, which is information that specifies what the program should do and how.

For example, in a program that searches for duplicate photos, we might have a dictionary called `config` that contains the name of the directory to search, the name of another directory where it should store the results, and a list of file extensions it should use to identify image files.

Here's what it might look like:

```
config = {
    'photo_dir': 'photos',
    'data_dir': 'photo_info',
    'extensions': ['jpg', 'jpeg'],
}
```

To write this data in a text file, we could use f-strings, as in the previous section. But it is easier to use a module called `yaml` that is designed for just this sort of thing.

The `yaml` module provides functions to work with YAML files, which are text files formatted to be easy for humans *and* programs to read and write.

Here's an example that uses the `dump` function to write the `config` dictionary to a YAML file.

```
import yaml

config_filename = 'config.yaml'
writer = open(config_filename, 'w')
yaml.dump(config, writer)
writer.close()
```

If we read back the contents of the file, we can see what the YAML format looks like.

```
readback = open(config_filename).read()
print(readback)
```

```
data_dir: photo_info
extensions:
- jpg
- jpeg
photo_dir: photos
```

Now, we can use `safe_load` to read back the YAML file.

```
reader = open(config_filename)
config_readback = yaml.safe_load(reader)
config_readback
```

```
{'data_dir': 'photo_info',
 'extensions': ['jpg', 'jpeg'],
 'photo_dir': 'photos'}
```

The result is new dictionary that contains the same information as the original, but it is not the same dictionary.

```
config is config_readback
```

```
False
```

Converting an object like a dictionary to a string is called **serialization**. Converting the string back to an object is called **deserialization**. If you serialize and then deserialize an object, the result should be equivalent to the original.

13.4. Shelve

So far we've been reading and writing text files – now let's consider databases. A **database** is a file that is organized for storing data. Some databases are organized like a table with rows and columns of information. Others are organized like a dictionary that maps from keys to values; they are sometimes called **key-value stores**.

The `shelve` module provides functions for creating and updating a key-value store called a "shelf". As an example, we'll create a shelf to contain captions for the figures in the `photos` directory. We'll use the `config` dictionary to get the name of the directory where we should put the shelf.

```
config['data_dir']
```

```
'photo_info'
```

We can use `os.makedirs` to create this directory, if it doesn't already exist.

```
os.makedirs(config['data_dir'], exist_ok=True)
```

And `os.path.join` to make a path that includes the name of the directory and the name of the shelf file, `captions`.

```
db_file = os.path.join(config['data_dir'], 'captions')
db_file
```

```
'photo_info/captions'
```

Now we can use `shelve.open` to open the shelf file. The argument `c` indicates that the file should be created if necessary.

```
import shelve

db = shelve.open(db_file, 'c')
db
```

```
<shelve.DbfilenameShelf at 0x7fcc902cc430>
```

The return value is officially a `DbfilenameShelf` object, more casually called a shelf object.

The shelf object behaves in many ways like a dictionary. For example, we can use the bracket operator to add an item, which is a mapping from a key to a value.

```
key = 'jan-2023/photo1.jpg'
db[key] = 'Cat nose'
```

In this example, the key is the path to an image file and the value is a string that describes the image.

We also use the bracket operator to look up a key and get the corresponding value.

```
value = db[key]
value
```

```
'Cat nose'
```

If you make another assignment to an existing key, `shelve` replaces the old value.

```
db[key] = 'Close up view of a cat nose'
db[key]
```

```
'Close up view of a cat nose'
```

Some dictionary methods, like `keys`, `values` and `items`, also work with shelf objects.

```
list(db.keys())
```

```
['jan-2023/photo1.jpg']
```

```
list(db.values())
```

```
['Close up view of a cat nose']
```

We can use the `in` operator to check whether a key appears in the shelf.

```
key in db
```

```
True
```

And we can use a `for` statement to loop through the keys.

```
for key in db:
    print(key, ':', db[key])
```

```
jan-2023/photo1.jpg : Close up view of a cat nose
```

As with other files, you should close the database when you are done.

```
db.close()
```

Now if we list the contents of the data directory, we see two files.

```
os.listdir(config['data_dir'])
```

```
['captions.dir', 'captions.dat']
```

`captions.dat` contains the data we just stored. `captions.dir` contains information about the organization of the database that makes it more efficient to access. The suffix `dir` stands for "directory", but it has nothing to do with the directories we've been working with that contain files.

13.5. Storing data structures

In the previous example, the keys and values in the shelf are strings. But we can also use a shelf to contain data structures like lists and dictionaries.

As an example, let's revisit the anagram example from an exercise in Chapter 11. Recall that we made a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, the key `'opst'` maps to the list `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

We'll use the following function to sort the letters in a word.

```
def sort_word(word):
    return ''.join(sorted(word))
```

And here's an example.

```
word = 'pots'
key = sort_word(word)
key
```

```
'opst'
```

Now let's open a shelf called `anagram_map`. The argument `'n'` means we should always create a new, empty

shelf, even if one already exists.

```
db = shelve.open('anagram_map', 'n')
```

Now we can add an item to the shelf like this.

```
db[key] = [word]
db[key]
```

```
['pots']
```

In this item, the key is a string and the value is a list of strings.

Now suppose we find another word that contains the same letters, like `tops`

```
word = 'tops'
key = sort_word(word)
key
```

```
'opst'
```

The key is the same as in the previous example, so we want to append a second word to the same list of strings. Here's how we would do it if `db` were a dictionary.

```
db[key].append(word)          # INCORRECT
```

But if we run that and then look up the key in the shelf, it looks like it has not been updated.

```
db[key]
```

```
['pots']
```

Here's the problem: when we look up the key, we get a list of strings, but if we modify the list of strings, it does not affect the shelf. If we want to update the shelf, we have to read the old value, update it, and then write the new value back to the shelf.

```
anagram_list = db[key]
anagram_list.append(word)
db[key] = anagram_list
```

Now the value in the shelf is updated.

```
db[key]
```

```
['pots', 'tops']
```

As an exercise, you can finish this example by reading the word list and storing all of the anagrams in a shelf.

13.6. Checking for equivalent files

Now let's get back to the goal of this chapter: searching for different files that contain the same data. One way to check is to read the contents of both files and compare.

If the files contain images, we have to open them with mode `'rb'`, where `'r'` means we want to read the contents and `'b'` indicates **binary mode**. In binary mode, the contents are not interpreted as text – they are treated as a sequence of bytes.

Here's an example that opens and reads an image file.

```
path1 = 'photos/jan-2023/photo1.jpg'
data1 = open(path1, 'rb').read()
type(data1)
```

```
bytes
```

The result from `read` is a `bytes` object – as the name suggests, it contains a sequence of bytes.

In general the contents of an image file and not human-readable. But if we read the contents from a second file, we can use the `==` operator to compare.

```
path2 = 'photos/jan-2023/photo2.jpg'
data2 = open(path2, 'rb').read()
data1 == data2
```

```
False
```

These two files are not equivalent.

Let's encapsulate what we have so far in a function.

```
def same_contents(path1, path2):
    data1 = open(path1, 'rb').read()
    data2 = open(path2, 'rb').read()
    return data1 == data2
```

If we have only two files, this function is a good option. But suppose we have a large number of files and we

want to know whether any two of them contain the same data. It would be inefficient to compare every pair of files.

An alternative is to use a **hash function**, which takes the contents of a file and computes a **digest**, which is usually a large integer. If two files contain the same data, they will have the same digest. If two files differ, they will *almost always* have different digests.

The `hashlib` module provides several hash functions – the one we'll use is called `md5`. We'll start by using `hashlib.md5` to create a `HASH` object.

```
import hashlib

md5_hash = hashlib.md5()
type(md5_hash)
```

```
_hashlib.HASH
```

The `HASH` object provides an `update` method that takes the contents of the file as an argument.

```
md5_hash.update(data1)
```

Now we can use `hexdigest` to get the digest as a string of hexadecimal digits that represent an integer in base 16.

```
digest = md5_hash.hexdigest()
digest
```

```
'aa1d2fc25b7ae247b2931f5a0882fa37'
```

The following function encapsulates these steps.

```
def md5_digest(filename):
    data = open(filename, 'rb').read()
    md5_hash = hashlib.md5()
    md5_hash.update(data)
    digest = md5_hash.hexdigest()
    return digest
```

If we hash the contents of a different file, we can confirm that we get a different digest.

```
filename2 = 'photos/feb-2023/photo2.jpg'
md5_digest(filename2)
```

```
'6a501b11b01f89af9c3f6591d7f02c49'
```

Now we have almost everything we need to find equivalent files. The last step is to search a directory and find all of the images files.

13.7. Walking directories

The following function takes as an argument the directory we want to search. It uses `listdir` to loop through the contents of the directory. When it finds a file, it prints its complete path. When it finds a directory, it calls itself recursively to search the subdirectory.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        elif os.path.isdir(path):
            walk(path)
```

We can use it like this:

```
walk('photos')
```

```
photos/digests.dat
photos/digests.dir
photos/notes.txt
photos/new_notes.txt
photos/mar-2023/photo2.jpg
photos/mar-2023/photo1.jpg
photos/digests.bak
photos/jan-2023/photo3.jpg
photos/jan-2023/photo2.jpg
photos/jan-2023/photo1.jpg
photos/feb-2023/photo2.jpg
photos/feb-2023/photo1.jpg
```

The order of the results depends on details of the operating system.

13.8. Debugging

When you are reading and writing files, you might run into problems with whitespace. These errors can be hard to debug because whitespace characters are normally invisible. For example, here's a string that contains spaces, a tab represented by the sequence `\t`, and a newline represented by the sequence `\n`. When we print it, we don't see the whitespace characters.

```
s = '1 2\t 3\n 4'
print(s)
```

```
1 2	3
 4
```

The built-in function `repr` can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences.

```
print(repr(s))
```

```
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented `\n`. Others use a return character, represented `\r`. Some use both. If you move files between different systems, these inconsistencies can cause problems.

File name capitalization is another issue you might encounter if you work with different operating systems. In macOS and UNIX, file names can contain lowercase and uppercase letters, digits, and most symbols. But many Windows applications ignore the difference between lowercase and uppercase letters, and several symbols that are allowed in macOS and UNIX are not allowed in Windows.

13.9. Glossary

ephemeral: An ephemeral program typically runs for a short time and, when it ends, its data are lost.

persistent: A persistent program runs indefinitely and keeps at least some of its data in permanent storage.

directory: A collection of files and other directories.

current working directory: The default directory used by a program unless another directory is specified.

path: A string that specifies a sequence of directories, often leading to a file.

relative path: A path that starts from the current working directory, or some other specified directory.

absolute path: A path that does not depend on the current directory.

f-string: A string that has the letter `f` before the opening quotation mark, and contains one or more expressions in curly braces.

configuration data: Data, often stored in a file, that specifies what a program should do and how.

serialization: Converting an object to a string.

deserialization: Converting a string to an object.

database: A file whose contents are organized to perform certain operations efficiently.

key-value stores: A database whose contents are organized like a dictionary with keys that correspond to values.

binary mode: A way of opening a file so the contents are interpreted as sequence of bytes rather than a sequence of characters.

hash function: A function that takes and object and computes an integer, which is sometimes called a digest.

digest: The result of a hash function, especially when it is used to check whether two objects are the same.

13.10. Exercises

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

```
Exception reporting mode: Verbose
```

13.10.1. Ask a virtual assistant

There are several topics that came up in this chapter that I did not explain in detail. Here are some questions you can ask a virtual assistant to get more information.

- "What are the differences between ephemeral and persistent programs?"
- "What are some examples of persistent programs?"
- "What's the difference between a relative path and an absolute path?"
- "Why does the `yaml` module have functions called `load` and `safe_load`?"
- "When I write a Python shelf, what are the files with suffixes `dat` and `dir`?"
- "Other than key-values stores, what other kinds of databases are there?"
- "When I read a file, what's the difference between binary mode and text mode?"
- "What are the differences between a bytes object and a string?"
- "What is a hash function?"
- "What is an MD5 digest?"

As always, if you get stuck on any of the following exercises, consider asking a VA for help. Along with your question, you might want to paste in the relevant functions from this chapter.

13.10.2. Exercise

Write a function called `replace_all` that takes as arguments a pattern string, a replacement string, and two filenames. It should read the first file and write the contents into the second file (creating it if necessary). If the pattern string appears anywhere in the contents, it should be replaced with the replacement string.

Here's an outline of the function to get you started.

```
def replace_all(old, new, source_path, dest_path):
    # read the contents of the source file
    reader = open(source_path)

    # replace the old string with the new

    # write the result into the destination file
```

To test your function, read the file `photos/notes.txt`, replace `'photos'` with `'images'`, and write the result to the file `photos/new_notes.txt`.

13.10.3. Exercise

In a previous section, we used the `shelve` module to make a key-value store that maps from a sorted string of letters to a list of anagrams. To finish the example, write a function called `add_word` that takes as arguments a string and a shelf object.

It should sort the letters of the word to make a key, then check whether the key is already in the shelf. If not, it should make a list that contains the new word and add it to the shelf. If so, it should append the new word to the existing value.

13.10.4. Exercise

In a large collection of files, there may be more than one copy of the same file, stored in different directories or with different file names. The goal of this exercise is to search for duplicates. As an example, we'll work with image files in the `photos` directory.

Here's how it will work:

- We'll use the `walk` function from Walking directories to search this directory for files that end with one of the extensions in `config['extensions']`.
- For each file, we'll use `md5_digest` from Checking for equivalent files to compute a digest of the contents.
- Using a shelf, we'll make a mapping from each digest to a list of paths with that digest.
- Finally, we'll search the shelf for any digests that map to multiple files.
- If we find any, we'll use `same_contents` to confirm that the files contain the same data.

I'll suggest some functions to write first, then we'll bring it all together.

1. To identify image files, write a function called `is_image` that takes a path and a list of file extensions, and

returns `True` if the path ends with one of the extensions in the list. Hint: Use `os.path.splitext` – or ask a virtual assistant to write this function for you.

2. Write a function called `add_path` that takes as arguments a path and a shelf. It should use `md5_digest` to compute a digest of the file contents. Then it should update the shelf, either creating a new item that maps from the digest to a list containing the path, or appending the path to the list if it exists.

3. Write a version of `walk` called `walk_images` that takes a directory and walks through the files in the directory and its subdirectories. For each file, it should use `is_image` to check whether it's an image file and `add_path` to add it to the shelf.

When everything is working, you can use the following program to create the shelf, search the `photos` directory and add paths to the shelf, and then check whether there are multiple files with the same digest.

```
db = shelve.open('photos/digests', 'n')
walk_images('photos')

for digest, paths in db.items():
    if len(paths) > 1:
        print(paths)
```

You should find one pair of files that have the same digest. Use `same_contents` to check whether they contain the same data.

Think Python: 3rd Edition