Chapter 18
Reading and Writing Files

18.1 Introduction

Reading data from and writing data to a file is very common within many programs. Python provides a large amount of support for working with files of various types. This chapter introduces you to the core file IO functionality in Python.

18.2 Obtaining References to Files

Reading from, and writing to, text files in Python is relatively straightforward. The built in `open()` function creates a file object for you that you can use to read and/ or write data from and/ or to a file.

The function requires as a minimum the name of the file you want to work with.

Optionally you can specify the access mode (e.g. read, write, append etc.). If you do not specify a mode then the file is open in read-only mode. You can also specify whether you want the interactions with the file to be buffered which can improve performance by grouping data reads together.

The syntax for the `open()` function is

```
file_object = open(file_name, access_mode, buffering)
```

Where

- `file_name` indicates the file to be accessed.
- `access_mode` The `access_mode` determines the mode in which the file is to be opened, i.e. read, write, append, etc. A complete list of possible values is given below in the table. This is an optional parameter and the default file access mode is read (r).

J. Hunt, *Advanced Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-25943-3_18

- `buffering` If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.

The `access_mode` values are given in the following table.

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode
r+	Opens a file for both reading and writing. The file pointer placed at the beginning of the file
rb+	Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing
wb+	Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing
a+	Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing

The file object itself has several useful attributes such as

- `file.closed` returns `True` if the file has been closed (can no longer be accessed because the `close()` method has been called on it).
- `file.mode` returns the access mode with which the file was opened.
- `file.name` The name of the file.

The `file.close()` method is used to close the file once you have finished with it. This will flush any unwritten information to the file (this may occur because of buffering) and will close the reference from the file object to the actual underlying operating system file. This is important to do as leaving a reference to a file open can cause problems in larger applications as typically there are only a certain number of file references possible at one time and over a long period of time these

may all be used up resulting in future errors being thrown as files can no longer be opened.

The following short code snippet illustrates the above ideas:

```
file = open('myfile.txt', 'r+')
print('file.name:', file.name)
print('file.closed:', file.closed)
print('file.mode:', file.mode)
file.close()
print('file.closed now:', file.closed)
```

The output from this is:

```
file.name: myfile.txt
file.closed: False
file.mode: r+
file.closed now: True
```

18.3 Reading Files

Of course, having set up a file object we want to be able to either access the contents of the file or write data to that file (or do both). Reading data from a text file is supported by the `read()`, `readline()` and `readlines()` methods:

- The `read()` method This method will return the entire contents of the file as a single string.
- The `readline()` method reads the next line of text from a file. It returns all the text on one line up to and including the newline character. It can be used to read a file a line at a time.
- The `readlines()` method returns a list of all the lines in a file, where each item of the list represents a single line.

Note that once you have read some text from a file using one of the above operations then that line is not read again. Thus using `readlines()` would result in a further `readlines()` returning an empty list whatever the contents of the file.

The following illustrates using the `readlines()` method to read all the text in a text file into a program and then print each line out in turn:

```
file = open('myfile.txt', 'r')
lines = file.readlines()
for line in lines:
    print(line, end='')
file.close()
```

Notice that within the `for` loop we have indicated to the `print` function that we want the end character to be `' '` rather than a newline; this is because the line string already possesses the newline character read from the file.

18.4 File Contents Iteration

As suggested by the previous example; it is very common to want to process the contents of a file one line at a time. In fact Python makes this extremely easy by making the file object support iteration. File iteration accesses each line in the file and makes that line available to the `for` loop. We can therefore write:

```
file = open('myfile.txt', 'r')
for line in file:
    print(line, end='')
file.close()
```

It is also possible to use the *list comprehension* to provide a very concise way to load and process lines in a file into a list. It is similar to the effect of `readlines()` but we are now able to pre-process the data before creating the list:

```
file = open('myfile.txt', 'r')
lines = [line.upper() for line in file]
file.close()
print(lines)
```

18.5 Writing Data to Files

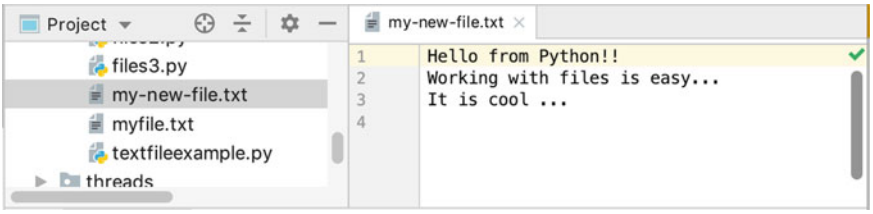
Writing a string to a file is supported by the `write()` method. Of course, the file object we create must have an access mode that allows writing (such as `'w'`). Note that the write method *does not* add a newline character (represented as `'\n'`) to the end of the string—you must do this manually.

An example short program to write a text file is given below:

```
print('Writing file')
f = open('my-new-file.txt', 'w')
f.write('Hello from Python!!\n')
f.write('Working with files is easy...\n')
f.write('It is cool ...\n')
f.close()
```

This creates a new file called `my-new-file.txt`. It then writes three strings to the file each with a newline character on the end; it then closes the file.

The effect of this is to create a new file called `myfile.txt` with three lines in it:



18.6 Using Files and with Statements

Like several other types where it is important to shut down resources; the file object class implements the *Context Manager Protocol* and thus can be used with the `with` statement. It is therefore common to write code that will open a file using the `with as` structure thus ensuring that the file will be closed when the block of code is finished with, for example:

```
with open('my-new-file.txt', 'r') as f:
    lines = file.readlines()
    for line in lines:
        print(line, end='')
```

18.7 The Fileinput Module

In some situations, you may need to read the input from several files in one go. You could do this by opening each file independently and then reading the contents and appending that contents to a list etc. However, this is a common enough requirement that the `fileinput` module provides a function `fileinput.input()` that can take a list of files and treat all the files as a single input significantly simplifying this process, for example:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

Features provided by the `fileinput` module include

- Return the name of the file currently being read.
- Return the integer "file descriptor" for the current file.
- Return the cumulative line number of the line that has just been read.
- Return the line number in the current file. Before the first line has been read this returns 0.
- A boolean function that indicates if the current line just read is the first line of its file

Some of these are illustrated below:

```
with fileinput.input(files=('textfile1.txt',
'textfile2.txt')) as f:
        line = f.readline()
        print('f.filename():', f.filename())
        print('f.isfirstline():', f.isfirstline())
        print('f.lineno():', f.lineno())
        print('f.filelineno():', f.filelineno())
        for line in f:
            print(line, end='')
```

18.8 Renaming Files

A file can be renamed using the `os.rename()` function. This function takes two arguments, the current filename and the new filename. It is part of the Python `os` module which provides methods that can be used to perform a range of file-processing operations (such as renaming a file). To use the module, you will first need to import it. An example of using the rename function is given below:

```
import os
os.rename('myfileoriginalname.txt','myfilenewname.txt')
```

18.9 Deleting Files

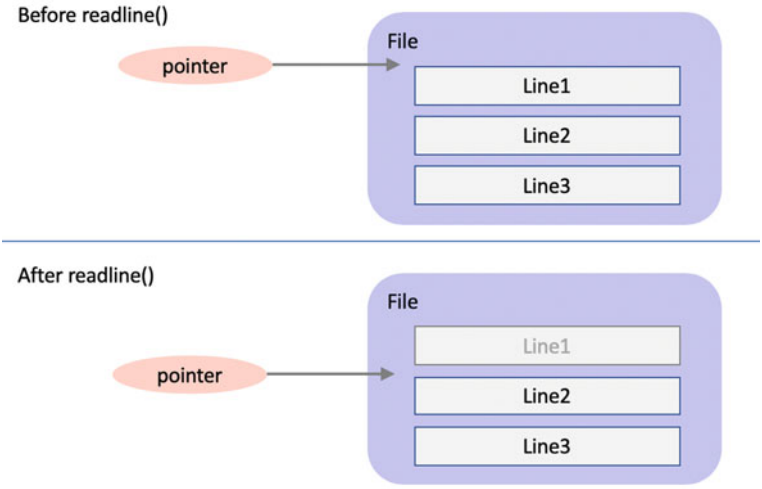
A file can be deleted using the `os.remove()` method. This method deletes the file specified by the filename passed to it. Again, it is part of the `os` module and therefore this must be imported first:

```
import os
os.remove('somefilename.txt')
```

18.10 Random Access Files

All the examples presented so far suggest that files are accessed sequentially, with the first line read before the second and so on. Although this is (probably) the most common approach it is not the only approach supported by Python; it is also possible to use a random-access approach to the contents within a file.

To understand the idea of random file access it is useful to understand that we can maintain a pointer into a file to indicate where we are in that file in terms of reading or writing data. Before anything is read from a file the pointer is before the beginning of the file and reading the first line of text would for example, advance the point to the start of the second line in the file etc. This idea is illustrated below:



When randomly accessing the contents of a file the programmer manually moves the pointer to the location required and reads or writes text relative to that pointer. This means that they can move around in the file reading and writing data.

The random-access aspect of a file is provided by the seek method of the file object:

- `file.seek (offset, whence)` this method determines where the next read or write operation (depending on the mode used in the `open()` call) takes place.

In the above the offset parameter indicates the position of the read/ write pointer within the file. The move can also be forwards or backwards (represented by a negative offset).

The optional whence parameter indicates where the offset is relative to. The values used for whence are:

- `0` indicates that the offset is relative to start of file (the default).
- `1` means that the offset is relative to the current pointer position.
- `2` indicates the offset is relative to end of file.

Thus, we can move the pointer to a position relative to the start of the file, to the end of the file, or to the current position.

For example, in the following sample code we create a new text file and write a set of characters into that file. At this point the pointer is positioned after the 'z' in the file. However, we then use `seek()` to move the point to the 10th character in the file and now write '`Hello`', next we reposition the pointer to the 6th character in the file and write out '`BOO`'. We then close the file. Finally, we read all the lines from the file using a `with as` statement and the `open()` function and from this we will see that the text is the file is now `abcdef`**`BOO`**`j`**`HELLO`**`pqrstuvwxyz`:

```
f = open('text.txt', 'w')
f.write('abcdefghijklmnopqrstuvwxyz\n')
f.seek(10,0)
f.write('HELLO')
f.seek(6, 0)
f.write ('BOO')
f.close()
with open('text.txt', 'r') as f:
    for line in f:
        print(line, end='')
```

18.11 Directories

Both Unix like systems and Windows operating systems are hierarchical structures comprising directories and files. The `os` module has several functions that can help with creating, removing and altering directories. These include:

- `mkdir()` This function is used to create a directory, it takes the name of the directory to create as a parameter. If the directory already exists `FileExistsError` is raised.
- `chdir()` This function can be used to change the current working directory. This is the directory that the application will read from/ write to by default.
- `getcwd()` This function returns a string representing the name of the current working directory.
- `rmdir()` This function is used to remove/ delete a directory. It takes the name of the directory to delete as a parameter.
- `listdir()` This function returns a list containing the names of the entries in the directory specified as a parameter to the function (if no name is given the current directory is used).

A simple example illustrates the use of some of these functions is given below:

```
import os
print('os.getcwd(:', os.getcwd())
print('List contents of directory')
print(os.listdir())
print('Create mydir')
os.mkdir('mydir')
print('List the updated contents of directory')
print(os.listdir())
print('Change into mydir directory')
os.chdir('mydir')
print('os.getcwd(:', os.getcwd())
print('Change back to parent directory')
os.chdir('..')
print('os.getcwd(:', os.getcwd())
print('Remove mydir directory')
os.rmdir('mydir')
print('List the resulting contents of directory')
print(os.listdir())
```

Note that '. .' is a short hand for the parent directory of the current directory and '.' is short hand for the current directory.

An example of the type of output generated by this program for a specific set up on a Mac is given below:

```
os.getcwd(:
/Users/Shared/workspaces/pycharm/pythonintro/textfiles
List contents of directory
['my-new-file.txt', 'myfile.txt', 'textfile1.txt',
'textfile2.txt']
Create mydir
List the updated contents of directory
['my-new-file.txt', 'myfile.txt', 'textfile1.txt',
'textfile2.txt', 'mydir']
Change into mydir directory
os.getcwd(:
/Users/Shared/workspaces/pycharm/pythonintro/textfiles/mydir
Change back to parent directory
os.getcwd(:
/Users/Shared/workspaces/pycharm/pythonintro/textfiles
Remove mydir directory
List the resulting contents of directory
['my-new-file.txt', 'myfile.txt', 'textfile1.txt',
'textfile2.txt']
```

18.12 Temporary Files

During the execution of many applications it may be necessary to create a temporary file that will be created at one point and deleted before the application finishes. It is of course possible to manage such temporary files yourself however, the `tempfile` module provides a range of facilities to simplify the creation and management of these temporary files.

Within the `tempfile` module `TemporaryFile`, `NamedTemporaryFile`, `TemporaryDirectory`, and `SpooledTemporaryFile` are high-level file objects which provide automatic cleanup of temporary files and directories. These objects implement the *Context Manager Protocol*.

The `tempfile` module also provides the lower-level function `mkstemp()` and `mkdtemp()` that can be used to create temporary files that require the developer to management them and delete them at an appropriate time.

The high-level feature for the `tempfile` module are:

- `TemporaryFile(mode='w+b')` Return an anonymous file-like object that can be used as a temporary storage area. On completion of the managed context (via a `with` statement) or destruction of the file object, the temporary file will be removed from the filesystem. Note that by default all data is written to the temporary file in binary format which is generally more efficient.
- `NamedTemporaryFile(mode='w+b')` This function operates exactly as `TemporaryFile()` does, except that the file has s visible name in the file system.
- `SpooledTemporaryFile(max_size=0, mode='w+b')` This function operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with `TemporaryFile()`.
- `TemporaryDirectory(suffix=None, prefix=None, dir=None)` This function creates a temporary directory. On completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed from the filesystem.

The lower level functions include:

- `mkstemp()` Creates a temporary file that is only readable or writable by the user who created it.
- `mkdtemp()` Creates a temporary directory. The directory is readable, writable, and searchable only by the creating user ID.
- `gettempdir()` Return the name of the directory used for temporary files. This defines the default value for the default temporary directory to be used with the other functions in this module.

An example of using the `TemporaryFile` function is given below. This code imports the `tempfile` module then prints out the default directory used for

temporary files. It then creates a `TemporaryFile` object and prints its name and mode (the default mode is binary but for this example we have overwritten this so that plain text is used). We have then written a line to the file. Using seek we are repositioning ourselves at the start of the file and then reading the line we have just written.

```
import tempfile

print('tempfile.gettempdir():', tempfile.gettempdir())
temp = tempfile.TemporaryFile('w+')
print('temp.name:', temp.name)
print('temp.mode:', temp.mode)
temp.write('Hello world!')
temp.seek(0)
line = temp.readline()
print('line:', line)
```

The output from this when run on an Apple Mac is:

```
tempfile.gettempdir():
/var/folders/6n/8nrnt9f93pn66ypg9s5dq8y80000gn/T
temp.name: 4
temp.mode: w+
line: Hello world!
```

Note that the file name is '4' and that the temporary directory is not a meaningful name!

18.13 Working with Paths

The `pathlib` module provides a set of classes representing filesystem paths; that is paths through the hierarchy of directories and files within an operating systems file structure. It was introduced in Python 3.4. The core class in this module is the `Path` class.

A `Path` object is useful because it provides operations that allow you to manipulate and manage the path to a file or directory. The `Path` class also replicates some of the operations available from the `os` module (such as `mkdir`, `rename` and `rmdir`) which means that it is not necessary to work directly with the `os` module.

A path object is created using the `Path` constructor function; this function actually returns a specific type of `Path` depending on the type of operating system being used such as a `WindowsPath` or a `PosixPath` (for Unix style systems).

The `Path()` constructor takes the path to create for example '`D:/mydir`' (on Windows) or '`/Users/user1/mydir`' on a Mac or '`/var/temp`' on Linux etc.

You can then use several different methods on the Path object to obtain information about the path such as:

- `exists()` returns `True` of `False` depending on whether the path points to an existing file or directory.
- `is_dir()` returns `True` if the path points to a directory. `False` if it references a file. `False` is also returned if the path does not exist.
- `is_file()` returns `True` of the path points to a file, it returns `False` if the path does not exist or the path references a directory.
- `absolute()` A `Path` object is considered absolute if it has both a root and (if appropriate) a drive.
- `is_absolute()` returns a `Boolean` value indicating whether the Path is absolute or not.

An example of using some of these methods is given below:

```
from pathlib import Path
print('Create Path object for current directory')
p = Path('.')
print('p:', p)
print('p.exists():', p.exists())
print('p.is_dir():', p.is_dir())
print('p.is_file():', p.is_file())
print('p.absolute():', p.absolute())
```

Sample output produced by this code snippet is:

```
Create Path object for current directory
p: .
p.exists(): True
p.is_dir(): True
p.is_file(): False
p.absolute():
/Users/Shared/workspaces/pycharm/pythonintro/textfiles
```

There are also several methods on the `Path` class that can be used to create and remove directories and files such as:

- `mkdir()` is used to create a directory path if it does not exist. If the path already exists, then a `FileExistsError` is raised.
- `rmdir()` remove this directory; the directory must be empty otherwise an error will be raised.

- `rename(target)` rename this file or directory to the given target.
- `unlink()` removes the file referenced by the path object.
- `joinpath(*other)` appends elements to the path object e.g. path.joinpath('/temp').
- `with_name(new_name)` return a new path object with the name changed.
- The '`/`' operator can also be used to create new path objects from existing paths for example path/ 'test'/ 'output' which would append the directories test and out to the path object.

Two `Path` class methods can be used to obtain path objects representing key directories such as the current working directory (the directory the program is logically in at that point) and the home directory of the user running the program:

- `Path.cwd()` return a new path object representing the current directory.
- `Path.home() return` a new path object representing the user's home directory.

An example using several of the above features is given below. This example obtains a path object representing the current working directory and then appends 'text' to this. The result path object is then checked to see if the path exists (on the computer running the program), assuming that the path does not exist it is created and the `exists()` method is rerun.

```
p = Path.cwd()
print('Set up new directory')
newdir = p / 'test'
print('Check to see if newdir exists')
print('newdir.exists():', newdir.exists())
print('Create new dir')
newdir.mkdir()
print('newdir.exists():', newdir.exists())
```

The effect of creating the directory can be seen in the output:

```
Set up new directory
Check to see if newdir exists
newdir.exists(): False
Create new dir
newdir.exists(): True
```

A very useful method in the `Path` object is the `glob(pattern)` method. This method returns all elements within the path that meet the pattern specified.

For example `path.glob('*.py')` will return all the files ending `.py` within the current path.

Note that '`**/*.py`' would indicate the current directory and any sub directory. For example, the following code will return all files where the file name ends with '`.txt`' for a given path:

```
print('-' * 10)

for file in path.glob('*.txt'):
    print('file:', file)

print('-' * 10)
```

An example of the output generated by this code is:

```
----------
file: my-new-file.txt
file: myfile.txt
file: textfile1.txt
file: textfile2.txt
----------
```

Paths that reference a file can also be used to read and write data to that file. For example the `open()` method can be used to open a file that by default allows a file to be read:

- `open(mode='r') this` can be used to open the file referenced by the path object.

This is used below to read the contents of a file a line at a time (note that `with as` statement is used here to ensure that the file represented by the `Path` is closed):

```
p = Path('mytext.txt')
with p.open() as f:
    print(f.readline())
```

However, there are also some high-level methods available that allow you to easily write data to a file or read data from a file. These include the Path methods `write_text` and `read_text` methods:

- `write_text(data)` opens the file pointed to in text mode and writes the data to it and then closes the file.
- `read_text()` opens the file in read mode, reads the text and closes the file; it then returns the contents of the file as a string.

These are used below

```
dir = Path('./test')
print('Create new file')
newfile = dir / 'text.txt'
print('Write some text to file')
newfile.write_text('Hello Python World!')
print('Read the text back again')
print(newfile.read_text())
print('Remove the file')
newfile.unlink()
```

Which generates the following output:

```
Create new file
Write some text to file
Read the text back again
Hello Python World!
Remove the file
```

18.14 Online Resources

See the following online resources for information on the topics in this chapter:

- https://docs.python.org/3/tutorial/inputoutput.html for the Python Standard Tutorial on file input and output.
- https://pymotw.com/3/os.path/index.html for platform independent manipulation of filenames.
- https://pymotw.com/3/pathlib/index.html for information filesystem Path objects.
- https://pymotw.com/3/glob/index.html for filename pattern matching using glob.
- https://pymotw.com/3/tempfile/index.html for temporary file system objects.
- https://pymotw.com/3/gzip/index.html for information on reading and writing GNU Zip files.

18.15 Exercise

The aim of this exercise is to explore the creation of, and access to, the contents of a file.

You should write two programs, these programs are outlined below:

1. Create a program that will write todays date into a file – the name of the file can be hard coded or supplied by the user. You can use the `datetime.today()`

function to obtain the current date and time. You can use the `str()` function to convert this date time object into a string so that it can be written out to a file.
2. Create a second program to reload the date from the file and convert the string into a date object. You can use the `datetime.strptime()` function to convert a string into a date time object (see https://docs.python.org/3/library/datetime.html#datetime.datetime.strptime for documentation on this function). This functions takes a string containing a date and time in it and a second string which defines the format expected. If you use the approach outlined in step 1 above to write the string out to a file then you should find that the following defines an appropriate format to parse the `date_str` so that a date time object can be created:

```
datetime_object = datetime.strptime(date_str, '%Y-%m-%d
%H:%M:%S.%f')
```