## FILES

Python Variables are good to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file. You can think of a file's contents as a single string value. In this chapter, you will learn how to use Python to create, read, and save files on the hard drive.

Before we can work with files, we need to learn what file paths are. So, lets dive in …

## FILE PATHS

Every file in your computer is stored in a particular location. Lets say, you save all your photos in *F:drive* inside *photos* folder. This is the location where you will find all your photos. This is how you organise your files and remember the locations.

The **file path** is a string that specifies the *location of a file* on the computer. It has three major parts:

- **Folder Path:** the file folder location on the file system where subsequent folders are separated by a forward slash (in Unix) or backslash (in Windows)
- **File Name:** the actual name of the file
- **Extension:** the end of the file path pre-pended with a *period ( . )* used to indicate the file type.

For example, you have a photo called *'random.png'* in the path *F:\photos*

*photos* is a folder (also called directory). Folders can contain files and other folders.

In Windows, the root folder starts with drive letters (*C:*, *D:*, etc.), also called the *C:drive*, and *D:drive*, respectively. In MacOS and Linux, the root folder starts with `/`.

There is another important thing to note. In Windows, paths are written using backslashes ( `\` ) as the separator between folder names. MacOS and Linux, however, use the forward slash ( `/` ) as their path separator. The good news is that, you don't have to worry about it. Because Python lets you use MacOS/Linux style slashes `/` even in Windows. Therefore, you can refer *'random.png'* as `F:/Photos/random.png`.

## THE CURRENT WORKING DIRECTORY

Every program that runs on your computer has a *current working directory*, or *cwd*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory. You can get the current working directory as a string value with the `os.getcwd()` function.

```
import os
os.getcwd()
'F:\\AnacondaProjects\\Ai Adventures\\courses complete\\New
Courses\\course_books\\python_book\\course_material'
```

**Note**

You see two backslashes instead of one, because one is used an escape character and the other is the actual separator. You can read more about escape characters here.

You can change the *current working directory* with `os.chdir()`.

## ABSOLUTE VS. RELATIVE PATHS

There are two ways to specify a file path. An **absolute path**, which always begins with the root folder. And a **relative path**, which is relative to the program's current working directory

There are also the *dot ( . )* and *dot-dot ( .. )* folders. These are not real folders but special names that can be used in a path. A single period (or dot) for a folder name is shorthand for "this directory". Two periods (or dot-dot) means "the parent folder".

You can access any file using *absolute path* or *relative path*. Here is a table to showing you both the paths when *current working directory* is `F:/photos`

| Absolute Path | Relative Path |
|---|---|
| `F:/photos` | . (dot) |
| `F:` | . . (dot-dot) |
| `F:/photos/random.png` | `random.png` |
| `F:/aiadventures` | `../aiadventures` |

**Note**

Always check your *cwd* before using relative paths.

**Task**

For practice, take some absolute paths, check your current working directory, and write relative paths for the absolute paths. You can verify your results with the `os.path.relpath()` method. It takes an absolute path and returns the relative path w.r.t current working directory. For example,

```
os.path.relpath('F:/AnacondaProjects/Ai Adventures/courses complete/New Courses/python_course/nbs/sample2.txt')
'nbs\\sample2.txt'
```

## FILES

Once you are comfortable working with folders and relative paths, you'll be able to specify the location of files to read and write. Its very important that you feel comfortable with absolute and relative paths before moving ahead. If not, practice some more and also ask your mentor for help.

Generally speaking, there are two types of files:

- **Text files** contain only basic text characters and do not include font, size, or color information.
- **Binary files** are all other file types, such as mp4, mp3, word, excel, powerpoint, PDFs, images, and executable programs.

The functions covered in the next few sections will apply to plaintext files.

## OPENING FILES IN PYTHON

Python has a built-in `open()` function to open a file. This function returns a file object, also called a **handle,** as it is used to read or modify the file accordingly.

This function creates a **file** object, which would be utilized to call other support methods associated with it.

**SYNTAX**

**file object = open(file_name [, access_mode][, buffering])**

Here are parameter details –

- **file_name** – The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** – The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).

- **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. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behaviour).

**EXAMPLE:**

```
 f = open("test.txt")                   # open file in current directory
f = open("C:/Python38/README.txt")      # specifying full path
f = open("test.txt")                    # equivalent to 'r' or 'rt'
f = open("test.txt",'w')                 # write in text mode
f = open("img.bmp",'r+b')               # read and write in binary mode
```

Note that the file handle is different from a file. It can be related with an analogy of TV (file) and the TV remote control(handle). You can use the remote control to switch channels, change the volume, etc. but whatever changes you try to do with the remote are actually applied to the TV. So your file handle (file object) acts a s a remote control of your file (TV). Whatever changes you want to perform on the file is actually carried out through the file object.

We can specify the mode while opening a file. In mode, we specify whether we want to read `r`, write `w` or append `a` to the file. We can also specify if we want to open the file in text mode or binary mode.

## TEXT VS BINARY

The default is reading in text mode. In this mode, **we get strings when reading from the file.** In a text file each line of data ends with a newline character. Each file ends with a special character called the end-of-file (EOF) marker.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with **non-text files like images or executable files.** Binary files store data in the internal representation format. As a consequence, an integer value will be stores in binary format as a 4-byte value. The same format is used to store data in memory as well as in file. Like text file, binary file also ends with an EOF marker. Binary files are mainly used to store data beyond text such as images, executables, etc.

## ACCESS MODES:

| Sr.No. | Modes & Description |
|---|---|
| 1 | **r**<br><br>Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode. |
| 2 | **rb**<br><br>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. |
| 3 | **r+**<br><br>Opens a file for both reading and writing. The file pointer placed at the beginning of the file. |
| 4 | **rb+**<br><br>Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file. |
| 5 | **w**<br><br>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. |
| 6 | **wb**<br><br>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. |
| 7 | **w+**<br><br>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. |
| 8 | **wb+**<br><br>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. |
| 9 | **a**<br><br>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. |
| 10 | **ab**<br><br>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. |
| 11 | **a+**<br><br>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. |
| 12 | **ab+**<br><br>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. |

## CAUTION:

Unlike other languages, the character `a` does not imply the number 97 until it is encoded using `ASCII` (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is `cp1252` but `utf-8` in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

## OPENING FILES USING with KEYWORD:

The advantage of opening files using this technique is that the file is properly closed immediately after it is used, even if an error occurs during read or write operation or if one forgets to close the file explicitly.

```
with open("welcome.txt") as file:       # Use file to refer to the file object
          data = file.read()
          do something with data
```

**Opens output.txt in write mode**

```
with open('output.txt', 'w') as file:       # Use file to refer to the file object
  file.write('Hi there!')
```

Notice, that we didn't have to write **file.close()**.That will automatically be called.

## CLOSING FILES IN PYTHON:

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the close() method available in Python.

Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

```
f = open("test.txt", encoding = 'utf-8')       # perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

```
try:
  f = open("test.txt", encoding = 'utf-8')
  # perform file operations
finally:
  f.close()
```

This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

The best way to close a file is by using the `with` statement. This ensures that the file is closed when the block inside the `with` statement is exited.

We don't need to explicitly call the `close()` method. It is done internally.
You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

## OPEN ( ) FUNCTION

To open a file with the `open()` function, you pass it a string path indicating the path of the file you want to open; it can be either an absolute or relative path.
The `open()` function returns a **File object**.

Before we can read a file, we will have to create it.

- Check your *current working directory* (cwd).
- Create a new text file named *sample.txt* in your *current working directory*.
- Open *sample.txt* and write some random text in it.
- Save the changes and close the file.

Lets try to open *sample.txt*. The file path can be written as *sample.txt*.

**Note:** The path might be different, if you have created the file in some other directory.

```
f = open('sample.txt')
```

The above command will open the file in read mode. When a file is opened in read mode, Python lets you only read data from the file; you can't write or modify it in any way. Read mode is the default mode for files you open in Python.

The call to `open()` returns a *File object / File descriptor*. A **File object** represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with. File object can be thought of as a "read/write pointer" and you can also perform operations like moving the "read/write pointer", which determines where in the file data is read from and where it is written to.

In the previous example, we stored the *File object* in the variable `f`. Now, whenever you want to interact with the file, you can do so by calling methods on the File object in variable `f`.

## READ() FUNCTION

If you want to read the entire contents of a file as a string value, use the File object's `read()` method.

```
text = f.read()
text
'Here is some random text.\nAgain some random text.\nSorry, random text
again!'
```

## CLOSE() FUNCTION

```
f.close()
```

**Note**

It's important to remember that it's your responsibility to close the file. You can close your file by calling `close()` method of the file object. Not closing the file can have unintended effects.
Instead of `read()` you can use the `readlines()` method to get a list of string values from the file, one string for each line of text.

```
f = open('sample.txt')
lines = f.readlines()
print(lines)
f.close()
['Here is some random text.\n', 'Again some random text.\n', 'Sorry,
random text again!']
```

**Note**

Except for the last line of the file, each of the string values ends with a newline character `\n`.

## WRITE() FUNCTION

Python allows you to write content to a file in a way similar to how the `print()` function writes strings to the screen. You can't write to a file you've opened in *read mode*, though. Instead, you need to open it in *write mode* and *append mode*.

**Write mode** will overwrite the existing file and start from scratch, just like when you overwrite a variable's value with a new value. Pass `'w'` as the second argument to `open()` to open the file in write mode.

**Append mode**, on the other hand, will append text to the end of the existing file. You can think of this as appending to a list in a variable, rather than overwriting the variable altogether. Pass `'a'` as the second argument to `open()` to open the file in append mode.

If the filename passed to `open()` does not exist, both *write* and *append* mode will create a new, blank file. After reading or writing a file, call the `close()` method before opening the file again.

```
fw = open('sample2.txt', 'w') # write mode
fw.write('This is written by python.')
fw.close()
fw = open('sample2.txt', 'a') # append mode
fw.write('This is an appended text')
fw.close()
f = open('sample2.txt') # reading mode
text = f.read()
f.close()
text
'This is written by python.This is an appended text'
```