File handling in Python

What Is a File?

Before we can go into how to work with files in Python, it's important to understand what exactly a file is and how modern operating systems handle some of their aspects.

At its core, **a file is a contiguous set of bytes used to store data permanently**. This data is organized in a specific format and can be anything as simple as a text file or as complicated as an executable program. In the end, these byte files are then translated into binary 1 and 0 for easier processing by the computer.

Files on most modern file systems are composed of three main parts:

1. **Header:** metadata about the contents of the file (file name, size, type, and so on)
2. **Data:** contents of the file as written by the creator or editor
3. **End of file (EOF):** special character that indicates the end of the file

What this data represents depends on the format specification used, which is typically represented by an extension. For example, a file that has an extension of **.gif** most likely conforms to the **Graphics Interchange Format** specification. There are hundreds, if not thousands, of file extensions out there.

In Python, there is **no need for importing external libraries to read and write files**. Python provides an inbuilt function for creating, writing, and reading files

The file handling plays an important role when the data needs to be stored permanently into the file. **A file is a named location on disk to store related information**. We can access the stored information (non-volatile) after the program termination.

Files are treated in **two modes as text or binary**. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write - Performing operation
- Close the file

Opening a file

Python provides an **open()** function that accepts two arguments, **file name and access mode** in which the file is accessed. The function **returns a file object** which can be used to **perform various operations** like reading, writing, etc.
Syntax:

file object = open(<file-name>, <access-mode>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

Mode	Description
r	It opens the file to **read-only mode**. The file **pointer exists at the beginning.** The file is by default open in this mode if no access mode is passed.
rb	It opens the file to **read-only in binary format**. The file **pointer exists at the beginning** of the file.
r+	It opens the file to **read and write both**. The **file pointer exists at the beginning** of the file.

rb+	It opens the file to **read and write both in binary format**. The **file pointer exists at the beginning** of the file.
w	It opens the file to **write only**. It **overwrites** the file if it previously exists or **creates a new one** if no file exists with the same name. The file **pointer exists at the beginning** of the file.
wb	It opens the file to **write only in binary format**. It o**verwrites** the file if it exists previously or **creates a new one** if no file exists. The file **pointer exists at the beginning** of the file.
w+	It opens the file to **write and read both**. It is different from r+ in the sense that it overwrites the previous file if one exists whereas **r+ doesn't overwrite the previously written file**. It creates a new file if no file exists. The file pointer exists at the beginning of the file.
wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
a	It opens the file in the **append mode**. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name.
ab	It opens the file in the **append mode in binary** format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
a+	It opens a file to **append and read both**. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
ab+	It opens a file to **append and read both in binary format**. The file pointer remains at the end of the file.

The open() method

```
#opens the file file.txt in read mode
     >>> fileptr = open("file.txt","r")
      >>> if fileptr:
```

```
...    print("file is opened successfully")
```

Output:

```
file is opened successfully
```

In the above code, we have passed **filename as a first argument** and **opened file in read mode** as we mentioned “**r**” as the second argument. The **fileptr holds the file object** and if the file is opened successfully, it will execute the print statement

The close() method

Once all the operations are done on the file, we must close it through our Python script using the **close() method**. Any unwritten information gets destroyed once the close() method is called on a file object.

We can perform any operation on the file externally using the file system which is currently opened in Python; hence it is good practice to close the file once all the operations are done.

The **syntax to use the close() method** is given below.

Syntax

```
fileobject.close()
```

Consider the following example.

```
# opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
   print("file is opened successfully")

#closes the opened file
fileptr.close()
```

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

The with statement

The with statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using the **with** statement is given below.

```
with open(<file name>, <access mode>) as <file-pointer>:
        #statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always **suggestible to use the with statement** in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, **we don't need to write the close() function**. It doesn't let the file to corrupt.

Consider the following example.

Example

```
with open("file.txt",'r') as f:
     content = f.read()
     print(content)
```

Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

Example

```
     # open the file.txt in append mode. Create a new file if no such file exists.
     fileptr = open("file2.txt", "w")

     # appending the content to the file
     fileptr.write('''Python is the modern day language.
                     It makes things so simple.
                     It is the fastest-growing programing language''')

     # closing the opened the file
     fileptr.close()
```

Output:

file2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

Example 2

```
     #open the file.txt in write mode.
     fileptr = open("file2.txt","a")

     #overwriting the content of the file
     fileptr.write("Python has an easy syntax and user-friendly interaction.")

     #closing the opened file
     fileptr.close()
```

Output:

```
Python is the modern day language. It makes things so simple.
 It is the fastest growing programing language Python has an easy
 syntax and user-friendly interaction.
```

Read from a file

To read a file using the Python script, the Python provides the read() method. The read() method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

Syntax:

```
fileobj.read(<count>)
```

Here, the **count is the number of bytes to be read from the file starting from the beginning** of the file. **If the count is not specified**, then it may read the content of the file until the end.

Consider the following example.

Example

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r")
#stores first 10 byte data of the file into the variable content
content = fileptr.read(10)
# prints the type of the data stored in the file
print(type(content))
#prints the content of the file
print(content)
#closes the opened file
fileptr.close()
```

Output:

```
<class 'str'>
Python is
```

In the above code, we have read the content of file2.txt by using the read() function. We have passed count value as ten which means it will read the first ten characters from the file. If we use the following line, then it will print all content of the file.

```
content = fileptr.read()
print(content)
```

Output:

```
Python is the modern-day language. It makes things so simple.
It is the fastest-growing programing language Python has easy an
syntax and user-friendly interaction.
```

Read file through for loop

We can read the file using for loop. Consider the following example.

```
#open the file.txt in read mode. causes an error if no such file
exists.
     fileptr = open("file2.txt","r");
     #running a for loop
     for i in fileptr:
          print(i) # i contains each line of the file
```

Output:

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

Read Lines of the file

Python facilitates reading the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the

readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file "**file2.txt**" containing three lines. Consider the following example.

Example 1: Reading lines using readline() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");
#stores all the data of the file into the variable content
content = fileptr.readline()
content1 = fileptr.readline()
#prints the content of the file
print(content)
print(content1)
#closes the opened file
fileptr.close()
```

Output:

```
Python is the modern day language.

It makes things so simple.
```

We called the **readline()** function two times that's why it read two lines from the file.
Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2: Reading Lines Using readlines() function

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","r");
#stores all the data of the file into the variable content
content = fileptr.readlines()
#prints the content of the file
print(content)
  #closes the opened file
fileptr.close()
```

Output:
['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly interaction.']

Creating a new file

The new file can be created by using one of the following access modes with the function open().

x: it creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example 1

```
        #open the file.txt in read mode. causes error if no such file
exists.
        fileptr = open("file2.txt","x")
        print(fileptr)
        if fileptr:
            print("File created successfully")
```

Output:

<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>

File created successfully

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")
 #initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())
#reading the content of the file
content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

The syntax to use the seek() method is given below.

Syntax:

```
<file-ptr>.seek(offset[, from)
```

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

Example

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.
fileptr.seek(10);

#tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

Getting a Directory Listing

Suppose your current working directory has a subdirectory called `my_directory` that has the following contents:

```
my_directory/
|
├── sub_dir/
|   ├── bar.py
|   └── foo.py
|
├── sub_dir_b/
|   └── file4.txt
|
```

```
├── sub_dir_c/
|   ├── config.py
|   └── file5.txt
|
├── file1.py
├── file2.csv
└── file3.txt
```

The built-in `os` module has a number of useful functions that can be used to list directory contents and filter the results. To get a list of all the files and folders in a particular directory in the filesystem, use `os.listdir()` in legacy versions of Python or `os.scandir()` in Python 3.x. `os.scandir()` is the preferred method to use if you also want to get file and directory properties such as file size and modification date.

Directory Listing using *os.listdir()* # in Legacy Python Versions

In versions of Python prior to Python 3, ***os.listdir()*** is the method to use to get a directory listing:

```
>>> import os
>>> entries = os.listdir('my_directory/')
```

os.listdir() returns a Python list containing the names of the **files and subdirectories** in the directory given by the path argument:

```
>>> os.listdir('my_directory/')
['sub_dir_c', 'file1.py', 'sub_dir_b', 'file3.txt', 'file2.csv', 'sub_dir']
```

Directory Listing using *os.scandir()* # in Modern Python Versions

In modern versions of Python, an alternative to **os.listdir()** is to use **os.scandir()** and **pathlib.Path()**.

os.scandir() was introduced in Python 3.5 and it returns an iterator as opposed to a list when called:

```
>>> import os
>>> entries = os.scandir('my_directory/')
>>> entries
<posix.ScandirIterator object at 0x7f5b047f3690>
```

The `ScandirIterator` points to all the entries in the current directory. You can loop over the contents of the iterator and print out the filenames:

```
import os

with os.scandir('my_directory/') as entries:
    for entry in entries:
        print(entry.name)
```

Here, os.scandir() is used in conjunction with the with statement because it supports the context manager protocol. Using a context manager closes the iterator and frees up acquired resources automatically after the iterator has been exhausted. The result is a print out of the filenames in my_directory/ just like you saw in the os.listdir() example:

Directory listing using *pathlib* module

```
from pathlib import Path
entries = Path('my_directory/')
for entry in entries.iterdir():
    print(entry.name)
```

The objects returned by Path are either PosixPath or WindowsPath objects depending on the OS.

pathlib.Path() objects have an .iterdir() method for creating an iterator of all files and folders in a directory. Each entry

yielded by .iterdir() contains information about the file or directory such as its name and file attributes.

Listing All Files in a Directory

This section will show you how to print out the names of files in a directory using os.listdir(), os.scandir(), and pathlib.Path().

To filter out directories and only list files from a directory listing produced by os.listdir(), use os.path:

```
import os
# List all files in a directory using os.listdir
basepath = 'my_directory/'
for entry in os.listdir(basepath):
    if os.path.isfile(os.path.join(basepath, entry)):
        print(entry)
```

Here, the call to **os.listdir()** returns a list of everything in the specified path, and then that list is filtered by **os.path.isfile()** to only print out files and not directories.

output:

```
file1.py
file3.txt
file2.csv
```

An easier way to list files in a directory is to use os.scandir() or pathlib.Path():

```
import os
# List all files in a directory using scandir()
basepath = 'my_directory/'
with os.scandir(basepath) as entries:
    for entry in entries:
        if entry.is_file():
            print(entry.name)
```

Using `os.scandir()` has the advantage of looking cleaner and being easier to understand than using `os.listdir()`, even though it is one line of code longer. Calling `entry.is_file()` on each item in the `ScandirIterator` returns `True` if the object is a file. Printing out the names of all files in the directory gives you the following output:

```
file1.py
file3.txt
file2.csv
```

Here's how to list files in a directory using `pathlib.Path()`:

```
from pathlib import Path
basepath = Path('my_directory/')
files_in_basepath = basepath.iterdir()
for item in files_in_basepath:
        if item.is_file():
            print(item.name)
```

Here, you call .is_file() on each entry yielded by .iterdir(). The output produced is the same:

```
file1.py
file3.txt
file2.csv
```

The code above can be made more concise if you combine the for loop and the if statement into a single generator expression.

The modified version looks like this:

```
from pathlib import Path

# List all files in directory using pathlib
basepath = Path('my_directory/')
files_in_basepath = (entry for entry in basepath.iterdir() if entry.is_file())
for item in files_in_basepath:
    print(item.name)
```

This produces exactly the same output as the example before it. This section showed that filtering files or directories using os.scandir() and pathlib.Path() feels more intuitive and looks cleaner than using os.listdir() in conjunction with os.path.

Listing Subdirectories

To list subdirectories instead of files, use one of the methods below. Here's how to use os.listdir() and os.path():

```
import os
# List all subdirectories using os.listdir
basepath = 'my_directory/'
for entry in os.listdir(basepath):
    if os.path.isdir(os.path.join(basepath, entry)):
        print(entry)
```

Manipulating filesystem paths this way can quickly become cumbersome when you have multiple calls to **os.path.join()**.

```
sub_dir_c
sub_dir_b
sub_dir
```

Here's how to use **`os.scandir()`**:

```
import os

# List all subdirectories using scandir()
basepath = 'my_directory/'
with os.scandir(basepath) as entries:
    for entry in entries:
        if entry.is_dir():
            print(entry.name)
```

As in the file listing example, here you call **.is_dir()** on each entry returned by **os.scandir()**. If the entry is a directory, **.is_dir()** returns True, and the directory's name is printed out. The output is the same as above:

```
sub_dir_c
sub_dir_b
sub_dir
```

Here's how to use pathlib.Path():

```
from pathlib import Path
# List all subdirectory using pathlib
basepath = Path('my_directory/')
for entry in basepath.iterdir():
     if entry.is_dir():
          print(entry.name)
```

Calling .is_dir() on each entry of the basepath iterator checks if an entry is a file or a directory. If the entry is a directory, its name is printed out to the screen, and the output produced is the same as the one from the previous example:

sub_dir_c
sub_dir_b
sub_dir

The rename() file method

os.rename(current_file, new_file)
The <rename()> method takes two arguments, the current filename and the new filename.
Following is the example to rename an existing file <app.log> to <app1.log>.
Example:

```
import os

#Rename a file from <app.log> to <app1.log>
os.rename( "app.log", "app1.log" )
```

The remove() file method

os.remove(file_name)
The <remove()> method deletes a file which it receives in the argument.
Following is the example to delete an existing file, the <app1.log>.
Example:

```
import os

#Delete a file <app1.log>
os.remove( "app1.log" )
```