https://www.datacamp.com/community/tutorials/reading-writing-files-python

Hafsa Jabeen

April 11th, 2018

Reading and Writing Files in Python Tutorial

Learn how to open, read and write data into flat files, such as JSON and text files, as well as binary files in Python with the io and os modules.

As a data scientist, you'll surely work with a lot of data! You will receive this data from multiple sources, from databases, from Excel to flat files. You will need to know how to open, read and write data into a flat files so that you can perform analyses on them.

That's exactly what this tutorial will cover! You'll learn about:

- The Python file object;
- How to open a file, so that you can access the data;
- How to read data from a file;
- Once you have accessed the data, you can also close the file
- And also write data to a file;
- You'll also see some Python file object attributes,
- And take a look at other methods of the `File` object
- Lastly, you'll dig into the Python `os` module.

Tip: if you want to learn more about importing files in Python, check out DataCamp's Importing Data in Python course.

Flat Files Versus Text Files

To start with, you should first know what flat files are and how they differ from text files.

Flat files are data files that contain records with no structured relationships between the records and there's also no structure for indexing, like you typically find it in relational databases. These files can contain only basic formatting, have a small fixed number of fields, and can or can not have a file format.

A flat file can be a plain text file or a binary file. In the former case, the files usually contain one record per line:

- Comma Separated Values (CSV) files, which contain data values that are separated by `,` for example:

  ```
  NAME,ADDRESS,EMAIL

  ABC,CITY A,abc@xyz.com

  LMN,CITY B,lmn@xyz.com

  PQR,CITY C,pqr@xyz.com
  ```

- Delimited files, which contain data values with a user-specified delimiter. This can be a `\t` tab or a symbol (#,&,||), for example:

  ```
  NAME||ADDRESS||EMAIL

  ABC||CITY A||abc@xyz.com

  LMN||CITY B||lmn@xyz.com

  PQR||CITY C||pqr@xyz.com
  ```

But what does this mean for Python?

Python File Objects

Python has in-built functions to create and manipulate files. The `io` module is the default module for accessing files and you don't need to import it. The module consists of `open(filename, access_mode)` that returns a file object, which is called "handle". You can use this handle to read from or write to a file. Python treats the file as an object, which has its own attributes and methods.

As you already read before, there are two types of flat files, text and binary files:

1. As you might have expected from reading the previous section, text files have an End-Of-Line (EOL) character to indicate each line's termination. In Python, the new line character (`\n`) is default EOL terminator.

2. Since binary files store data after converting it into binary language (0s and 1s), there is no EOL character. This file type returns bytes. This is the file to be used when dealing with non-text files such as images or `exe`.

In this tutorial, you will focus more on text files.

`Open()`

The built-in Python function `open()` has the following arguments:

```
             `open(file, mode='r', buffering=-1, encoding=None, errors=None,
newline=None, closefd=True, opener=None)`
```

The above `open()` tells also the default values for each argument. First, let's get hands-on with reading and writing files with first two parameters `open(file, mode)` and go through other parameters one by one.

Note that, in this tutorial, `opener` will not be discussed because it is used for low-level I/O operations.

`file`

`file` is an argument that you have to provide to the `open` function. All other arguments are optional and have default values. Now, this argument is basically the path where your file resides.

If the path is in current working directory, you can just provide the filename, just like in the following examples:

```
my_file_handle=open("mynewtextfile.txt")
```

If the file resides in a directory other than that, you have to provide the full path with the file name:

```
my_file_handle=open("D:\\new_dir\\anotherfile.txt")

my_file_handle.read()
```

```
'Hi,\nI am in D Drive Datacamp folder'
```

Make sure file name and path given is correct, otherwise you'll get a `FileNotFoundError`:

```
my_file_handle=open("D:\\new_dir1\\anotherfile.txt")
```

```
---------------------------------------------------------------------------

FileNotFoundError                         Traceback (most recent call last)

<ipython-input-3-f10ef385d072> in <module>()

----> 1 my_file_handle=open("D:\\new_dir1\\anotherfile.txt")


FileNotFoundError: [Errno 2] No such file or directory: 'D:\\new_dir1\\anotherfile.txt'
```

You can catch the exception with a try-finally block:

```
try:

    my_file_handle=open("D:\\new_dir1\\anotherfile.txt")

except IOError:
    print("File not found or path is incorrect")
finally:
    print("exit")
```

Access Modes

Access modes define in which way you want to open a file, you want to open a file for read only, write only or for both. It specifies from where you want to start reading or writing in the file.

You specify the access mode of a file through the `mode` argument. You use `'r'`, the default mode, to read the file. In other cases where you want to write or append, you use `'w'` or `'a'`, respectively.

There are of course more access modes! Take a look at the following table:

Character	Function
r	Open file for reading only. Starts reading from beginning of file. This default mode.
rb	Open a file for reading only in binary format. Starts reading from beginning of file.
r+	Open file for reading and writing. File pointer placed at beginning of the file.
w	Open file for writing only. File pointer placed at beginning of the file. Overwrites existing file and creates a new one if it does not exists.
wb	Same as **w** but opens in binary mode.
w+	Same as **w** but also alows to read from file.
wb+	Same as **wb** but also alows to read from file.
a	Open a file for appending. Starts writing at the end of file. Creates a new file if file does not exist.
ab	Same as **a** but in binary format. Creates a new file if file does not exist.
a+	Same a **a** but also open for reading.
ab+	Same a **ab** but also open for reading.

As you have seen in the first section, there are two types of flat files and this is also why there's also an option to specify in which format you want to open file, such as text or binary. Of course, the former is the default.

Reading from a file

Let's try out all the reading methods for reading from a file and you will also explore the access modes along! There are three ways to read from a file.

- `read([n])`
- `readline([n])`
- `readlines()`

Note: that `n` is the number of bytes to be read.

Create a file as below:

```
1st line

2nd line

3rd line

4th line

5th line
```

Let's see what each read method does:

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")

my_file.read()
```

The `read()` method just outputs the entire file if number of bytes are not given in the argument. If you execute `my_file.read(3)`, you will get back the first three characters of the file

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")

my_file.read(3)
```

`readline(n)` outputs at most `n` bytes of a single line of a file. It does not read more than one line.

```
my_file.close()

my_file=open("D:\\new_dir\\multiplelines.txt","r")

#Use print to print the line else will remain in buffer and replaced by next
statement

print(my_file.readline())

# outputs first two characters of next line

print(my_file.readline(2))
```

Closing Python Files with `close()`

Use the `close()` method with file handle to close the file. When you use this method, you clear all buffer and close the file.

```
my_file.close()
```

You can use a `for` loop to read the file line by line:

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")

#Use print to print the line else will remain in buffer and replaced by next statement

for line in my_file:

    print(line)

my_file.close()
```

The `readlines()` method maintains a list of each line in the file:

```
my_file=open("D:\\new_dir\\multiplelines.txt","r")

my_file.readlines()
```

Writing to a file

You can use three methods to write to a file in Python:

- `write(string) (for text) or write(byte_string) (for binary)`
- `writelines(list)`

Let's create a new file within a folder in the "D" drive. Following will create a new file in the specified folder because it does not exist. **Remember** to give correct path with correct filename otherwise you will get error:

Create a notepad file and write some text in it. Make sure to save file as `.txt` and save it to the working directory of Python.

```
my_file_handle=open("mynewtextfile.txt")

my_file_handle.read()


new_file=open("D:\\new_dir\\newfile.txt",mode="w",encoding="utf-8")


new_file.write("Writing to a new file\n")

new_file.write("Writing to a new file\n")
new_file.write("Writing to a new file\n")
new_file.close()
```

Now let's write a list to this file with `a+` mode:

```
fruits=["Orange\n","Banana\n","Apple\n"]
new_file=open("D:\\new_dir\\newfile.txt",mode="a+",encoding="utf-8")
new_file.writelines(fruits)
for line in new_file:
    print(line)
new_file.close()
```

Note that reading from a file does not print anything because the file cursor is at the end of the file. To set the cursor at the beginning, you can use the `seek()` method of file object:

```
cars=["Audi\n","Bentely\n","Toyota\n"]
new_file=open("D:\\new_dir\\newfile.txt",mode="a+",encoding="utf-8")
for car in cars:
    new_file.write(car)
print("Tell the byte at which the file cursor is:",new_file.tell())
new_file.seek(0)
for line in new_file:
    print(line)
```

The `tell()` method of file object tells at which byte the file cursor is located. In `seek(offset,reference_point)` the reference points are `0` (the beginning of the file and is default), `1` (the current position of file) and `2` (the end of the file).

Let's try out passing another reference point and offset and see the output:

```
new_file.seek(4,0)
print(new_file.readline())
new_file.close()
```

Note the use of `.seek()` and `.truncate()`: the argument in `.truncate()` is `5` that says that truncate the file till 5 bytes of text are left. And output shows exactly 5 bytes of text left including space. You are only left with `next()` method so let's complete this section of the tutorial! Here you are using same file created above with name `multiplelines.txt`.

End-relative seeks such as `seek(-2,2)` are not allowed if file mode does not include `'b'`, which indicates binary format. Only forward operations such `seek(0,2)` are allowed when file object is dealt as text file.

```
file=open("D:\\new_dir\\multiplelines.txt","r")

for index in range(5):

    line=next(file)

    print(line)

file.close()
```

Note that `write()` doesn't actually write data to a file but to a buffer, it does, but only when the `close()` is called. This latter method flushes the buffer and writes the content to the file. If you wish to not close the file use `fileObject.flush()` method to clear buffer and write back to file.

You can also write your data to `.json` files.

Remember: Javascript Object Notation (JSON) has become popular method for exchange of structured information over a network and sharing information across platforms. It is basically text with some structure and saving it as `.json` tells how to read the structure otherwise it is just a plain text file. It stores data as key:value pairs. The structure can be simple to complex.

Take a look at following simple JSON for countries and their capitals:

```
{

"Algeria":"Algiers",

"Andorra":"Andorra la Vella",

"Nepal":"Kathmandu",

"Netherlands":"Amsterdam",

}
```

Anything before `:` is called key and after `:` is called value. This is very similar to Python dictionaries, isn't it! You can see that the data are separated by `,` and that curly braces define objects. Square brackets are used to define arrays in more complex JSON files, as you can see in the following excerpt:

```
{
  "colors": [
    {
      "color": "black",       "category": "hue",       "type": "primary",
      "code": {         "rgba": [255,255,255,1],         "hex": "#000"       }
    },
    {
      "color": "white",      "category": "value",
      "code": {        "rgba": [0,0,0,1],        "hex": "#FFF"      }
    },
    {
      "color": "red",      "category": "hue",      "type": "primary",
      "code": {        "rgba": [255,0,0,1],        "hex": "#FF0"      }
    },
    {
      "color": "blue",      "category": "hue",      "type": "primary",
      "code": {        "rgba": [0,0,255,1],        "hex": "#00F"      }
    },
    {
      "color": "yellow",      "category": "hue",      "type": "primary",
      "code": {        "rgba": [255,255,0,1],        "hex": "#FF0"      }
    },
    {
      "color": "green",      "category": "hue",      "type": "secondary",
      "code": {        "rgba": [0,255,0,1],        "hex": "#0F0"      }
    },
  ]
}
```

Note that JSON files can hold different data types in one object as well!

When you read the file with `read()`, you read strings from file. That means that when you read numbers, you would need to convert them to integers with data type conversion functions like `int()`. For more complex use cases, you can always use the `json` module.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
# Importing json module
import json
my_data=["Hafsa Jabeen","Reading and writing files in python",78546]
json.dumps(my_data)
```

To write the JSON in a file, you can use the `.dump()` method:

```
with open("jsonfile.json","w") as f:
    json.dump(my_data,f)
f.close()
```

Note: that it is a good practice to use with-open method to open a file because it closes the file properly if any exception is raised on the way. It is a good alternative to try-finally blocks.

The file has been made and the `json` package has been loaded. If a JSON file is opened for reading, you can decode it with `load(file)` as follows:

```
with open("jsonfile.json","r") as f:

    jsondata=json.load(f)

    print(jsondata)

f.close()
```

Similarly, more complex dictionaries can be stored using the `json` module. You can find more information here.

Now, you will see some other parameters of `open()` method, which you have already seen in the previous sections. Let's start with `buffering`.

`buffering`

A buffer holds a chunk of data from operating system's file stream until it is used upon which more data comes in, which is similar to video buffering.

Buffering is useful when you don't know the size of file you are working with, if the file size is greater than computer memory then processing unit will not function properly. The buffer size tells how much data can be held at a time until it is used.`io.DEFAULT_BUFFER_SIZE` can tell default buffer size of your platform.

Optionally, you can pass an integer to `buffering` to set the buffering policy:

1. `0` to switch off buffering (only allowed in binary mode)
2. `1` to select line buffering (only usable in text mode)
3. Any integer that is bigger than `1` to indicate the size in bytes of a fixed-size chunk buffer
4. Use negative values to set the buffering policy to the system default

When you don't specify any policy, the default is:

1. Binary files are buffered in fixed-size chunks
2. The size of the buffer is chosen depending on underlying device's "block size". On many systems, the buffer will typically be 4096 or 8192 bytes long.

3. “Interactive” text files (files for which `isatty()` returns `True`) use line buffering. Other text files use the policy described above for binary files. Note that `isatty()` can be used to see if you’re connected to a Tele-TYpewriter(-like) device.

```
import io

print("Default buffer size:",io.DEFAULT_BUFFER_SIZE)

file=open("mynewtextfile.txt",mode="r",buffering=5)

print(file.line_buffering)

file_contents=file.buffer

for line in file_contents:

    print(line)
```

Note that if you are using all arguments in the order that is specified in

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

, you don't need to write argument name! If you skip arguments because you want to keep the default values, it’s better to write everything out in full.

errors

An optional string that specifies how encoding and decoding errors are to be handled. This argument cannot be used in binary mode. A variety of standard error handlers are available (listed under Error Handlers).

```
file=open("mynewtextfile.txt",mode="r",errors="strict")

print(file.read())

file.close()
```

`errors="strict"` raises `ValueErrorException` if there is encoding error.

newline

`newline` controls how universal newlines mode works (it only applies to text mode). It can be None, '', '\n', '\r', and '\r\n'. In the example above, you see that passing `None` to `newline` translates `'\r\n'` to `'\n'`.

1. **None**:universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into default line separator
2. **" "**:universal newlines mode is enabled, but line endings are returned not translated
3. **'\n','\r', '\r\n'**:Input lines are only terminated by the given string, and the line ending is not translated.

Note that universal newlines are a manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention '\n', the Windows convention '\r\n', and the old Macintosh convention '\r'.

Note also that `os.linesep` returns the system's default line separator:

```
file=open("mynewtextfile.txt",mode="r",newline="")

file.read()


file=open("mynewtextfile.txt",mode="r",newline=None)

file.read()


file.close()
```

encoding

`encoding` represents the character encoding, which is the coding system that uses bits and byte to represent a character. This concept frequently pops up when you're talking about data storage, data transmission and computation.

As default encoding is operating system dependent for Microsoft Windows it is cp1252 but UTF-8 in Linux. So when dealing with text files, it is a good practice to specify the character encoding. Note that the binary mode doesn't take an `encoding` argument.

Earlier, you read that you can use the `errors` parameter to handle encoding and decoding error and that you use `newline` to deal with line endings. Now, try out the following code for these:

```
with open("mynewtextfile.txt",mode="r") as file:

    print("Default encoding:",file.encoding)

    file.close()

##change encoding to utf-8

with open("mynewtextfile.txt",mode="r",encoding="utf-8") as file:
    print("New encoding:",file.encoding)
    file.close()
```

closefd

If `closefd` is `False` and a file descriptor, rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given, `closefd` has to be set to `True`, which is the default. Otherwise, you'll probably get an error. You use this argument to wrap an existing file descriptor into a real file object.

Note that a file descriptor is simply an integer assigned to a file object by operating system so that Python can request I/O operations. The method `.fileno()` returns this integer.

If you have an integer file descriptor already open for a I/O channel you can wrap a file object around it as below:

```
file=open("mynewtextfile.txt","r+")
fd=file.fileno()
print("File descriptor assigned:",fd)

# Turn the file descriptor into a file object
filedes_object=open(fd,"w")
filedes_object.write("Data sciences\r\nPython")
filedes_object.close()
```

In this case `filedes_object` will close the underlying file object `file`. Closing `file` will give `OSError Bad file descriptor`:

```
file.close()
```

To prevent closing the underlying file object, you can use `closefd=False`:

```
file=open("mynewtextfile.txt","r+")
fd=file.fileno()
print("File descriptor assigned:",fd)

# Turn the file descriptor into a file object
filedes_object=open(fd,"w",closefd=False)
```

```
filedes_object.write("Hello")

filedes_object.close()

file.close()
```

Up until now, you have learned pretty much all about reading text files in Python, but as you have read many times throughout this tutorial, these are not the only files that you can import: there's also binary files.

But what are these binary files exactly?

Binary files store data in `0's` and `1's` that is machine readable. A byte is collection of 8-bits. One character stores one byte in the memory that is 8-bits. For example, the binary representation of character 'H' is `01001000` and convert this 8-bit binary string into decimal gives you `72`.

```
binary_file=open("D:\\new_dir\\binary_file.bin",mode="wb+")

text="Hello 123"

encoded=text.encode("utf-8")

binary_file.write(encoded)

binary_file.seek(0)

binary_data=binary_file.read()

print("binary:",binary_data)

text=binary_data.decode("utf-8")

print("Decoded data:",text)
```

When you open a file for reading in binary mode `b`, it returns bytes of data.

If you ever need to read or write text from a binary-mode file, make sure you remember to decode or encode it like above. You can access each byte through iteration like below and it will return integer byte values (decimal of the 8-bit binary representation of each character) instead of byte strings:

```
for byte in binary_data:

    print(byte)
```

Python File Object Attributes

File attributes give information about the file and file state.

Attribute	Function
`name`	Returns the name of the file
`closed`	Returns true if file is closed. False otherwise.

Attribute	Function
`mode`	The mode in which file is open.
`softspace`	Returns a Boolean that indicates whether a space character needs to be printed before another value when using the print statement.
`encoding`	The encoding of the file

```
# This is just another way you can open  a file
with open("D:\\new_dir\\anotherfile.txt") as file:
    print("Name of the file:",file.name)
    print("Mode of the file:",file.mode)
    print("Mode of the file:",file.encoding)
    file.close()
print("Closed?",file.closed)
```

Other Methods of File object

Method	Function
`readable()`	Returns `True`/`False` whether file is readable
`writable()`	Returns `True`/`False` whether file is writable
`fileno()`	Return the Integer descriptor used by Python to request I/O operations from Operating System
`flush()`	Clears the internal buffer for the file.
`isatty()`	Returns `True` if file is connected to a Tele-TYpewriter (TTY) device or something similar.
`truncate([size)`	Truncate the file, up to specified bytes.
`next(iterator, [default])`	Iterate over a file when file is used as an iterator, stops iteration when reaches end-of-file (EOF) for reading.

Let's try out all of these methods:

```
with open("mynewtextfile.txt","w+") as f:
    f.write("We are learning python\nWe are learning python\nWe are learning python")
    f.seek(0)
    print(f.read())
    print("Is readable:",f.readable())
    print("Is writeable:",f.writable())
    print("File no:",f.fileno())
    print("Is connected to tty-like device:",f.isatty())
    f.truncate(5)
    f.flush()
    f.seek(0)
    print(f.read())
f.close()
```

Handling files through `os` module

The `os` module of Python allows you to perform Operating System dependent operations such as making a folder, listing contents of a folder, know about a process, end a process etc. It has methods to view environment variables of the Operating System on which Python is working on and many more. Here is the Python documentation for the `os` module.

Let's see some useful `os` module methods that can help you to handle files and folders in your program.

Method	Function
`os.makedirs()`	Create a new folder
`os.listdir()`	List the contents of a folder
`os.getcwd()`	Show current working directory
`os.path.getsize()`	show file size in bytes of file passed in parameter
`os.path.isfile()`	Is passed parameter a file
`os.path.isdir()`	Is passed parameter a folder
`os.chdir`	Change directory/folder
`os.rename(current,new)`	Rename a file
`os.remove(file_name)`	Delete a file

Let's see some examples of these methods:

```
import os

os.getcwd()

'C:\\Users\\HAFSA-J'

os.makedirs("D:\\my_folder")

---------------------------------------------------------------------------

FileExistsError                           Traceback (most recent call last)

<ipython-input-6-346b12771465> in <module>()

----> 1 os.makedirs("D:\\my_folder")



C:\Users\HAFSA-J\Anaconda3\lib\os.py in makedirs(name, mode, exist_ok)

    218             return

    219     try:

--> 220         mkdir(name, mode)

    221     except OSError:

    222         # Cannot rely on checking for EEXIST, since the operating system



FileExistsError: [WinError 183] Cannot create a file when that file already exists: 'D:\\my_folder'
```

The next code chunk will create a folder named My_folder in the D Drive:

```
open("D:\\my_folder\\newfile.txt","w")

print("Contents of folder D:\\my_folder\n",os.listdir("D:\\my_folder"))

print("---------------------------------")

print("Size of folder D:\my_folder (in bytes)",os.path.getsize("D:\\my_folder"))

print("Is file?",os.path.isfile("D:\\new_dir"))

print("Is folder?",os.path.isdir("D:\\new_dir\\anotherfile.txt"))
```

```
os.chdir("D:\\my_folder")

os.rename("newfile.txt","hello.txt")

print("New Contents of folder D:\\my_folder\n",os.listdir("D:\\my_folder"))
```

```
Contents of folder D:\my_folder

 ['hello.txt', 'newfile.txt']

--------------------------------

Size of folder D:\my_folder (in bytes) 0

Is file? False

Is folder? False

---------------------------------------------------------------------------

FileExistsError                           Traceback (most recent call last)


<ipython-input-9-4884bcddc38c> in <module>()

      6 print("Is folder?",os.path.isdir("D:\\new_dir\\anotherfile.txt"))

      7 os.chdir("D:\\my_folder")

----> 8 os.rename("newfile.txt","hello.txt")

      9 print("New Contents of folder D:\\my_folder\n",os.listdir("D:\\my_folder"))



FileExistsError: [WinError 183] Cannot create a file when that file already exists: 'newfile.txt' -> 'hello.txt'
```

If you create a filename that already exists Python will give `FileExistsError` error. To delete a file use, you can use `os.remove(filename)`:

```
os.getcwd()

os.remove("hello.txt")

print("New Contents of folder D:\\my_folder\n",os.listdir("D:\\my_folder"))

os.chdir("C:\\Users\\Hafsa-J")
```

This was all about flat files! if you want to learn how to import data from an Excel sheet check out this tutorial.

Wohooo!

Before you go, you should definitely check out Python’s awesome data manipulation library`pandas`: check out this tutorial for more great ways to handle data.

That's the end of the tutorial! Now you know how to handle files in Python and their manipulation from creation to operating system level handling.