Aditya Sharma
May 25th, 2020

PYTHON

Reading and Writing Files in Python

Learn how to read and write data into flat files, such as CSV, JSON, text files, and binary files in Python using io and os modules.

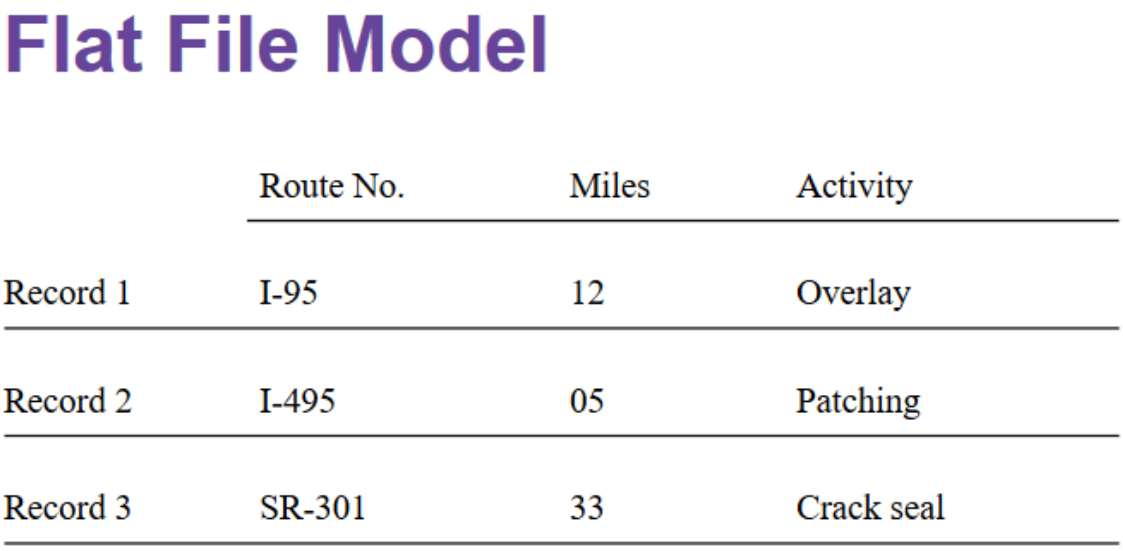
As a data scientist, you handle a lot of data daily. And this data could be from multiple sources like from databases, from Excel to flat files, from public websites like kaggle. Not just sources it could be in any file format like `.csv`, `.txt`, `.parquet`, etc. Before you start making sense of the data, you will need to know the basic three things: how to open, read and write data into flat files so that you can then perform analyses on them.

You would also learn about the following topics in this tutorial:

- Python file object
- How to `open` a basic flat file like `.csv`, `json`, etc. and `read` data from a file
- Write data to a file
- You'll also see some Python file object attributes
- You would also dig into the Python os module
- You would also learn about the `NumPy` library and how it can be used to import `Image` datasets

Flat Files vs. Non-Flat 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.



Source

Though in both flat and non-flat files, the data is usually in a tabular row-column fashion.

A non-flat file is a file where an index is assigned to every record. The exact location of the record can be known using the index of that record. You would normally need some applications like a database management system to read this type of file.

`XML` is an example of a `non-flat` file.

A flat file can be a plain text file having a `TSV`, `CSV` format, or a binary file format. 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
```

`\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
```

Let's now understand how Python creates and reads these types of file formats having specific delimiters.

Python File Objects

Python has in-built functions to create, read, write, and manipulate accessible files. The `io` module is the default module for accessing files that can be used off the shelf without even importing it. Before you read, write, or manipulate the file, you need to make use of the module `open(filename, access_mode)` that returns a file object called "handle". After which you can simply use this handle to read from or write to a file. Like everything else, files in Python are also treated as an object, which has its own attributes and methods.

An IOError exception is raised if, while opening the file, the operation fails. It could be due to various reasons like trying to read a file that is opened in write mode or accessing a file that is already closed.

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

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

Let's now understand the Python file objects in detail, along with necessary examples.

Open()

The `open()` function has almost 8 parameters along with their default values for each argument as shown above.

You would be focusing on the first and second parameters for now, which are essential for reading and writing files. And go through other parameters one by one as the tutorial progresses.

Let's understand the first argument, i.e., `file` .

file

`file` is a mandatory argument that you have to provide to the `open` function while rest all arguments are optional and use their default values.

To put it simply, the `file` argument represents the path where your file resides in your system.

If the path is in the current working directory, you can just provide the filename. If not then you have to provide the absolute path of the file, just like in the following examples: my_file_handle=open("mynewtextfile.txt") If the file resides in a directory other than the current directory, you have to provide the absolute path with the file name:

```
my_file_handle=open("D://test.txt")
my_file_handle.read()
```

```
"Welcome to DataCamp's Tutorial on Reading and Writing Files in Python!"
```

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

```
my_file_handle=open("folder/test.txt")
my_file_handle.read()
```

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

```
<ipython-input-2-a0d1ea891918> in <module>
----> 1 my_file_handle=open("folder/test.txt")
      2 my_file_handle.read()


FileNotFoundError: [Errno 2] No such file or directory: 'folder/test.txt'
```

Exception Handling in files

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

```
try:
    my_file_handle=open("folder/test.txt")
except IOError:
    print("File not found or path is incorrect")
finally:
    print("exit")


File not found or path is incorrect
exit
```

Let's understand the second argument of the `open` function, i.e., `access modes` .

Access Modes

Access modes define in which way you want to open a file, whether you want to open a file in:

- read-only mode
- write-only mode
- append mode
- both read and write mode

Though a lot of access modes exist as shown in the below table, the most commonly used ones are read and write modes. It specifies where you want to start reading or writing in

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. This is also why there's an option to specify which format you want to open, such as text or binary. Of course, the former is the default. When you add `'b'` to the access modes, you can read the file in binary format rather than the default text format. It is used when the file to be accessed is not in text format.

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 with it! There are three ways to read from a file.

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

Here `n` is the number of bytes to be read. If nothing is passed to `n`, then the complete file is considered to be read.

Create a file as below: 1st line 2nd line 3rd line 4th line 5th line Let's understand what each read method does:

```
my_file=open("test1.txt","r")
print(my_file.read())
```

```
1st line
2nd line
3rd line
4th line
5th line
```

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

```
my_file=open("test1.txt","r")
print(my_file.read(3))
```

```
1st
```

`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("test1.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))
```

```
2n
```

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("test1.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()
```

```
1st line

2nd line

3rd line

4th line

5th line
```

The `readlines()` method maintains a list of each line in the file which can be iterated using a for loop:

```
my_file=open("test1.txt","r")
my_file.readlines()
```

```
['1st line\n', '2nd line\n', '3rd line\n', '4th line\n', '5th line']
```

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. The 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 an error:

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

```
new_file=open("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()
```

Append Mode

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

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

Seek Method

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:

```
new_file=open( "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)
```

```
Tell the byte at which the file cursor is: 115
Writing to a new file

Writing to a new file

Writing to a new file

Orange

Banana

Apple

Audi

Bentley

Toyota
```

The `tell()` method of a 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 the 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()
```

next Method

You are only left with the `next()` method, so let's complete this section of the tutorial! Here you are using the same file created above with the name `test1.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 as `seek(0,2)` are allowed when the file object is dealt with as a text file.

```
file=open("test1.txt","r")
for index in range(5):
    line=next(file)
    print(line)
file.close()
```

```
1st line

2nd line

3rd line

4th line

5th line
```

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

Importing the Moby Dick Novel

Moby Dick is an 1851 novel by American writer Herman Melville. You'll be working with the file moby_dick.txt. It is a text file that contains the opening sentences of Moby Dick, one of the great American novels! Here you'll get experience opening a text file, printing its contents, and, finally, closing it.

You will do the following things:

- Open the moby_dick.txt file in read-only mode and store it in the variable file
- Print the contents of the file
- Check whether the file is closed
- Close the file using the close() method
- Check again whether the file is closed

```
# Open a file: file
file = open('moby_dick.txt', 'r')

# Print it
print(file.read())
print('\n')
# Check whether file is closed
print('Is the file closed?:', file.closed)

# Close file
file.close()
print('\n')
# Check whether file is closed
print('Is the file closed?:', file.closed)
```

```
CHAPTER 1. Loomings.

Call me Ishmael. Some years ago--never mind how long precisely--having little or no money i
the world. It is a way I have of driving off the spleen and regulating the circulation. Whe
such an upper hand of me, that it requires a strong moral principle to prevent me from deli
take to the ship. There is nothing surprising in this. If they but knew it, almost all men


Is the file closed?: False


Is the file closed?: True
```

Reading the Moby Dick Novel using Context Manager

You can bind a file object by using a context manager construct, and you don't need to worry about closing the file. The file can not be accessed outside the context manager and is deemed closed.

Let's print the first three lines of the moby dick text file using the `readline()` method. Note that the file is opened by default in a `read` mode.

```
with open('moby_dick.txt') as file:
    print(file.readline())
    print(file.readline())
    print(file.readline())
```

```
CHAPTER 1. Loomings.



Call me Ishmael. Some years ago--never mind how long precisely--having
```

Writing to a JSON File

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

Remember: Javascript Object Notation (JSON) has become a popular method for the 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 the following simple JSON for countries and their capitals:

```
{
"Algeria":"Algiers",
"Andorra":"Andorra la Vella",
"Nepal":"Kathmandu",
```

Since JSON consists of an array of `key: value` pairs as shown in below code cell, 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 is 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",
```

```
        "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 a 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=["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: It is good practice to use the with-open method to open a file because it closes the file properly if any exception is raised on the way.

Let's now open the `JSON` file you created using the `dump` method. 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()
```

```
['Reading and writing files in python', 78546]
```

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 the `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 the 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 the file you are working with if the file size is greater than computer memory then the 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 the default buffer size of your platform.

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

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

- Binary files are buffered in fixed-size chunks
- The size of the buffer is chosen depending on the underlying device's "block size". On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "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("test1.txt",mode="r",buffering=5)
print(file.line_buffering)
file_contents=file.buffer
for line in file_contents:
    print(line)
```

```
Default buffer size: 8192
False
b'1st line\r\n'
b'2nd line\r\n'
b'3rd line\r\n'
b'4th line\r\n'
b'5th line'
```

`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("test1.txt",mode="r",errors="strict")
print(file.read())
file.close()
```

```
1st line
2nd line
3rd line
4th line
5th line
```

`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'`.

- 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
- " ":universal newlines mode is enabled, but line endings are returned not translated
- '\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

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

```
file=open("test1.txt",mode="r",newline="")
file.read()
```

```
'1st line\r\n2nd line\r\n3rd line\r\n4th line\r\n5th line'
```

```
file=open("test1.txt",mode="r",newline=None)
file.read()
```

```
'1st line\n2nd line\n3rd line\n4th line\n5th line'
```

```
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("test1.txt",mode="r") as file:
    print("Default encoding:",file.encoding)
    file.close()
##change encoding to utf-8
with open("test1.txt",mode="r",encoding="utf-8") as file:
    print("New encoding:",file.encoding)
    file.close()
```

```
Default encoding: cp1252
New encoding: utf-8
```

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 the 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 an I/O channel you can wrap a file object around it as below:

```
file=open("test1.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()
```

```
File descriptor assigned: 6
```

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

```
file=open("test1.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")
```

```
File descriptor assigned: 6
```

You have learned a lot about reading text files in Python, but as you have read repeatedly throughout this tutorial, these are not the only files that you can import: there are also binary files.

But what are these binary files exactly?

Binary files store data in `0's` and `1's` that are machine-readable. A byte is a 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("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)
```

```
binary: b'Hello 123'
Decoded data: Hello 123
```

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:

```
    print(byte)
```

```
72
101
108
108
111
32
49
50
51
```

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.
`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("test1.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)
```

```
Name of the file: test1.txt
Mode of the file: r
```

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()
```

```
We are learning python
We are learning python
We are learning python
Is readable: True
Is writeable: True
```

```
We ar
```

Handling files through the 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\\hda3kor\\Documents\\Reading_and_Writing_Files'
```

```
os.makedirs("my_folder")
```

```
FileExistsError                           Traceback (most recent call last)

<ipython-input-12-f469e8a88f1b> in <module>
----> 1 os.makedirs("my_folder")


C:\Program Files\Anaconda3\lib\os.py in makedirs(name, mode, exist_ok)
    219             return
    220     try:
--> 221         mkdir(name, mode)
    222     except OSError:
    223         # Cannot rely on checking for EEXIST, since the operating system


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

The next code chunk will create a folder named my_folder:

```
open("my_folder\\newfile.txt","w")
print("Contents of folder my_folder\n",os.listdir("my_folder"))
print("---------------------------------")
print("Size of folder my_folder (in bytes)",os.path.getsize("my_folder"))
print("Is file?",os.path.isfile("test1.txt"))
print("Is folder?",os.path.isdir("my_folder"))
os.chdir("my_folder")
os.rename("newfile.txt","hello.txt")
print("New Contents of folder my_folder\n",os.listdir("my_folder"))
```

```
Contents of folder my_folder
 ['hello.txt', 'newfile.txt']
---------------------------------
Size of folder my_folder (in bytes) 0
Is file? True
Is folder? True
```

```
FileExistsError                           Traceback (most recent call last)

<ipython-input-13-6d2da66512fd> in <module>
      6 print("Is folder?",os.path.isdir("my_folder"))
      7 os.chdir("my_folder")
----> 8 os.rename("newfile.txt","hello.txt")
      9 print("New Contents of folder my_folder\n",os.listdir("my_folder"))


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

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")
```

Importing flat files using NumPy

Numerical Python, or more commonly known as NumPy arrays, is the Python standard for storing numerical data. They are efficient, fast, and clean. They are widely used in linear algebra, statistics, machine learning, and deep learning. NumPy arrays act as a backbone for reading image datasets.

It is also useful for packages like `Pandas` and `Scikit-learn` . NumPy consists of a lot of built-in functions which can be leveraged to do data analysis, manipulation: efficiently

MNIST data

The sample MNIST `.csv` dataset can be downloaded from here/_datasets/mnist_kaggle_some_rows.csv).

You can find more information about the MNIST dataset from here on the webpage of Yann LeCun.

You will first import the NumPy module and then use the `loadtxt` method to import the MNIST data, as shown below:

```
import numpy as np
data = np.loadtxt('mnist.csv', delimiter=',')
print(data)
```

```
[[1. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]
 ...
 [2. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [5. 0. 0. ... 0. 0. 0.]]
```

If your dataset has a header with string values, you can use the `skiprows` parameter and skip the first row. Similarly, you can use the `usecols` parameter to read only some specific columns.

You can also pass in the `dtype`, i.e., datatype in which you want to import your data either integer, float, string, etc.

Note that NumPy arrays are capable of handling only one type of datatype, meaning it cannot have mixed data types in a single array.

Let's check the number of rows and columns this dataset has:

```
data.shape
```

If you would like to learn more great ways to handle data in Python then check out this tutorial.

Conclusion

Congratulations on finishing the tutorial.

Now you know how to handle files in Python and their manipulation from creation to operating system level handling.

You might want to try experimenting with various NumPy functionalities that could be leveraged to understand numerical and imagery datasets. You could further analyze the dataset graphically using the Matplotlib plotting library.

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