

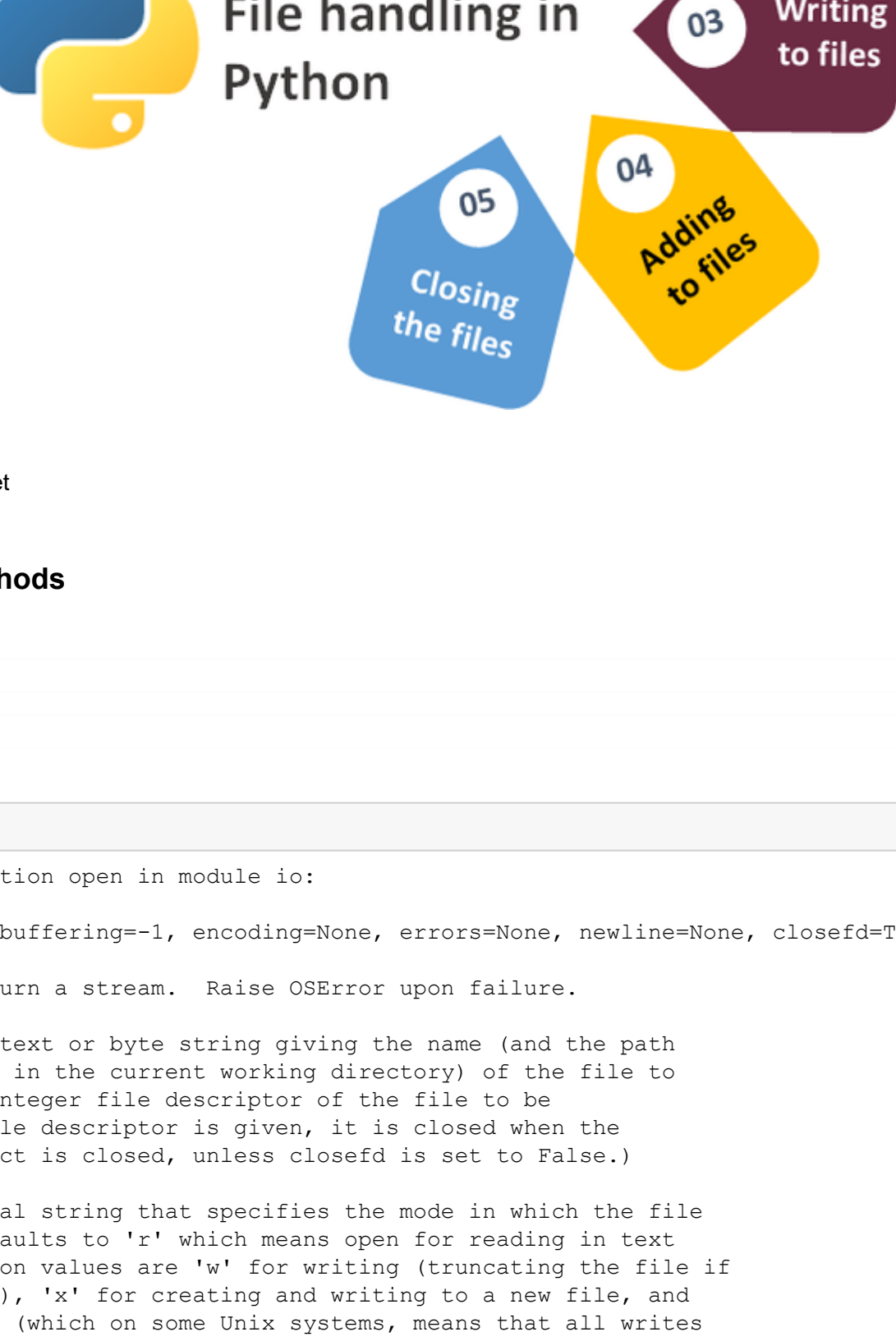
File Handling in Python

File handling in python is simply **reading** and **writing** a file.

What exactly are we writing into file?


We can write anything we want. But whatever we write - is termed as **data** ; the crucial thing for understanding the resources.


Procedure to handle files



Credits - Image from Internet

Main functions / methods

`open()`  function

`close()`  method

In [1]: `help(open)`

Help on built-in function open in module io:

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

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped (I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

=====

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+''	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

=====

The default mode is 'rt' (open for reading text). For binary random access, the mode 'rb+' opens and truncates the file to 0 bytes, while 'rb+' opens the file without truncation. The 'x' mode implies 'w' and raises an 'FileExistsError' if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

* Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on 'io.DEFAULT_BUFFER_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.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

errors is an optional argument that specifies how encoding errors are to be handled---this argument should not be used in binary mode. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register or run 'help(codecs.Codec)' for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', '\n', '\r', and '\r\n'. It works as follows:

* On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in '\n', '\r', or '\r\n', and these are translated into '\n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.

* On output, if newline is None, any '\n' characters written are translated to the system default line separator, os.linesep. If newline is '' or '\n', no translation takes place. If newline is any of the other legal values, any '\n' characters written are translated to the given string.

If closefd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

A custom opener can be used by passing a callable as "opener". The underlying file descriptor for the file object is then obtained by calling "opener" with ("file", "flags"). "opener" must return an open file descriptor (passing os.open as "opener" results in functionality similar to passing None).

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

Well known modes to handle files

- **r** - reading
- **w** - writing
- **a** - appending
- **+** - for both reading and writing

Reading

In [2]: `my_file = open(file='basic_sample.txt', mode='r')`

#####

`data = my_file.read()`

#####

`my_file.close()`

In [3]: `# print data`

`print(len(data))`

0

The data is **Nil** because there is no content in it. Let's add some content in the file.

- content to add → **Python is superb**
- Open the file and type manually

In [4]: `my_file = open(file='basic_sample.txt', mode='r')`

#####

`data = my_file.read()`

#####

`my_file.close()`

In [5]: `# print data`

`print(len(data))`

`print(data)`

`print("type - ", type(data))`

16

Python is superb

type = <class 'str'>

Writing

Let's write something in a new file. We use the mode **w** for writing into a file.

- content to write → **Python is amazing for data science**
- Writing happens from python code

In [6]: `my_file = open(file='writing_sample.txt', mode='w')`

#####

`my_file.write('Python is amazing for data science')`

#####

`my_file.close()`

Let's read `writing_sample.txt` in python

In [7]: `my_file = open(file='writing_sample.txt', mode='r')`

#####

`data = my_file.read()`

#####

`my_file.close()`

In [8]: `# print data`

`print(len(data))`

`print(data)`

34

Python is amazing for data science

Appending

In [9]: `my_file = open(file='append_sample.txt', mode='a')`

#####

`my_file.write("Simple is better than complex")`

`my_file.write("\nComplex is better than complicated")`

#####

`my_file.close()`

When used **a** -append mode, make sure to provide **\n** to make the data written in next line, otherwise it will consider to continue writing in the same line.

Every time we open a file, we have to make sure it is closed as well. We do this manually and sometimes we may forget to close the file which is the most essential thing not to forget. Is there any alternative??

If used **with** statement, we need not close the file manually. It will automatically close the file when the entire process is finished.

Example of using with

In [10]: `with open(file='writing_sample.txt', mode='r') as my_file:`

`data = my_file.read()`

`dl = data.lower()`

`print(dl)`

python is amazing for data science

In [11]: `# print data`

`print(len(data))`

`print(data)`

34

Python is amazing for data science

We can apply the same modes like **a**, **w**, **r** and so on.

But, what about readlines () function?

In [12]: `with open(file='sheep_rhyme.txt', mode='r') as sheep:`

`poem = sheep.readlines()`

In [13]: `poem`

Out[13]: ['Baa, baa, black sheep,\n',
'Have you any wool?\n',
'Yes sir, yes sir,\n',
'Three bags full,\n',
'One for the master,\n',
'One for the dame,\n',
'One for the little boy\n',
'Who lives down the lane.\n',
'Baa, baa, white sheep,\n',
'Have you any wool?\n',
'Yes sir, yes sir,\n',
'Three bags full,\n',
'One for the master,\n',
'One for the dame,\n',
'And one for the little Girl\n',
'Who lives down the lane.\n',
'Baa, baa, brown sheep,\n',
'Have you any wool?\n',
'Yes sir, yes sir,\n',
'Three bags full,\n',
'One for the master,\n',
'One for the dame,\n',
'And one for the Old Man\n',
'Who lives down the lane.']

Alternative - .split ()

In [14]: `with open(file='sheep_rhyme.txt', mode='r') as sheep:`

`p_ = sheep.read()`

In [15]: `p_`

Out[15]: ['Baa, baa, black sheep,\nHave you any wool?\nYes sir, yes sir,\n\nThree bags full,\n\nOne for the master,\n\nOne for the dame,\n\nOne for the little boy\n\nWho lives down the lane,\n\nBaa, baa, white sheep,\n\nHave you any wool?\n\nYes sir, yes sir,\n\nThree bags full,\n\nOne for the master,\n\nOne for the dame,\n\nAnd one for the little Girl\n\nWho lives down the lane,\n\nBaa, baa, brown sheep,\n\nHave you any wool?\n\nYes sir, yes sir,\n\nThree bags full,\n\nOne for the master,\n\nOne for the dame,\n\nAnd one for the Old Man\n\nWho lives down the lane.']

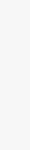
In [16]: `p_.split('\n')`

Out[16]: ['Baa, baa, black sheep,',
'Have you any wool?',
'Yes sir, yes sir,',
'Three bags full,',
'One for the master,',
'One for the dame,',
'One for the little boy',
'Who lives down the lane.',
'Baa, baa, white sheep,',
'Have you any wool?',
'Yes sir, yes sir,',
'Three bags full,',
'One for the master,',
'One for the dame,',
'And one for the little Girl',
'Who lives down the lane.',
'Baa, baa, brown sheep,',
'Have you any wool?',
'Yes sir, yes sir,',
'Three bags full,',
'One for the master,',
'One for the dame,',
'And one for the Old Man',
'Who lives down the lane.']

In [17]: `type(poem)`

Out[17]: list

Alice and the wonderland

 Image by author

In [18]: `with open(file='alice_wonderland.txt', mode='r') as alice:`

`alice_story = alice.readlines()`

In [19]: `# display first 10 items`

`alice_story[:10]`

Out[19]: ["Alice's Adventures in Wonderland\n",
'
'ALICE'S ADVENTURES IN WONDERLAND\n',
'
'Lewis Carroll\n',
'
'THE MILLENNIUM FULCRUM EDITION 3.0\n',
'
'
'CHAPTER I\n',
'Down the Rabbit-Hole\n',
'Alice was beginning to get very tired of sitting by her sister\n',
'on the bank, and of having nothing to do: once or twice she had\n',
'peeped into the book her sister was reading, but it had no\n',
'pictures or conversations in it, 'and what is the use of a book,'"]

In [20]: `# check length`

`len(alice_story)`

Out[20]: 3600

Data cleaning

In [21]: `# remove "\n" (single item)`

`non_slashes = []`

`for each_line in alice_story:`

`if each_line != '\n':`

`non_slashes.append(each_line)`

In [22]: `# display first 10 items`

`non_slashes[:10]`

Out[22]: ["Alice's Adventures in Wonderland\n",
'
'ALICE'S ADVENTURES IN WONDERLAND\n',
'
'Lewis Carroll\n',
'
'THE MILLENNIUM FULCRUM EDITION 3.0\n',
'
'
'CHAPTER I\n',
'Down the Rabbit-Hole\n',
'Alice was beginning to get very tired of sitting by her sister\n',
'on the bank, and of having nothing to do: once or twice she had\n',
'peeped into the book her sister was reading, but it had no\n',
'pictures or conversations in it, 'and what is the use of a book,'"]

In [23]: `# check length`

`len(non_slashes)`

Out[23]: 2726

In [24]: `# remove "\n" from each item at the end`

`# remove extra spaces from each item (at start and at end)`

`# convert each item into lower case`

`non_slash_new_lines = []`

`for each_line in non_slashes:`

`each_line = each_line.replace("\n", "").strip().casefold()`

`non_slash_new_lines.append(each_line)`

In [25]: `# display first 10 items`

`non_slash_new_lines[:10]`

Out[25]: ["alice's adventures in wonderland",
'alice's adventures in wonderland',
'lewis carroll',
'the millennium fulcrum edition 3.0',
'chapter i',
'down the rabbit-hole',
'alice was beginning to get very tired of sitting by her sister',
'on the bank, and of having nothing to do: once or twice she had',
'peeped into the book her sister was reading, but it had no',
'pictures or conversations in it, 'and what is the use of a book,'"]

In [26]: `# check length`

`len(non_slash_new_lines)`

Out[26]: 2726

 Image by author

In [27]: `# show enumerate example with list`

`d = [1, 3, 43, 100]`

`f = dict(enumerate(d))`

`f.items()`

Out[27]: dict_items([(0, 1), (1, 3), (2, 43), (3, 100)])

In [28]: `# make a dictionary with enumerate and count the word`

`line_dict = {}`

`my_word = 'a'`

`my_word = 'i' + my_word + '`

`for index, line in dict(enumerate(non_slash_new_lines)).items():`

`line_dict[index + 1] = line.count(my_word)`

`for i in range(1, 11):`

`print(i, "=", line_dict[i])`

1 → 0
2 → 0
3 → 0
4 → 0
5 → 0
6 → 0
7 → 0
8 → 0
9 → 0
10 → 1

In [29]: `# plot the result for the first 100 items`

`X = list(line_dict.keys())`

`Y = list(line_dict.values())`

`# py -m pip install matplotlib`

`# py -m pip install plotly`

`from matplotlib import pyplot as plt`

`plt.plot(X, y)`

`plt.show()`

In [31]: `True`

How to debug errors?

- Know about the errors
- Learn what errors mean
- Errors are the new beginning to learn programming

1. Do not google the error exactly as it is. But know where it occurred, why it occurred and the reason behind its occurrence.

2. Then google about the exact reason how to tackle the same.

Things that can be developed

- **Web Development**
 - Flask
 - Dash
 - Django
- **Game Development**
 - Pygame
 - OpenGL
- **Numeric & Scientific Computation**
 - NumPy
 - SciPy
 - OpenCV
 - Pandas
- **Data Analysis**
 - Pandas
 - Statmodel
 - Statistics
- **Data Visualization**
 - Matplotlib
 - Plotly
 - Bokeh
- **Machine Learning and Artificial Intelligence (Research & Development)**
 - Scikit-learn
 - Tensorflow
 - Pytorch
 - Keras
 - Skikit-image
- **Mobile App Development**
 - Kivy
- **Desktop Application**
 - Tkinter
- **Web Scraping**
 - BeautifulSoup
 - Selenium (Automation)
 - Scrapy

The list goes on ...

“ Always try to learn something new everyday ”

-Lex Fridman