Python: Files

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts of Southern Switzerland

Files

- Files (archives) are data structures that allow you to permanently store information on the hard disk (or other permanent media).
- The files are managed by the operating system.
- The files are organized in the file system.
- Different operating systems use different file systems (ex: Linux: ext2; WinNT: NTFS; DOS: FAT32; MacOS: MacOS extended).
- The management of the files of a programming language (cross-platform) must mask the file system as much as possible.

Files in Python

The python standard library gives us many features to work with files.

It is for example possible to:

- check if the file or directory exists
- check whether it is a file or a directory
- create a directory if it does not exist
- read the length of a file
- rename or move a file
- delete a file
- request the list of files present in a directory

Working with data files

Why do we need to read and write files from our Python programs?

A large amount of different data (weather, traffic,...) can be retrieved from textual files that are available on the internet or that are generated by some device.

For a data scientist, retrieving, manipulating and analyzing data contained in files are very important tasks.

As part of this course, we will look at some simple ways to handle open, read and write files.

Opening a file

In order to access a file it is first necessary to use the `open(filename, mode)` function to create a Python file object.

The first argument is the name of the file, to be provided using absolute or relative directory paths in form of a string.

The second argument represents the mode in which we are opening the file. This is also a String.

Opening a file: modes

The modes that we can use are:

- `'r'` : to open for reading. This is the default value.
- `'w'` : to create or open (will be truncated) a file for writing.
- `'a'`: to open a file to append content (add at the end).
- `'x'`: only to create a new empty file. Fails if the file exists.
- `'+'`: to open a file in both read and write mode. This char can be added to r and w modes.

An additional char can be specified to choose how the file should be handled:

- `'t'` : textual mode (default). All input and outputs are handled as strings. This is the default.
- `'b'` : binary mode. Unicode translation and end-line are turned off.

Opening a file: examples

```
# open in read mode
my_file = open('my_file.txt', 'r')

# open in write mode
my_file = open('my_file.txt', 'w')

# open in append mode
my_file = open('my_file.txt', 'a')

# open in read + write mode
my_file = open('my_file.txt', 'r+')

# open in read binary mode
my_file = open('my_file.txt', 'rb')

# open in write binary mode
my_file = open('my_file.txt', 'wb')
```

The returned file object is a particular data type that provide us with methods that we can use to interact with a file.

Features of Python files

When using a Python file object it is important to know some particularities:

- To read a file it is best to use iterators and read line by line, instead of reading the entire content
- Data read from a file is always of type String. If necessary, you have to manually convert from string to the desired data type.
- Files are buffered: when you write to a file, the operation will not be immediate. A call to `close` or `flush` forces the writing of the buffered data to file.
- Files are seekable: it is possible to access any location in the file (random-access) by using a byte offset.
- In order to free the resources, it is important to always close the file when finished with it. This can be done using the `close` function or with a context manager

Closing a file

After opening and performing operations on a file we must close it in order to free the resources.

A file can be closed manually by means of the `close()` method available on the file object.

```
my_file = open('my_file.txt', 'r')
# do some useful stuff with your file
my_file.close()
```

Another option to automatically close files is to use a `file context manager`.

Closing a file: file context manager

A file context manager is a feature that we can use in order to wrap the logic of file handling, so that the file will be automatically closed.

To use it we need to use the `with` and the `as` keywords in conjunction with the `open` function.

```
with open('my_file.txt', 'r') as my_file:
    # do some useful stuff with your file

# when exiting the context manager block
# the file will be automatically closed
```

This is usually the preferred way of deal with closing a file.

Reading entire file content

Once a file is opened there are many methods (functions) that we can use to interact with the file.

To read the whole content of a file it is possible to use the `read()` function on a file object.

Example:

```
with open('my_file.txt', 'r') as my_file:
    file_content = my_file.read()
    print(file_content)

    # prints the full content of the file
```

Reading a single line

To read a single line at a time it is possible to use the `readline()` method.

Let's assume to read a file containing 2 lines of text.

```
with open('my_file.txt', 'r') as my_file:
    my_file.readline()
    # reads the content of the first line

    my_file.readline()
    # reads the content of the second line

    my_file.readline()
    # if the line is the last it will read ''
```

Reading line by line with an iterator

When we need to read the entire content of a file line by line the best option is to use a `file iterator`.

Using this approach it is basically only necessary to open the file and use a `for` loop to cycle through the rows as in the following example.

```
with open('my_file.txt') as my_file:
    for line in my_file:
        print(line, end='')

# Outputs:
# content of the first row
# content of the second line
```

Done in this simple way, the file read performs well since it take advantage of `iterators` (we will not cover iterators in this course).

Writing a file

In order to write a file it is necessary to use the mode `w` in combination with the open function.

It is then possible to use the `write(somecontent)` function to store data to the file.

```
with open('write.txt', 'w') as my_file:
    my_file.write("This is\nthe first data\n")
    my_file.write("that we write\nin a file!")

# Creates a file named write.txt with the following content:

# This is
# the first data
# that we write
# in a file!
```

Textual and binary files

In the examples that we have seen so far we used only textual files since it is the default mode.

The content of textual files is represented as `str` data types.

Furthermore, also Unicode character encoding and decoding, as well as end-of-line manipulations are done automatically out of the box.

If dealing with non textual file types, like for instance images, we need to use different options so that python handles the content as a `binary file`.

Reading a binary file

To read a file in binary mode, it is possible to add a `b` to the mode when using the `open` function.

For example, if we want to set the mode as read binary we can use `'rb'` for the mode.

```
with open('my_file.bin', mode='rb') as bin_file:
    # File opened in read binary mode.
```

When opened in binary mode, the content of the files is represented as `bytes` and it is unaltered (not translated/manipulated in any way).

Reading a binary file: example

```
with open('my_file.bin', mode='rb') as bin_file:
    data = bin_file.read()
    print(data)

# output example:
# b'\x01\x02\x037<b'
```

When opened in binary mode, the content of the file is represented as `bytes` and it is unaltered (not translated in any way).

Writing a binary file

In order to write a binary file we just need to open it in writing mode, and add the b flag to specify we will write binary code.

```
with open('my_file.bin', mode='wb') as bin_file:
    my_list = [1, 2, 8]
    # convert to bytes
    byte_list = bytes(my_list)
    bin_file.write(byte_list)

# File written in binary mode
```

The content of the file will be saved in binary mode and not as text.

Note that it is necessary to transform the data into bytes using the `bytes()` function.

Read exercise

Let's assume that we have a file containing the following textual data

```
Alice, Bob, Daisy, Ed
1,4,2,3
```

What should we do in order to read from the file and construct a dictionary like this:

```
{'Alice': 1, 'Bob': 4, 'Daisy': 2, 'Ed': 3}
```

Do we need to take care of something in particular?
What if we had the dictionary in the file and we wanted to obtain two lists?

Storing native Python objects

If you need to store and retrieve Python object in file, for instance a dictionary or a list, you should take into account that the parsing (extraction) and conversion of the data can be difficult to carry on.

Some of the possible solutions use for example the `eval()` function to obtain the Python object from a string.

This function basically takes any string expression and evaluates (executes) it.

This is something that can cause a lot of problems in terms of security, since we often cannot rely on the goodness of the data read by a file.

We really do not want to risk executing possible malicious commands.

Storing native Python objects to files: pickle

To solve the problem of storing native Python objects, the standard library provide us with the `pickle` module.

This module basically provides a tool to store and retrieve almost every python object to and from a file. This is what is known as `object serialization`.

To store an object in a file we can use the `dump(object, file_obj)` function as in the following example:

```
import pickle

with open('my_file.pkl', mode='wb') as bin_file:
    my_dict = {'Alice': 1, 'Bob': 4, 'Daisy': 2, 'Ed': 3}
    pickle.dump(my_dict, bin_file)

# Dictionary written in binary mode using pickle
```

Loading native Python objects from files: pickle

To load any object from a file we can use the `load(file_obj)` function as in the following example:

```
import pickle

with open('my_file.pkl', mode='rb') as bin_file:
    my_dict = pickle.load(bin_file)
    print(my_dict)

# Prints {'Alice': 1, 'Bob': 4, 'Daisy': 2, 'Ed': 3}
```

Other tools to read/write files

The Python standard library provides us out of the box with a large number of modules to handle different data types.

Here some examples of modules that could be of interest:

- `json`: provides methods to handle json type files.
- `csv`: provides methods to handle csv (comma separated values) files.
- `struct`: provides methods to handle packed binary data files.
- `zipfile`: provides methods to handle zip archives.

Other external libraries as `pandas` can provide you additional file support, for example to load files such as `excel, open documents,...`

Summary

- Files
- Opening and Closing files
- Modes
- Reading files
- Writing files
- Binary files
- Reading binary files
- Writing binary files
- Storing and retrieving native Python objects
- Other tools

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapter 9
- Python Crash Course, no starch press - Eric Matthes: Chapters 10
- Python Official Documentation: https://docs.python.org/3/tutorial/
- LearnByExample: https://www.learnbyexample.org/python/