

Real Python

Reading and Writing Files in Python (Guide)

by James Mertz 25m 18 Comments

 intermediate  python

Mark as Completed



Share

Table of Contents

- [What Is a File?](#)
 - [File Paths](#)

— FREE Email Series —

Python Tricks

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Email...

[Get Python Tricks »](#)

 No spam. Unsubscribe any time.

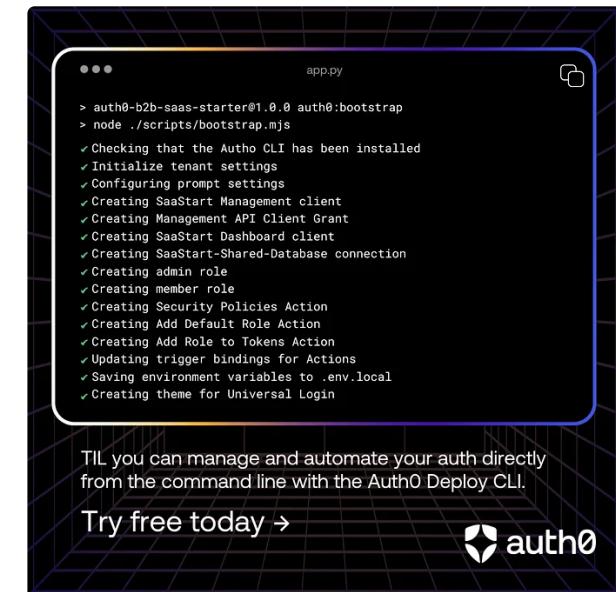
 [Browse Topics](#)  [Guided Learning Paths](#)

 [Basics](#)  [Intermediate](#)  [Advanced](#)

[ai](#) [algorithms](#) [api](#) [best-practices](#) [career](#)
[community](#) [databases](#) [data-science](#)
[data-structures](#) [data-viz](#) [devops](#) [django](#)
[docker](#) [editors](#) [flask](#) [front-end](#) [gamedev](#)
[gui](#) [machine-learning](#) [news](#) [numpy](#)

- Line Endings
- Character Encodings
- Opening and Closing a File in Python
 - Text File Types
 - Buffered Binary File Types
 - Raw File Types
- Reading and Writing Opened Files
 - Iterating Over Each Line in the File
 - Working With Bytes
 - A Full Example: dos2unix.py
- Tips and Tricks
 - __file__
 - Appending to a File
 - Working With Two Files at the Same Time
 - Creating Your Own Context Manager
- Don't Re-Invent the Snake
- You're a File Wizard Harry!

projects python stdlib testing tools
web-dev web-scraping





Secure access for AI agents.
But not too much.

[Try free today](#)

[Remove ads](#)

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Reading and Writing Files in Python](#)

One of the most common tasks that you can do with Python is reading and writing files. Whether it's writing to a simple text file, reading a complicated server log, or even analyzing raw byte data, all of these situations require reading or writing a file.

Table of Contents

- What Is a File?
- Opening and Closing a File in Python
- Reading and Writing Opened Files
- Tips and Tricks
- Don't Re-Invent the Snake
- You're a File Wizard Harry!

[Mark as Completed](#) 

In this tutorial, you'll learn:

- What makes up a file and why that's important in Python
- The basics of reading and writing files in Python
- Some basic scenarios of reading and writing files

This tutorial is mainly for beginner to intermediate Pythonistas, but there are some tips in here that more advanced programmers may appreciate as well.

Free Bonus: [Click here to get our free Python Cheat Sheet](#) that shows you the basics of Python 3, like working with data types, dictionaries, lists, and Python functions.

 **Take the Quiz:** Test your knowledge with our interactive “Reading and Writing Files in Python” quiz. You’ll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Reading and Writing Files in Python

A quiz used for testing the user's knowledge of the topics covered in the Reading and Writing Files in Python article.

Recommended Video Course

Reading and Writing Files in Python



Join Real Python and Unlock Learning Paths, Courses, Live Q&As, and More:

[Become a Python Expert »](#)

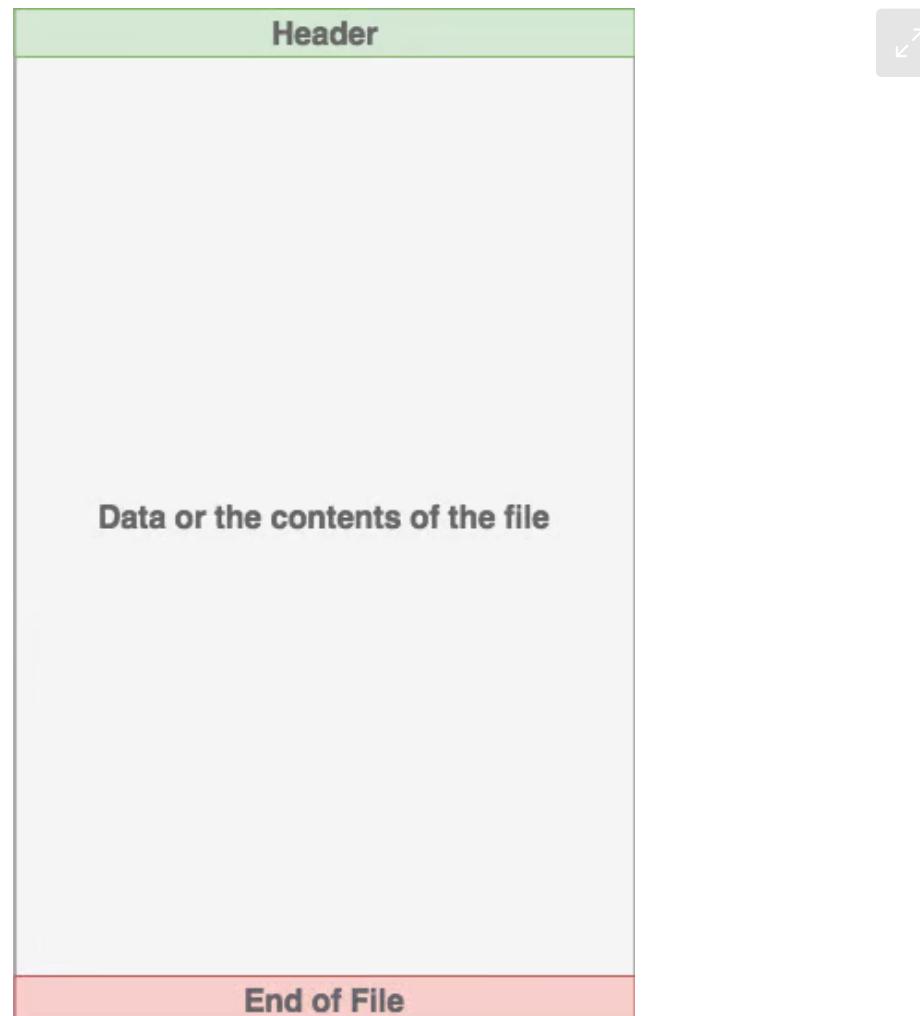
What Is a File?

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](#). This data is organized in a specific format and can be anything as simple as a text file or as complicated as a program executable. In the end, these byte files are then translated into binary 1 and 0 for easier processing by the computer.

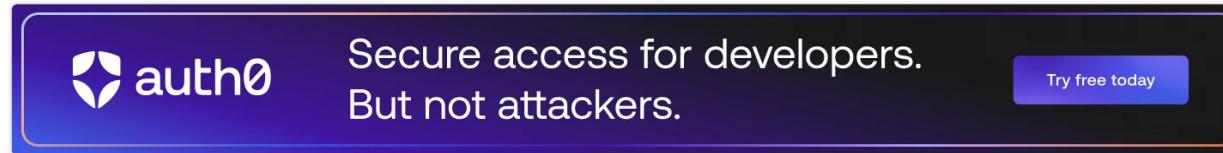
Files on most modern file systems are composed of three main parts:

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. For this tutorial, you'll only deal with `.txt` or `.csv` file extensions.



The banner features the Auth0 logo (a white diamond shape with a smaller diamond inside) and the text "auth0 Secure access for developers. But not attackers." in white. A blue button on the right says "Try free today".

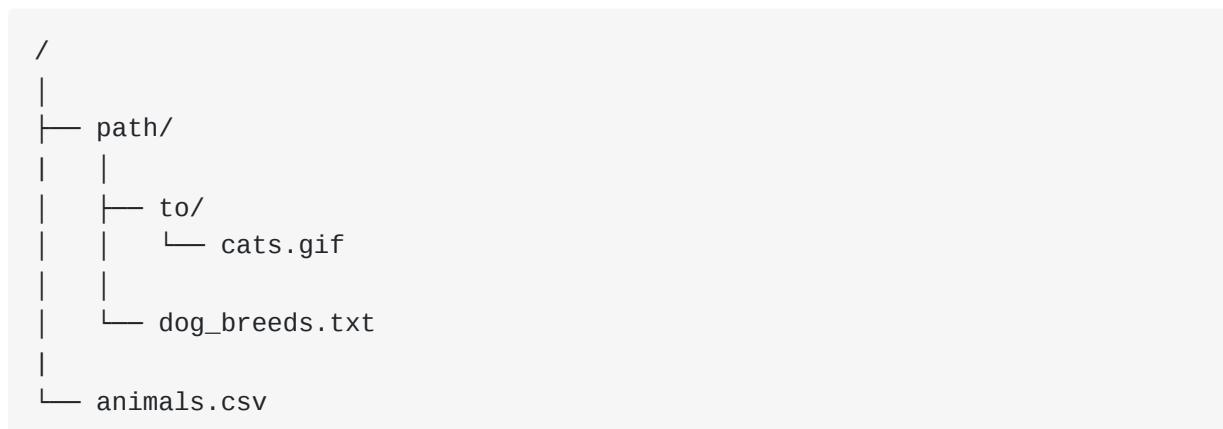
[i Remove ads](#)

File Paths

When you access a file on an operating system, a file path is required. The file path is a string that represents the location of a file. It's broken up into three major parts:

- 1. Folder Path:** the file folder location on the file system where subsequent folders are separated by a forward slash `/` (Unix) or backslash `\` (Windows)
- 2. File Name:** the actual name of the file
- 3. Extension:** the end of the file path pre-pended with a period `(.)` used to indicate the file type

Here's a quick example. Let's say you have a file located within a file structure like this:



Let's say you wanted to access the `cats.gif` file, and your current location was in the same folder as `path`. In order to access the file, you need to go through the `path` folder and then the `to` folder, finally arriving at the `cats.gif` file. The Folder Path is `path/to/`. The File Name is `cats`. The File Extension is `.gif`. So the full path is `path/to/cats.gif`.

Now let's say that your current location or current working directory (`cwd`) is in the `to` folder of our example folder structure. Instead of referring to the `cats.gif` by the full path of `path/to/cats.gif`, the file can be simply referenced by the file name and extension `cats.gif`.

```
/  
|  
|   path/  
|   |  
|   |   to/  ← Your current working directory (cwd) is here  
|   |   |   cats.gif  ← Accessing this file  
|   |  
|   |   dog_breeds.txt  
|  
|  
└   animals.csv
```

But what about `dog_breeds.txt`? How would you access that without using the full path? You can use the special characters double-dot (`..`) to move one directory up. This means that `../dog_breeds.txt` will reference the `dog_breeds.txt` file from the directory of `to`:

```
/  
|  
|   path/  ← Referencing this parent folder  
|   |  
|   |   to/  ← Current working directory (cwd)  
|   |   |   cats.gif  
|  
|   |   dog_breeds.txt  ← Accessing this file
```

```
|  
└── animals.csv
```

The double-dot (..) can be chained together to traverse multiple directories above the current directory. For example, to access `animals.csv` from the `to` folder, you would use `../../animals.csv`.

Line Endings

One problem often encountered when [working with file data](#) is the representation of a new line or line ending. The line ending has its roots from back in the Morse Code era, [when a specific pro-sign was used to communicate the end of a transmission or the end of a line](#).

Later, this [was standardized for teleprinters](#) by both the International Organization for Standardization (ISO) and the American Standards Association (ASA). ASA standard states that line endings should use the sequence of the Carriage Return (CR or \r) *and* the Line Feed (LF or \n) characters (CR+LF or \r\n). The ISO standard however allowed for either the CR+LF characters or just the LF character.

[Windows uses the CR+LF characters](#) to indicate a new line, while Unix and the newer Mac versions use just the LF character. This can cause some complications when you're processing files on an operating system that is different than the file's source. Here's a quick example. Let's say that we examine the file `dog_breeds.txt` that was created on a Windows system:

```
Pug\r\n
Jack Russell Terrier\r\n
English Springer Spaniel\r\n
German Shepherd\r\n
Staffordshire Bull Terrier\r\n
Cavalier King Charles Spaniel\r\n
Golden Retriever\r\n
West Highland White Terrier\r\n
```

```
Boxer\r\nBorder Terrier\r\n
```

This same output will be interpreted on a Unix device differently:

```
Pug\r\n\nJack Russell Terrier\r\n\nEnglish Springer Spaniel\r\n\nGerman Shepherd\r\n\nStaffordshire Bull Terrier\r\n\nCavalier King Charles Spaniel\r\n\nGolden Retriever\r\n\nWest Highland White Terrier\r\n\nBoxer\r\n\nBorder Terrier\r\n\n
```

This can make iterating over each line problematic, and you may need to account for situations like this.

Character Encodings

Another common problem that you may face is the encoding of the byte data. An encoding is a translation from byte data to human readable characters. This is typically done by assigning a numerical value to represent a character. The two most common encodings are

the [ASCII](#) and [UNICODE](#) Formats. [ASCII can only store 128 characters](#), while [Unicode can contain up to 1,114,112 characters](#).

ASCII is actually a subset of [Unicode](#) (UTF-8), meaning that ASCII and Unicode share the same numerical to character values. It's important to note that parsing a file with the incorrect character encoding can lead to failures or misrepresentation of the character. For example, if a file was created using the UTF-8 encoding, and you try to parse it using the ASCII encoding, if there is a character that is outside of those 128 values, then an error will be thrown.



**Master Real-World Python Skills
With a Community of Experts**
Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons

Watch Now »

 Remove ads

Opening and Closing a File in Python

When you want to work with a file, the first thing to do is to open it. This is done by invoking the [open\(\) built-in function](#). `open()` has a single required argument that is the path to the file. `open()` has a single return, the [file object](#):

Python

```
file = open('dog_breeds.txt')
```

After you open a file, the next thing to learn is how to close it.

Warning: You should *always* make sure that an open file is properly closed. To learn why, check out the [Why Is It Important to Close Files in Python?](#) tutorial.

It's important to remember that it's your responsibility to close the file. In most cases, upon termination of an application or script, a file will be closed eventually. However, there is no

guarantee when exactly that will happen. This can lead to unwanted behavior including resource leaks. It's also a best practice within Python (Pythonic) to make sure that your code behaves in a way that is well defined and reduces any unwanted behavior.

When you're manipulating a file, there are two ways that you can use to ensure that a file is closed properly, even when encountering an error. The first way to close a file is to use the `try-finally` block:

Python

```
reader = open('dog_breeds.txt')
try:
    # Further file processing goes here
finally:
    reader.close()
```

If you're unfamiliar with what the `try-finally` block is, check out [Python Exceptions: An Introduction](#).

The second way to close a file is to use the `with statement`:

Python

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

The `with` statement automatically takes care of closing the file once it leaves the `with` block, even in cases of error. I highly recommend that you use the `with` statement as much as possible, as it allows for cleaner code and makes handling any unexpected errors easier for you.

Most likely, you'll also want to use the second positional argument, `mode`. This argument is a `string` that contains multiple characters to represent how you want to open the file. The

default and most common is 'r', which represents opening the file in read-only mode as a text file:

Python

```
with open('dog_breeds.txt', 'r') as reader:  
    # Further file processing goes here
```

Other options for modes are [fully documented online](#), but the most commonly used ones are the following:

Character	Meaning
'r'	Open for reading (default)
'w'	Open for writing, truncating (overwriting) the file first
'rb' or 'wb'	Open in binary mode (read/write using byte data)

Let's go back and talk a little about file objects. A file object is:

“an object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource.” ([Source](#))

There are three different categories of file objects:

- Text files
- Buffered binary files
- Raw binary files

Each of these file types are defined in the `io` module. Here's a quick rundown of how everything lines up.



[i Remove ads](#)

Text File Types

A text file is the most common file that you'll encounter. Here are some examples of how these files are opened:

Python

```
open('abc.txt')

open('abc.txt', 'r')

open('abc.txt', 'w')
```

With these types of files, `open()` will return a `TextIOWrapper` file object:

Python

```
>>> file = open('dog_breeds.txt')
>>> type(file)
<class '_io.TextIOWrapper'>
```

This is the default file object returned by `open()`.

Buffered Binary File Types

A buffered binary file type is used for reading and writing binary files. Here are some examples of how these files are opened:

Python

```
open('abc.txt', 'rb')
```

```
open('abc.txt', 'wb')
```

With these types of files, `open()` will return either a `BufferedReader` or `BufferedWriter` file object:

Python

```
>>> file = open('dog_breeds.txt', 'rb')
>>> type(file)
<class '_io.BufferedReader'>
>>> file = open('dog_breeds.txt', 'wb')
>>> type(file)
<class '_io.BufferedWriter'>
```

Raw File Types

A raw file type is:

“generally used as a low-level building-block for binary and text streams.” ([Source](#))

It is therefore not typically used.

Here’s an example of how these files are opened:

Python

```
open('abc.txt', 'rb', buffering=0)
```

With these types of files, `open()` will return a `FileIO` file object:

Python

```
>>> file = open('dog_breeds.txt', 'rb', buffering=0)
>>> type(file)
<class '_io.FileIO'>
```

Reading and Writing Opened Files

Once you've opened up a file, you'll want to read or write to the file. First off, let's cover reading a file. There are multiple methods that can be called on a file object to help you out:

Method	What It Does
<code>.read(size=-1)</code>	This reads from the file based on the number of <code>size</code> bytes. If no argument is passed or <code>None</code> or <code>-1</code> is passed, then the entire file is read.
<code>.readline(size=-1)</code>	This reads at most <code>size</code> number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or <code>None</code> or <code>-1</code> is passed, then the entire line (or rest of the line) is read.
<code>.readlines()</code>	This reads the remaining lines from the file object and returns them as a list.

Using the same `dog_breeds.txt` file you used above, let's go through some examples of how to use these methods. Here's an example of how to open and read the entire file using `.read()`:

Python

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read & print the entire file
>>>     print(reader.read())
```

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Here's an example of how to read 5 bytes of a line each time using the Python `.readline()` method:

```
Python
```

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read & print the first 5 characters of the line 5 times
>>>     print(reader.readline(5))
>>>     # Notice that line is greater than the 5 chars and continues
>>>     # down the line, reading 5 chars each time until the end of the
>>>     # line and then "wraps" around
>>>     print(reader.readline(5))
>>>     print(reader.readline(5))
>>>     print(reader.readline(5))
>>>     print(reader.readline(5))
Pug
Jack
Russe
ll Te
rrier
```

Here's an example of how to read the entire file as a list using the Python `.readlines()` method:

```
Python
```

```
>>> f = open('dog_breeds.txt')
>>> f.readlines() # Returns a list object
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shep
```

The above example can also be done by using `list()` to create a list out of the file object:

```
Python
```

```
>>> f = open('dog_breeds.txt')
>>> list(f)
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shep
```



[i Remove ads](#)

Iterating Over Each Line in the File

A common thing to do while reading a file is to iterate over each line. Here's an example of how to use the Python `.readline()` method to perform that iteration:

```
Python
```

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read and print the entire file line by line
>>>     line = reader.readline()
>>>     while line != '': # The EOF char is an empty string
>>>         print(line, end='')
>>>         line = reader.readline()
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
```

```
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Another way you could iterate over each line in the file is to use the Python `.readlines()` method of the file object. Remember, `.readlines()` returns a list where each element in the list represents a line in the file:

```
Python
```

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     for line in reader.readlines():
>>>         print(line, end='')

Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

However, the above examples can be further simplified by iterating over the file object itself:

```
Python
```

```
>>> with open('dog_breeds.txt', 'r') as reader:
>>>     # Read and print the entire file line by line
>>>     for line in reader:
>>>         print(line, end='')

Pug
Jack Russell Terrier
English Springer Spaniel
```

```
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

This final approach is more Pythonic and can be quicker and more memory efficient.

Therefore, it is suggested you use this instead.

Note: Some of the above examples contain `print('some text', end='')`. The `end=''` is to prevent Python from adding an additional newline to the text that is being printed and only `print` what is being read from the file.

Now let's dive into writing files. As with reading files, file objects have multiple methods that are useful for writing to a file:

Method	What It Does
<code>.write(string)</code>	This writes the string to the file.
<code>.writelines(seq)</code>	This writes the sequence to the file. No line endings are appended to each sequence item. It's up to you to add the appropriate line ending(s).

Here's a quick example of using `.write()` and `.writelines()`:

Python

```
with open('dog_breeds.txt', 'r') as reader:
    # Note: readlines doesn't trim the line endings
    dog_breeds = reader.readlines()
```

```
with open('dog_breeds_reversed.txt', 'w') as writer:  
    # Alternatively you could use  
    # writer.writelines(reversed(dog_breeds))  
  
    # Write the dog breeds to the file in reversed order  
    for breed in reversed(dog_breeds):  
        writer.write(breed)
```

Working With Bytes

Sometimes, you may need to work with files using [byte strings](#). This is done by adding the 'b' character to the mode argument. All of the same methods for the file object apply.

However, each of the methods expect and return a bytes object instead:

Python

```
>>> with open('dog_breeds.txt', 'rb') as reader:  
>>>     print(reader.readline())  
b'Pug\n'
```

Opening a text file using the b flag isn't that interesting. Let's say we have this cute picture of a Jack Russell Terrier (jack_russell.png):



Image: CC BY 3.0 (<https://creativecommons.org/licenses/by/3.0/>), from Wikimedia Commons

You can actually open that file in Python and examine the contents! Since the [.png file format](#) is well defined, the header of the file is 8 bytes broken up like this:

Value	Interpretation
0x89	A “magic” number to indicate that this is the start of a PNG
0x50 0x4E 0x47	PNG in ASCII
0x0D 0x0A	A DOS style line ending \r\n
0x1A	A DOS style EOF character
0x0A	A Unix style line ending \n

Sure enough, when you open the file and read these bytes individually, you can see that this is indeed a .png header file:

```
Python

>>> with open('jack_russell.png', 'rb') as byte_reader:
>>>     print(byte_reader.read(1))
>>>     print(byte_reader.read(3))
>>>     print(byte_reader.read(2))
>>>     print(byte_reader.read(1))
>>>     print(byte_reader.read(1))

b'\x89'
b'PNG'
b'\r\n'
b'\x1a'
b'\n'
```



A Full Example: dos2unix.py

Let's bring this whole thing home and look at a full example of how to read and write to a file. The following is a [dos2unix](#) like tool that will convert a file that contains line endings of `\r\n` to `\n`.

This tool is broken up into three major sections. The first is `str2unix()`, which converts a string from `\r\n` line endings to `\n`. The second is `dos2unix()`, which converts a string that contains `\r\n` characters into `\n`. `dos2unix()` calls `str2unix()` internally. Finally, there's the `__main__` block, which is called only when the file is executed as a script. Think of it as the `main` function found in other programming languages.

```
"""
A simple script and library to convert files or strings from dos like
line endings with Unix like line endings.

"""

import argparse
import os

def str2unix(input_str: str) -> str:
    """
    Converts the string from \r\n line endings to \n

    Parameters
    -----
    input_str
        The string whose line endings will be converted

    Returns
    -----
        The converted string
    """
    r_str = input_str.replace('\r\n', '\n')
    return r_str

def dos2unix(source_file: str, dest_file: str):
    """
    Converts a file that contains Dos like line endings into Unix like

    Parameters
    -----
    source_file
        The path to the source file to be converted
    dest_file
        The path to the converted file for output
    """

```

```

# NOTE: Could add file existence checking and file overwriting
# protection
with open(source_file, 'r') as reader:
    dos_content = reader.read()

unix_content = str2unix(dos_content)

with open(dest_file, 'w') as writer:
    writer.write(unix_content)

if __name__ == "__main__":
    # Create our Argument parser and set its description
    parser = argparse.ArgumentParser(
        description="Script that converts a DOS like file to an Unix like file"
    )

    # Add the arguments:
    #   - source_file: the source file we want to convert
    #   - dest_file: the destination where the output should go

    # Note: the use of the argument type of argparse.FileType could
    # streamline some things
    parser.add_argument(
        'source_file',
        help='The location of the source '
    )

    parser.add_argument(
        '--dest_file',
        help='Location of dest file (default: source_file appended with `._unix'
        default=None
    )

    # Parse the args (argparse automatically grabs the values from
    # sys.argv)
    args = parser.parse_args()

```

```
s_file = args.source_file
d_file = args.dest_file

# If the destination file wasn't passed, then assume we want to
# create a new file based on the old one
if d_file is None:
    file_path, file_extension = os.path.splitext(s_file)
    d_file = f'{file_path}_unix{file_extension}'

dos2unix(s_file, d_file)
```

Tips and Tricks

Now that you've mastered the basics of reading and writing files, here are some tips and tricks to help you grow your skills.

__file__

The `__file__` attribute is a [special attribute](#) of modules, similar to `__name__`. It is:

“the pathname of the file from which the module was loaded, if it was loaded from a file.” ([Source](#)

Note: To re-iterate, `__file__` returns the path *relative* to where the initial Python script was called. If you need the full system path, you can use `os.getcwd()` to get the current working directory of your executing code.

Here's a real world example. In one of my past jobs, I did multiple tests for a hardware device. Each test was written using a Python script with the test script file name used as a title. These scripts would then be executed and could print their status using the `__file__` special attribute. Here's an example folder structure:

```
project/
|
└── tests/
    ├── test_commanding.py
    ├── test_power.py
    ├── test_wireHousing.py
    └── test_leds.py
|
└── main.py
```

Running `main.py` produces the following:

```
>>> python main.py
tests/test_commanding.py Started:
tests/test_commanding.py Passed!
tests/test_power.py Started:
tests/test_power.py Passed!
tests/test_wireHousing.py Started:
tests/test_wireHousing.py Failed!
tests/test_leds.py Started:
tests/test_leds.py Passed!
```

I was able to run and get the status of all my tests dynamically through use of the `__file__` special attribute.

Appending to a File

Sometimes, you may want to append to a file or start writing at the end of an already populated file. This is easily done by using the 'a' character for the mode argument:

Python

```
with open('dog_breeds.txt', 'a') as a_writer:
    a_writer.write('\nBeagle')
```

When you examine `dog_breeds.txt` again, you'll see that the beginning of the file is unchanged and `Beagle` is now added to the end of the file:

Python

```
>>> with open('dog_breeds.txt', 'r') as reader:  
>>>     print(reader.read())  
Pug  
Jack Russell Terrier  
English Springer Spaniel  
German Shepherd  
Staffordshire Bull Terrier  
Cavalier King Charles Spaniel  
Golden Retriever  
West Highland White Terrier  
Boxer  
Border Terrier  
Beagle
```



[i Remove ads](#)

Working With Two Files at the Same Time

There are times when you may want to read a file and write to another file at the same time. If you use the example that was shown when you were learning how to write to a file, it can actually be combined into the following:

Python

```
d_path = 'dog_breeds.txt'  
d_r_path = 'dog_breeds_reversed.txt'  
with open(d_path, 'r') as reader, open(d_r_path, 'w') as writer:
```

```
dog_breeds = reader.readlines()
writer.writelines(reversed(dog_breeds))
```

Creating Your Own Context Manager

There may come a time when you'll need finer control of the file object by placing it inside a custom class. When you do this, using the `with` statement can no longer be used unless you add a few magic methods: `__enter__` and `__exit__`. By adding these, you'll have created what's called a [context manager](#).

`__enter__()` is invoked when calling the `with` statement. `__exit__()` is called upon exiting from the `with` statement block.

Here's a template that you can use to make your custom class:

Python

```
class my_file_reader():
    def __init__(self, file_path):
        self.__path = file_path
        self.__file_object = None

    def __enter__(self):
        self.__file_object = open(self.__path)
        return self

    def __exit__(self, type, val, tb):
        self.__file_object.close()

    # Additional methods implemented below
```

Now that you've got your custom class that is now a context manager, you can use it similarly to the `open()` built-in:

Python

```
with my_file_reader('dog_breeds.txt') as reader:  
    # Perform custom class operations  
    pass
```

Here's a good example. Remember the cute Jack Russell image we had? Perhaps you want to open other .png files but don't want to parse the header file each time. Here's an example of how to do this. This example also uses custom iterators. If you're not familiar with them, check out [Python Iterators](#):

Python

```
class PngReader():  
    # Every .png file contains this in the header.  Use it to verify  
    # the file is indeed a .png.  
    _expected_magic = b'\x89PNG\r\n\x1a\n'  
  
    def __init__(self, file_path):  
        # Ensure the file has the right extension  
        if not file_path.endswith('.png'):  
            raise NameError("File must be a '.png' extension")  
        self.__path = file_path  
        self.__file_object = None  
  
    def __enter__(self):  
        self.__file_object = open(self.__path, 'rb')  
  
        magic = self.__file_object.read(8)  
        if magic != self._expected_magic:  
            raise TypeError("The File is not a properly formatted .png file!")  
  
        return self  
  
    def __exit__(self, type, val, tb):  
        self.__file_object.close()  
  
    def __iter__(self):  
        # This and __next__() are used to create a custom iterator
```

```

# See https://dbader.org/blog/python-iterators
return self

def __next__(self):
    # Read the file in "Chunks"
    # See https://en.wikipedia.org/wiki/Portable_Network_Graphics#%22Chunks%22

    initial_data = self.__file_object.read(4)

    # The file hasn't been opened or reached EOF. This means we
    # can't go any further so stop the iteration by raising the
    # StopIteration.
    if self.__file_object is None or initial_data == b'':
        raise StopIteration
    else:
        # Each chunk has a len, type, data (based on len) and crc
        # Grab these values and return them as a tuple
        chunk_len = int.from_bytes(initial_data, byteorder='big')
        chunk_type = self.__file_object.read(4)
        chunk_data = self.__file_object.read(chunk_len)
        chunk_crc = self.__file_object.read(4)
        return chunk_len, chunk_type, chunk_data, chunk_crc

```

You can now open .png files and properly parse them using your custom context manager:

Python

```

>>> with PngReader('jack_russell.png') as reader:
>>>     for l, t, d, c in reader:
>>>         print(f"{l:05}, {t}, {c}")
00013, b'IHDR', b'v\x121k'
00001, b'sRGB', b'\xae\xce\x1c\xe9'
00009, b'pHYs', b'(<]\x19'
00345, b'iTExt', b'L\xc2'Y"
16384, b'IDAT', b'i\x99\x0c('
16384, b'IDAT', b'\xb3\xfa\x9a$'
16384, b'IDAT', b'\xff\xbf\xd1\n'
16384, b'IDAT', b'\xc3\x9c\xb1'

```

```
16384, b'IDAT', b'\xe3\x02\xba\x91'
16384, b'IDAT', b'\xa0\x99='
16384, b'IDAT', b'\xf4\x8b.\x92'
16384, b'IDAT', b'\x17i\xfc\xde'
16384, b'IDAT', b'\x8fb\x0e\xe4'
16384, b'IDAT', b')3={'
01040, b'IDAT', b'\xd6\xb8\xc1\x9f'
00000, b'IEEND', b'\xaeB`\x82'
```

Don't Re-Invent the Snake

There are common situations that you may encounter while working with files. Most of these cases can be handled using other modules. Two common file types you may need to work with are `.csv` and `.json`. *Real Python* has already put together some great articles on how to handle these:

- [Reading and Writing CSV Files in Python](#)
- [Working With JSON Data in Python](#)

Additionally, there are built-in libraries out there that you can use to help you:

- **wave**: read and write WAV files (audio)
- **aifc**: read and write AIFF and AIFC files (audio)
- **sunau**: read and write Sun AU files
- **tarfile**: read and write tar archive files
- **zipfile**: work with ZIP archives
- **configparser**: easily create and parse configuration files
- **xml.etree.ElementTree**: create or read XML based files
- **msilib**: read and write Microsoft Installer files
- **plistlib**: generate and parse Mac OS X `.plist` files

There are plenty more out there. Additionally there are even more third party tools available on PyPI. Some popular ones are the following:

- **PyPDF2**: PDF toolkit
- **xlwings**: read and write Excel files
- **Pillow**: image reading and manipulation



[i Remove ads](#)

You're a File Wizard Harry!

You did it! You now know how to work with files with Python, including some advanced techniques. Working with files in Python should now be easier than ever and is a rewarding feeling when you start doing it.

In this tutorial you've learned:

- What a file is
- How to open and close files properly
- How to read and write files
- Some advanced techniques when working with files
- Some libraries to work with common file types

If you have any questions, hit us up in the comments.

Take the Quiz: Test your knowledge with our interactive “Reading and Writing Files in Python” quiz. You’ll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Reading and Writing Files in Python

A quiz used for testing the user's knowledge of the topics covered in the Reading and Writing Files in Python article.

[Mark as Completed](#)[Share](#)

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Reading and Writing Files in Python](#)



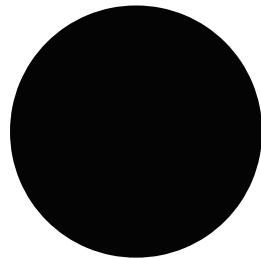
Python Tricks

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

[Send Me Python Tricks »](#)

About James Mertz



James is a passionate Python developer at NASA's Jet Propulsion Lab who also writes on the side for Real Python.

[» More about James](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:



Aldren



David



Joanna

Master Real-World Python Skills
With Unlimited Access to Real Python

Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

[Level Up Your Python Skills »](#)

What Do You Think?

Rate this article:



LinkedIn

Twitter

Bluesky

Facebook

Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next “Office Hours” Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: [intermediate](#) [python](#)

Recommended Video Course: [Reading and Writing Files in Python](#)

Related Tutorials:

- [Working With Files in Python](#)
- [Reading and Writing CSV Files in Python](#)
- [Working With JSON Data in Python](#)
- [Python Exceptions: An Introduction](#)
- [Logging in Python](#)

Learn Python	Courses & Paths	Community	Membership	Company
Start Here	Learning Paths	Podcast	Plans & Pricing	About Us
Learning Resources	Quizzes & Exercises	Newsletter	Team Plans	Team
Code Mentor	Browse Topics	Community Chat	For Business	Mission & Values
Python Reference	Live Courses	Office Hours	For Schools	Editorial Guidelines
Python Cheat Sheet	Books	Learner Stories	Reviews	Sponsorships
Support Center				Careers
				Press Kit
				Merch



[Privacy Policy](#) [Terms of Use](#) [Security](#) [Contact](#)

 Happy Pythoning!

© 2012–2026 DevCademy Media Inc. DBA Real Python. All rights reserved.

REALPYTHON™ is a trademark of DevCademy Media Inc.

