Working with Files in Python

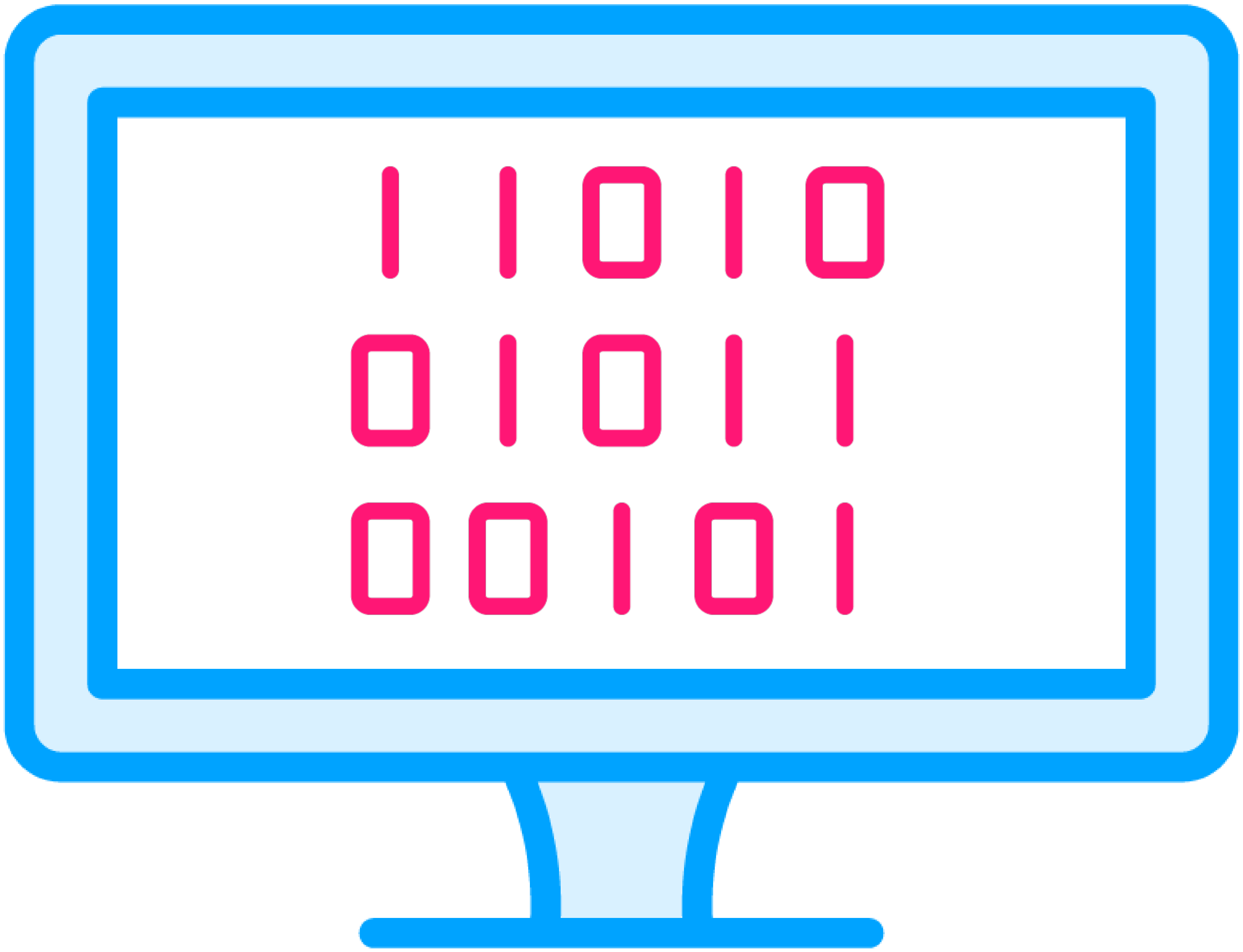
Creating, Reading, Writing, Deleting, and Managing Files and Directories

Xavier Morera

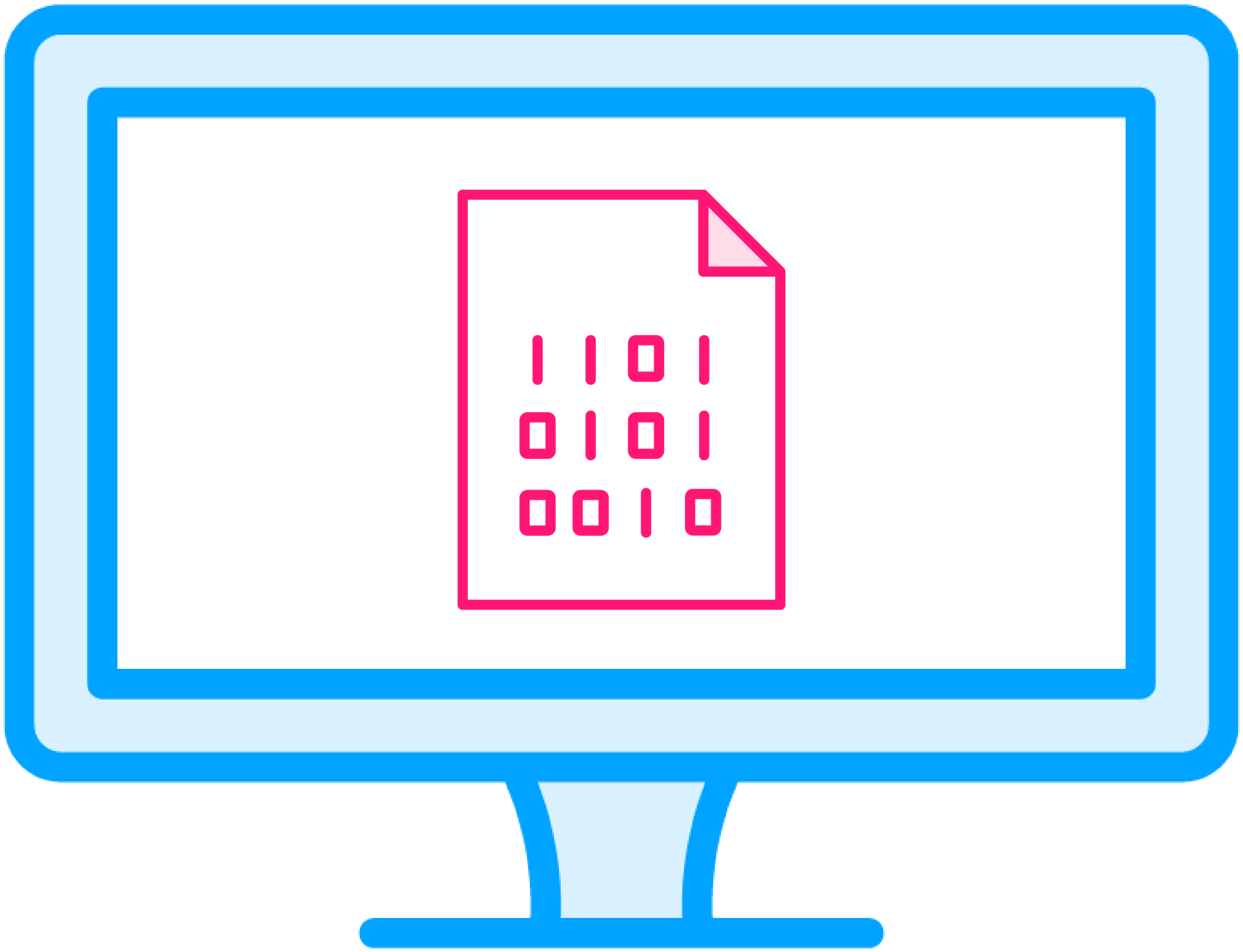
Helping developers create amazing applications

@xmorera / www.xaviermorera.com / www.bigdatainc.org

Why Are Files Important?

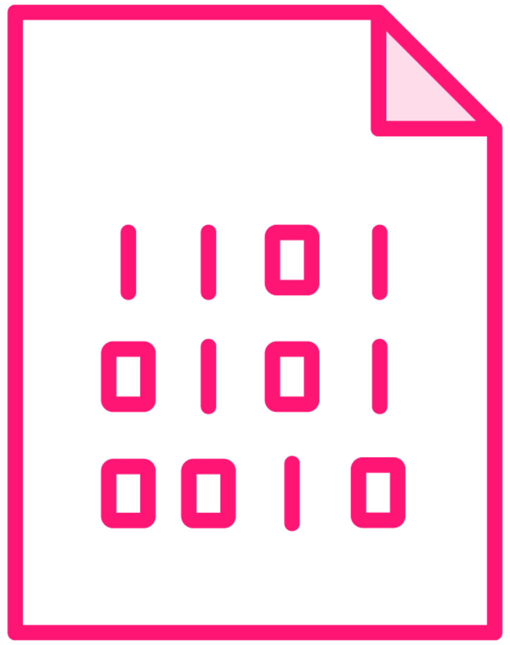


Why Are Files Important?



Files

Used for persistent storage



Define: File

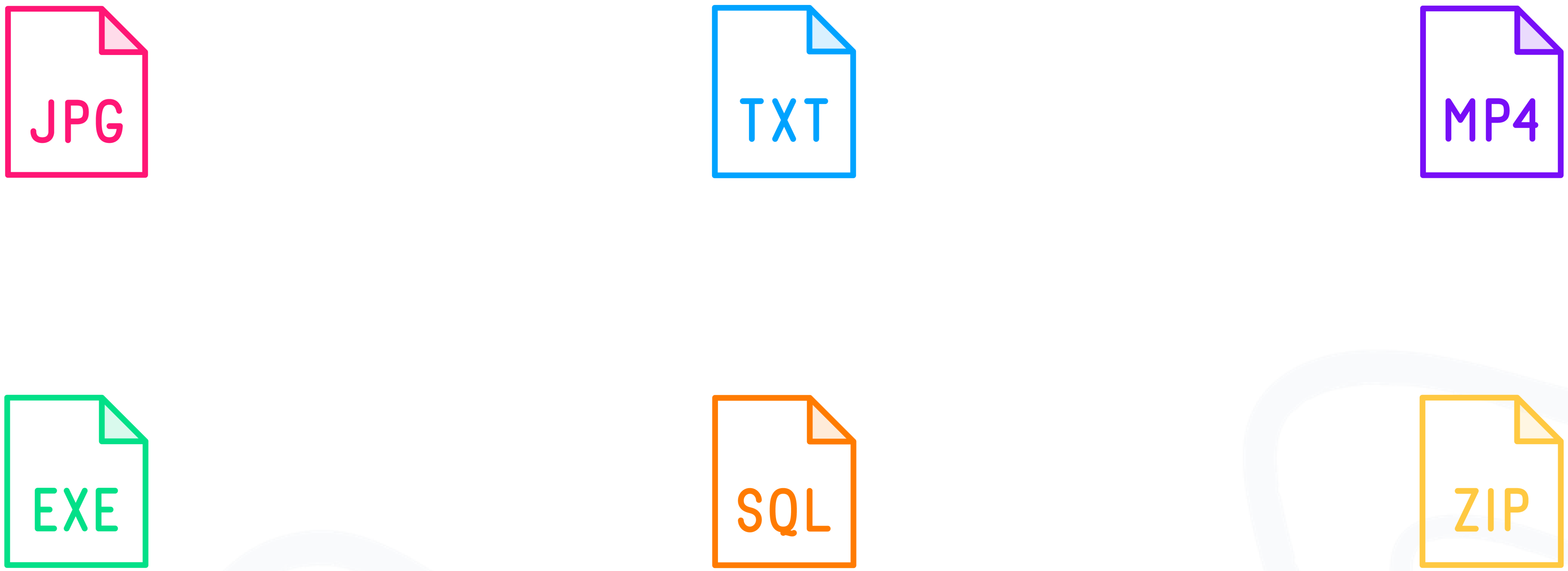
A computer file is a computer resource for recording data in a computer storage device, primarily identified by its file name.

Just as words can be written to paper, so can data be written to a computer file.

Files can be shared with and transferred between computers and mobile devices via removable media, networks, or the Internet.

https://en.wikipedia.org/wiki/Computer_file

Different Types of Files



Reasons for Working with Files

- Data storage and retrieval
- Data processing
- Logging and debugging
- Data transmission
- Other related scenarios

Files

Working With Files

Directories

Processes

Environment variables

The Libraries

os

shutil

psutil

The demo environment

Processes with psutil

psutil

Cross-platform library

- Linux, MacOS, Windows
- FreeBSD, OpenBSD, NetBSD, Sun Solaris, AIX

Not part of the Python Standard Library

Search projects

Help Sponsors Log in Register

psutil 5.9.4

```
pip install psutil
```

Latest version

Released: Nov 7, 2022

Cross-platform lib for process and system monitoring in Python.

Navigation

- Project description
- Release history
- Download files

Project links

- Homepage

Statistics

GitHub statistics:

- **Stars:** 9115
- **Forks:** 1310

Project description

downloads 63M/month | stars 9.1k | forks 1.3k | contributors 166 | coverage 93%
pypi v5.9.4 | python 2.7 | 3 | in repositories 38 | license BSD-3-Clause
Linux, macOS, FreeBSD https://github.com/badges/shields/issues/8671 | Windows passing | docs passing | follow | lifted!

Quick links

- Home page
- Install
- Documentation
- Download
- Forum
- StackOverflow
- Blog
- What’s new

psutil

Used to retrieve system information and interact with running processes

- CPU, memory, disks, network, sensors
- Manage system processes
- Information on system users and sessions
- Monitoring system utilization in real-time

```python
import psutil

psutil.process_iter()
```

Retrieve Running Processes

Use process_iter to get a list of processes currently executing

- Can iterate and inspect each process

Disks

`psutil.disk_partitions(all=False)`

Return all mounted disk partitions as a list of named tuples including device, mount point and filesystem type, similarly to "df" command on UNIX. If *all* parameter is `False` it tries to distinguish and return physical devices only (e.g. hard disks, cd-rom drives, USB keys) and ignore all others (e.g. pseudo, memory, duplicate, inaccessible filesystems). Note that this may not be fully reliable on all systems (e.g. on BSD this parameter is ignored). See disk_usage.py script providing an example usage. Returns a list of named tuples with the following fields:

- **device**: the device path (e.g. `"/dev/hda1"`). On Windows this is the drive letter (e.g. `"C:\\"`).
- **mountpoint**: the mount point path (e.g. `"/"`). On Windows this is the drive letter (e.g. `"C:\\"`).
- **fstype**: the partition filesystem (e.g. `"ext3"` on UNIX or `"NTFS"` on Windows).
- **opts**: a comma-separated string indicating different mount options for the drive/partition. Platform-dependent.
- **maxfile**: the maximum length a file name can have.
- **maxpath**: the maximum length a path name (directory name + base file name) can have.

```
>>> import psutil
>>> psutil.disk_partitions()
[sdiskpart(device='/dev/sda3', mountpoint='/', fstype='ext4', opts='rw,errors=remount-ro', maxfile=255, maxpath=4096),
 sdiskpart(device='/dev/sda7', mountpoint='/home', fstype='ext4', opts='rw', maxfile=255, maxpath=4096)]
```

Changed in version 5.7.4: added *maxfile* and *maxpath* fields

`psutil.disk_usage(path)`

Return disk usage statistics about the partition which contains the given *path* as a named tuple including **total**, **used** and **free** space expressed in bytes, plus the **percentage** usage. `OSError` is raised if *path* does not exist. Starting from Python 3.3 this is also available as shutil.disk_usage (see BPO-12442). See disk_usage.py script providing an example usage.

```
>>> import psutil
>>> psutil.disk_usage('/')
sdiskusage(total=21378641920, used=4809781248, free=15482871808, percent=22.5)
```

Note: UNIX usually reserves 5% of the total disk space for the root user. *total* and *used* fields on UNIX refer to the overall total and used space, whereas *free* represents the space available for the **user** and *percent* represents the **user** utilization (see source code). That is why *percent* value may look 5% bigger than what you would expect it to be. Also note that both 4 values match "df" cmdline utility.

Changed in version 4.3.0: percent value takes root reserved space into account.

`psutil.disk_io_counters(perdisk=False, nowrap=True)`

Working with Files Using psutil

disk_partitions

disk_usage

disk_io_counters

Disks with psutils

Operations related to files

```
import psutil

psutil.disk_partitions()

psutil.disk_usage('/')

psutil.disk_io_counters(perdisk=False, nowrap=True)
psutil.disk_io_counters(perdisk=True)
```

Processes and disks with psutil

Platform Information and Environment Variables

os — Miscellaneous operating system interfaces

Source code: Lib/os.py

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see `open()`, if you want to manipulate paths, see the `os.path` module, and if you want to read all the lines in all the files on the command line see the `fileinput` module. For creating temporary files and directories see the `tempfile` module, and for high-level file and directory handling see the `shutil` module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about *path* in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the `os` module, but using them is of course a threat to portability.
- All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.
- On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.
- On WebAssembly platforms `wasm32-emscripten` and `wasm32-wasi`, large parts of the `os` module are not available or behave differently. API related to processes (e.g. `fork()`, `execve()`), signals (e.g. `kill()`, `wait()`), and resources (e.g. `nice()`) are not available. Others like `getuid()` and `getpid()` are emulated or stubs.

Note: All functions in this module raise `OSError` (or subclasses thereof) in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception `os.error`

An alias for the built-in `OSError` exception.

os

Portable way of using operating system dependent functionality

- Platform independent
- Write portable code

Provides functions for

- Files and directories
    - Creating, deleting, renaming, and listing
    - Check if they exist and permissions
- Work with environment variables

Platform Information and Environment Variables with os

```python
import os

os.name

os.listdir("/users/xavier/Downloads")
os.listdir("c:\Users\Xavier\Downloads")

os.getcwd()

os.environ
os.environ["TMPDIR"]
os.getenv("TMPDIR")
os.getenv("HOMEPATH")
os.environ["HOMEPATH"]
```

Platform information and environment variables with os

Understanding the Differences in File Structures

Differences in File Structures

Windows

MacOS

Linux

Differences in File Structures

File structure is different

!=

File contents is the same (usually)

Different File Structures

Path Separators

Root Directory

Case Sensitivity

Hidden Files

Different File Structures

File Extensions

File Permissions

File System Types

Path Separators

/

vs.

\

Linux and Mac

- Use forward slashes
- `/home/user/documents/file.txt`

Windows

- Use backslashes
- `C:\Users\user\Documents\file.txt`

Root Directory

/

vs.

C:\

Linux and Mac

- Denoted by a single forward slash
- /

Windows

- Drive letter followed by a backslash
- C:\

Case Sensitivity

File

vs.

file.txt

Linux and Mac

- Case sensitive
- `file.txt` and `File.txt` are different

Windows

- Case insensitive
- `file.txt` and `File.txt` is the same file

Hidden Files

.dir

Linux and Mac

- Preceded by a dot
- `.hidden_file`

Windows

- Marked as hidden in their properties

File Extensions

.txt

Linux and Mac

- Can use file header to determine file type

Windows

- Typically relies on extensions to determine file type

File Permissions

rwx

vs.

ACL

Linux and Mac

- Permission system based on read/write/execute for owner and group

Windows

- Access Control List (ACL)

File System Types

Ext4

APFS

NTFS

Linux

- Typically uses Ext4

Mac

- APFS

Windows

- NTFS or FAT32

Differences in file structures between operating systems

Working with Directories and Locating Files

Working with Directories and Locating Files

- Creating, locating, and changing directory
- Listing files in a directory
- Finding a file or several files
- Copying, moving, and renaming files and directories

Working with Directories and Locating Files

os

shutil

pathlib

shutil

High-level operations

- Mainly copying, moving, and deleting
- Files and collections of files
- Directories

Stands for "shell utilities"

- Similar functions to cp, mv, rm…

Table of Contents

Previous topic

`linecache` — Random access to text lines

Next topic

Data Persistence

This Page

Report a Bug
Show Source

`shutil` — High-level file operations

Source code: Lib/shutil.py

The `shutil` module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the `os` module.

Warning: Even the higher-level file copying functions (`shutil.copy()`, `shutil.copy2()`) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

Directory and files operations

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])

Copy the contents of the file-like object *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

shutil.**copyfile**(*src*, *dst*, *, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst* in the most efficient way possible. *src* and *dst* are path-like objects or path names given as strings.

dst must be the complete target file name; look at `copy()` for a copy that accepts a target directory path. If *src* and *dst* specify the same file, `SameFileError` is raised.

The destination location must be writable; otherwise, an `OSError` exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot

pathlib

Provides classes representing filesystem paths

- Suitable for different operating systems

Divided between

- Pure paths
  - Provide purely computational operations
  - Without I/O
- Concrete paths
  - Inherit from pure paths
  - Also provide I/O operations

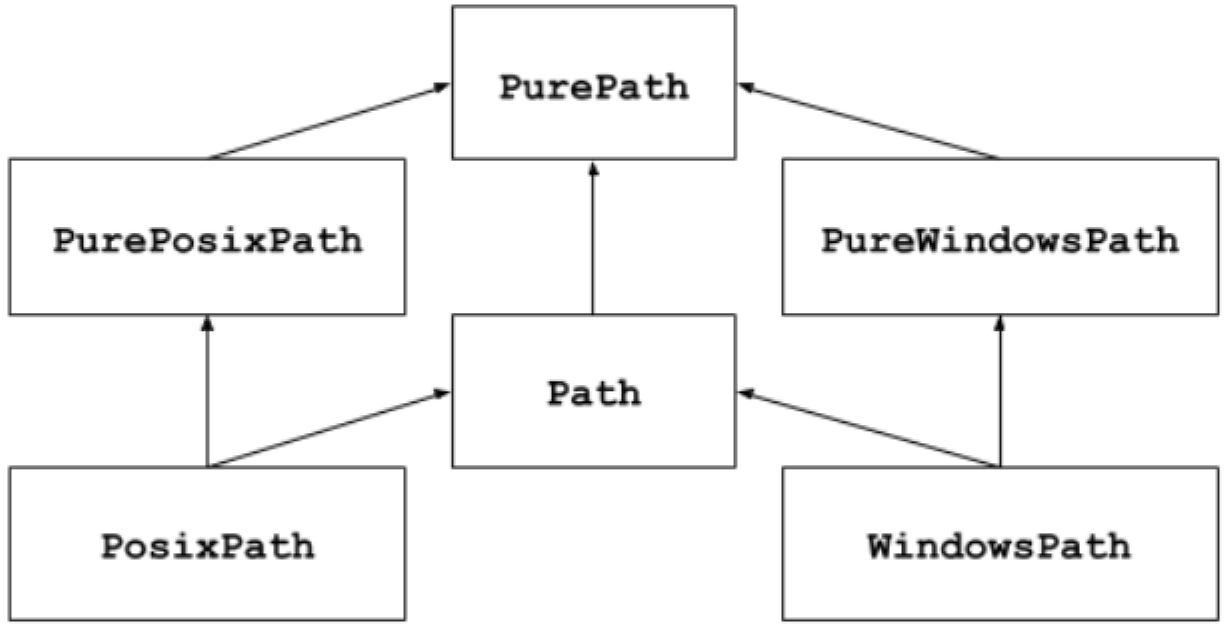
Recommended way to work with paths

`pathlib` — Object-oriented filesystem paths

New in version 3.4.

Source code: Lib/pathlib.py

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.



If you've never used this module before or just aren't sure which class is right for your task, `Path` is most likely what you need. It instantiates a concrete path for the platform the code is running on.

Pure paths are useful in some special cases; for example:

1. If you want to manipulate Windows paths on a Unix machine (or vice versa). You cannot instantiate a `WindowsPath` when running on Unix, but you can instantiate `PureWindowsPath`.
2. You want to make sure that your code only manipulates paths without actually accessing the OS. In this case, instantiating one of the pure classes may be useful since those simply don't have any OS-accessing operations.

os, shutil, and pathlib

os

- rename
- mkdir / makedirs
- getcwd
- path.join
- path.abspath
- listdir
- chdir

shutil

- copy
- copyfile / copytree
- move
- rmtree
- chown* (Unix)
- which

** Platform dependent ops*

pathlib

- PurePath
  - drive, parts, root…
- home
- cwd
- exists
- rename
- mkdir

Working with directories and locating files

Evaluating and Opening Files

Evaluating and Opening Files

Open an existing file

Create a new file

Open

Access an existing file

- Reading, writing, or for appending

Use the `open( )` **function**

Takes two arguments

- Name of the file to be opened
- In which mode the file should be opened
    - Mode used to specify type of operation to perform on the opened file

Returns a file object on success

An exception is raised if the file does not exist

Create

Create a new file

- Provide the name of the new file

Also uses the `open()` **function**

- Use either `x` or `w` mode

Mode `x`**, which means create**

- If the file exists, an error is raised

Mode `w` **, which means write**

- Overwrites the file if it exists

Mode

x	r	w	a
Create	Read	Write	Append

Mode

t

Text

b

Binary

Mode

rt

Text

rb

Binary

file is a path-like object giving the pathname (absolute or relative to the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned 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 exclusive creation, 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.getencoding()` 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'`	open for exclusive creation, failing if the file already exists
`'a'`	open for writing, appending to the end of file if it exists
`'b'`	binary mode
`'t'`	text mode (default)
`'+'`	open for updating (reading and writing)

The default mode is `'r'` (open for reading text, a synonym of `'rt'`). Modes `'w+'` and `'w+b'` open and truncate the file. Modes `'r+'` and `'r+b'` open the file with no truncation.

Opening and Creating Files

```python
# Open, check if the file exists
open("file.txt", "x")
```

Opening and Creating Files

file mode

```
open("file.txt", "x")
```

Opening and Creating Files

```
open("file.txt", "x",            )
```

file

mode

buffering
encoding
errors
newline
closed
opener

```python
f = open("file.txt", buffering=1)
```

Buffering

Specifies if the file should be buffered

- -1 is the default value, means the file will be fully buffered
- 0 for no buffering
- 1 for line buffering, or specify a positive integer for a buffer size in bytes

```python
f = open("file.txt", encoding='utf-8')
```

Encoding

Specifies encoding used to decode the file

Default is None

- File read and written as binary data

A sample encoding is utf-8

```python
f = open("file.txt", errors='ignore')
```

Errors

Specifies how errors are handled during decoding
Default is strict
Other options are ignore and replace

```python
f = open("file.txt", newline='\n')
```

Newline

Specify how line endings should be handled

- Default is None
- Uses the default character for the current platform

```
f = open(fd, closefd=False)
```

Closefd

Specify whether the file descriptor should be closed when the file is closed

- Default is true
- Can set to false to leave it open

Used with a file descriptor instead of a file name

```python
# Example opener function
def custom_opener(filename, mode):
    filename = f"prefix_{filename}"
    return open(filename, mode)

f = open("file.txt", opener=custom_opener)
```

Opener

Specify a custom opener for files

- Default value is None
- Built-in open function will be used

Opening and creating files

Opening and Creating Files

```
open("file.txt", "w")
```

Opening and Creating Files

```
open("file.txt", "r")
```

Opening and Creating Files

```
open("file.txt", encoding="utf-8")
```

Opening and Creating Files

```
open("file.txt", newline="\n")
```

Opening and Creating Files

```
open("file.txt", errors="ignore")
```

Reading and Writing Files

Reading and Writing to Files

Read

Write

Open the File First

```
open("file.txt", "r")
```

Read

```
open("file.txt", "w")
open("file.txt", "a")
open("file.txt", "x")
```

Write

File Object

```
file = open("file.txt", "r")
```

file object

An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object

A synonym for file object.

filesystem encoding and error handler

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The filesystem encoding and error handler are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the locale encoding.

finder

```python
# Read the entire contents of the file
content = file.read()

# Read a single line from the file
line = file.readline()

# Iterate over the file object line by line
for line in file:
    print(line)
```

Read

Read the contents of a file using the `read()` method
Or can read one line at a time using `readline()`
Also possible to iterate over the file object

```python
# Append a single line of text using write
f.write("This is line 2\n")

# Append multiple lines of text using writelines and a list
# it's important to include the newline character \n at the end of each line of text, as
writelines does not add it automatically.
lines = ["This is line 3\n", "This is line 4\n", "This is line 5\n"]
f.writelines(lines)

# Append multiple lines of text using writelines and a generator expression
f.writelines(f"This is line {i}\n" for i in range(6, 9))
```

Write and Append

Required to open in `w` , `a`, or `x` mode

Two methods available

- `write()` expects a single string
- `writelines()` expects an iterable of strings

Reading and writing files

Closing Files

```python
# Open the file in append mode
file = open("file.txt", "a")

# Write data to the file
file.write("This is a new line in the file.\n")

# Close the file
file.close()
```

Close

When done reading or writing to a file, it is necessary to close the file
- Using the `close()` method

Any operation on a closed file will raise a ValueError

Do not leave files open unnecessarily

What happens if I don't close files that I open?

This Happens if You Do Not Close a File

Resource leakage

File locks

Data corruption

```python
# Using 'with' keyword with 'close' method
with open('example.txt', 'r') as file:
    data = file.read()
    print(data)

# File is automatically closed outside the 'with' block
```

Using with

Used to create a context
File is automatically closed when the block is exited

Closing files

Deleting Files and Directories

```python
# Delete a file
os.remove("file.txt")

# os.unlink() is an alias to os.remove()
os.unlink("file.txt")

# Another way to remove a file
pathlib.Path("file.txt").unlink()
```

Deleting Files

Use the os.remove() function

- A FileNotFoundError exception raised if file does not exist
- Can also use os.unlink(), which is an alias

Additionally, can also use `pathlib.Path.unlink()` to delete files

```python
# Delete the file
os.remove("file.txt")

# Delete the directory (must be empty)
os.rmdir("my_files")

# Delete the directory and all its contents
shutil.rmtree("my_files")
```

Deleting Directories

- Delete a directory using `os.rmdir()`
  - Directory needs to empty, else an error is raised
- Can delete a directory and all of its contents using `shutil.rmtree()`

Deleting files and directories

Final Takeaway

Final Takeaway

Ability to create and manipulate files

- Fundamental aspect of programming

Requires knowing about other aspects

- Including directories, processes, and environment variables

Final Takeaway

Processes access files

Use `psutil` to manage and retrieve information

- CPU and memory
- System users and sessions
- Disk-related information

Final Takeaway

Retrieve and manage platform information using the os module

- Write portable code
- Work with files and directories
  - Create, delete, rename, list
  - Check if they exist
  - Permissions
- Access environment variables

Final Takeaway

Several operating systems available

- Windows, MacOS, Linux
- File contents is (usually) the same

Differences in file structures for each operating system

- Path separators, root directory, case sensitivity, hidden files, file extensions, file permissions, file system types

Final Takeaway

Working with directories and locating files

- os, shutil, pathlib

shutil

- Mainly for copying, moving, and deleting
- Files and collections of files

Stands for shell utilities

- cp, mv, rm

Final Takeaway

pathlib

- Provides classes for representing filesystem paths

Pure paths

- Provide purely computational operations

Concrete paths

- Inherit from pure paths
- Additionally provide I/O operations

Final Takeaway

Open and create files

- Use the `open()` function
- Set the mode
  - x for create
  - w for write
  - a for append

Other available parameters

- `buffering`, `encoding`, `errors`, `newline`, `closed`, `opener`

Final Takeaway

Reading files

- Read the contents of a file using the `read()` method
- Or can read one line at a time using `readline()`
- Also, possible to iterate over the file object

Final Takeaway

Writing to files

- Required to open in `w` , a, or x mode

Two methods available

- `write()` expects a single string
- writelines() expects an iterable of strings

Final Takeaway

Close files after you are done

- Use the `close()` method

Or use a context manager

- The `with` keyword

Final Takeaway

Delete files using `os.remove()`

Other options available

- `os.unlink()`
- `pathlib.Path.unlink()`

Delete directories using `os.rmdir()`

- Needs to be empty

Remove non-empty directories

- Using `shutil.rmtree()`

Thanks for watching!

@xmorera

www.xaviermorera.com

“What you learn is yours for life.”