Python File IO

Table of contents

Opening a file 1
Reading 1
Closing 3
Passing files around: `str` vs `TextIO` 3
Writing and appending 4

Opening a file

```
file = open("path", "<mode>")
```

Mode	Description
`r`	read
`w`	write (creates or **overwrites**)
`a`	append (creates or adds to the end)

A bare file name is resolved relative to the directory the running Python is in; `os.getcwd()` reports that directory and `os.chdir()` changes it.

Reading

`file.readline()` returns one line (including its trailing `\n`, except possibly the last line of the file), then `''` forever once exhausted. A for-loop iterates line by line and is the idiomatic way to consume a whole file:

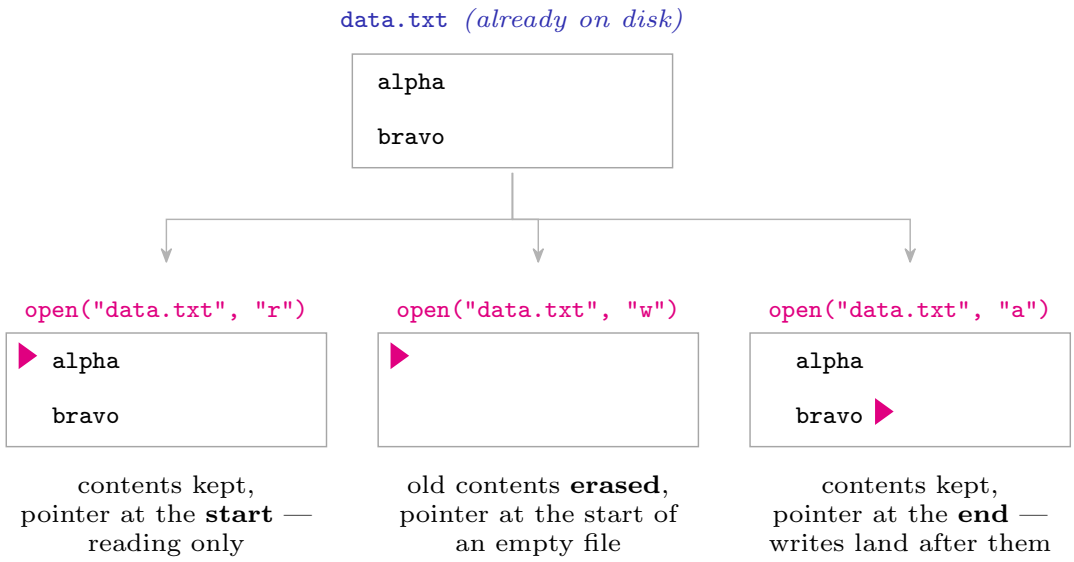

Figure 1: What each mode does to a file that already exists, and where it leaves the file-pointer. This is the difference that matters in practice: `w` destroys what was there before anything is written, while `a` keeps it and starts writing past the end.

```
with open("path", "r") as file:
    for line in file:
        ...
```

Reading always gives back **strings** — cast (`int(...)`, `float(...)`, …) as needed.

A file-pointer only moves forward: once a `for` loop (or repeated `readline()` calls) has consumed a file, iterating again yields nothing further unless the file is reopened or the pointer is reset with `file.seek(0)`. One consequence is useful: a lone `file.readline()` before the loop is how you skip a CSV header, because the call's only lasting effect is on the pointer.

Closing

`file.close()` releases the file; forgetting to close leaves it vulnerable to side effects (missing or extra data). `with open(...) as file:` closes it automatically — even if the block raises an exception — so it's preferred over manual `open`/`close`.

Passing files around: `str` vs `TextIO`

A function that does something to a file can take either the file's **name** or an already-open **file-pointer**, and the two are genuinely different interfaces:

```
from typing import TextIO

def f(path: str) -> ...:          # takes a name; opens (and closes) the file itself
    with open(path, "r") as file:
        ...

def g(file_pointer: TextIO) -> ...:   # takes an open file; the caller owns it
    for line in file_pointer:
        ...
```

`TextIO` is the type hint (imported from `typing`) for an open text file. A function of the second shape reads from wherever the pointer currently is, and leaves it wherever it stops — so its behaviour depends on what the caller has already read.

Writing and appending

```
with open("path", "w") as file:
    file.write("a single string\n")
    file.writelines(["one string\n", "per element\n"])
```

`"w"` overwrites any existing file; `"a"` instead appends to the end without touching existing content. Both write exactly the characters given — any newlines have to be part of the strings.