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

Open file and return a corresponding file object. If the file cannot be opened, an OSError is raised.

file is either a string or bytes 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.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' open for exclusive creation, failing if the file already exists

'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 newlines mode (deprecated)

The default mode is 'r' (open for reading text, synonym of 'rt'). For binary read-write access, the mode 'w+b' opens and truncates the file to 0 bytes. 'r+b' opens the file without truncation.

As mentioned in the Overview, Python distinguishes between binary and text I/O. Files opened in binary mode (including 'b' in the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is included in the mode argument), the contents of the file are returned as str, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

Note:

Python doesn’t depend on the underlying operating system’s notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

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 in bytes 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 (whatever locale.getpreferredencoding() returns), but any encoding supported by Python can be used. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding and decoding errors are to be handled–this cannot be used in binary mode. A variety of standard error handlers are available, though any error handling name that has been registered with codecs.register\_error() is also valid. The standard names are:

•'strict' to raise a ValueError exception if there is an encoding error. The default value of None has the same effect.

•'ignore' ignores errors. Note that ignoring encoding errors can lead to data loss.

•'replace' causes a replacement marker (such as '?') to be inserted where there is malformed data.

•'surrogateescape' will represent any incorrect bytes as code points in the Unicode Private Use Area ranging from U+DC80 to U+DCFF. These private code points will then be turned back into the same bytes when the surrogateescape error handler is used when writing data. This is useful for processing files in an unknown encoding.

•'xmlcharrefreplace' is only supported when writing to a file. Characters not supported by the encoding are replaced with the appropriate XML character reference &#nnn;.

•'backslashreplace' (also only supported when writing) replaces unsupported characters with Python’s backslashed escape sequences.

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

•When reading input from the stream, 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 newlines 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.

•When writing output to the stream, 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 and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given closefd has no effect and must be True (the default).

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).

The newly created file is non-inheritable.

The following example uses the dir\_fd parameter of the os.open() function to open a file relative to a given directory:

>>>>>> import os

>>> dir\_fd = os.open('somedir', os.O\_RDONLY)

>>> def opener(path, flags):

... return os.open(path, flags, dir\_fd=dir\_fd)

...

>>> with open('spamspam.txt', 'w', opener=opener) as f:

... print('This will be written to somedir/spamspam.txt', file=f)

...

>>> os.close(dir\_fd) # don't leak a file descriptor

The type of file object returned by the open() function depends on the mode. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a subclass of io.TextIOBase (specifically io.TextIOWrapper). When used to open a file in a binary mode with buffering, the returned class is a subclass of io.BufferedIOBase. The exact class varies: in read binary mode, it returns a io.BufferedReader; in write binary and append binary modes, it returns a io.BufferedWriter, and in read/write mode, it returns a io.BufferedRandom. When buffering is disabled, the raw stream, a subclass of io.RawIOBase, io.FileIO, is returned.

See also the file handling modules, such as, fileinput, io (where open() is declared), os, os.path, tempfile, and shutil.

Changed in version 3.3: The opener parameter was added. The 'x' mode was added. IOError used to be raised, it is now an alias of OSError. FileExistsError is now raised if the file opened in exclusive creation mode ('x') already exists.

Changed in version 3.4: The file is now non-inheritable.

The 'U' mode.

Deprecated since version 3.4, will be removed in version 4.0.