Chapter 11. File and Text Operations

This chapter covers issues related to files and filesystems in Python. A *file* is a stream of text or bytes that a program can read and/or write; a *filesystem* is a hierarchical repository of files on a computer system.

OTHER CHAPTERS THAT ALSO DEAL WITH FILES

Files are a crucial concept in programming: so, although this chapter is one of the largest in the book, other chapters also have material relevant to handling specific kinds of files. In particular, **Chapter 12** deals with many kinds of files related to persistence and database functionality (CSV files in **Chapter 12**, JSON files in **"The json Module"**, pickle files in **"The pickle Module"**, shelve files in **"The shelve Module"**, DBM and DBM-like files in **"The dbm Package"**, and SQLite database files in **"SQLite"**), **Chapter 22** deals with files in HTML format, and **Chapter 23** deals with files in XML format.

Files and streams come in many flavors. Their contents can be arbitrary bytes, or text. They may be suitable for reading, writing, or both, and they may be *buffered,* so that data is temporarily held in memory on the way to or from the file. Files may also allow *random access*, moving forward and back within the file, or jumping to read or write at a particular location in the file. This chapter covers each of these topics.

In addition, this chapter also covers the polymorphic concept of file-like objects (objects that are not actually files but behave to some extent like files), modules that deal with temporary files and file-like objects, and modules that help you access the contents of text and binary files and support compressed files and other data archives. Python's standard library supports several kinds of **lossless compression**, including (ordered by the typical ratio of compression on a text file, from highest to lowest):

- LZMA (used, for example, by the `xz` program), see module `lzma`
- bzip2 (used, for example, by the `bzip2` program), see module `bz2`
- deflate (used, for example, by the `gzip` and `zip` programs), see modules `zlib`, `gzip`, and `zipfile`

The `tarfile` module lets you read and write TAR files compressed with any one of these algorithms. The `zipfile` module lets you read and write ZIP files and also handles bzip2 and LZMA compressions. We cover both of these modules in this chapter. We don't cover the details of compression in this book; for details, see the online docs.

In the rest of this chapter, we will refer to all files and file-like objects as files.

In modern Python, input/output (I/O) is handled by the standard library's `io` module. The `os` module supplies many of the functions that operate on the filesystem, so this chapter also introduces that module. It then covers operations on the filesystem (comparing, copying, and deleting directories and files; working with filepaths; and accessing low-level file descriptors) provided by the `os` module, the `os.path` module, and the new and preferable `pathlib` module, which provides an object-oriented approach to filesystem paths. For a cross-platform interprocess communication (IPC) mechanism known as *memory-mapped files*, see the module `mmap`, covered in Chapter 15.

While most modern programs rely on a graphical user interface (GUI), often via a browser or a smartphone app, text-based, nongraphical "command-line" user interfaces are still very popular for their ease, speed of use, and scriptability. This chapter concludes with a discussion of non-GUI text input and output in Python in "Text Input and Output", terminal text I/O in "Richer-Text I/O", and, finally, how to build software showing text understandable to different users, across languages and cultures, in "Internationalization".

The io Module

As mentioned in this chapter's introduction, `io` is the standard library module in Python that provides the most common ways for your Python programs to read or write files. In modern Python, the built-in function `open` is an alias for the function `io.open`. Use `io.open` (or its built-in alias `open`) to make a Python file object to read from, and/or write to, a file as seen by the underlying operating system. The parameters you pass to open determine what type of object is returned. This object can be an instance of `io.TextIOWrapper` if textual, or, if binary, one of `io.BufferedReader`, `io.BufferedWriter`, or `io.BufferedRandom`, depending on whether it's read-only, write-only, or read/write. This section covers the various types of file objects, as well as the important issue of making and using *temporary* files (on disk, or even in memory).

I/O ERRORS RAISE OSERROR

Python reacts to any I/O error related to a file object by raising an instance of built-in exception class `OSError` (many useful subclasses exist, as covered in **"OSError subclasses"**). Errors causing this exception include a failing `open` call, calls to a method on a file to which the method doesn't apply (e.g., `write` on a read-only file, or `seek` on a nonseekable file), and actual I/O errors diagnosed by a file object's methods.

The `io` module also provides the underlying classes, both abstract and concrete, that, by inheritance and by composition (also known as *wrapping*), make up the file objects that your program generally uses. We do not cover these advanced topics in this book. If you have access to unusual channels for data, or nonfilesystem data storage, and want to provide a file interface to those channels or storage, you can ease your task (through appropriate subclassing and wrapping) using other classes in the `io` module. For assistance with such advanced tasks, consult the **online docs**.

Creating a File Object with open

To create a Python file object, call open with the following syntax:

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

file can be a string or an instance of pathlib.Path (any path to a file as seen by the underlying OS), or an int (an OS-level *file descriptor* as returned by os.open, or by whatever function you pass as the opener argument). When file is a path (a string or pathlib.Path instance), open opens the file thus named (possibly creating it, depending on the mode argument—despite its name, open is not just for opening existing files: it can also create new ones). When file is an integer, the underlying OS file must already be open (via os.open).

OPENING A FILE PYTHONICALLY

open is a context manager: use **with** open(...) **as** *f*:, *not* *f* = open(...), to ensure the file *f* gets closed as soon as the **with** statement's body is done.

---

open creates and returns an instance *f* of the appropriate io module class, depending on the mode and buffering settings. We refer to all such instances as file objects; they are polymorphic with respect to each other.

mode

mode is an optional string indicating how the file is to be opened (or created). The possible values for mode are listed in Table 11-1.

Table 11-1. mode settings

Mode	Meaning
'a'	The file is opened in write-only mode. The file is kept intact if it already exists, and the data you write is

Mode	Meaning
	appended to the existing contents. The file is created if it does not exist. Calling *f*.`seek` on the file changes the result of the method *f*.`tell`, but does not change the write position in the file opened in this `mode`: that write position always remains at the end of the file.
`'a+'`	The file is opened for both reading and writing, so all methods of *f* can be called. The file is kept intact if it already exists, and the data you write is appended to the existing contents. The file is created if it does not exist. Calling *f*.`seek` on the file, depending on the underlying operating system, may have no effect when the next I/O operation on *f* *writes* data, but does work normally when the next I/O operation on *f* *reads* data.
`'r'`	The file must already exist, and it is opened in read-only mode (this is the default).
`'r+'`	The file must exist and is opened for both reading and writing, so all methods of *f* can be called.
`'w'`	The file is opened in write-only mode. The file is truncated to zero length and overwritten if it already exists, or created if it does not exist.
`'w+'`	The file is opened for both reading and writing, so all methods of *f* can be called. The file is truncated to zero length and overwritten if it already exists, or created if it does not exist.

Binary and text modes

The `mode` string may include any of the values in **Table 11-1**, followed by a `b` or `t`. `b` indicates that the file should be opened (or created) in binary mode, while `t` indicates text mode. When neither `b` nor `t` is included, the default is text (i.e., `'r'` is like `'rt'`, `'w+'` is like `'w+t'`, and so on), but per **The Zen of Python**, "explicit is better than implicit."

Binary files let you read and/or write strings of type `bytes`, and text files let you read and/or write Unicode text strings of type `str`. For text files, when the underlying channel or storage system deals in bytes (as most do), `encoding` (the name of an encoding known to Python) and `errors` (an error-handler name such as `'strict'`, `'replace'`, and so on, as covered under `decode` in **Table 9-1**) matter, as they specify how to translate between text and bytes, and what to do on encoding and decoding errors.

Buffering

`buffering` is an integer value that denotes the buffering policy you're requesting for the file. When `buffering` is `0`, the file (which must be binary mode) is unbuffered; the effect is as if the file's buffer is flushed every time you write anything to the file. When `buffering` is `1`, the file (which *must* be open in text mode) is line buffered, which means the file's buffer is flushed every time you write `\n` to the file. When `buffering` is greater than `1`, the file uses a buffer of about `buffering` bytes, often rounded up to some value convenient for the driver software. When `buffering` is `<0`, a default is used, depending on the type of file stream. Normally, this default is line buffering for files that correspond to interactive streams, and a buffer of `io.DEFAULT_BUFFER_SIZE` bytes for other files.

Sequential and nonsequential ("random") access

A file object *f* is inherently sequential (a stream of bytes or text). When you read, you get bytes or text in the sequential order in which they are present. When you write, the bytes or text you write are added in the order in which you write them.

For a file object *f* to support nonsequential access (also known as random access), it must keep track of its current position (the position in the storage where the next read or write operation starts transferring data), and the underlying storage for the file must support setting the current position. `f.seekable` returns **`True`** when *f* supports nonsequential access.

When you open a file, the default initial read/write position is at the start of the file. Opening *f* with a `mode` of `'a'` or `'a+'` sets *f*'s read/write position to the end of the file before writing data to *f*. When you write or read *n* bytes to/from file object *f*, *f*'s position advances by *n*. You can query the current position by calling `f.tell`, and change the position by calling `f.seek`, both covered in the next section.

When calling `f.seek` on a text-mode *f*, the offset you pass must be `0` (to position *f* at the start or end, depending on `f.seek`'s second parameter), or the opaque result returned by an earlier call to `f.tell`,[1] to position *f* back to a spot you had thus "bookmarked" before.

Attributes and Methods of File Objects

A file object *f* supplies the attributes and methods documented in Table 11-2.

Table 11-2. Attributes and methods of file objects

`close`	`close()` Closes the file. You can call no other method on *f* after `f.close`. Multiple calls to `f.close` are allowed and innocuous.
`closed`	`f.closed` is a read-only attribute that is **`True`** when `f.close()` has been called; otherwise, it is **`False`**.
`encoding`	`f.encoding` is a read-only attribute, a string naming the encoding (as covered in "Unicode"). The attribute does not exist on binary files.
`fileno`	`fileno()` Returns the file descriptor of *f*'s file at operating

	system level (an integer). File descriptors are covered in **“File and directory functions of the os module”**.
`flush`	`flush()` Requests that *f*’s buffer be written out to the operating system, so that the file as seen by the system has the exact contents that Python’s code has written. Depending on the platform and the nature of *f*’s underlying file, `f.flush` may not be able to ensure the desired effect.
`isatty`	`isatty()` Returns **`True`** when *f*’s underlying file is an interactive stream, such as to or from a terminal; otherwise, returns **`False`**.
`mode`	`f.mode` is a read-only attribute that is the value of the `mode` string used in the `io.open` call that created *f*.
`name`	`f.name` is a read-only attribute that is the value of the file (`str` or `bytes`) or `int` used in the `io.open` call that created *f*. When `io.open` was called with a `pathlib.Path` instance *p*, `f.name` is `str(p)`.
`read`	`read(size=-1, /)` When *f* is open in binary mode, reads up to *size* bytes from *f*’s file and returns them as a bytestring. `read` reads and returns less than *size* bytes if the file ends before *size* bytes are read. When *size* is less than `0`, `read` reads and returns all bytes up to the end of the file. `read` returns an empty string when the file’s current position is at the end of the file or when *size* equals `0`. When *f* is open in text mode, *size* is a number of characters, not bytes, and `read` returns a text string.

`readline`	`readline(size=-1, /)` Reads and returns one line from *f*'s file, up to the end of line (`\n`), included. When *size* is greater than or equal to `0`, reads no more than *size* bytes. In that case, the returned string might not end with `\n`. `\n` might also be absent when `readline` reads up to the end of the file without finding `\n`. `readline` returns an empty string when the file's current position is at the end of the file or when *size* equals `0`.
`readlines`	`readlines(size=-1, /)` Reads and returns a list of all lines in *f*'s file, each a string ending in `\n`. If `size > 0`, `readlines` stops and returns the list after collecting data for a total of about *size* bytes rather than reading all the way to the end of the file; in that case, the last string in the list might not end in `\n`.
`seek`	`seek(pos, how=io.SEEK_SET, /)` Sets *f*'s current position to the integer byte offset *pos* away from a reference point. *how* indicates the reference point. The `io` module has attributes named `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`, to specify that the reference point is, respectively, the file's beginning, current position, or end. When *f* is opened in text mode, `f.seek` must have a *pos* of `0`, or, for `io.SEEK_SET` only, a *pos* that is the result of a previous call to `f.tell`. When *f* is opened in mode `'a'` or `'a+'`, on some but not all platforms, data written to *f* is appended to the data that is already in *f*, regardless of calls to `f.seek`.
`tell`	`tell()` Returns *f*'s current position: for a binary file this is an integer offset in bytes from the start of the file, and for

	a text file it's an opaque value usable in future calls to *f*.seek to position *f* back to the position that is now current.
truncate	truncate(*size*=None, /) Truncates *f*'s file, which must be open for writing. When *size* is present, truncates the file to be at most *size* bytes. When *size* is absent, uses *f*.tell() as the file's new size. *size* may be larger than the current file size; in this case, the resulting behavior is platform dependent.
write	write(*s*, /) Writes the bytes of string *s* (binary or text, depending on *f*'s mode) to the file.
writelines	writelines(*lst*, /) Like: `for line in lst: f.write(line)` It does not matter whether the strings in iterable *lst* are lines: despite its name, the method writelines just writes each of the strings to the file, one after the other. In particular, writelines does not add line-ending markers: such markers, if required, must already be present in the items of *lst*.

Iteration on File Objects

A file object *f*, open for reading, is also an iterator whose items are the file's lines. Thus, the loop:

```
for line in f:
```

iterates on each line of the file. Due to buffering issues, interrupting such a loop prematurely (e.g., with **break**), or calling `next(f)` instead of `f.readline()`, leaves the file's position set to an arbitrary value. If you want to switch from using *f* as an iterator to calling other reading methods on *f*, be sure to set the file's position to a known value by appropriately calling `f.seek`. On the plus side, a loop directly on *f* has very good performance, since these specifications allow the loop to use internal buffering to minimize I/O without taking up excessive amounts of memory even for huge files.

File-Like Objects and Polymorphism

An object *x* is file-like when it behaves *polymorphically* to a file object as returned by `io.open`, meaning that we can use *x* "as if" *x* were a file. Code using such an object (known as *client code* of the object) usually gets the object as an argument, or by calling a factory function that returns the object as the result. For example, if the only method that client code calls on *x* is `x.read`, without arguments, then all *x* needs to supply in order to be file-like enough for that code is a method `read` that is callable without arguments and returns a string. Other client code may need *x* to implement a larger subset of file methods. File-like objects and polymorphism are not absolute concepts: they are relative to demands placed on an object by some specific client code.

Polymorphism is a powerful aspect of object-oriented programming, and file-like objects are a good example of polymorphism. A client-code module that writes to or reads from files can automatically be reused for data residing elsewhere, as long as the module does not break polymorphism by type checking. When we discussed the built-ins `type` and `isinstance` in **Table 8-1**, we mentioned that type checking is often best avoided, as it blocks Python's normal polymorphism. Often, to support polymorphism in your client code, you just need to avoid type checking.

You can implement a file-like object by coding your own class (as covered in Chapter 4) and defining the specific methods needed by client code, such as `read`. A file-like object *fl* need not implement all the attributes and methods of a true file object *f*. If you can determine which methods the client code calls on *fl*, you can choose to implement only that subset. For example, when *fl* is only going to be written, *fl* doesn't need "reading" methods, such as `read`, `readline`, and `readlines`.

If the main reason you want a file-like object instead of a real file object is to keep the data in memory, rather than on disk, use the `io` module's classes `StringIO` or `BytesIO`, covered in "In-Memory Files: io.StringIO and io.BytesIO". These classes supply file objects that hold data in memory and largely behave polymorphically to other file objects. If you're running multiple processes that you want to communicate via file-like objects, consider `mmap`, covered in Chapter 15.

The tempfile Module

The `tempfile` module lets you create temporary files and directories in the most secure manner afforded by your platform. Temporary files are often a good idea when you're dealing with an amount of data that might not comfortably fit in memory, or when your program must write data that another process later uses.

The order of the parameters for the functions in this module is a bit confusing: to make your code more readable, always call these functions with named-argument syntax. The `tempfile` module exposes the functions and classes outlined in Table 11-3.

Table 11-3. Functions and classes of the `tempfile` module

`mkdtemp`	`mkdtemp(suffix=None, prefix=None, dir=Non` Securely creates a new temporary directory tha readable, writable, and searchable only by the c user, and returns the absolute path to the tempo directory. You can optionally pass arguments to strings to use as the start (`prefix`) and end (`suf`

the temporary file's filename, and the path to th directory in which the temporary file is created Ensuring that the temporary directory is remov when you're done with it is your program's responsibility.

`mkdtemp` *(cont.)*

Here is a typical usage example that creates a temporary directory, passes its path to another function, and finally ensures the directory (and contents) are removed:

```
import tempfile, shutil
path = tempfile.mkdtemp()
try:
    use_dirpath(path)
finally:
    shutil.rmtree(path)
```

`mkstemp`

`mkstemp(suffix=None, prefix=None, dir=Non text=False)`

Securely creates a new temporary file that is rea and writable only by the current user, is not executable, and is not inherited by subprocesse returns a pair (*fd*, *path*), where *fd* is the file descriptor of the temporary file (as returned by `os.open`, covered in Table 11-18) and the string the absolute path to the temporary file. The opti arguments `suffix`, `prefix`, and `dir` are like for t function `mkdtemp`. If you want the temporary file text file, explicitly pass the argument `text=True` Ensuring that the temporary file is removed wh you're done using it is up to you. `mkstemp` is not context manager, so you can't use a **with** statem best to use **try**/**finally** instead. Here is a typica example that creates a temporary text file, close

passes its path to another function, and finally e
the file is removed:

```
import tempfile, os
fd, path = tempfile.mkstemp(suffix='.tx
                            text=True)
try:
    os.close(fd)
    use_filepath(path)
finally:
    os.unlink(path)
```

NamedTemporaryFile	NamedTemporaryFile(mode='w+b', bufsize=-1, suffix=**None**, prefix=**None**, dir=**None**) Like TemporaryFile (covered later in this table), that the temporary file does have a name on the filesystem. Use the name attribute of the file obje access that name. Some platforms (mainly Winc do not allow the file to be opened again; therefo usefulness of the name is limited if you want to that your program works cross-platform. If you pass the temporary file's name to another progr opens the file, you can use the function mkstemp of NamedTemporaryFile to guarantee correct cro platform behavior. Of course, when you choose mkstemp, you do have to take care to ensure the removed when you're done with it. The file obje returned from NamedTemporaryFile is a context manager, so you can use a with statement.
SpooledTemporary File	SpooledTemporaryFile(mode='w+b', bufsize=- suffix=**None**, prefix=**None**, dir=**None**) Like TemporaryFile (see below), except that the object that SpooledTemporaryFile returns can s

	memory, if space permits, until you call its `filer` method (or its `rollover` method, which ensures gets written to disk, whatever its size). As a resu performance can be better with `SpooledTempora` as long as you have enough memory that's not otherwise in use.
`TemporaryDirectory`	`TemporaryDirectory(suffix=None, prefix=Nor dir=None, ignore_cleanup_errors=False)` Creates a temporary directory, like `mkdtemp` (pas optional arguments `suffix`, `prefix`, and `dir`). Th returned directory object is a context manager, can use a **`with`** statement to ensure it's removed as you're done with it. Alternatively, when you'r using it as a context manager, use its built-in cla method `cleanup` (*not* `shutil.rmtree`) to explicit remove and clean up the directory. Set `ignore_cleanup_errors` to **`True`** to ignore unhar exceptions during cleanup. The temporary direc and its contents are removed as soon as the dire object is closed (whether implicitly on garbage collection or explicitly by a `cleanup` call).
`TemporaryFile`	`TemporaryFile(mode='w+b', bufsize=-1, suffix=None, prefix=None, dir=None)` Creates a temporary file with `mkstemp` (passing t `mkstemp` the optional arguments `suffix`, `prefix`, `dir`), makes a file object from it with `os.fdopen`, covered in **Table 11-18** (passing to `fdopen` the op arguments `mode` and `bufsize`), and returns the f object. The temporary file is removed as soon as object is closed (implicitly or explicitly). For grea security, the temporary file has no name on the filesystem, if your platform allows that (Unix-lik platforms do; Windows doesn't). The file object

returned from `TemporaryFile` is a context mana
you can use a **`with`** statement to ensure it's remo
soon as you're done with it.

Auxiliary Modules for File I/O

File objects supply the functionality needed for file I/O. Other Python library modules, however, offer convenient supplementary functionality, making I/O even easier and handier in several important cases. We'll look at two of those modules here.

The fileinput Module

The `fileinput` module lets you loop over all the lines in a list of text files. Performance is good—comparable to the performance of direct iteration on each file—since buffering is used to minimize I/O. You can therefore use this module for line-oriented file input whenever you find its rich functionality convenient, with no worry about performance. The key function of the module is `input`; `fileinput` also supplies a `FileInput` class whose methods support the same functionality. Both are described in **Table 11-4**.

Table 11-4. Key classes and functions of the `fileinput` module

`FileInput`	`class FileInput(files=None, inplace=False, backup='', mode='r', openhook=None, encoding=None, errors=None)` Creates and returns an instance *f* of class `FileInput`. The arguments are the same as for `fileinput.input` covered next, and methods of *f* have the same names, arguments, and semantics as the other functions of the `fileinput` module (see **Table 11-5**). *f* also supplies a `readline` method, which reads and returns the next line. Use the `FileInput` class to nest or mix loops that read lines from multiple sequences of files.

`input`	`input(files=None, inplace=False, backup='', mode='r', openhook=None, encoding=None, errors=None)` Returns an instance of `FileInput`, an iterable yielding lines in `files`; that instance is the global state, so all other functions of the `fileinput` module (see **Table 11-5**) operate on the same shared state. Each function of the `fileinput` module corresponds directly to a method of the class `FileInput`. `files` is a sequence of filenames to open and read one after the other, in order. When `files` is a string, it's a single filename to open and read. When `files` is **`None`**, `input` uses `sys.argv[1:]` as the list of filenames. The filename `'-'` means standard input (`sys.stdin`). When the sequence of filenames is empty, `input` reads `sys.stdin` instead. When `inplace` is **`False`** (the default), `input` just reads the files. When `inplace` is **`True`**, `input` moves each file being read (except standard input) to a backup file and redirects standard output (`sys.stdout`) to write to a new file with the same path as the original one of the file being read. This way, you can simulate overwriting files in place. If `backup` is a string that starts with a dot, `input` uses `backup` as the extension of the backup files and does not remove the backup files. If `backup` is an empty string (the default), `input` uses *.bak* and deletes each backup file as the input files are closed. The keyword argument `mode` may be `'r'`, the default, or `'rb'`. You may optionally pass an `openhook` function to use as an alternative to `io.open`. For example, `openhook=fileinput.hook_compressed` decompresses any input file with extension *.gz* or *.bz2* (not compatible with `inplace=True`). You can write your own `openhook` function to decompress other file types, for example

		using LZMA decompression[a] for *.xz* files; use the **Python source for `fileinput.hook_compressed`** as a template. 3.10+ You can also pass `encoding` and `errors`, which will be passed to the hook as keyword arguments .

[a] LZMA support may require building Python with optional additional libraries.

The functions of the `fileinput` module listed in **Table 11-5** work on the global state created by `fileinput.input`, if any; otherwise, they raise `RuntimeError`.

Table 11-5. Additional functions of the `fileinput` module

`close`	`close()` Closes the whole sequence so that iteration stops and no file remains open.
`filelineno`	`filelineno()` Returns the number of lines read so far from the file now being read. For example, returns `1` if the first line has just been read from the current file.
`filename`	`filename()` Returns the name of the file now being read, or **`None`** if no line has been read yet.
`isfirstline`	`isfirstline()` Returns **`True`** or **`False`**, just like `filelineno() == 1`.
`isstdin`	`isstdin()` Returns **`True`** when the current file being read is `sys.stdin`; otherwise, returns **`False`**.

`lineno`	`lineno()` Returns the total number of lines read since the call to `input`.
`nextfile`	`nextfile()` Closes the file being read: the next line to read is the first one of the next file.

Here's a typical example of using `fileinput` for a "multifile search and replace," changing one string into another throughout the text files whose names were passed as command-line arguments to the script:

```
import fileinput
for line in fileinput.input(inplace=True):
    print(line.replace('foo', 'bar'), end='')
```

In such cases it's important to include the `end=''` argument to `print`, since each `line` has its line-end character `\n` at the end, and you need to ensure that `print` doesn't add another (or else each file would end up "double-spaced").

You may also use the `FileInput` instance returned by `fileinput.input` as a context manager. Just as with `io.open`, this will close all files opened by the `FileInput` upon exiting the **`with`** statement, even if an exception occurs:

```
with fileinput.input('file1.txt', 'file2.txt') as infile:
    dostuff(infile)
```

The struct Module

The `struct` module lets you pack binary data into a bytestring, and unpack the bytes of such a bytestring back into the Python data they represent. This is useful for many kinds of low-level programming. Often, you use `struct` to interpret data records from binary files that have some specified format, or to prepare records to write to such binary files. The module's name comes from C's keyword `struct`, which is usable for related purposes. On any error, functions of the module `struct` raise exceptions that are instances of the exception class `struct.error`.

The `struct` module relies on *struct format strings* following a specific syntax. The first character of a format string gives the **byte order**, size, and alignment of the packed data; the options are listed in **Table 11-6**.

Table 11-6. Possible first characters in a `struct` format string

Character	Meaning
`@`	Native byte order, native data sizes, and native alignment for the current platform; this is the default if the first character is none of the characters listed here (note that the format `P` in **Table 11-7** is available only for this kind of `struct` format string). Look at the string `sys.byteorder` when you need to check your system's byte order; most CPUs today use `'little'`, but `'big'` is the "network standard" for TCP/IP, the core protocols of the internet.
`=`	Native byte order for the current platform, but standard size and alignment.
`<`	Little-endian byte order; standard size and alignment.
`>`, `!`	Big-endian/network standard byte order; standard size and alignment.

Standard sizes are indicated in **Table 11-7**. Standard alignment means no forced alignment, with explicit padding bytes used as needed. Native sizes and alignment are whatever the platform's C compiler uses. Native byte order can put the most significant byte at either the lowest (big-endian) or highest (little-endian) address, depending on the platform.

After the optional first character, a format string is made up of one or more format characters, each optionally preceded by a count (an integer represented by decimal digits). Common format characters are listed in Table 11-7; see the online docs for a complete list. For most format characters, the count means repetition (e.g., `'3h'` is exactly the same as `'hhh'`). When the format character is `s` or `p`—that is, a bytestring—the count is not a repetition: it's the total number of bytes in the string. You can freely use whitespace between formats, but not between a count and its format character. The format `s` means a fixed-length bytestring as long as its count (the Python string is truncated, or padded with copies of the null byte `b'\0'`, if needed). The format `p` means a "Pascal-like" bytestring: the first byte is the number of significant bytes that follow, and the actual contents start from the second byte. The count is the total number of bytes, including the length byte.

Table 11-7. Common format characters for `struct`

Character	C type	Python type	Standard size
`B`	`unsigned char`	`int`	1 byte
`b`	`signed char`	`int`	1 byte
`c`	`char`	`bytes` (length 1)	1 byte
`d`	`double`	`float`	8 bytes
`f`	`float`	`float`	4 bytes
`H`	`unsigned short`	`int`	2 bytes
`h`	`signed short`	`int`	2 bytes
`I`	`unsigned int`	`long`	4 bytes

Character	C type	Python type	Standard size
`i`	`signed int`	`int`	4 bytes
`L`	`unsigned long`	`long`	4 bytes
`l`	`signed long`	`int`	4 bytes
`P`	`void*`	`int`	N/A
`p`	`char[]`	`bytes`	N/A
`s`	`char[]`	`bytes`	N/A
`x`	`padding byte`	No value	1 byte

The `struct` module supplies the functions covered in **Table 11-8**.

Table 11-8. Functions of the `struct` module

`calcsize`	`calcsize(`*`fmt`*`, /)` Returns the size, in bytes, corresponding to format string *`fmt`*.
`iter_unpack`	`iter_unpack(`*`fmt`*`, `*`buffer`*`, /)` Unpacks iteratively from *`buffer`* per format string *`fmt`*. Returns an iterator that will read equally sized chunks from *`buffer`* until all its contents are consumed; each iteration yields a tuple as specified by *`fmt`*. *`buffer`*'s size must be a multiple of the size required by the format, as reflected in `struct.calcsize(`*`fmt`*`)`.
`pack`	`pack(`*`fmt`*`, *`*`values`*`, /)` Packs the values per format string *`fmt`*, and returns

		the resulting bytestring. *values* must match in number and type the values required by *fmt*.
`pack_into`	`pack_into(`*fmt*`, `*buffer*`, `*offset*`, *`*values*`, /)`	Packs the values per format string *fmt* into writable buffer *buffer* (usually an instance of `bytearray`) starting at index *offset*. *values* must match in number and type the values required by *fmt*. `len(buffer[offset:])` must be `>=struct.calcsize(`*fmt*`)`.
`unpack`	`unpack(`*fmt*`, `*s*`, /)`	Unpacks bytestring *s* per format string *fmt*, and returns a tuple of values (if just one value, a one-item tuple). `len(`*s*`)` must equal `struct.calcsize(`*fmt*`)`.
`unpack_from`	`unpack_from(`*fmt*`, /, buffer, offset=0)`	Unpacks bytestring (or other readable buffer) `buffer`, starting from offset `offset`, per format string *fmt*, returning a tuple of values (if just one value, a one-item tuple). `len(buffer[offset:])` must be `>=struct.calcsize(`*fmt*`)`.

The `struct` module also offers a `Struct` class, which is instantiated with a format string as an argument. Instances of this class implement `pack`, `pack_into`, `unpack`, `unpack_from`, and `iter_unpack` methods corresponding to the functions described in the preceding table; they take the same arguments as the corresponding module functions, but omitting the *fmt* argument, which was provided on instantiation. This allows the class to compile the format string once and reuse it. Struct objects also have a `format` attribute that holds the format string for the object, and a `size` attribute that holds the calculated size of the structure.

In-Memory Files: io.StringIO and io.BytesIO

You can implement file-like objects by writing Python classes that supply the methods you need. If all you want is for data to reside in memory, rather than in a file as seen by the operating system, use the classes `StringIO` or `BytesIO` of the `io` module. The difference between them is that instances of `StringIO` are text-mode files, so reads and writes consume or produce text strings, while instances of `BytesIO` are binary files, so reads and writes consume or produce bytestrings. These classes are especially useful in tests and other applications where program output should be redirected for buffering or journaling; **"The print Function"** includes a useful context manager example, `redirect`, that demonstrates this.

When you instantiate either class you can optionally pass a string argument, respectively `str` or `bytes`, to use as the initial content of the file. Additionally, you can pass the argument `newline='\n'` to `StringIO` (but not `BytesIO`) to control how line endings are handled (like in **`TextIoWrapper`**); if `newline` is `None`, newlines are written as `\n` on all platforms. In addition to the methods described in **Table 11-2**, an instance *f* of either class supplies one extra method:

`getvalue`	`getvalue()` Returns the current data contents of *f* as a string (text or bytes). You cannot call *f*`.getvalue` after you call *f*`.close`: `close` frees the buffer that *f* internally keeps, and `getvalue` needs to return the buffer as its result.

Archived and Compressed Files

Storage space and transmission bandwidth are increasingly cheap and abundant, but in many cases you can save such resources, at the expense of some extra computational effort, by using compression. Computational power grows cheaper and more abundant even faster than some other resources, such as bandwidth, so compression's popularity keeps growing. Python makes it easy for your programs to support compression. We don't cover the details of compression in this book, but you can find details on the relevant standard library modules in the **online docs**.

The rest of this section covers "archive" files (which collect in a single file a collection of files and optionally directories), which may or may not be compressed. Python's stdlib offers two modules to handle two very popular archive formats: `tarfile` (which, by default, does not compress the files it bundles), and `zipfile` (which, by default, does compress the files it bundles).

The tarfile Module

The `tarfile` module lets you read and write**TAR files** (archive files compatible with those handled by popular archiving programs such as `tar`), optionally with gzip, bzip2, or LZMA compression. TAR files are typically named with a *.tar* or *.tar.(compression type)* extension. **3.8+** The default format of new archives is POSIX.1-2001 (pax). **`python -m tarfile`** offers a useful command-line interface to the module's functionality: run it without arguments to get a brief help message.

The `tarfile` module supplies the functions listed in **Table 11-9**. When handling invalid TAR files, functions of `tarfile` raise instances of `tarfile.TarError`.

Table 11-9. Classes and functions of the `tarfile` module

`is_tarfile`	`is_tarfile(`*`filename`*`)` Returns **`True`** when the file named by *`filename`* (which may be a `str`, **3.9+** or a file or file-like object) appears to be a valid TAR file (possibly with compression), judging by the first few bytes; otherwise, returns **`False`**.

open	`open(name=None, mode='r', fileobj=None, bufsize=10240, **kwargs)` Creates and returns a `TarFile` instance *f* to read or create a TAR file through file-like object `fileobj`. When `fileobj` is **None**, `name` may be a string naming a file or a path-like object; `open` opens the file with the given `mode` (by default, `'r'`), and *f* wraps the resulting file object. `open` may be used as a context manager (e.g., **`with`** `tarfile.open(...)` **`as`** *f*). **F.CLOSE MAY NOT CLOSE FILEOBJ** Calling *f*`.close` does *not* close `fileobj` when *f* was opened with a `fileobj` that is not **None**. This behavior of *f*`.close` is important when `fileobj` is an instance of `io.BytesIO`: you can call `fileobj.getvalue` after *f*`.close` to get the archived and possibly compressed data as a string. This behavior also means that you have to call `fileobj.close` explicitly after calling *f*`.close`. `mode` can be `'r'` to read an existing TAR file with whatever compression it has (if any); `'w'` to write a new TAR file, or truncate and rewrite an existing one, without compression; or `'a'` to append to an existing TAR file, without compression. Appending to compressed TAR files is not supported. To write a new TAR file with compression, `mode` can be `'w:gz'` for gzip compression, `'w:bz2'` for bzip2 compression, or `'w:xz'` for LZMA compression. You can use mode strings `'r:'` or `'w:'` to read or write uncompressed, nonseekable TAR files using a buffer of `bufsize` bytes; for reading TAR files use plain `'r'`, since this will automatically uncompress as necessary. In the mode strings specifying compression, you can use a vertical bar (`|`) instead of a colon (`:`) to force sequential processing and fixed-size blocks; this is

useful in the (admittedly very unlikely) case that you ever find yourself handling a tape device!

The TarFile class

`TarFile` is the underlying class for most `tarfile` methods, but is not used directly. A `TarFile` instance *f*, created using `tarfile.open`, supplies the methods detailed in **Table 11-10**.

Table 11-10. Methods of a `TarFile` instance *f*

`add`	*f*`.add(`*name*`, arcname=`**`None`**`, recursive=`**`True`**`, *, filter=`**`None`**`)` Adds to archive *f* the file named by *name* (can be any type of file, a directory, or a symbolic link). When `arcname` is not **`None`**, it's used as the archive member name in lieu of *name*. When *name* is a directory, and `recursive` is **`True`**, `add` recursively adds the whole filesystem subtree rooted in that directory in sorted order. The optional (named-only) argument `filter` is a function that is called on each object to be added. It takes a `TarInfo` object argument and returns either the (possibly modified) `TarInfo` object, or **`None`**. In the latter case the `add` method excludes this `TarInfo` object from the archive.
`addfile`	*f*`.addfile(`*tarinfo*`, fileobj=`**`None`**`)` Adds to archive *f* a `TarInfo` object *tarinfo*. If `fileobj` is not **`None`**, the first *tarinfo*`.size` bytes of binary file-like object `fileobj` are added.
`close`	*f*`.close()` Closes archive *f*. You must call `close`, or else an incomplete, unusable TAR file might be left on disk. Such mandatory finalization is best performed with a **`try`**`/`**`finally`**, as covered in **"try/finally"**, or, even better, a **`with`** statement, covered in **"The with**

	Statement and Context Managers". Calling *f*.close does *not* close fileobj if *f* was created with a non-**None** fileobj. This matters especially when fileobj is an instance of io.BytesIO: you can call fileobj.getvalue after *f*.close to get the compressed data string. So, you always have to call fileobj.close (explicitly, or implicitly by using a **with** statement) *after* *f*.close.
extract	*f*.extract(*member*, path='', set_attrs=**True**, numeric_owner=**False**) Extracts the archive member identified by *member* (a name or a TarInfo instance) into a corresponding file in the directory (or path-like object) named by path (the current directory by default). If set_attrs is **True**, the owner and timestamps will be set as they were saved in the TAR file; otherwise, the owner and timestamps for the extracted file will be set using the current user and time values. If numeric_owner is **True**, the UID and GID numbers from the TAR file are used to set the owner/group for the extracted files; otherwise, the named values from the TAR file are used. (The **online docs** recommend using extractall over calling extract directly, since extractall does additional error handling internally.)
extractall	*f*.extractall(path='.', members=**None**, numeric_owner=**False**) Similar to calling extract on each member of TAR file *f*, or just those listed in the members argument, with additional error checking for chown, chmod, and utime errors that occur while writing the extracted members.

DON'T USE EXTRACTALL ON A TARFILE FROM AN UNTRUSTED SOURCE

`extractall` does not check the paths of extracted files, so there is a risk that an extracted file will have an absolute path (or include one or more `..` components) and thus overwrite a potentially sensitive file.[a] It is best to read each member individually and only extract it if it has a safe path (i.e., no absolute paths or relative paths with any `..` path component).

`extractfile`	`f.extractfile(member)` Extracts the archive member identified by *member* (a name or a `TarInfo` instance) and returns an `io.BufferedReader` object with the methods `read`, `readline`, `readlines`, `seek`, and `tell`.
`getmember`	`f.getmember(name)` Returns a `TarInfo` instance with information about the archive member named by the string *name*.
`getmembers`	`f.getmembers()` Returns a list of `TarInfo` instances, one for each member in archive *f*, in the same order as the entries in the archive itself.
`getnames`	`f.getnames()` Returns a list of strings, the names of each member in archive *f*, in the same order as the entries in the archive itself.
`gettarinfo`	`f.gettarinfo(name=None, arcname=None, fileobj=None)` Returns a `TarInfo` instance with information about the open file object `fileobj`, when not **None**, or else the existing file whose path is the string `name`. `name` may be a path-like object. When `arcname` is not **None**,

		it's used as the `name` attribute of the resulting `TarInfo` instance.
`list`		*f*`.list(verbose=`**`True`**`, *, members=`**`None`**`)` Outputs a directory of the archive *f* to `sys.stdout`. If the optional argument `verbose` is **`False`**, outputs only the names of the archive's members. If the optional argument `members` is given, it must be a subset of the list returned by `getmembers`.
`next`		*f*`.next()` Returns the next available archive member as a `TarInfo` instance; if none are available, returns **`None`**.

[a] Described further in CVE-2007-4559.

The TarInfo class

The methods `getmember` and `getmembers` of `TarFile` instances return instances of `TarInfo`, supplying information about members of the archive. You can also build a `TarInfo` instance with a `TarFile` instance's method `gettarinfo`. The *name* argument may be a path-like object. The most useful attributes and methods supplied by a `TarInfo` instance *t* are listed in Table 11-11.

Table 11-11. Useful attributes of a `TarInfo` instance *t*

`isdir()`	Returns **`True`** if the file is a directory
`isfile()`	Returns **`True`** if the file is a regular file
`issym()`	Returns **`True`** if the file is a symbolic link
`linkname`	Target file's name (a string), when *t*`.type` is `LNKTYPE` or `SYMTYPE`

`mode`	Permission and other mode bits of the file identified by *t*
`mtime`	Time of last modification of the file identified by *t*
`name`	Name in the archive of the file identified by *t*
`size`	Size, in bytes (uncompressed), of the file identified by *t*
`type`	File type—one of many constants that are attributes of the `tarfile` module (`SYMTYPE` for symbolic links, `REGTYPE` for regular files, `DIRTYPE` for directories, and so on; see the online docs for a complete list)

The zipfile Module

The `zipfile` module can read and write ZIP files (i.e., archive files compatible with those handled by popular compression programs such as `zip` and `unzip`, `pkzip` and `pkunzip`, `WinZip`, and so on, typically named with a *.zip* extension). **`python -m zipfile`** offers a useful command-line interface to the module's functionality: run it without further arguments to get a brief help message.

Detailed information about ZIP files is available on the PKWARE and Info-ZIP websites. You need to study that detailed information to perform advanced ZIP file handling with `zipfile`. If you do not specifically need to interoperate with other programs using the ZIP file standard, the modules `lzma`, `gzip`, and `bz2` are usually better ways to deal with compression, as is `tarfile` to create (optionally compressed) archives.

The `zipfile` module can't handle multidisk ZIP files, and cannot create encrypted archives (it can decrypt them, albeit rather slowly). The module also cannot handle archive members using compression types besides the usual ones, known as *stored* (a file copied to the archive without compression) and *deflated* (a file compressed using the ZIP format's default algorithm). `zipfile` also handles the bzip2 and LZMA compression types, but beware: not all tools can handle those, so if you use them you're sacrificing some portability to get better compression.

The `zipfile` module supplies function `is_zipfile` and class `Path`, as listed in Table 11-12. In addition, it supplies classes `ZipFile` and `ZipInfo`, described later. For errors related to invalid ZIP files, functions of `zipfile` raise exceptions that are instances of the exception class `zipfile.error`.

Table 11-12. Auxiliary function and class of the `zipfile` module

`is_zipfile`	`is_zipfile(`*`file`*`)` Returns **`True`** when the file named by string, path-like object, or file-like object *`file`* seems to be a valid ZIP file, judging by the first few and last bytes of the file; otherwise, returns **`False`**.
`Path`	**`class`** `Path(`*`root`*`, at='')` 3.8+ A `pathlib`-compatible wrapper for ZIP files. Returns a `pathlib.Path` object *p* from *`root`*, a ZIP file (which may be a `ZipFile` instance or file suitable for passing to the `ZipFile` constructor). The string argument `at` is a path to specify the location of *p* in the ZIP file: the default is the root. *p* exposes several `pathlib.Path` methods: see the online docs for details.

The ZipFile class

The main class supplied by `zipfile` is `ZipFile`. Its constructor has the following signature:

`ZipFile`	**`class`** `ZipFile(`*`file`*`, mode='r', compression=zipfile.ZIP_STORED, allowZip64=`**`True`**`, compresslevel=`**`None`**`, *, strict_timestamps=`**`True`**`)` Opens a ZIP file named by *`file`* (a string, file-like object, or path-like object). `mode` can be `'r'` to read an existing

ZIP file, `'w'`, to write a new ZIP file or truncate and rewrite an existing one, or `'a'` to append to an existing file. It can also be `'x'`, which is like `'w'` but raises an exception if the ZIP file already existed—here, `'x'` stands for "exclusive."

When `mode` is `'a'`, *`file`* can name either an existing ZIP file (in which case new members are added to the existing archive) or an existing non-ZIP file. In the latter case, a new ZIP file-like archive is created and appended to the existing file. The main purpose of this latter case is to let you build an executable file that unpacks itself when run. The existing file must then be a pristine copy of a self-unpacking executable prefix, as supplied by *www.info-zip.org* and by other purveyors of ZIP file compression tools.

`compression` is the ZIP compression method to use in writing the archive: `ZIP_STORED` (the default) requests that the archive use no compression, and `ZIP_DEFLATED` requests that the archive use the *deflation* mode of compression (the most usual and effective compression approach used in ZIP files). It can also be `ZIP_BZIP2` or `ZIP_LZMA` (sacrificing portability for more compression; these require the `bz2` or `lzma` module, respectively). Unrecognized values will raise `NotImplementedError`.

`ZipFile` *(cont.)*	When `allowZip64` is **`True`** (the default), the `ZipFile` instance is allowed to use the ZIP64 extensions to produce an archive larger than 4 GB; otherwise, any attempt to produce such a large archive raises a `LargeZipFile` exception. `compresslevel` is an integer (ignored when using `ZIP_STORED` or `ZIP_LZMA`) from `0` for `ZIP_DEFLATED` (`1` for `ZIP_BZIP2`), which requests modest compression but fast operation, to `9` to request the best compression at the cost of more computation. **3.8+** Set `strict_timestamps` to **`False`** to store files older

	than 1980-01-01 (sets the timestamp to 1980-01-01) or beyond 2107-12-31 (sets the timestamp to 2107-12-31).

`ZipFile` is a context manager; thus, you can use it in a **`with`** statement to ensure the underlying file gets closed when you're done with it. For example:

```
with zipfile.ZipFile('archive.zip') as z:
    data = z.read('data.txt')
```

In addition to the arguments with which it was instantiated, a `ZipFile` instance *z* has the attributes `fp` and `filename`, which are the file-like object *z* works on and its filename (if known); `comment`, the possibly empty string that is the archive's comment; and `filelist`, the list of `ZipInfo` instances in the archive. In addition, *z* has a writable attribute called `debug`, an `int` from `0` to `3` that you can assign to control how much debugging output to emit to `sys.stdout`:[2] from nothing when *z*`.debug` is `0`, to the maximum amount of information available when *z*`.debug` is `3`.

A `ZipFile` instance *z* supplies the methods listed in **Table 11-13**.

Table 11-13. Methods supplied by an instance *z* of `ZipFile`

`close`	`close()` Closes archive file *z*. Make sure to call *z*`.close()`, or an incomplete and unusable ZIP file might be left on disk. S mandatory finalization is generally best performed witl **`try`**/**`finally`** statement, as covered in **"try/finally"**, or—even better—a **`with`** statement, covered in **"The with Statement and Context Managers"**.
`extract`	`extract(`*`member`*`, path=`**`None`**`, pwd=`**`None`**`)` Extracts an archive member to disk, to the directory or p like object `path` or, by default, to the current working directory; *`member`* is the member's full name, or an insta of `ZipInfo` identifying the member. `extract` normalizes

	path info within *member*, turning absolute paths into rela ones, removing any `..` component, and, on Windows, turning characters that are illegal in filenames into underscores (_). `pwd`, if present, is the password to use to decrypt an encrypted member. `extract` returns the path to the file it has created (or overwritten if it already existed), or to the directory it ha created (or left alone if it already existed). Calling `extra` on a closed `ZipFile` raises `ValueError`.
`extractall`	`extractall(path=None, members=None, pwd=None)` Extracts archive members to disk (by default, all of them directory or path- like object `path` or, by default, to the current working directory; `members` optionally limits wh members to extract, and must be a subset of the list of strings returned by `z.namelist`. `extractall` normalizes path info within members it extracts, turning absolute p into relative ones, removing any `..` component, and, on Windows, turning characters that are illegal in filename into underscores (_). `pwd`, if present, is the password to u to decrypt encrypted members, if any.
`getinfo`	`getinfo(name)` Returns a `ZipInfo` instance that supplies information ab the archive member named by the string *name*.
`infolist`	`infolist()` Returns a list of `ZipInfo` instances, one for each membe archive *z*, in the same order as the entries in the archive
`namelist`	`namelist()` Returns a list of strings, the name of each member in archive *z*, in the same order as the entries in the archive
`open`	`open(name, mode='r', pwd=None, *, force_zip64=Fa` Extracts and returns the archive member identified by *n*

	(a member name string or `ZipInfo` instance) as a (mayb read-only) file-like object. *mode* may be `'r'` or `'w'`. `pwd`, i present, is the password to use to decrypt an encrypted member. Pass `force_zip64=`**`True`** when an unknown file may exceed 2 GiB, to ensure the header format is capabl supporting large files. When you know in advance the la file size, use a `ZipInfo` instance for *name*, with `file_siz` appropriately.
`printdir`	`printdir()` Outputs a textual directory of the archive *z* to `sys.stdou`
`read`	`read(`*`name`*`, `*`pwd`*`)` Extracts the archive member identified by *name* (a meml name string or `ZipInfo` instance) and returns the bytest of its contents (raises `ValueError` if called on a closed `ZipFile`). *pwd*, if present, is the password to use to decry an encrypted member.
`setpassword`	`setpassword(`*`pwd`*`)` Sets string *pwd* as the default password to use to decrypt encrypted files.
`testzip`	`testzip()` Reads and checks the files in archive *z*. Returns a string the name of the first archive member that is damaged, c **None** if the archive is intact.

`write`	`write(`*`filename`*`, arcname=`**`None`**`, compress_type=`**`None`**`,`
	`compresslevel=`**`None`**`)`
	Writes the file named by string *`filename`* to archive *z*, w archive member name `arcname`. When `arcname` is **None**, `write` uses *`filename`* as the archive member name. Whe `compress_type` or `compresslevel` is **None** (the default), `w` uses *z*'s compression type and level; otherwise, `compress_type` and/or `compresslevel` specify how to

	compress the file. *z* must be opened for modes `'w'`, `'x'`, `'a'`; otherwise `ValueError` is raised.
`writestr`	`writestr(`*`zinfo_arc`*`,` *`data`*`, compress_type=`**`None`**`, compresslevel=`**`None`**`)` Adds a member to archive *z* using the metadata specifie *`zinfo_arc`* and the data in *`data`*. *`zinfo_arc`* must be eith `ZipInfo` instance specifying at least *`filename`* and *`date_time`*, or a string to be used as the archive member name with the date and time are set to the current mom *`data`* is an instance of `bytes` or `str`. When `compress_typ` `compresslevel` is **`None`** (the default), `writestr` uses *z*'s compression type and level; otherwise, `compress_type` and/or `compresslevel` specify how to compress the file. must be opened for modes `'w'`, `'x'`, or `'a'`; otherwise `ValueError` is raised. When you have data in memory and need to write the d to the ZIP file archive *z*, it's simpler and faster to use *z*`.writestr` than *z*`.write`. The latter would require you write the data to disk first and later remove the useless file; with the former you can just code:

```
import zipfile
with zipfile.ZipFile('z.zip', 'w') as zz:
    data = 'four score\nand seven\nyears ago\r
    zz.writestr('saying.txt', data)
```

Here's how you can print a list of all files contained in tl ZIP file archive created by the previous example, follow by each file's name and contents:

```
with zipfile.ZipFile('z.zip') as zz:
    zz.printdir()
```

```
    for name in zz.namelist():
        print(f'{name}: {zz.read(name)!r}')
```

The ZipInfo class

The methods `getinfo` and `infolist` of `ZipFile` instances return instances of class `ZipInfo` to supply information about members of the archive. **Table 11-14** lists the most useful attributes supplied by a `ZipInfo` instance *`z`*.

Table 11-14. Useful attributes of a `ZipInfo` instance z

`comment`	A string that is a comment on the archive member
`compress_size`	The size, in bytes, of the compressed data for the archive member
`compress_type`	An integer code recording the type of compression of the archive member
`date_time`	A tuple of six integers representing the time of the last modification to the file: the items are year (`>=1980`), month, day (`1+`), hour, minute, second (`0+`)
`file_size`	The size, in bytes, of the uncompressed data for the archive member
`filename`	The name of the file in the archive

The os Module

`os` is an umbrella module presenting a nearly uniform cross-platform view of the capabilities of various operating systems. It supplies low-level ways to create and handle files and directories, and to create, manage, and destroy processes. This section covers filesystem-related functions of `os`; **"Running Other Programs with the os Module"** covers process-related functions. Most of the time you can use other modules at higher levels of abstraction and gain productivity, but understanding what is "underneath" in the low-level `os` module can still be quite useful (hence our coverage).

The `os` module supplies a `name` attribute, a string that identifies the kind of platform on which Python is being run. Common values for `name` are `'posix'` (all kinds of Unix-like platforms, including Linux and macOS) and `'nt'` (all kinds of Windows platforms); `'java'` is for the old but still-missed Jython. You can exploit some unique capabilities of a platform through functions supplied by `os`. However, this book focuses on cross-platform programming, not platform-specific functionality, so we cover neither parts of `os` that exist only on one platform, nor platform-specific modules: functionality covered in this book is available at least on `'posix'` and `'nt'` platforms. We do, though, cover some of the differences among the ways in which a given functionality is provided on various platforms.

Filesystem Operations

Using the `os` module, you can manipulate the filesystem in a variety of ways: creating, copying, and deleting files and directories; comparing files; and examining filesystem information about files and directories. This section documents the attributes and methods of the `os` module that you use for these purposes, and covers some related modules that operate on the filesystem.

Path-string attributes of the os module

A file or directory is identified by a string, known as its *path*, whose syntax depends on the platform. On both Unix-like and Windows platforms, Python accepts Unix syntax for paths, with a slash (`/`) as the directory separator. On non-Unix-like platforms, Python also accepts platform-specific path syntax. On Windows, in particular, you may use a backslash (`\`) as the separator. However, you then need to double up each backslash as `\\` in string literals, or use raw string literal syntax (as covered in **"Strings"**); you also needlessly lose portability. Unix path syntax is handier and usable everywhere, so we strongly recommend that you *always* use it. In the rest of this chapter, we use Unix path syntax in both explanations and examples.

The `os` module supplies attributes that provide details about path strings on the current platform, detailed in **Table 11-15**. You should typically use the higher-level path manipulation operations covered in **"The os.path Module"**[3] rather than lower-level string operations based on these attributes. However, these attributes may be useful at times.

Table 11-15. Attributes supplied by the `os` module

`curdir`	The string that denotes the current directory (`'.'` on Unix and Windows)
`defpath`	The default search path for programs, used if the environment lacks a `PATH` environment variable
`extsep`	The string that separates the extension part of a file's name from the rest of the name (`'.'` on Unix and Windows)
`linesep`	The string that terminates text lines (`'\n'` on Unix; `'\r\n'` on Windows)
`pardir`	The string that denotes the parent directory (`'..'` on Unix and Windows)
`pathsep`	The separator between paths in lists of paths expressed as strings, such as those used for the environment variable `PATH` (`':'` on Unix; `';'` on Windows)
`sep`	The separator of path components (`'/'` on Unix; `'\\'` on Windows)

Permissions

Unix-like platforms associate nine bits with each file or directory: three each for the file's owner, its group, and everybody else (aka "others" or "the world"), indicating whether the file or directory can be read, written, and executed by the given subject. These nine bits are known as the file's *permission bits*, and are part of the file's *mode* (a bit string that includes other bits that describe the file). You often display these bits in octal notation, which groups three bits per digit. For example, mode `0o664` indicates a file that can be read and written by its owner and group, and that anybody else can read, but not write. When any process on a Unix-like system creates a file or directory, the operating system applies to the specified mode a bit mask known as the process's *umask*, which can remove some of the permission bits.

Non-Unix-like platforms handle file and directory permissions in very different ways. However, the `os` functions that deal with file permissions accept a *`mode`* argument according to the Unix-like approach described in the previous paragraph. Each platform maps the nine permission bits in a way appropriate for it. For example, on Windows, which distinguishes only between read-only and read/write files and does not record file ownership, a file's permission bits show up as either `0o666` (read/write) or `0o444` (read-only). On such a platform, when creating a file, the implementation looks only at bit `0o200`, making the file read/write when that bit is `1` and read-only when it is `0`.

File and directory functions of the os module

The `os` module supplies several functions (listed in **Table 11-16**) to query and set file and directory status. In all versions and platforms, the argument *`path`* to any of these functions can be a string giving the path of the file or directory involved, or it can be a path-like object (in particular, an instance of `pathlib.Path`, covered later in this chapter). There are also some particularities on some Unix platforms:

- Some of the functions also support a *file descriptor* (*fd*)—an `int` denoting a file as returned, for example, by `os.open`—as the *path* argument. The module attribute `os.supports_fd` is the set of functions in the `os` module that support this behavior (the module attribute is missing on platforms lacking such support).

- Some functions support the optional keyword-only argument `follow_symlinks`, defaulting to **`True`**. When this argument is **`True`**, if *`path`* indicates a symbolic link, the function follows it to reach an actual file or directory; when it's **`False`**, the function operates on the symbolic link itself. The module attribute `os.supports_follow_symlinks`, if present, is the set of functions in the `os` module that support this argument.
- Some functions support the optional named-only argument `dir_fd`, defaulting to `None`. When `dir_fd` is present, *`path`* (if relative) is taken as being relative to the directory open at that file descriptor; when missing, *`path`* (if relative) is taken as relative to the current working directory. If *`path`* is absolute, `dir_fd` is ignored. The module attribute `os.supports_dir_fd`, if present, is the set of functions of the `os` module that support this argument.

Additionally, on some platforms the named-only argument `effective_ids`, defaulting to **`False`**, lets you choose to use effective rather than real user and group identifiers. Check whether it is available on your platform with `os.supports_effective_ids`.

Table 11-16. `os` module functions

`access`	`access(`*`path, mode`*`, *, dir_fd=`**`None`**`, effective_ids=`**`F`** `follow_symlinks=`**`True`**`)` Returns **`True`** when the file or path-like object *`path`* has al permissions encoded in integer *`mode`*; otherwise, returns *`mode`* can be `os.F_OK` to test for file existence, or one or m `os.R_OK`, `os.W_OK`, and `os.X_OK` (joined with the bitwise O operator `|`, if more than one) to test permissions to read, and execute the file. If `dir_fd` is not **`None`**, `access` operate relative to the provided directory (if *`path`* is absolute, `dir` ignored). Pass the keyword-only argument `effective_id` (the default is **`False`**) to use effective rather than real use group identifiers (this may not work on all platforms). If

	`follow_symlinks=False` and the last element of *path* is a link, `access` operates on the symbolic link itself, not on th pointed to by the link. `access` does not use the standard interpretation for its *mo* argument, covered in the previous section. Rather, `acces` only if this specific process's real user and group identifie the requested permissions on the file. If you need to stud permission bits in more detail, see the function `stat`, cov later in this table. Don't use `access` to check if a user is authorized to open a before opening it; this might be a security hole.
`chdir`	`chdir(path)` Sets the current working directory of the process to *path*, may be a file descriptor or path-like object.

`chmod`,
`lchmod`
	`chmod(path, mode, *, dir_fd=None, follow_symlinks=T`
	`lchmod(path, mode)`
	Changes the permissions of the file (or file descriptor or p object) *path*, as encoded in integer *mode*. *mode* can be zero of `os.R_OK`, `os.W_OK`, and `os.X_OK` (joined with the bitwise operator `|`, if more than one) for read, write, and execute permissions. On Unix-like platforms, *mode* can be a richer pattern (as covered in the previous section) to specify dif permissions for user, group, and other, as well as having special, rarely used bits defined in the module `stat` and l the **online docs**. Pass `follow_symlinks=False` (or use `lch` change permissions of a symbolic link, not the target of th

`DirEntry`	An instance *d* of class `DirEntry` supplies attributes *name* a holding the item's base name and full path, respectively, a several methods, of which the most frequently used are `i` `is_file`, and `is_symlink`. `is_dir` and `is_file` by default symbolic links: pass `follow_symlinks=False` to avoid this behavior. *d* avoids system calls as much as feasible, and w needs one, it caches the results. If you need information t

	guaranteed to be up-to-date, you can call `os.stat(d.path` use the `stat_result` instance it returns; however, this sa `scandir`'s potential performance improvements. For mor complete information, see the **online docs**.
`getcwd`, `getcwdb`	`getcwd()`, `getcwdb()` `getcwd` returns a `str`, the path of the current working dir `getcwdb` returns a `bytes` string (3.8+ with UTF-8 encodin Windows).
`link`	`link(src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True)` Creates a *hard* link named *dst*, pointing to *src*. Both may like objects. Set `src_dir_fd` and/or `dst_dir_fd` for `link` t on relative paths, and pass `follow_symlinks=False` to on operate on a symbolic link, not the target of that link. To symbolic ("soft") link, use the `symlink` function, covered l this table.
`listdir`	`listdir(path='.')` Returns a list whose items are the names of all files and subdirectories in the directory, file descriptor (referring t directory), or path-like object `path`. The list is in arbitrary and does *not* include the special directory names `'.'` (cu directory) and `'..'` (parent directory). When `path` is of ty `bytes`, the filenames returned are also of type `bytes`; othe they are of type `str`. See also the alternative function `sca` covered later in this table, which can offer performance improvements in some cases. Don't remove or add files t directory during the call of this function: that may produ unexpected results.
`mkdir`, `makedirs`	`mkdir(path, mode=0777, dir_fd=None)`, `makedirs(path, mode=0777, exist_ok=False)` `mkdir` creates only the rightmost directory of *path* and ra

	OSError if any of the previous directories in *path* do not e mkdir accepts dir_fd for paths relative to a file descripto makedirs creates all directories that are part of *path* and yet exist (pass exist_ok=**True** to avoid raising FileExists Both functions use mode as permission bits of directories create, but some platforms, and some newly created intermediate-level directories, may ignore mode; use chmo explicitly set permissions.
remove, unlink	remove(*path*, *, dir_fd=**None**), unlink(*path*, *, dir_fd=**None**) Removes the file or path-like object *path* , which may be r to dir_fd. See rmdir later in this table to remove a direct rather than a file. unlink is a synonym of remove.
removedirs	removedirs(*path*) Loops from right to left over the directories that are part which may be a path-like object, removing each one. The ends when a removal attempt raises an exception, genera because a directory is not empty. removedirs does not pr the exception, as long as it has removed at least one direc
rename, renames	rename(src, dst, *, src_dir_fd=**None**, dst_dir_fd=**No** renames(*src*, *dst*, /) Renames ("moves") the file, path-like object, or directory src to dst. If dst already exists, rename may either replac raise an exception; to guarantee replacement, instead cal function os.replace. To use relative paths, pass src_dir_ and/or dst_dir_fd. renames works like rename, except it creates all intermedi directories needed for *dst*. After renaming, renames remo empty directories from the path *src* using removedirs. It propagate any resulting exception; it's not an error if the renaming does not empty the starting directory of *src*. re cannot accept relative path arguments.

`rmdir`	`rmdir(path, *, dir_fd=None)` Removes the empty directory or path-like object named *p* (which may be relative to `dir_fd`). Raises `OSError` if the r fails, and, in particular, if the directory is not empty.
`scandir`	`scandir(path='.')` Returns an iterator yielding `os.DirEntry` instances for ea in `path`, which may be a string, a path-like object, or a file descriptor. Using `scandir` and calling each resulting item methods to determine its characteristics can provide perf improvements compared to using `listdir` and `stat`, dep on the underlying platform. `scandir` may be used as a co manager: e.g., **with** `os.scandir(path)` **as** *itr*: to ensure of the iterator (freeing up resources) when done.
`stat`, `lstat`, `fstat`	`stat(path, *, dir_fd=None, follow_symlinks=True)`, `lstat(path, *, dir_fd=None)`, `fstat(fd)` `stat` returns a value *x* of type `stat_result`, which provid least) 10 items of information about `path`. `path` may be a descriptor (in this case you can use `stat(fd)` or `fstat`, w accepts file descriptors), path-like object, or subdirectory. may be a relative path of `dir_fd`, or a symlink (if `follow_symlinks=False`, or if using `lstat`; on Windows, **reparse points** that the OS can resolve are followed unle `follow_symlinks=False`). The `stat_result` value is a tup values that also supports named access to each of its cont values (similar to a `collections.namedtuple`, though not implemented as such). Accessing the items of `stat_resul` their numeric indices is possible but not advisable, becau resulting code is not readable; use the corresponding attr names instead. **Table 11-17** lists the main 10 attributes of `stat_result` instance and the meaning of the correspond items.

Table 11-17. Items (attributes) of a `stat_result` instance

Item index	Attribute name	Meaning
0	`st_mode`	Protection and other bits
1	`st_ino`	Inode number
2	`st_dev`	Device ID
3	`st_nlink`	Number of hard link
4	`st_uid`	User ID of owner
5	`st_gid`	Group ID of owner
6	`st_size`	Size, in bytes
7	`st_atime`	Time of last access
8	`st_mtime`	Time of last modifica
9	`st_ctime`	Time of last status ch

For example, to print the size, in bytes, of file *path*, you c
any of:

```
import os
print(os.stat(path)[6])        # works but uncle
print(os.stat(path).st_size)   # easier to under
print(os.path.getsize(path))   # convenience fur
                              # that wraps stat
```

	Time values are in seconds since the epoch, as covered in **Chapter 13** (`int`, on most platforms). Platforms unable to meaningful value for an item use a dummy value. For oth platform-dependent attributes of `stat_result` instances, **online docs**.
`symlink`	`symlink(`*`target`*`,` *`symlink_path`*`, target_is_directory=` `*, dir_fd=`**`None`**`)` Creates a symbolic link named *`symlink_path`* to the file, d or path-like object *`target`*, which may be relative to `dir_` `target_is_directory` is used only on Windows systems, specify whether the created symlink should represent a f directory; this argument is ignored on non-Windows syst (Calling `os.symlink` typically requires elevated privileges run on Windows.)
`utime`	`utime(`*`path`*`, times=`**`None`**`, *, [`*`ns`*`, ]dir_fd=`**`None`**`,` `follow_symlinks=`**`True`**`)` Sets the accessed and modified times of file, directory, or object *`path`*, which may be relative to `dir_fd`, and may be symlink if `follow_symlinks=`**`False`**. If `times` is **`None`**, `utim` the current time. Otherwise, `times` must be a pair of num seconds since the epoch, as covered in **Chapter 13**) in the (*`accessed`*`,` *`modified`*). To specify nanoseconds instead, p (*`acc_ns`*`,` *`mod_ns`*), where each member is an `int` express nanoseconds since the epoch. Do *not* specify both `times` a
`walk`, `fwalk`	`walk(`*`top`*`, topdown=`**`True`**`, onerror=`**`None`**`, followlinks=` `fwalk(top='.', topdown=`**`True`**`, onerror=`**`None`**`, *,` `follow_symlinks=`**`False`**`, dir_fd=`**`None`**`)` `walk` is a generator yielding an item for each directory in whose root is the directory or path-like object *`top`*. When is **`True`**, the default, `walk` visits directories from the tree's downward; when `topdown` is **`False`**, `walk` visits directories the tree's leaves upward. By default, `walk` catches and ign `OSError` exception raised during the tree-walk; set `onerro`

callable in order to catch any `OSError` exception raised du
tree-walk and pass it as the only argument in a call to `one`
which may process it, ignore it, or **`raise`** it to terminate th
walk and propagate the exception (the filename is availal
`filename` attribute of the exception object).

Each item `walk` yields is a tuple of three subitems: *`dirpat`*
string that is the directory's path; *`dirnames`*, a list of name
subdirectories that are immediate children of the directo
(special directories `'.'` and `'..'` are *not* included); and
`filenames`, a list of names of files that are directly in the

If `topdown` is **`True`**, you can alter list *`dirnames`* in place, re
some items and/or reordering others, to affect the tree-wa
subtree rooted at *`dirpath`*; `walk` iterates only on subdirec
in *`dirnames`*, in the order in which they're left. Such altera
have no effect if `topdown` is **`False`** (in this case, `walk` has a
visited all subdirectories by the time it visits the current
and yields its item).

`walk`, `fwalk` *(cont.)*	By default, `walk` does not walk down symbolic links that r directories. To get such extra walking, pass `followlinks=` beware: this can cause infinite looping if a symbolic link to a directory that is its ancestor. `walk` doesn't take precau against this anomaly.

FOLLOWLINKS VERSUS FOLLOW_SYMLINKS

Note that, for `os.walk` *only*, the argument that is named `follow_`
everywhere else is instead named `followlinks`.

`fwalk` (Unix only) works like `walk`, except that *`top`* may be
relative path of file descriptor `dir_fd`, and `fwalk` yields *fo*
member tuples: the first three members (*`dirpath`*, *`dirnam`*
`filenames`) are identical to `walk`'s yielded values, and the
member is *`dirfd`*, a file descriptor of *`dirpath`*. Note that b
and `fwalk` default to *not* following symlinks.

File descriptor operations

In addition to the many functions covered earlier, the `os` module supplies several that work specifically with file descriptors. A *file descriptor* is an integer that the operating system uses as an opaque handle to refer to an open file. While it is usually best to use Python file objects (covered in "The io Module") for I/O tasks, sometimes working with file descriptors lets you perform some operations faster, or (at the possible expense of portability) in ways not directly available with `io.open`. File objects and file descriptors are not interchangeable.

To get the file descriptor *n* of a Python file object *f*, call *n* = *f*`.fileno()`. To create a new Python file object *f* using an existing open file descriptor *fd*, use *f* = `os.fdopen(`*fd*`)`, or pass *fd* as the first argument of `io.open`. On Unix-like and Windows platforms, some file descriptors are preallocated when a process starts: `0` is the file descriptor for the process's standard input, `1` for the process's standard output, and `2` for the process's standard error. Calling `os` module methods such as `dup` or `close` on these preallocated file descriptors can be useful for redirecting or manipulating standard input and output streams.

The `os` module provides many functions for dealing with file descriptors; some of the most useful are listed in Table 11-18.

Table 11-18. Useful `os` module functions to deal with file descriptors

`close`	`close(`*fd*`)` Closes file descriptor *fd*.
`closerange`	`closerange(`*fd_low*`, `*fd_high*`)` Closes all file descriptors from *fd_low*, included, to *fd_high*, excluded, ignoring any errors that may occur.
`dup`	`dup(`*fd*`)` Returns a file descriptor that duplicates file descriptor *fd*.

`dup2`	`dup2(`*`fd`*`, `*`fd2`*`)` Duplicates file descriptor *`fd`* to file descriptor *`fd2`*. When file descriptor *`fd2`* is already open, `dup2` first closes *`fd2`*.
`fdopen`	`fdopen(`*`fd`*`, `*`*a`*`, `*`**k`*`)` Like `io.open`, except that *`fd`* *must* be an `int` that is an open file descriptor.
`fstat`	`fstat(`*`fd`*`)` Returns a `stat_result` instance *`x`*, with information about the file open on file descriptor *`fd`*. **Table 11-17** covers *`x`*'s contents.
`lseek`	`lseek(`*`fd`*`, `*`pos`*`, `*`how`*`)` Sets the current position of file descriptor *`fd`* to the signed integer byte offset *`pos`* and returns the resulting byte offset from the start of the file. *`how`* indicates the reference (point `0`). When *`how`* is `os.SEEK_SET`, a *`pos`* of `0` means the start of the file; for `os.SEEK_CUR` it means the current position, and for `os.SEEK_END` it means the end of the file. For example, `lseek(`*`fd`*`, 0, os.SEEK_CUR)` returns the current position's byte offset from the start of the file without affecting the current position. Normal disk files support seeking; calling `lseek` on a file that does not support seeking (e.g., a file open for output to a terminal) raises an exception.

`open`	`open(`*`file`*`, `*`flags`*`, mode=0o777)`
	Returns a file descriptor, opening or creating a file named by string *`file`*. When `open` creates the file, it uses `mode` as the file's permission bits. *`flags`* is an `int`, normally the bitwise OR (with operator `|`) of one or more of the following attributes of `os`:
	
	`O_APPEND`

Appends any new data to *file*'s current contents

`O_BINARY`

Opens *file* in binary rather than text mode on Windows platforms (raises an exception on Unix-like platforms)

`O_CREAT`

Creates *file* if *file* does not already exist

`O_DSYNC`, `O_RSYNC`, `O_SYNC`, `O_NOCTTY`

Set the synchronization mode accordingly, if the platform supports this

`O_EXCL`

Raises an exception if *file* already exists

`O_NDELAY`, `O_NONBLOCK`

Opens *file* in nonblocking mode, if the platform supports this

`O_RDONLY`, `O_WRONLY`, `O_RDWR`

Opens *file* for read-only, write-only, or read/write access, respectively (mutually exclusive: exactly one of these attributes *must* be in *flags*)

`O_TRUNC`

Throws away previous contents of *file* (incompatible with `O_RDONLY`)

`pipe`	`pipe()` Creates a pipe and returns a pair of file descriptors (*r_fd, w_fd*), respectively open for reading and writing.
`read`	`read(fd, n)` Reads up to *n* bytes from file descriptor *fd* and returns them as a bytestring. Reads and returns *m < n* bytes when only *m* more bytes are currently available for reading from the file. In particular, returns the empty string when no more bytes are currently available from the file, typically because the file is finished.

`write`	`write(`*`fd`*`, `*`s`*`)` Writes all bytes from bytestring *`s`* to file descriptor *`fd`* and returns the number of bytes written.

The os.path Module

The `os.path` module supplies functions to analyze and transform path strings and path-like objects. The most commonly useful functions from the module are listed in **Table 11-19**.

Table 11-19. Frequently used functions of the `os.path` module

`abspath`	`abspath(`*`path`*`)`
	Returns a normalized absolute path string equivalent to *`path`*, just like (in the case where *`path`* is the name of a file in the current directory):
	`os.path.normpath(os.path.join(os.getcwd(), `*`path`*`))`
	For example, `os.path.abspath(os.curdir)` is the same as `os.getcwd()`.

`basename`	`basename(`*`path`*`)` Returns the base name part of *`path`*, just like `os.path.split(`*`path`*`)[1]`. For example, `os.path.basename('b/c/d.e')` returns `'d.e'`.
`commonpath`	`commonpath(`*`list`*`)` Accepts a sequence of strings or path-like objects, and returns the longest common subpath. Unlike `commonprefix`, only returns a valid path; raises `ValueError` if *list* is empty, contains a mixture of absolute and relative paths, or contains paths on different drives.
`commonprefix`	`commonprefix(`*`list`*`)` Accepts a list of strings or pathlike objects and

	returns the longest string that is a prefix of all items in the list, or `'.'` if *list* is empty. For example, `os.path.commonprefix(['foobar', 'foolish'])` returns `'foo'`. May return an invalid path; see `commonpath` if you want to avoid this.
`dirname`	`dirname(`*path*`)` Returns the directory part of *path*, just like `os.path.split(`*path*`)[0]`. For example, `os.path.dirname('b/c/d.e')` returns `'b/c'`.
`exists`, `lexists`	`exists(`*path*`)`, `lexists(`*path*`)` `exists` returns **`True`** when *path* names an existing file or directory (*path* may also be an open file descriptor or path-like object); otherwise, returns **`False`**. In other words, `os.path.exists(`*x*`)` is the same as `os.access(`*x*, `os.F_OK)`. `lexists` is the same, but also returns **`True`** when *path* names an existing symbolic link that indicates a nonexistent file or directory (sometimes known as a *broken symlink*), while `exists` returns **`False`** in such cases. Both return **`False`** for paths containing characters or bytes that are not representable at the OS level.
`expandvars`, `expanduser`	`expandvars(`*path*`)`, `expanduser(`*path*`)` Returns a copy of string or path-like object *path*, where each substring of the form `$`*name* or `${`*name*`}` (and `%`*name*`%` on Windows only) is replaced with the value of environment variable *name*. For example, if environment variable `HOME` is set to `/u/alex`, the following code:

```
import os
print(os.path.expandvars('$HOME/foo/'))
```

	emits `/u/alex/foo/`. `os.path.expanduser` expands a leading `~` or `~user`, if any, to the path of the home directory of the current user.
`getatime`, `getctime`, `getmtime`, `getsize`	`getatime(`*`path`*`)`, `getctime(`*`path`*`)`, `getmtime(`*`path`*`)`, `getsize(`*`path`*`)` Each of these functions calls `os.stat(`*`path`*`)` and returns an attribute from the result: respectively, `st_atime`, `st_ctime`, `st_mtime`, and `st_size`. See **Table 11-17** for more details about these attributes.
`isabs`	`isabs(`*`path`*`)` Returns **`True`** when *`path`* is absolute. (A path is absolute when it starts with a (back)slash (`/` or `\`), or, on some non-Unix-like platforms, such as Windows, with a drive designator followed by `os.sep`.) Otherwise, `isabs` returns **`False`**.
`isdir`	`isdir(`*`path`*`)` Returns **`True`** when *`path`* names an existing directory (`isdir` follows symlinks, so `isdir` and `islink` may both return **`True`**); otherwise, returns **`False`**.
`isfile`	`isfile(`*`path`*`)` Returns **`True`** when *`path`* names an existing regular file (`isfile` follows symlinks, so `islink` may also be **`True`**); otherwise, returns **`False`**.
`islink`	`islink(`*`path`*`)` Returns **`True`** when *`path`* names a symbolic link; otherwise, returns **`False`**.

`ismount`	`ismount(`*`path`*`)` Returns **`True`** when *`path`* names a **mount point**; otherwise, returns **`False`**.

`join`	`join(`*`path, *paths`*`)`
	Returns a string that joins the arguments (strings or path-like objects) with the appropriate path separator for the current platform. For example, on Unix, exactly one slash character `/` separates adjacent path components. If any argument is an absolute path, `join` ignores previous arguments. For example:
	
	`print(os.path.join('a/b', 'c/d', 'e/f'))`
	`# on Unix prints: a/b/c/d/e/f`
	`print(os.path.join('a/b', '/c/d', 'e/f'))`
	`# on Unix prints: /c/d/e/f`
	
	The second call to `os.path.join` ignores its first argument `'a/b'`, since its second argument `'/c/d'` is an absolute path.

`normcase`	`normcase(`*`path`*`)` Returns a copy of *`path`* with case normalized for the current platform. On case-sensitive filesystems (typical in Unix-like systems), *`path`* is returned unchanged. On case-insensitive filesystems (typical in Windows), it lowercases the string. On Windows, `normcase` also converts each `/` to a `\\`.
`normpath`	`normpath(`*`path`*`)` Returns a normalized pathname equivalent to *`path`*, removing redundant separators and path-navigation aspects. For example, on Unix, `normpath` returns `'a/b'` when *`path`* is any of `'a//b'`, `'a/./b'`, or

	`'a/c/../b'`. `normpath` makes path separators appropriate for the current platform. For example, on Windows, separators become `\\`.
`realpath`	`realpath(`*`path`*`, *, strict=`**`False`**`)` Returns the actual path of the specified file or directory or path-like object, resolving symlinks along the way. 3.10+ Set `strict=`**`True`** to raise `OSError` when *path* doesn't exist, or when there is a loop of symlinks.
`relpath`	`relpath(`*`path`*`, start=os.curdir)` Returns a path to the file or directory *path* (a `str` or path-like object) relative to directory `start`.
`samefile`	`samefile(`*`path1`*`, `*`path2`*`)` Returns **`True`** if both arguments (strings or path-like objects) refer to the same file or directory.
`sameopenfile`	`sameopenfile(`*`fd1`*`, `*`fd2`*`)` Returns **`True`** if both arguments (file descriptors) refer to the same file or directory.
`samestat`	`samestat(`*`stat1`*`, `*`stat2`*`)` Returns **`True`** if both arguments (instances of `os.stat_result`, typically results of `os.stat` calls) refer to the same file or directory.
`split`	`split(`*`path`*`)` Returns a pair of strings `(`*`dir`*`, `*`base`*`)` such that `join(`*`dir`*`, `*`base`*`)` equals *path*. *base* is the last component and never contains a path separator. When *path* ends in a separator, *base* is `''`. *dir* is the leading part of *path*, up to the last separator excluded. For example, `os.path.split('a/b/c/d')` returns `('a/b/c', 'd')`.

`splitdrive`	`splitdrive(`*`path`*`)` Returns a pair of strings `(`*`drv`*`, `*`pth`*`)` such that *`drv`*`+`*`pth`* equals *`path`*. *`drv`* is a drive specification, or `''`; it is always `''` on platforms without drive specifications, e.g. Unix-like systems. On Windows, `os.path.splitdrive('c:d/e')` returns `('c:', 'd/e')`.
`splitext`	`splitext(`*`path`*`)` Returns a pair `(`*`root`*`, `*`ext`*`)` such that *`root`*`+`*`ext`* equals *`path`*. *`ext`* is either `''` or starts with a `'.'` and has no other `'.'` or path separator. For example, `os.path.splitext('a.a/b.c.d')` returns the pair `('a.a/b.c', '.d')`.

OSError Exceptions

When a request to the operating system fails, `os` raises an exception, an instance of `OSError`. `os` also exposes the built-in exception class `OSError` with the synonym `os.error`. Instances of `OSError` expose three useful attributes, detailed in **Table 11-20**.

Table 11-20. Attributes of `OSError` instances

`errno`	The numeric error code of the operating system error
`filename`	The name of the file on which the operation failed (file-related functions only)
`strerror`	A string that briefly describes the error

`OSError` has subclasses to specify what the problem was, as discussed in **“OSError subclasses”**.

os functions can also raise other standard exceptions, such as TypeError or ValueError, when called with invalid argument types or values, so that they didn't even attempt the underlying operating system functionality.

The errno Module

The errno module supplies dozens of symbolic names for error code numbers. Use errno to handle possible system errors selectively, based on error codes; this will enhance your program's portability and readability. However, a selective **except** with the appropriate OSError subclass often works better than errno. For example, to handle "file not found" errors, while propagating all other kinds of errors, you could use:

```
import errno
try:
    os.some_os_function_or_other()
except FileNotFoundError as err:
    print(f'Warning: file {err.filename!r} not found; continuing')
except OSError as oserr:
    print(f'Error {errno.errorcode[oserr.errno]}; continuing')
```

errno supplies a dictionary named errorcode: the keys are error code numbers, and the corresponding values are the error names, strings such as 'ENOENT'. Displaying errno.errorcode[err.errno] as part of the explanation behind some OSError instance's err can often make the diagnosis clearer and more understandable to readers who specialize in the specific platform.

The pathlib Module

The pathlib module provides an object-oriented approach to filesystem paths, pulling together a variety of methods for handling paths and files as objects, not as strings (unlike os.path). For most use cases, pathlib.Path will provide everything you'll need. On rare occasions, you’ll want to instantiate a platform-specific path, or a “pure” path that doesn’t interact with the operating system; see the **online docs** if you need such advanced functionality. The most commonly useful functions of `pathlib.Path` are listed in **Table 11-21**, with examples for a `pathlib.Path` object *p*. On Windows, `pathlib.Path` objects are returned as `WindowsPath`; on Unix, as `PosixPath`, as shown in the examples in **Table 11-21**. (For clarity, we are simply importing `pathlib` rather than using the more common and idiomatic **`from`** `pathlib` **`import`** `Path`.)

PATHLIB METHODS RETURN PATH OBJECTS, NOT STRINGS

Keep in mind that `pathlib` methods typically return a path object, not a string, so results of similar methods in `os` and `os.path` do *not* test as being identical.

Table 11-21. Commonly used methods of `pathlib.Path`

`chmod`,
`lchmod`
	`p.chmod(mode, follow_symlinks=True)`,
	`p.lchmod(mode)`
	`chmod` changes the file mode and permissions, like `os.ch`
	Table 11-16). On Unix platforms, **3.10+** set `follow_syml`
	to change permissions on the symbolic link rather than
	use `lchmod`. See the **online docs** for more information o
	settings.`lchmod` is like `chmod` but, when *p* points to a sym
	changes the symbolic link rather than its target. Equival
	`pathlib.Path.chmod(follow_symlinks=False)`.

`cwd`	`pathlib.Path.cwd()` Returns the current working directory as a path object.
`exists`	`p.exists()` Returns **`True`** when *p* names an existing file or directory symbolic link pointing to an existing file or directory); o returns **`False`**.
`expanduser`	`p.expanduser()` Returns a new path object with a leading ~ expanded to

	the home directory of the current user, or `~user` expand path of the home directory of the given user. See also ho this table.
`glob`, `rglob`	*p*.glob(*pattern*), *p*.rglob(*pattern*) Yield all matching files in directory *p* in arbitrary order. include `**` to allow recursive globbing in *p* or any subdi always performs recursive globbing in *p* and all subdire *pattern* started with `'**/'`. For example:

```
>>> sorted(td.glob('*'))
```

```
[WindowsPath('tempdir/bar'),
WindowsPath('tempdir/foo')]
```

```
>>> sorted(td.glob('**/*'))
```

```
[WindowsPath('tempdir/bar'),
WindowsPath('tempdir/bar/baz'),
WindowsPath('tempdir/bar/boo'),
WindowsPath('tempdir/foo')]
```

```
>>> sorted(td.glob('*/**/*')) # expanding at 2
```

```
[WindowsPath('tempdir/bar/baz'),
WindowsPath('tempdir/bar/boo')]
```

```
>>> sorted(td.rglob('*'))  # just like glob('*
```

```
[WindowsPath('tempdir/bar'),
WindowsPath('tempdir/bar/baz'),
WindowsPath('tempdir/bar/boo'),
WindowsPath('tempdir/foo')]
```

`hardlink_to`	*`p`*`.hardlink_to(`*`target`*`)` 3.10+ Makes *p* a hard link to the same file as *`target`*. Re deprecated `link_to` 3.8+, `-3.10` Note: the order of argu `link_to` was like `os.link`, described in **Table 11-16**; for like for `symlink_to` later in this table, it's the reverse.
`home`	`pathlib.Path.home()` Returns the user's home directory as a path object.
`is_dir`	*`p`*`.is_dir()` Returns **`True`** when *p* names an existing directory (or a s to a directory); otherwise, returns **`False`**.
`is_file`	*`p`*`.is_file()` Returns **`True`** when *p* names an existing file (or a symbo file); otherwise, returns **`False`**.
`is_mount`	*`p`*`.is_mount()` Returns **`True`** when *p* is a *mount point* (a point in a filesy different filesystem has been mounted); otherwise, retu the **online docs** for details. Not implemented on Windo
`is_symlink`	*`p`*`.is_symlink()` Returns **`True`** when *p* names an existing symbolic link; o returns **`False`**.

`iterdir`	`p.iterdir()` Yields path objects for the contents of directory *p* (`'.'` a included) in arbitrary order. Raises `NotADirectoryErro` not a directory. May produce unexpected results if you r from *p*, or add a file to *p*, after you create the iterator an you're done using it.

`mkdir`	`p.mkdir(mode=0o777, parents=False, exist_ok=Fals`
	Creates a new directory at the path. Use `mode` to set file r access flags. Pass `parents=True` to create any missing pa needed. Pass `exist_ok=True` to ignore `FileExistsError`
	For example:
	`>>> td=pathlib.Path('tempdir/')`
	`>>> td.mkdir(exist_ok=True)`
	`>>> td.is_dir()`
	`True`
	See the **online docs** for thorough coverage.

`open`	`p.open(mode='r', buffering=-1, encoding=None, er`
	`newline=None)`
	Opens the file pointed to by the path, like the built-in op other args the same).

`read_bytes`	`p.read_bytes()` Returns the binary contents of *p* as a `bytes` object.
`read_text`	`p.read_text(encoding=None, errors=None)` Returns the decoded contents of *p* as a string.

`readlink`	*p*`.readlink()` **3.9+** Returns the path to which a symbolic link points.
`rename`	*p*`.rename(`*target*`)` Renames *p* to *target* and **3.8+** returns a new `Path` insta to *target*. *target* may be a string, or an absolute or rela however, relative paths are interpreted relative to the *c* working directory, *not* the directory of *p*. On Unix, when existing file or empty directory, `rename` replaces it silent user has permission; on Windows, `rename` raises `FileEx`
`replace`	*p*`.replace(`*target*`)` Like *p*`.rename(`*target*`)`, but, on any platform, when *tar* existing file (or, except on Windows, an empty directory replaces it silently when the user has permission. For ex

```
>>> p.read_text()
```

```
'spam'
```

```
>>> t.read_text()
```

```
'and eggs'
```

```
>>> p.replace(t)
```

```
WindowsPath('C:/Users/annar/testfile.txt')
```

```
>>> t.read_text()
```

```
'spam'
```

```
>>> p.read_text()
```

```
Traceback (most recent call last):
```

```
...
```

```
FileNotFoundError: [Errno 2] No such file...
```

resolve   *p*.resolve(strict=**False**)

Returns a new absolute path object with symbolic links eliminates any `'..'` components. Set `strict=True` to rai `FileNotFoundError` when the path does not exist, or Rur when it encounters an infinite loop. For example, on the directory created in the `mkdir` example earlier in this ta

```
>>> td.resolve()
```

```
PosixPath('/Users/annar/tempdir')
```

`rmdir`	*p*.rmdir() Removes directory *p*. Raises `OSError` if *p* is not empty.
`samefile`	*p*.samefile(*target*) Returns **True** when *p* and *target* indicate the same file; returns **False**. *target* may be a string or a path object.
`stat`	p.stat(*, follow_symlinks=**True**) Returns information about the path object, including pe and size; see `os.stat` in **Table 11-16** for return values. a symbolic link itself, rather than its target, pass follow_symlinks=**False**.
`symlink_to`	*p*.symlink_to(*target*, target_is_directory=**False**) Makes *p* a symbolic link to *target*. On Windows, you mu target_is_directory=**True** if *target* is a directory. (POS this argument.) (On Windows 10+, like `os.symlink`, requ Developer Mode permissions; see the **online docs** for de the order of arguments is the reverse of the order for os `os.symlink`, described in **Table 11-16**.
`touch`	*p*.touch(mode=0o666, exist_ok=**True**) Like `touch` on Unix, creates an empty file at the given pa file already exists, updates the modification time to the if exist_ok=**True**; if exist_ok=**False**, raises FileExistsE example:

```
>>> d
```

```
WindowsPath('C:/Users/annar/Documents')
```

```
>>> f = d / 'testfile.txt'
```

```
>>> f.is_file()
```

```
False
```

```
>>> f.touch()
>>> f.is_file()
```

```
True
```

unlink	*p*.unlink(missing_ok=**False**) Removes file or symbolic link *p*. (Use rmdir for directori described earlier in this table.) **3.8+** Pass missing_ok=T **FileExistsError**.
write_bytes	*p*.write_bytes(*data*) Opens (or, if need be, creates) the file pointed to in bytes *data* to it, then closes the file. Overwrites the file if it alr
write_text	*p*.write_text(data, encoding=**None**, errors=**None**, n Opens (or, if need be, creates) the file pointed to in text r *data* to it, then closes the file. Overwrites the file if it alr **3.10+** When newline is **None** (the default), translates an system default line separator; when '\r' or '\r\n', tran the given string; when '' or '\n', no translation takes p

pathlib.Path objects also support the attributes listed in **Table 11-22** to access the various component parts of the path string. Note that some attributes are strings, while others are Path objects. (For brevity, OS-specific types such as PosixPath or WindowsPath are shown simply using the abstract Path class.)

Table 11-22. Attributes of an instance `p` of `pathlib.Path`

Attribute	Description	Value for Unix path Path('/usr/bin/python')	Value for Windows path Path(r'c:\Python3\python.exe')
`anchor`	Combination of `drive` and `root`	`'/'`	`'c:\\'`
`drive`	Drive letter of *p*	`''`	`'c:'`
`name`	End component of *p*	`'python'`	`'python.exe'`
`parent`	Parent directory of *p*	`Path('/usr/bin')`	`Path('c:\\Python3`
`parents`	Ancestor directories of *p*	`(Path('/usr/bin'), Path('/usr'), Path('/'))`	`(Path('c:\\Python Path('c:\\'))`
`parts`	Tuple of all components of *p*	`('/', 'usr', 'bin', 'python')`	`('c:\\', 'Python3 'python.exe')`
`root`	Root directory of *p*	`'/'`	`'\\'`

Attribute	Description	Value for Unix path Path('/usr/bin/python')	Value for Windows path Path(r'c:\Python3\python.exe')
stem	Name of *p*, minus suffix	'python'	'python'
suffix	Ending suffix of *p*	''	'.exe'
suffixes	List of all suffixes of *p*, as delimited by '.' characters	[]	['.exe']

The **online documentation** includes more examples for paths with additional components, such as filesystem and UNC shares.

pathlib.Path objects also support the '/' operator, an excellent alternative to os.path.join or Path.joinpath from the Path module. See the example code in the description of Path.touch in **Table 11-21**.

The stat Module

The function os.stat (covered in **Table 11-16**) returns instances of stat_result, whose item indices, attribute names, and meaning are also covered there. The stat module supplies attributes with names like those of stat_result's attributes in uppercase, and corresponding values that are the corresponding item indices.

The more interesting contents of the stat module are functions to examine the st_mode attribute of a stat_result instance and determine the kind of file. `os.path` also supplies functions for such tasks, which operate directly on the file's *path*. The functions supplied by `stat`, shown in **Table 11-23**, are faster than `os`'s when you perform several tests on the same file: they require only one `os.stat` system call at the start of a series of tests to obtain the file's `st_mode`, while the functions in `os.path` implicitly ask the operating system for the same information at each test. Each function returns **`True`** when *mode* denotes a file of the given kind; otherwise, it returns **`False`**.

Table 11-23. `stat` module functions for examining `st_mode`

`S_ISBLK`	`S_ISBLK(`*mode*`)` Indicates whether *mode* denotes a special-device file of the block kind
`S_ISCHR`	`S_ISCHR(`*mode*`)` Indicates whether *mode* denotes a special-device file of the character kind
`S_ISDIR`	`S_ISDIR(`*mode*`)` Indicates whether *mode* denotes a directory
`S_ISFIFO`	`S_ISFIFO(`*mode*`)` Indicates whether *mode* denotes a FIFO (also known as a "named pipe")
`S_ISLNK`	`S_ISLNK(`*mode*`)` Indicates whether *mode* denotes a symbolic link
`S_ISREG`	`S_ISREG(`*mode*`)` Indicates whether *mode* denotes a normal file (not a directory, special device-file, etc.)
`S_ISSOCK`	`S_ISSOCK(`*mode*`)` Indicates whether *mode* denotes a Unix-domain socket

Several of these functions are meaningful only on Unix-like systems, since other platforms do not keep special files such as devices and sockets in the same namespace as regular files; Unix-like systems do.

The `stat` module also supplies two functions that extract relevant parts of a file's *mode* (*x*`.st_mode`, for some result *x* of function `os.stat`), listed in **Table 11-24**.

Table 11-24. `stat` module functions for extracting bits from mode

`S_IFMT`	`S_IFMT(`*mode*`)` Returns those bits of *mode* that describe the kind of file (i.e., the bits that are examined by the functions `S_ISDIR`, `S_ISREG`, etc.)
`S_IMODE`	`S_IMODE(`*mode*`)` Returns those bits of *mode* that can be set by the function `os.chmod` (i.e., the permission bits and, on Unix-like platforms, a few other special bits such as the `set-user-id` flag)

The `stat` module supplies a utility function, `stat.filemode(`*mode*`)`, that converts a file's mode to a human readable string of the form `'-rwxrwxrwx'`.

The filecmp Module

The `filecmp` module supplies a few functions that are useful for comparing files and directories, listed in **Table 11-25**.

Table 11-25. Useful functions of the `filecmp` module

`clear_cache`	`clear_cache()` Clears the `filecmp` cache, which may be useful in quick file comparisons.

`cmp`	`cmp(`*`f1`*`, `*`f2`*`, shallow=`**`True`**`)` Compares the files (or `pathlib.Paths`) identified by path strings *`f1`* and *`f2`*. If the files are deemed to be equal, `cmp` returns **`True`**; otherwise, it returns **`False`**. If `shallow` is **`True`**, files are deemed to be equal if their `stat` tuples are equal. When `shallow` is **`False`**, `cmp` reads and compares the contents of files whose `stat` tuples are equal.
`cmpfiles`	`cmpfiles(`*`dir1`*`, `*`dir2`*`, `*`common`*`, shallow=`**`True`**`)` Loops on the sequence *`common`*. Each item of *`common`* is a string that names a file present in both directories *`dir1`* and *`dir2`*. `cmpfiles` returns a tuple whose items are three lists of strings: `(`*`equal`*`, `*`diff`*`,` and *`errs`*`)`. *`equal`* is the list of names of files that are equal in both directories, *`diff`* is the list of names of files that differ between directories, and *`errs`* is the list of names of files that it could not compare (because they do not exist in both directories, or there is no permission to read one or both of them). The argument `shallow` is the same as for `cmp`.

The `filecmp` module also supplies the class `dircmp`. The constructor for this class has the signature:

`dircmp`	**`class`** `dircmp(`*`dir1`*`, `*`dir2`*`, ignore=`**`None`**`, hide=`**`None`**`)` Creates a new directory-comparison instance object comparing directories *`dir1`* and *`dir2`*, ignoring names listed in `ignore` and hiding names listed in `hide` (defaulting to `'.'` and `'..'` when `hide=`**`None`**). The default value for `ignore` is supplied by the `DEFAULT_IGNORE` attribute of the `filecmp` module; at the time of this writing it is `['RCS', 'CVS', 'tags', '.git', '.hg', '.bzr', '_darcs', '__pycache__']`. Files in the

directories are compared like with `filecmp.cmp` with `shallow=True`.

A `dircmp` instance *d* supplies three methods, detailed in **Table 11-26**.

Table 11-26. Methods supplied by a `dircmp` instance `d`

`report`	`report_full_closure()` Outputs to `sys.stdout` a comparison between *dir1* and *dir2* and all their common subdirectories, recursively
`report_full_closure`	`report_full_closure()` Outputs to `sys.stdout` a comparison between *dir1* and *dir2* and all their common subdirectories, recursively
`report_partial_closure`	`report_partial_closure()` Outputs to `sys.stdout` a comparison between *dir1* and *dir2* and their common immediate subdirectories

In addition, *d* supplies several attributes, covered in **Table 11-27**. These attributes are computed "just in time" (i.e., only if and when needed, thanks to a `__getattr__` special method) so that using a `dircmp` instance incurs no unnecessary overhead.

Table 11-27. Attributes supplied by a `dircmp` instance `d`

`common`	Files and subdirectories that are in both *dir1* and *dir2*
`common_dirs`	Subdirectories that are in both *dir1* and *dir2*
`common_files`	Files that are in both *dir1* and *dir2*

`common_funny`	Names that are in both *`dir1`* and *`dir2`* for which `os.stat` reports an error or returns different kinds for the versions in the two directories
`diff_files`	Files that are in both *`dir1`* and *`dir2`* but with different contents
`funny_files`	Files that are in both *`dir1`* and *`dir2`* but could not be compared
`left_list`	Files and subdirectories that are in *`dir1`*
`left_only`	Files and subdirectories that are in *`dir1`* and not in *`dir2`*
`right_list`	Files and subdirectories that are in *`dir2`*
`right_only`	Files and subdirectories that are in *`dir2`* and not in *`dir1`*
`same_files`	Files that are in both *`dir1`* and *`dir2`* with the same contents
`subdirs`	A dictionary whose keys are the strings in `common_dirs`; the corresponding values are instances of `dircmp` (or **3.10+** of the same `dircmp` subclass as *d*) for each subdirectory

The fnmatch Module

The `fnmatch` module (an abbreviation for *filename match*) matches filename strings or paths with patterns that resemble the ones used by Unix shells, as listed in **Table 11-28**.

Table 11-28. `fnmatch` pattern matching conventions

Pattern	Matches
`*`	Any sequence of characters
`?`	Any single character
`[`*`chars`*`]`	Any one of the characters in *`chars`*
`[!`*`chars`*`]`	Any one character not among those in *`chars`*

`fnmatch` does *not* follow other conventions of Unix shell pattern matching, such as treating a slash (`/`) or a leading dot (`.`) specially. It also does not allow escaping special characters: rather, to match a special character, enclose it in brackets. For example, to match a filename that's a single close bracket, use `'[]]'`.

The `fnmatch` module supplies the functions listed in **Table 11-29**.

Table 11-29. Functions of the `fnmatch` module

`filter`	`filter(`*`names`*`, `*`pattern`*`)` Returns the list of items of *`names`* (a sequence of strings) that match *`pattern`*.
`fnmatch`	`fnmatch(`*`filename`*`, `*`pattern`*`)` Returns **`True`** when string *`filename`* matches *`pattern`*; otherwise, returns **`False`**. The match is case sensitive when the platform is (for example, typical Unix-like systems), and otherwise (for example, on Windows) case insensitive; beware of that, if you're dealing with a filesystem whose case-sensitivity doesn't match your platform (for example, macOS is Unix-like; however, its typical filesystems are case insensitive).

`fnmatchcase`	`fnmatchcase(`*`filename`*`, `*`pattern`*`)` Returns **`True`** when string *`filename`* matches *`pattern`*; otherwise, returns **`False`**. The match is always case-sensitive on any platform.
`translate`	`translate(`*`pattern`*`)` Returns the regular expression pattern (as covered in **"Pattern String Syntax"**) equivalent to the `fnmatch` pattern *`pattern`*.

The glob Module

The `glob` module lists (in arbitrary order) the pathnames of files that match a *path pattern*, using the same rules as `fnmatch`; in addition, it treats a leading dot (`.`), separator (`/`), and `**` specially, like Unix shells do. **Table 11-30** lists some useful functions provided by the `glob` module.

Table 11-30. Functions of the `glob` module

`escape`	`escape(`*`pathname`*`)` Escapes all special characters (`'?'`, `'*'`, and `'['`), so you can match an arbitrary literal string that may contain special characters.
`glob`	`glob(`*`pathname`*`, *, root_dir=`**`None`**`, dir_fd=`**`None`**`, recursive=`**`False`**`)` Returns the list of pathnames of files that match the pattern *`pathname`*. `root_dir` (if not **`None`**) is a string or path-like object specifying the root directory for searching (this works like changing the current directory before calling `glob`). If *`pathname`* is relative, the paths returned are relative to `root_dir`. To search paths relative to directory descriptors, pass `dir_fd` instead. Optionally pass named argument `recursive=`**`True`** to

	have path component ** recursively match zero or more levels of subdirectories.
iglob	iglob(*pathname*, *, root_dir=**None**, dir_fd=**None**, recursive=**False**) Like glob, but returns an iterator yielding one relevant pathname at a time.

The shutil Module

The shutil module (an abbreviation for *shell utilities*) supplies functions to copy and move files, and to remove an entire directory tree. On some Unix platforms, most of the functions support the optional keyword-only argument follow_symlinks, defaulting to **True**. When follow_symlinks=**True**, if a path indicates a symbolic link, the function follows it to reach an actual file or directory; when **False**, the function operates on the symbolic link itself. Table 11-31 lists the functions provided by the shutil module.

Table 11-31. Functions of the shutil module

copy	copy(*src*, *dst*) Copies the contents of the file named by *src*, which must exist, and creates or overwrites the file *dst* (*src* and *dst* are strings or instances of pathlib.Path). If *dst* is a directory, the target is a file with the same base name as *src*, but located in *dst*. copy also copies permission bits, but not last access and modification times. Returns the path to the destination file it has copied to.
copy2	copy2(*src*, *dst*) Like copy, but also copies last access time and modification time.
copyfile	copyfile(*src*, *dst*) Copies just the contents (not permission bits, nor last

	access and modification times) of the file named by *src*, creating or overwriting the file named by *dst*.
copyfileobj	`copyfileobj(`*`fsrc`*`, `*`fdst`*`, bufsize=16384)` Copies all bytes from file object *fsrc*, which must be open for reading, to file object *fdst*, which must be open for writing. Copies up to `bufsize` bytes at a time if *bufsize* is greater than `0`. File objects are covered in **“The io Module”**.
copymode	`copymode(`*`src`*`, `*`dst`*`)` Copies permission bits of the file or directory named by *src* to the file or directory named by *dst*. Both *src* and *dst* must exist. Does not change *dst*’s contents, nor its status as being a file or a directory.
copystat	`copystat(`*`src`*`, `*`dst`*`)` Copies permission bits and times of last access and modification of the file or directory named by *src* to the file or directory named by *dst*. Both *src* and *dst* must exist. Does not change *dst*’s contents, nor its status as being a file or a directory.
copytree	`copytree(`*`src`*`, `*`dst`*`, symlinks=`**`False`**`, ignore=`**`None`**`, copy_function=copy2, ignore_dangling_symlinks=`**`False`**`, dirs_exist_ok=`**`False`**`)` Copies the directory tree rooted at the directory named by *src* into the destination directory named by *dst*. *dst* must not already exist: `copytree` creates it (as well as creating any missing parent directories). `copytree` copies each file using the function `copy2`, by default; you can optionally pass a different file-copy function as named argument `copy_function`. If any exceptions occur during the copy process, `copytree` will record them internally and continue, raising `Error` at the end containing the list of all

	the recorded exceptions. When `symlinks` is **`True`**, `copytree` creates symbolic links in the new tree when it finds symbolic links in the source tree. When `symlinks` is **`False`**, `copytree` follows each symbolic link it finds and copies the linked-to file with the link's name, recording an exception if the linked file does not exist (if `ignore_dangling_symlinks=`**`True`**, this exception is ignored). On platforms that do not have the concept of a symbolic link, `copytree` ignores the argument `symlinks`.

`copytree` *(cont.)*	When `ignore` is not **`None`**, it must be a callable accepting two arguments (a directory path and a list of the immediate children of the directory) and returning a list of the children to be ignored in the copy process. If present, `ignore` is often the result of a call to `shutil.ignore_patterns`. For example, this code:
	`import shutil`
	`ignore = shutil.ignore_patterns('.*', '*.bak')`
	`shutil.copytree('src', 'dst', ignore=ignore)`
	copies the tree rooted at directory `src` into a new tree rooted at directory `dst`, ignoring any file or subdirectory whose name starts with a dot and any file or subdirectory whose name ends with *.bak*.
	By default, `copytree` will record a `FileExistsError` exception if a target directory already exists. **3.8+** You can set `dirs_exist_ok` to **`True`** to allow `copytree` to write into existing directories found in the copying process (and potentially overwrite their contents).

`ignore_patterns`	`ignore_patterns(*patterns)` Returns a callable picking out files and subdirectories matching *`patterns`*, like those used in the `fnmatch` module

	(see **"The fnmatch Module"**). The result is suitable for passing as the `ignore` argument to the `copytree` function.
`move`	`move(`*`src`*`, `*`dst`*`, copy_function=copy2)` Moves the file or directory named by *`src`* to that named by *`dst`*. `move` first tries using `os.rename`. Then, if that fails (because *`src`* and *`dst`* are on separate filesystems, or because *`dst`* already exists), `move` copies *`src`* to *`dst`* (using `copy2` for a file or `copytree` for a directory by default; you can optionally pass a file-copy function other than `copy2` as the named argument `copy_function`), then removes *`src`* (using `os.unlink` for a file, `rmtree` for a directory).
`rmtree`	`rmtree(`*`path`*`, ignore_errors=`**`False`**`, onerror=`**`None`**`)` Removes the directory tree rooted at *`path`*. When `ignore_errors` is **`True`**, `rmtree` ignores errors. When `ignore_errors` is **`False`** and `onerror` is **`None`**, errors raise exceptions. When `onerror` is not **`None`**, it must be callable with three parameters: *`func`*, *`path`*, and *`ex`*. *`func`* is the function raising the exception (`os.remove` or `os.rmdir`), *`path`* is the path passed to *`func`*, and *`ex`* is the tuple of information `sys.exc_info` returns. When `onerror` raises an exception, `rmtree` terminates, and the exception propagates.

Beyond offering functions that are directly useful, the source file *shutil.py* in the Python stdlib is an excellent example of how to use many of the `os` functions.

Text Input and Output

Python presents non-GUI text input and output streams to Python programs as file objects, so you can use the methods of file objects (covered in **“Attributes and Methods of File Objects”**) to operate on these streams.

Standard Output and Standard Error

The `sys` module (covered in **“The sys Module”**) has the attributes `stdout` and `stderr`, which are writable file objects. Unless you are using shell redirection or pipes, these streams connect to the “terminal” running your script. Nowadays, actual terminals are very rare: a so-called terminal is generally a screen window that supports text I/O.

The distinction between `sys.stdout` and `sys.stderr` is a matter of convention. `sys.stdout`, known as *standard output*, is where your program emits results. `sys.stderr`, known as *standard error*, is where output such as error, status, or progress messages should go. Separating program output from status and error messages helps you use shell redirection effectively. Python respects this convention, using `sys.stderr` for its own errors and warnings.

The print Function

Programs that output results to standard output often need to write to `sys.stdout`. Python’s `print` function (covered in **Table 8-2**) can be a rich, convenient alternative to `sys.stdout.write`. `print` is fine for the informal output used during development to help you debug your code, but for production output, you may need more control of formatting than `print` affords. For example, you may need to control spacing, field widths, the number of decimal places for floating-point values, and so on. If so, you can prepare the output as an f-string (covered in **“String Formatting”**), then output the string, usually with the `write` method of the appropriate file object. (You can pass formatted strings to `print`, but `print` may add spaces and newlines; the `write` method adds nothing at all, so it’s easier for you to control what exactly gets output.)

If you need to direct output to a file *f* that is open for writing, just calling *f*`.write` is often best, while `print(..., file=`*f*`)` is sometimes a handy alternative. To repeatedly direct the output from `print` calls to a certain file, you can temporarily change the value of sys.stdout. The following example is a general-purpose redirection function usable for such a temporary change; in the presence of multitasking, make sure to also add a lock in order to avoid any contention (see also the contextlib.redirect_stdout decorator described in **Table 6-1**):

```
def redirect(func: Callable, *a, **k) -> (str, Any):
    """redirect(func, *a, **k) -> (func's results, return value)
    func is a callable emitting results to standard output.
    redirect captures the results as a str and returns a pair
    (output string, return value).
    """
    import sys, io
    save_out = sys.stdout
    sys.stdout = io.StringIO()
    try:
        retval = func(*args, **kwds)
        return sys.stdout.getvalue(), retval
    finally:
        sys.stdout.close()
        sys.stdout = save_out
```

Standard Input

In addition to stdout and stderr, the sys module provides the stdin attribute, which is a readable file object. When you need a line of text from the user, you can call the built-in function input (covered in **Table 8-2**), optionally with a string argument to use as a prompt.

When the input you need is not a string (for example, when you need a number), use input to obtain a string from the user, then other built-ins, such as int, float, or ast.literal_eval, to turn the string into the number you need. To evaluate an expression or string from an untrusted source, we recommend using the function literal_eval from the standard library module ast (as covered in the **online docs**). ast.literal_eval(*astring*) returns a valid Python value (such as an int, a float, or a list) for the given literal *astring* when it can (3.10+

stripping any leading spaces and tabs from string inputs), or else raises a `SyntaxError` or `ValueError` exception; it never has any side effects. To ensure complete safety, *`astring`* cannot contain any operator or any nonkeyword identifier; however, `+` and `-` may be accepted as positive or negative signs on numbers, rather than as operators. For example:

```
import ast
print(ast.literal_eval('23'))     # prints 23
print(ast.literal_eval('  23'))   # prints 23 (3.10++)
print(ast.literal_eval('[2,-3]')) # prints [2, -3]
print(ast.literal_eval('2+3'))    # raises ValueError
print(ast.literal_eval('2+'))     # raises SyntaxError
```

EVAL CAN BE DANGEROUS

Don't use `eval` on arbitrary, unsanitized user inputs: a nasty (or well-meaning but careless) user can breach security or otherwise cause damage this way. There is no effective defense—just avoid using `eval` (and `exec`) on input from sources you do not fully trust.

The getpass Module

Very occasionally, you may want the user to input a line of text in such a way that somebody looking at the screen cannot see what the user is typing. This may occur when you're asking the user for a password, for example. The `getpass` module provides a function for this, as well as one to get the current user's username (see Table 11-32).

Table 11-32. Functions of the `getpass` module

`getpass`	`getpass(prompt='Password: ')` Like `input` (covered in Table 8-2), except that the text the user inputs is not echoed to the screen as the user is typing, and the default `prompt` is different from `input`'s.

`getuser`	`getuser()` Returns the current user's username. `getuser` tries to get the username as the value of one of the environment variables `LOGNAME`, `USER`, `LNAME`, or `USERNAME`, in that order. If none of these variables are in `os.environ`, `getuser` asks the operating system.

Richer-Text I/O

The text I/O modules covered so far supply basic text I/O functionality on all platform terminals. Most platforms also offer enhanced text I/O features, such as responding to single keypresses (not just entire lines), printing text in any terminal row and column position, and enhancing the text with background and foreground colors and font effects like bold, italic, and underline. For this kind of functionality you'll need to consider a third-party library. We focus here on the `readline` module, then take a quick look at a few console I/O options, including `mscvrt`, with a brief mention of `curses`, `rich`, and `colorama`, which we do not cover further.

The readline Module

The `readline` module wraps the **GNU Readline Library**, which lets the user edit text lines during interactive input and recall previous lines for editing and re-entry. Readline comes preinstalled on many Unix-like platforms, and it's available online. On Windows, you can install and use the third-party module **`pyreadline`**.

When `readline` is available, Python uses it for all line-oriented input, such as `input`. The interactive Python interpreter always tries to load `readline` to enable line editing and recall for interactive sessions. Some `readline` functions control advanced functionality: particularly *history*, for recalling lines entered in previous sessions; and *completion*, for context-sensitive completion of the word being entered. (See the **Python `readline` docs** for complete details on configuration commands.) You can access the module's functionality using the functions in **Table 11-33**.

Table 11-33. Functions of the readline module

add_history	add_history(*s*, /) Adds string *s* as a line at the end of the history buffer. To temporarily disable add_history, call set_auto_history(**False**), which will disable add_history for this session only (it won't persist across sessions); set_auto_history is **True** by default.
append_history_file	append_history_file(*n*, *filename*='~/.history', /) Appends the last *n* items to existing file *filename*.
clear_history	clear_history() Clears the history buffer.
get_completer	get_completer() Returns the current completer function (as last set by set_completer), or **None** if no completer function is set.
get_history_length	get_history_length() Returns the number of lines of history to be saved to the history file. When the result is less than 0, all lines in the history are to be saved.
parse_and_bind	parse_and_bind(*readline_cmd*, /) Gives readline a configuration command. To let the user hit Tab to request completion, call parse_and_bind('tab: complete'). See the **readline documentation** for other useful values of the string *readline_cmd*. A good completion function is in the standard library module rlcompleter. In the interactive interpreter (or in the startup file executed at the

	start of interactive sessions, covered in **"Environment Variables"**), enter:
	`import readline, rlcompleter`
	`readline.parse_and_bind('tab: complete')`
	For the rest of this interactive session, you can hit the Tab key during line editing and get completion for global names and object attributes.

`read_history_file`	`read_history_file(filename='~/.history', /)` Loads history lines from the text file at path *`filename`*.
`read_init_file`	`read_init_file(filename=None, /)` Makes `readline` load a text file: each line is a configuration command. When *`filename`* is **`None`**, loads the same file as the last time.
`set_completer`	`set_completer(func, /)` Sets the completion function. When *`func`* is **`None`** or omitted, `readline` disables completion. Otherwise, when the user types a partial word *`start`*, then presses the Tab key, `readline` calls *`func`*`(`*`start, i`*`)`, with *`i`* initially `0`. *`func`* returns the *`i`*th possible word starting with *`start`*, or **`None`** when there are no more. `readline` loops, calling *`func`* with *`i`* set to `0`, `1`, `2`, etc., until *`func`* returns **`None`**.
`set_history_length`	`set_history_length(x, /)` Sets the number of lines of history that are to be saved to the history file. When *`x`* is less than `0`, all lines in the history are to be saved.

`write_history_file`	`write_history_file(filename='~/.history')` Saves history lines to the text file whose name or path is *`filename`*, overwriting any existing file.

Console I/O

As mentioned previously, “terminals” today are usually text windows on a graphical screen. You may also, in theory, use a true terminal, or (perhaps a tad less theoretically, but these days not by much) the console (main screen) of a personal computer in text mode. All such “terminals” in use today offer advanced text I/O functionality, accessed in platform-dependent ways. The low-level `curses` package works on Unix-like platforms. For a cross-platform (Windows, Unix, macOS) solution, you may use the third-party package **`rich`**; in addition to its excellent **online docs**, there are online **tutorials** to help you get started. To output colored text on the terminal, see `colorama`, available on **PyPI**. `msvcrt`, introduced next, provides some low-level (Windows only) functions.

curses

The classic Unix approach to enhanced terminal I/O is named `curses`, for obscure historical reasons.[4] The Python package `curses` lets you exert detailed control if required. We don’t cover `curses` in this book; for more information, see A.M. Kuchling’s and Eric Raymond’s online tutorial **“Curses Programming with Python”**.

The msvcrt module

The Windows-only `msvcrt` module (which you may need to install with `pip`) supplies a few low-level functions that let Python programs access proprietary extras supplied by the Microsoft Visual C++ runtime library *msvcrt.dll*. For example, the functions listed in **Table 11-34** let you read user input character by character rather than reading a full line at a time.

Table 11-34. Some useful functions of the `msvcrt` module

`getch`, `getche`	`getch()`, `getche()` Reads and returns a single-character bytes from keyboard input, and if necessary blocks until one is available (i.e., a key is pressed). `getche` echoes the character to screen (if printable), while `getch` does not. When the user presses a special key (arrows, function keys, etc.), it's seen as two characters: first a `chr(0)` or `chr(224)`, then a second character that, together with the first one, defines the special key the user pressed. This means that the program must call `getch` or `getche` twice to read these key presses. To find out what `getch` returns for any key, run the following small script on a Windows machine: (see code below)
`kbhit`	`kbhit()` Returns **`True`** when a character is available for reading (`getch`, when called, returns immediately); otherwise, returns **`False`** (`getch`, when called, waits).
`ungetch`	`ungetch(`*`c`*`)` "Ungets" character *c*; the next call to `getch` or `getche` returns *c*. It's an error to call `ungetch` twice without intervening calls to `getch` or `getche`.

```
import msvcrt
print("press z to exit, or any other key "
      "to see the key's code:")
while True:
    c = msvcrt.getch()
    if c == b'z':
        break
    print(f'{ord(c)} ({c!r})')
```

Internationalization

Many programs present some information to users as text. Such text should be understandable and acceptable to users in different locales. For example, in some countries and cultures, the date "March 7" can be concisely expressed as "3/7." Elsewhere, "3/7" indicates "July 3," and the string that means "March 7" is "7/3." In Python, such cultural conventions are handled with the help of the standard library module `locale`.

Similarly, a greeting might be expressed in one natural language by the string "Benvenuti," while in another language the string to use is "Welcome." In Python, such translations are handled with the help of the stdlib module `gettext`.

Both kinds of issues are commonly addressed under the umbrella term *internationalization* (often abbreviated *i18n*, as there are 18 letters between *i* and *n* in the full spelling in English)—a misnomer, since the same issues apply not just between nations, but also to different languages or cultures within a single nation.[5]

The locale Module

Python's support for cultural conventions imitates that of C, slightly simplified. A program operates in an environment of cultural conventions known as a *locale*. The locale setting permeates the program and is typically set at program startup. The locale is not thread-specific, and the `locale` module is not thread-safe. In a multithreaded program, set the program's locale in the main thread; i.e., set it before starting secondary threads.

LIMITATIONS OF LOCALE

`locale` is only useful for process-wide settings. If your application needs to handle multiple locales at the same time in a single process—whether in threads or asynchronously—`locale` is not the answer due to its process-wide nature. Consider, instead, alternatives such as **PyICU**, mentioned in **"More Internationalization Resources"**.

If a program does not call `locale.setlocale`, the *C locale* (so called due to Python's C language roots) is used; it's similar, but not identical, to the US English locale. Alternatively, a program can find out and accept the user's default locale. In this case, the `locale` module interacts with the operating system (via the environment or in other system-dependent ways) to try to find the user's preferred locale. Finally, a program can set a specific locale, presumably determining which locale to set on the basis of user interaction or via persistent configuration settings.

Locale setting is normally performed across the board for all relevant categories of cultural conventions. This common wide-spectrum setting is denoted by the constant attribute `LC_ALL` of the `locale` module. However, the cultural conventions handled by `locale` are grouped into categories, and, in some rare cases, a program can choose to mix and match categories to build up a synthetic composite locale. The categories are identified by the attributes listed in **Table 11-35**.

Table 11-35. Constant attributes of the `locale` module

`LC_COLLATE`	String sorting; affects functions `strcoll` and `strxfrm` in `locale`
`LC_CTYPE`	Character types; affects aspects of module `string` (and string methods) that have to do with lowercase and uppercase letters
`LC_MESSAGES`	Messages; may affect messages displayed by the operating system (for example, messages displayed by function `os.strerror` and module `gettext`)

`LC_MONETARY`	Formatting of currency values; affects functions `localeconv` and `currency` in `locale`
`LC_NUMERIC`	Formatting of numbers; affects functions `atoi`, `atof`, `format_string`, `localeconv`, and `str` in `locale`, as well as the number separators used in format strings (e.g., f-strings and str.format) when format character `'n'` is used
`LC_TIME`	Formatting of times and dates; affects the function `time.strftime`

The settings of some categories (denoted by `LC_CTYPE`, `LC_MESSAGES`, and `LC_TIME`) affect behavior in other modules (`string`, `os`, `gettext`, and `time`, as indicated). Other categories (denoted by `LC_COLLATE`, `LC_MONETARY`, and `LC_NUMERIC`) affect only some functions of `locale` itself (plus string formatting in the case of `LC_NUMERIC`).

The `locale` module supplies the functions listed in **Table 11-36** to query, change, and manipulate locales, as well as functions that implement the cultural conventions of locale categories `LC_COLLATE`, `LC_MONETARY`, and `LC_NUMERIC`.

Table 11-36. Useful functions of the `locale` module

`atof`	`atof(`*`s`*`)` Parses the string *`s`* into a floating-point number using the current `LC_NUMERIC` setting.
`atoi`	`atoi(`*`s`*`)` Parses the string *`s`* into an integer number using the current `LC_NUMERIC` setting.
`currency`	`currency(`*`data`*`, grouping=`**`False`**`, international=`**`False`** Returns the string or number *`data`* with a currency symb and, if `grouping` is **`True`**, uses the monetary thousands

separator and grouping. When `international` is **`True`**, us
`int_curr_symbol` and `int_frac_digits`, described later i
this table.

`format_string`	`format_string(`*`fmt`*`,` *`num`*`, grouping=`**`False`**`, monetary=`**`False`**`)` Returns the string obtained by formatting *num* according t the format string *fmt* and the `LC_NUMERIC` or `LC_MONETARY` settings. Except for cultural convention issues, the result i like old-style *fmt % num* string formatting, covered in **"Legacy String Formatting with %"**. If *num* is an instance of a number type and *fmt* is *%d* or *%f*, set `grouping` to **`True`** to group digits in the result string according to the `LC_NUMERIC` setting. If `monetary` is **`True`**, the string is formatted with `mon_decimal_point`, and *grouping* uses `mon_thousands_sep` and `mon_grouping` instead of the ones supplied by `LC_NUMERIC` (see `localeconv` later in this tabl for more information on these). For example:

```
>>> locale.setlocale(locale.LC_NUMERIC,
...                  'en_us')
```

```
'en_us'
```

```
>>> n=1000*1000
>>> locale.format_string('%d', n)
```

```
'1000000'
```

```
>>> locale.setlocale(locale.LC_MONETARY,
```

```
...                     'it_it')

'it_it'

>>> locale.format_string('%f', n)

'1000000.000000'  # uses decimal_point

>>> locale.format_string('%f', n,
...                      monetary=True)

'1000000,000000'  # uses mon_decimal_point

>>> locale.format_string('%0.2f', n,
...                      grouping=True)

'1,000,000.00'    # separators & decimal from
                  # LC_NUMERIC

>>> locale.format_string('%0.2f', n,
...                      grouping=True,
...                      monetary=True)

'1.000.000,00'    # separators & decimal from
                  # LC_MONETARY
```

	In this example, since the numeric locale is set to US English, when the argument `grouping` is **`True`**, `format_string` groups digits by threes with commas and uses a dot (`.`) for the decimal point. However, the moneta locale is set to Italian, so when the argument `monetary` is **`True`**, `format_string` uses a comma (`,`) for the decimal point and `grouping` uses a dot (`.`) for the thousands separator. Usually, the syntaxes for monetary and nonmonetary numbers are equal within any given locale
`getdefaultlocale`	`getdefaultlocale(envvars=('LANGUAGE', 'LC_ALL', 'LC_TYPE', 'LANG'))` Checks the environment variables whose names are specified by `envvars`, in order. The first one found in the environment determines the default locale. `getdefaultlocale` returns a pair of strings (*`lang`*, *`encoding`*) compliant with **RFC 1766** (except for the `'C'` locale), such as (`'en_US'`, `'UTF-8'`). Each item of the pair may be **`None`** if `gedefaultlocale` is unable to discover wh value the item should have.
`getlocale`	`getlocale(category=LC_CTYPE)` Returns a pair of strings (*`lang, encoding`*) with the current setting for the given `category`. The category canr be `LC_ALL`.
`localeconv`	`localeconv()` Returns a `dict` *d* with the cultural conventions specified categories `LC_NUMERIC` and `LC_MONETARY` of the current locale. While `LC_NUMERIC` is best used indirectly, via other functions of `locale`, the details of `LC_MONETARY` are accessible only through *d*. Currency formatting is differe for local and international use. For example, the `'$'` symbol is for *local* use only; it is ambiguous in *international* use, since the same symbol is used for many currencies called “dollars” (US, Canadian, Australian, Ho

Kong, etc.). In international use, therefore, the symbol for US currency is the unambiguous string `'USD'`. The functio temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale, or the `LC_MONETARY` locale if the locales are differe and the numeric or monetary strings are non-ASCII. This temporary change affects all threads. The keys into *d* to u for currency formatting are the following strings:

`'currency_symbol'`
: Currency symbol to use locally

`'frac_digits'`
: Number of fractional digits to use locally

`'int_curr_symbol'`
: Currency symbol to use internationally

`'int_frac_digits'`
: Number of fractional digits to use internationally

`'mon_decimal_point'`
: String to use as the "decimal point" (aka *radix poin* for monetary values

`'mon_grouping'`
: List of digit-grouping numbers for monetary value

`'mon_thousands_sep'`
: String to use as digit-groups separator for monetar values

`'negative_sign'`, `'positive_sign'`
: Strings to use as the sign symbol for negative (positive) monetary values

`'n_cs_precedes'`, `'p_cs_precedes'`
: **`True`** when the currency symbol comes before negative (positive) monetary values

`'n_sep_by_space'`, `'p_sep_by_space'`
: **`True`** when a space goes between the sign and negative (positive) monetary values

`'n_sign_posn'`, `'p_sign_posn'`
: See **Table 11-37** for a list of numberic codes for formating negative (positive) monetary values.

`CHAR_MAX`
: Indicates that the current locale does not specify a convention for this formatting

`localeconv` *(cont.)*	*d*`['mon_grouping']` is a list of numbers of digits to group when formatting a monetary value (but take care: in som locales, *d*`['mon_grouping']` may be an empty list). When *d*`['mon_grouping'][-1]` is `0`, there is no further grouping beyond the indicated numbers of digits. When *d*`['mon_grouping'][-1]` is `locale.CHAR_MAX`, grouping continues indefinitely, as if *d*`['mon_grouping'][-2]` were endlessly repeated. `locale.CHAR_MAX` is a constant used a the value for all entries in *d* for which the current locale does not specify any convention.
`localize`	`localize(`*`normstr`*`, grouping=`**`False`**`, monetary=`**`False`**`)` Returns a formatted string following `LC_NUMERIC` (or `LC_MONETARY`, when `monetary` is **`True`**) settings from normalized numeric string *`normstr`*.
`normalize`	`normalize(`*`localename`*`)` Returns a string, suitable as an argument to `setlocale`, that is the normalized form for *`localename`*. When `normalize` cannot normalize the string *`localename`*, it returns *`localename`* unchanged.
`reset` `locale`	`resetlocale(category=LC_ALL)` Sets the locale for `category` to the default given by `getdefaultlocale`.
`setlocale`	`setlocale(`*`category`*`, locale=`**`None`**`)` Sets the locale for *`category`* to `locale`, if not **`None`**, and returns the setting (the existing one when `locale` is **`None`**; otherwise, the new one). `locale` can be a string, or a pair `(`*`lang`*`, `*`encoding`*`)`. *`lang`* is normally a language code base on **ISO 639** two-letter codes (`'en'` is English, `'nl'` is Dutcl and so on). When `locale` is the empty string `''`, `setlocal`

		sets the user's default locale. To see valid locales, view the `locale.locale_alias` dictionary.
`str`	`str(num)` Like `locale.format_string('%f', num)`.
`strcoll`	`strcoll(str1, str2)` Respecting the `LC_COLLATE` setting, returns `-1` when *str1* comes before *str2* in collation, `1` when *str2* comes before *str1*, and `0` when the two strings are equivalent for collation purposes.
`strxfrm`	`strxfrm(s)` Returns a string *sx* such that Python's built-in comparison of two or more strings so transformed is like calling `locale.strcoll` on the originals. `strxfrm` lets you easily use the `key` argument for sorts and comparisons needing locale-conformant string comparisons. For example,

```
def locale_sort_inplace(list_of_strings):
    list_of_strings.sort(key=locale.strxfrm)
```

Table 11-37. Numeric codes to format monetary values

`0`	The value and the currency symbol are placed inside parentheses
`1`	The sign is placed before the value and the currency symbol
`2`	The sign is placed after the value and the currency symbol

3	The sign is placed immediately before the value
4	The sign is placed immediately after the value

The gettext Module

A key issue in internationalization is the ability to use text in different natural languages, a task known as *localization* (sometimes *l10n*). Python supports localization via the standard library module `gettext`, inspired by GNU `gettext`. The `gettext` module is optionally able to use the latter's infrastructure and APIs, but also offers a simpler, higher-level approach, so you don't need to install or study GNU `gettext` to use Python's `gettext` effectively.

For full coverage of `gettext` from a different perspective, see the **online docs**.

Using gettext for localization

`gettext` does not deal with automatic translation between natural languages. Rather, it helps you extract, organize, and access the text messages that your program uses. Pass each string literal subject to translation, also known as a *message*, to a function named _ (underscore) rather than using it directly. `gettext` normally installs a function named _ in the `builtins` module. To ensure that your program runs with or without `gettext`, conditionally define a do-nothing function, named _, that just returns its argument unchanged. Then you can safely use `_('message')` wherever you would normally use a literal `'message'` that should be translated, if feasible. The following example shows how to start a module for conditional use of `gettext`:

```
try:
    _
except NameError:
    def _(s): return s
```

```
def greet():
    print(_('Hello world'))
```

If some other module has installed `gettext` before you run this example code, the function `greet` outputs a properly localized greeting. Otherwise, `greet` outputs the string `'Hello world'` unchanged.

Edit your source, decorating message literals with the function _. Then use any of various tools to extract messages into a text file (normally named *messages.pot*) and distribute the file to the people who translate messages into the various natural languages your application must support. Python supplies a script *pygettext.py* (in the directory *Tools/i18n* in the Python source distribution) to perform message extraction on your Python sources.

Each translator edits *messages.pot* to produce a text file of translated messages, with extension *.po*. Compile the *.po* files into binary files with extension *.mo*, suitable for fast searching, using any of various tools. Python supplies a script *msgfmt.py* (also in *Tools/i18n*) for this purpose. Finally, install each *.mo* file with a suitable name in a suitable directory.

Conventions about which directories and names are suitable differ among platforms and applications. `gettext`'s default is subdirectory *share/locale/<lang>/LC_MESSAGES/* of directory *sys.prefix*, where <*lang*> is the language's code (two letters). Each file is named <*name*>*.mo*, where <*name*> is the name of your application or package.

Once you have prepared and installed your *.mo* files, you normally execute, at the time your application starts up, some code such as the following:

```
import os, gettext
os.environ.setdefault('LANG', 'en')  # application-default language
gettext.install('your_application_name')
```

This ensures that calls such as `_('message')` return the appropriate translated strings. You can choose different ways to access `gettext` functionality in your program; for example, if you also need to localize C-coded extensions, or to switch between languages during a run. Another important consideration is whether you're localizing a whole application, or just a package that is distributed separately.

Essential gettext functions

`gettext` supplies many functions. The most often used functions are listed in **Table 11-38**; see the **online docs** for a complete list.

Table 11-38. Useful functions of the `gettext` module

`install`	`install(`*`domain`*`, localedir=`**`None`**`, names=`**`None`**`)` Installs in Python's built-in namespace a function named _ to perform translations given in the file *<lang>/LC_MESSAGES/<domain>.mo* in the directory `localedir`, with language code *<lang>* as per `getdefaultlocale`. When `localedir` is **`None`**, `install` uses the directory `os.path.join(sys.prefix, 'share', 'locale')`. When `names` is provided, it must be a sequence containing the names of functions you want to install in the `builtins` namespace in addition to _. Supported names are `'gettext'`, `'lgettext'`, `'lngettext'`, `'ngettext'`, 3.8+ `'npgettext'`, and 3.8+ `'pgettext'`.
`translation`	`translation(`*`domain`*`, localedir=`**`None`**`, languages=`**`None`**`, class_=`**`None`**`, fallback=`**`False`**`)` Searches for a *.mo* file, like the `install` function; if it finds multiple files, `translation` uses later files as fallbacks for earlier ones. Set `fallback` to **`True`** to return a `NullTranslations` instance; otherwise, the function raises `OSError` when it doesn't find any *.mo* file. When `languages` is **`None`**, `translation` looks in the

	environment for the *<lang>* to use, like `install`. It examines, in order, the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG`, and splits the first nonempty one on `':'` to give a list of language names (for example, it splits `'de:en'` into `['de', 'en']`). When not **`None`**, `languages` must be a list of one or more language names (for example, `['de', 'en']`). `translation` uses the first language name in the list for which it finds a *.mo* file.
`translation` *(cont.)*	`translation` returns an instance object of a translation class (by default, `GNUTranslations`; if present, the class's constructor must take a single file object argument) that supplies the methods `gettext` (to translate a `str`) and `install` (to install `gettext` under the name _ in Python's `builtins` namespace). `translation` offers more detailed control than `install`, which is like `translation(`*`domain`*`,` *`localedir`*`).install(`*`unicode`*`)`. With `translation`, you can localize a single package without affecting the built-in namespace, by binding the name _ on a per-module basis—for example, with:

```
_ = translation(domain).ugettext
```

More Internationalization Resources

Internationalization is a very large topic. For a general introduction, see **Wikipedia**. One of the best packages of code and information for internationalization, which the authors happily recommend, is **ICU**, embedding also the Unicode Consortium's Common Locale Data Repository (CLDR) database of locale conventions and code to access the CLDR. To use ICU in Python, install the third-party package **PyICU**.

1 `tell`'s value is opaque for text files, since they contain variable-length characters. For binary files, it's simply a straight byte count.

2 Alas, yes—not `sys.stderr`, as common practice and logic would dictate!

3 Or, even better, the even-higher-level `pathlib` module, covered later in this chapter.

4 "Curses" does describe well the typical utterances of programmers faced with this complicated, low-level approach.

5 I18n includes the process of "localization," or adapting international software to local language and cultural conventions.