tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=- 1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

Return a *file-like object* that can be used as a temporary storage area. The file is created securely, using the same rules as `mkstemp()`. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The resulting object can be used as a context manager (see *Examples*). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *buffering*, *encoding*, *errors* and *newline* are interpreted as for `open()`.

The *dir*, *prefix* and *suffix* parameters have the same meaning and defaults as with `mkstemp()`.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose `file` attribute is the underlying true file object.

The `os.O_TMPFILE` flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

On platforms that are neither Posix nor Cygwin, TemporaryFile is an alias for NamedTemporaryFile.

Raises an *auditing event* `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.5: The `os.O_TMPFILE` flag is now used if available.

Changed in version 3.8: Added *errors* parameter.

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=- 1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*)

This function operates exactly as `TemporaryFile()` does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the `name` attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows). If *delete* is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

On POSIX (only), a process that is terminated abruptly with SIGKILL cannot automatically delete any NamedTemporaryFiles it created.

Raises an *auditing event* `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.8: Added *errors* parameter.

class tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=- 1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

This class operates exactly as `TemporaryFile()` does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with `TemporaryFile()`.

The resulting file has one additional method, `rollover()`, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose `_file` attribute is either an `io.BytesIO` or `io.TextIOWrapper` object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

Changed in version 3.3: the truncate method now accepts a `size` argument.

Changed in version 3.8: Added *errors* parameter.

Changed in version 3.11: Fully implements the *io.BufferedIOBase* and *io.TextIOBase* abstract base classes (depending on whether binary or text *mode* was specified).

class tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*, *ignore_cleanup_errors=False*)

This class securely creates a temporary directory using the same rules as *mkdtemp()*. The resulting object can be used as a context manager (see *Examples*). On completion of the context or destruction of the temporary directory object, the newly created temporary directory and all its contents are removed from the filesystem.

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a context manager, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

The directory can be explicitly cleaned up by calling the `cleanup()` method. If *ignore_cleanup_errors* is true, any unhandled exceptions during explicit or implicit cleanup (such as a *PermissionError* removing open files on Windows) will be ignored, and the remaining removable items deleted on a "best-effort" basis. Otherwise, errors will be raised in whatever context cleanup occurs (the `cleanup()` call, exiting the context manager, when the object is garbage-collected or during interpreter shutdown).

Raises an *auditing event* `tempfile.mkdtemp` with argument `fullpath`.

New in version 3.2.

Changed in version 3.10: Added *ignore_cleanup_errors* parameter.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the *os.O_EXCL* flag for *os.open()*. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike *TemporaryFile()*, the user of *mkstemp()* is responsible for deleting the temporary file when done with it.

If *suffix* is not `None`, the file name will end with that suffix, otherwise there will be no suffix. *mkstemp()* does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is not `None`, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of *gettempprefix()* or *gettempprefixb()*, as appropriate.

If *dir* is not `None`, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If any of *suffix*, *prefix*, and *dir* are not `None`, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass `suffix=b''`.

If *text* is specified and true, the file is opened in text mode. Otherwise, (the default) the file is opened in binary mode.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by *os.open()*) and the absolute pathname of that file, in that order.

Raises an *auditing event* `tempfile.mkstemp` with argument `fullpath`.

Changed in version 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

Changed in version 3.6: The *dir* parameter now accepts a *path-like object*.

tempfile.**mkdtemp**(*suffix=None*, *prefix=None*, *dir=None*)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of *mkdtemp()* is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for *mkstemp()*.

mkdtemp() returns the absolute pathname of the new directory if *dir* is None or is an absolute path. If *dir* is a relative path, *mkdtemp()* returns a relative path on Python 3.11 and lower. However, on 3.12 it will return an absolute path in all situations.

Raises an *auditing event* tempfile.mkdtemp with argument fullpath.

Changed in version 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to None to cause an appropriate default value to be used.

Changed in version 3.6: The *dir* parameter now accepts a *path-like object*.

tempfile.**gettempdir**()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the TMPDIR environment variable.
2. The directory named by the TEMP environment variable.
3. The directory named by the TMP environment variable.
4. A platform-specific location:
   - On Windows, the directories C:\TEMP, C:\TMP, \TEMP, and \TMP, in that order.
   - On all other platforms, the directories /tmp, /var/tmp, and /usr/tmp, in that order.
5. As a last resort, the current working directory.

The result of this search is cached, see the description of *tempdir* below.

Changed in version 3.10: Always returns a str. Previously it would return any *tempdir* value regardless of type so long as it was not None.

tempfile.**gettempdirb**()

Same as *gettempdir()* but the return value is in bytes.

New in version 3.5.

tempfile.**gettempprefix**()

Return the filename prefix used to create temporary files. This does not contain the directory component.

tempfile.**gettempprefixb**()

Same as *gettempprefix()* but the return value is in bytes.

New in version 3.5.

The module uses a global variable to store the name of the directory used for temporary files returned by *gettempdir()*. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a *dir* argument which can be used to specify the directory. This is the recommended approach that does not surprise other unsuspecting code by changing global API behavior.

tempfile.**tempdir**

When set to a value other than None, this variable defines the default value for the *dir* argument to the functions defined in this module, including its type, bytes or str. It cannot be a *path-like object*.

If `tempdir` is `None` (the default) at any call to any of the above functions except *gettempprefix()* it is initialized following the algorithm described in *gettempdir()*.

Note: Beware that if you set `tempdir` to a bytes value, there is a nasty side effect: The global default return type of *mkstemp()* and *mkdtemp()* changes to bytes when no explicit `prefix`, `suffix`, or `dir` arguments of type str are supplied. Please do not write code expecting or depending on this. This awkward behavior is maintained for compatibility with the historical implementation.

11.6.1 Examples

Here are some examples of typical usage of the *tempfile* module:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the *mktemp()* function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to *mktemp()* and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by *mkstemp()* and the other functions described above.

`tempfile.`**`mktemp`**(*suffix=''*, *prefix='tmp'*, *dir=None*)

Deprecated since version 2.3: Use *mkstemp()* instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are similar to those of *mkstemp()*, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

Warning: Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. *mktemp()*

usage can be replaced easily with *NamedTemporaryFile()*, passing it the `delete=False` parameter:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 `glob` — Unix style pathname pattern expansion

Source code: Lib/glob.py

The *glob* module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the *os.scandir()* and *fnmatch.fnmatch()* functions in concert, and not by actually invoking a subshell.

Note that files beginning with a dot (`.`) can only be matched by patterns that also start with a dot, unlike *fnmatch.fnmatch()* or *pathlib.Path.glob()*. (For tilde and shell variable expansion, use *os.path.expanduser()* and *os.path.expandvars()*.)

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

See also:

The *pathlib* module offers high-level path objects.

`glob.`**`glob`**(*pathname*, ***, *root_dir=None*, *dir_fd=None*, *recursive=False*, *include_hidden=False*)

Return a possibly empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell). Whether or not the results are sorted depends on the file system. If a file that satisfies conditions is removed or added during the call of this function, whether a path name for that file be included is unspecified.

If *root_dir* is not `None`, it should be a *path-like object* specifying the root directory for searching. It has the same effect on *glob()* as changing the current directory before calling it. If *pathname* is relative, the result will contain paths relative to *root_dir*.

This function can support *paths relative to directory descriptors* with the *dir_fd* parameter.

If *recursive* is true, the pattern "`**`" will match any files and zero or more directories, subdirectories and symbolic links to directories. If the pattern is followed by an *os.sep* or *os.altsep* then files will not match.

If *include_hidden* is true, "`**`" pattern will match hidden directories.

Raises an *auditing event* `glob.glob` with arguments `pathname`, `recursive`.

Raises an *auditing event* `glob.glob/2` with arguments `pathname`, `recursive`, `root_dir`, `dir_fd`.

Note: Using the "`**`" pattern in large directory trees may consume an inordinate amount of time.

Changed in version 3.5: Support for recursive globs using "`**`".