3.12.3

Quick search

Go

subprocess — Subprocess management

Source code: [Lib/subprocess.py](#)

The [subprocess](#) module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes. This module intends to replace several older modules and functions:

```
os.system
os.spawn*
```

Information about how the [subprocess](#) module can be used to replace these modules and functions can be found in the following sections.

> **See also:** [PEP 324](#) – PEP proposing the subprocess module

[Availability](#): not Emscripten, not WASI.

This module does not work or is not available on WebAssembly platforms `wasm32-emscripten` and `wasm32-wasi`. See [WebAssembly platforms](#) for more information.

Using the subprocess Module

The recommended approach to invoking subprocesses is to use the [run()](#) function for all use cases it can handle. For more advanced use cases, the underlying [Popen](#) interface can be used directly.

subprocess.**run**(*args, *, stdin=None, input=None, stdout=None, stderr=None, capture_output=False, shell=False, cwd=None, timeout=None, check=False, encoding=None, errors=None, text=None, env=None, universal_newlines=None, **other_popen_kwargs*)

> Run the command described by *args*. Wait for command to complete, then return a [CompletedProcess](#) instance.
>
> The arguments shown above are merely the most common ones, described below in [Frequently Used Arguments](#) (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the [Popen](#) constructor - most of the arguments to this function are passed through to that interface. (*timeout*, *input*, *check*, and *capture_output* are not.)
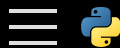
streams into one, set *stdout* to `PIPE` and *stderr* to `STDOUT`, instead of using *capture_output*.

A *timeout* may be specified in seconds, it is internally passed on to `Popen.communicate()`. If the timeout expires, the child process will be killed and waited for. The `TimeoutExpired` exception will be re-raised after the child process has terminated. The initial process creation itself cannot be interrupted on many platform APIs so you are not guaranteed to see a timeout exception until at least after however long process creation takes.

The *input* argument is passed to `Popen.communicate()` and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *text* is true. When used, the internal `Popen` object is automatically created with *stdin* set to `PIPE`, and the *stdin* argument may not be used as well.

If *check* is true, and the process exits with a non-zero exit code, a `CalledProcessError` exception will be raised. Attributes of that exception hold the arguments, the exit code, and stdout and stderr if they were captured.

If *encoding* or *errors* are specified, or *text* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the `io.TextIOWrapper` default. The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. It is passed directly to `Popen`. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like `os.environ` or `os.environb`.

Examples:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

New in version 3.5.
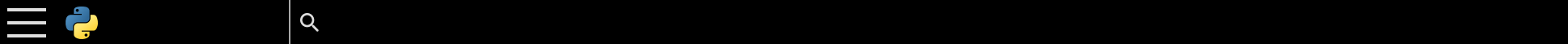
Changed in version 3.7: Added the *text* parameter, as a more understandable alias of *universal_newlines*. Added the *capture_output* parameter.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

class subprocess.**CompletedProcess**

> The return value from `run()`, representing a process that has finished.
>
> **args**
>
>> The arguments used to launch the process. This may be a list or a string.
>
> **returncode**
>
>> Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully.
>>
>> A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).
>
> **stdout**
>
>> Captured stdout from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or text=True. `None` if stdout was not captured.
>>
>> If you ran the process with `stderr=subprocess.STDOUT`, stdout and stderr will be combined in this attribute, and `stderr` will be `None`.
>
> **stderr**
>
>> Captured stderr from the child process. A bytes sequence, or a string if `run()` was called with an encoding, errors, or text=True. `None` if stderr was not captured.
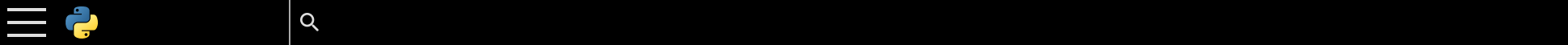>
> **check_returncode()**
>
>> If `returncode` is non-zero, raise a `CalledProcessError`.
>
> *New in version 3.5.*

subprocess.**DEVNULL**

Special value that can be used as the *stdin*, *stdout* or *stderr* argument to `Popen` and indicates that the special file `os.devnull` will be used.

subprocess.**PIPE**

>   Special value that can be used as the *stdin, stdout* or *stderr* argument to Popen and indicates that a pipe to the standard stream should be opened. Most useful with Popen.communicate().

subprocess.**STDOUT**

>   Special value that can be used as the *stderr* argument to Popen and indicates that standard error should go into the same handle as standard output.

exception subprocess.**SubprocessError**

>   Base class for all other exceptions from this module.

> | *New in version 3.3.*

exception subprocess.**TimeoutExpired**

>   Subclass of SubprocessError, raised when a timeout expires while waiting for a child process.

>   **cmd**
>
>   >   Command that was used to spawn the child process.

>   **timeout**
>
>   >   Timeout in seconds.

>   **output**
>
>   >   Output of the child process if it was captured by run() or check_output(). Otherwise, None. This is always bytes when any output was captured regardless of the text=True setting. It may remain None instead of b'' when no output was observed.

>   **stdout**
>
>   >   Alias for output, for symmetry with stderr.

>   **stderr**
>
>   >   Stderr output of the child process if it was captured by run(). Otherwise, None. This is always bytes when stderr output was captured regardless of the text=True setting. It may remain None instead of b'' when no stderr output was observed.
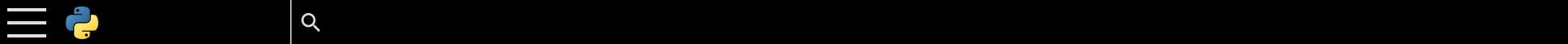
> | *New in version 3.3.*

exception subprocess.**CalledProcessError**

> Subclass of [SubprocessError](), raised when a process run by [check_call()](), [check_output()](), or [run()]() (with `check=True`) returns a non-zero exit status.

> **returncode**
>
> > Exit status of the child process. If the process exited due to a signal, this will be the negative signal number.

> **cmd**
>
> > Command that was used to spawn the child process.

> **output**
>
> > Output of the child process if it was captured by [run()]() or [check_output()](). Otherwise, `None`.

> **stdout**
>
> > Alias for output, for symmetry with [stderr]().

> **stderr**
>
> > Stderr output of the child process if it was captured by [run()](). Otherwise, `None`.

> *Changed in version 3.5:* *stdout* and *stderr* attributes added

Frequently Used Arguments

To support a wide variety of use cases, the [Popen]() constructor (and the convenience functions) accept a large number of optional arguments. For most typical use cases, many of these arguments can be safely left at their default values. The arguments that are most commonly needed are:

> *args* is required for all calls and should be a string, or a sequence of program arguments. Providing a sequence of arguments is generally preferred, as it allows the module to take care of any required escaping and quoting of arguments (e.g. to permit spaces in file names). If passing a single string, either *shell* must be [True]() (see below) or else the string must simply name the program to be executed without specifying any arguments.

> *stdin*, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), and an existing [file object]() with a valid file descriptor. With the default settings of `None`, no redirection will occur. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will
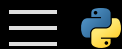
If *encoding* or *errors* are specified, or *text* (also known as *universal_newlines*) is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for `io.TextIOWrapper`.

For *stdin*, line ending characters `'\n'` in the input will be converted to the default line separator `os.linesep`. For *stdout* and *stderr*, all line endings in the output will be converted to `'\n'`. For more information see the documentation of the `io.TextIOWrapper` class when the *newline* argument to its constructor is `None`.

If text mode is not used, *stdin*, *stdout* and *stderr* will be opened as binary streams. No encoding or line ending conversion is performed.

> *Changed in version 3.6:* Added the *encoding* and *errors* parameters.

> *Changed in version 3.7:* Added the *text* parameter as an alias for *universal_newlines*.

> **Note:** The newlines attribute of the file objects `Popen.stdin`, `Popen.stdout` and `Popen.stderr` are not updated by the `Popen.communicate()` method.

If *shell* is `True`, the specified command will be executed through the shell. This can be useful if you are using Python primarily for the enhanced control flow it offers over most system shells and still want convenient access to other shell features such as shell pipes, filename wildcards, environment variable expansion, and expansion of `~` to a user's home directory. However, note that Python itself offers implementations of many shell-like features (in particular, `glob`, `fnmatch`, `os.walk()`, `os.path.expandvars()`, `os.path.expanduser()`, and `shutil`).

> *Changed in version 3.3:* When *universal_newlines* is `True`, the class uses the encoding `locale.getpreferredencoding(False)` instead of `locale.getpreferredencoding()`. See the `io.TextIOWrapper` class for more information on this change.

> **Note:** Read the Security Considerations section before using `shell=True`.

These options, along with all of the other options, are described in more detail in the `Popen` constructor documentation.

Popen Constructor

The underlying process creation and management in this module is handled by the `Popen` class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

restore_signals=True, start_new_session=False, pass_fds=(), *, group=None, extra_groups=None, user=None, umask=-1, encoding=None, errors=None, text=None, pipesize=-1, process_group=None)

Execute a child program in a new process. On POSIX, the class uses `os.execvpe()`-like behavior to execute the child program. On Windows, the class uses the Windows `CreateProcess()` function. The arguments to `Popen` are as follows.

args should be a sequence of program arguments or else a single string or [path-like object](#). By default, the program to execute is the first item in *args* if *args* is a sequence. If *args* is a string, the interpretation is platform-dependent and described below. See the *shell* and *executable* arguments for additional differences from the default behavior. Unless otherwise stated, it is recommended to pass *args* as a sequence.

> **Warning:** For maximum reliability, use a fully qualified path for the executable. To search for an unqualified name on `PATH`, use `shutil.which()`. On all platforms, passing `sys.executable` is the recommended way to launch the current Python interpreter again, and use the `-m` command-line format to launch an installed module.
>
> Resolving the path of *executable* (or the first item of *args*) is platform dependent. For POSIX, see `os.execvpe()`, and note that when resolving or searching for the executable path, *cwd* overrides the current working directory and *env* can override the `PATH` environment variable. For Windows, see the documentation of the `lpApplicationName` and `lpCommandLine` parameters of WinAPI `CreateProcess`, and note that when resolving or searching for the executable path with `shell=False`, *cwd* does not override the current working directory and *env* cannot override the `PATH` environment variable. Using a full path avoids all of these variations.

An example of passing some arguments to an external program as a sequence is:

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

On POSIX, if *args* is a string, the string is interpreted as the name or path of the program to execute. However, this can only be done if not passing arguments to the program.

> **Note:** It may not be obvious how to break a shell command into a sequence of arguments, especially in complex cases. `shlex.split()` can illustrate how to determine the correct tokenization for *args*:
>
> ```
> >>> import shlex, subprocess
> >>> command_line = input()
> /bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
> >>> args = shlex.split(command_line)
> >>> print(args)
> ```
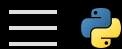
> Note in particular that options (such as *-input*) and arguments (such as *eggs.txt*) that are separated by whitespace in the shell go in separate list elements, while arguments that need quoting or backslash escaping when used in the shell (such as filenames containing spaces or the *echo* command shown above) are single list elements.

On Windows, if *args* is a sequence, it will be converted to a string in a manner described in [Converting an argument sequence to a string on Windows](#). This is because the underlying `CreateProcess()` operates on strings.

> *Changed in version 3.6:* *args* parameter accepts a [path-like object](#) if *shell* is `False` and a sequence containing path-like objects on POSIX.

> *Changed in version 3.8:* *args* parameter accepts a [path-like object](#) if *shell* is `False` and a sequence containing bytes and path-like objects on Windows.

The *shell* argument (which defaults to `False`) specifies whether to use the shell as the program to execute. If *shell* is `True`, it is recommended to pass *args* as a string rather than as a sequence.

On POSIX with `shell=True`, the shell defaults to `/bin/sh`. If *args* is a string, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them. If *args* is a sequence, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, `Popen` does the equivalent of:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

On Windows with `shell=True`, the `COMSPEC` environment variable specifies the default shell. The only time you need to specify `shell=True` on Windows is when the command you wish to execute is built into the shell (e.g. **dir** or **copy**). You do not need `shell=True` to run a batch file or console-based executable.

> **Note:** Read the [Security Considerations](#) section before using `shell=True`.

bufsize will be supplied as the corresponding argument to the `open()` function when creating the stdin/stdout/stderr pipe file objects:

- `0` means unbuffered (read and write are one system call and can return short)
- `1` means line buffered (only usable if `text=True` or `universal_newlines=True`)

Changed in version 3.3.1: bufsize now defaults to -1 to enable buffering by default to match the behavior that most code expects. In versions prior to Python 3.2.4 and 3.3.1 it incorrectly defaulted to `0` which was unbuffered and allowed short reads. This was unintentional and did not match the behavior of Python 2 as most code expected.

The *executable* argument specifies a replacement program to execute. It is very seldom needed. When `shell=False`, *executable* replaces the program to execute specified by *args*. However, the original *args* is still passed to the program. Most programs treat the program specified by *args* as the command name, which can then be different from the program actually executed. On POSIX, the *args* name becomes the display name for the executable in utilities such as **ps**. If `shell=True`, on POSIX the *executable* argument specifies a replacement shell for the default `/bin/sh`.

Changed in version 3.6: executable parameter accepts a [path-like object](#) on POSIX.

Changed in version 3.8: executable parameter accepts a bytes and [path-like object](#) on Windows.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.
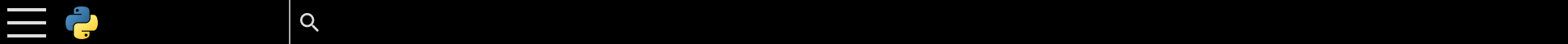
stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `None`, `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), and an existing [file object](#) with a valid file descriptor. With the default settings of `None`, no redirection will occur. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the applications should be captured into the same file handle as for *stdout*.

If *preexec_fn* is set to a callable object, this object will be called in the child process just before the child is executed. (POSIX only)

> **Warning:**   The *preexec_fn* parameter is NOT SAFE to use in the presence of threads in your application. The child process could deadlock before exec is called.

> **Note:**   If you need to modify the environment for the child use the *env* parameter rather than doing it in a *preexec_fn*. The *start_new_session* and *process_group* parameters should take the place of code using *preexec_fn* to call `os.setsid()` or `os.setpgid()` in the child.

Changed in version 3.8: The *preexec_fn* parameter is no longer supported in subinterpreters. The use of the parameter in a subinterpreter raises `RuntimeError`. The new restriction may affect applications that are deployed in mod_wsgi, uWSGI, and other embedded environments.

On Windows, if *close_fds* is true then no handles will be inherited by the child process unless explicitly passed in the `handle_list` element of `STARTUPINFO.lpAttributeList`, or by standard handle redirection.

> *Changed in version 3.2:* The default for *close_fds* was changed from `False` to what is described above.

> *Changed in version 3.7:* On Windows the default for *close_fds* was changed from `False` to `True` when redirecting the standard handles. It's now possible to set *close_fds* to `True` when redirecting the standard handles.

pass_fds is an optional sequence of file descriptors to keep open between the parent and child. Providing any *pass_fds* forces *close_fds* to be `True`. (POSIX only)

> *Changed in version 3.2:* The *pass_fds* parameter was added.

If *cwd* is not `None`, the function changes the working directory to *cwd* before executing the child. *cwd* can be a string, bytes or [path-like](#) object. On POSIX, the function looks for *executable* (or for the first item in *args*) relative to *cwd* if the executable path is a relative path.

> *Changed in version 3.6:* *cwd* parameter accepts a [path-like object](#) on POSIX.

> *Changed in version 3.7:* *cwd* parameter accepts a [path-like object](#) on Windows.

> *Changed in version 3.8:* *cwd* parameter accepts a bytes object on Windows.

If *restore_signals* is true (the default) all signals that Python has set to SIG_IGN are restored to SIG_DFL in the child process before the exec. Currently this includes the SIGPIPE, SIGXFZ and SIGXFSZ signals. (POSIX only)

> *Changed in version 3.2:* *restore_signals* was added.

If *start_new_session* is true the `setsid()` system call will be made in the child process prior to the execution of the subprocess.

[Availability](#): POSIX

> *Changed in version 3.2:* *start_new_session* was added.

If *process_group* is a non-negative integer, the `setpgid(0, value)` system call will be made in the child process prior to the execution of the subprocess.

Changed in version 3.11: `process_group` was added.

If *group* is not `None`, the setregid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via `grp.getgrnam()` and the value in `gr_gid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability: POSIX

New in version 3.9.

If *extra_groups* is not `None`, the setgroups() system call will be made in the child process prior to the execution of the subprocess. Strings provided in *extra_groups* will be looked up via `grp.getgrnam()` and the values in `gr_gid` will be used. Integer values will be passed verbatim. (POSIX only)

Availability: POSIX

New in version 3.9.

If *user* is not `None`, the setreuid() system call will be made in the child process prior to the execution of the subprocess. If the provided value is a string, it will be looked up via `pwd.getpwnam()` and the value in `pw_uid` will be used. If the value is an integer, it will be passed verbatim. (POSIX only)

Availability: POSIX

New in version 3.9.

If *umask* is not negative, the umask() system call will be made in the child process prior to the execution of the subprocess.

Availability: POSIX

New in version 3.9.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of the default behavior of inheriting the current process' environment. This mapping can be str to str on any platform or bytes to bytes on POSIX platforms much like `os.environ` or `os.environb`.

specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, or *text* is true, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified *encoding* and *errors*, as described above in [Frequently Used Arguments](). The *universal_newlines* argument is equivalent to *text* and is provided for backwards compatibility. By default, file objects are opened in binary mode.

> *New in version 3.6:* *encoding* and *errors* were added.

> *New in version 3.7:* *text* was added as a more readable alias for *universal_newlines*.

If given, *startupinfo* will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function.

If given, *creationflags*, can be one or more of the following flags:

- [CREATE_NEW_CONSOLE]()
- [CREATE_NEW_PROCESS_GROUP]()
- [ABOVE_NORMAL_PRIORITY_CLASS]()
- [BELOW_NORMAL_PRIORITY_CLASS]()
- [HIGH_PRIORITY_CLASS]()
- [IDLE_PRIORITY_CLASS]()
- [NORMAL_PRIORITY_CLASS]()
- [REALTIME_PRIORITY_CLASS]()
- [CREATE_NO_WINDOW]()
- [DETACHED_PROCESS]()
- [CREATE_DEFAULT_ERROR_MODE]()
- [CREATE_BREAKAWAY_FROM_JOB]()

pipesize can be used to change the size of the pipe when `PIPE` is used for *stdin*, *stdout* or *stderr*. The size of the pipe is only changed on platforms that support this (only Linux at this time of writing). Other platforms will ignore this parameter.

> *Changed in version 3.10:* Added the *pipesize* parameter.

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.
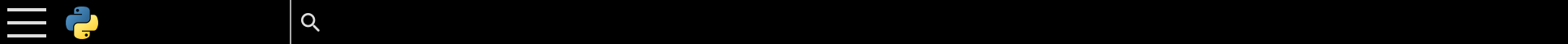
Popen and the other functions in this module that use it raise an [auditing event](#) `subprocess.Popen` with arguments `executable`, `args`, `cwd`, and `env`. The value for `args` may be a single string or a list of strings, depending on platform.

> *Changed in version 3.2:* Added context manager support.

> *Changed in version 3.6:* Popen destructor now emits a `ResourceWarning` warning if the child process is still running.

> *Changed in version 3.8:* Popen can use `os.posix_spawn()` in some cases for better performance. On Windows Subsystem for Linux and QEMU User Emulation, Popen constructor using `os.posix_spawn()` no longer raise an exception on errors like missing program, but the child process fails with a non-zero `returncode`.

Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent.

The most common exception raised is `OSError`. This occurs, for example, when trying to execute a non-existent file. Applications should prepare for `OSError` exceptions. Note that, when `shell=True`, `OSError` will be raised by the child only if the selected shell itself was not found. To determine if the shell failed to find the requested application, it is necessary to check the return code or output from the subprocess.

A `ValueError` will be raised if `Popen` is called with invalid arguments.

`check_call()` and `check_output()` will raise `CalledProcessError` if the called process returns a non-zero return code.

All of the functions and methods that accept a *timeout* parameter, such as `run()` and `Popen.communicate()` will raise `TimeoutExpired` if the timeout expires before the process exits.

Exceptions defined in this module all inherit from `SubprocessError`.

> *New in version 3.3:* The `SubprocessError` base class was added.

Security Considerations

Unlike some other popen functions, this library will not implicitly choose to call a system shell. This means that all characters, including shell metacharacters, can safely be passed to child processes. If the shell is invoked explicitly, via `shell=True`, it is the application's responsibility to ensure that
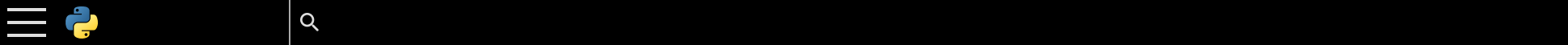
On Windows, batch files (`*.bat` or `*.cmd`) may be launched by the operating system in a system shell regardless of the arguments passed to this library. This could result in arguments being parsed according to shell rules, but without any escaping added by Python. If you are intentionally launching a batch file with arguments from untrusted sources, consider passing `shell=True` to allow Python to escape special characters. See gh-114539 for additional discussion.

Popen Objects

Instances of the Popen class have the following methods:

Popen.**poll()**

> Check if child process has terminated. Set and return `returncode` attribute. Otherwise, returns `None`.

Popen.**wait**(*timeout=None*)

> Wait for child process to terminate. Set and return `returncode` attribute.
>
> If the process does not terminate after *timeout* seconds, raise a `TimeoutExpired` exception. It is safe to catch this exception and retry the wait.
>
> > **Note:** This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `Popen.communicate()` when using pipes to avoid that.
>
> > **Note:** When the `timeout` parameter is not `None`, then (on POSIX) the function is implemented using a busy loop (non-blocking call and short sleeps). Use the `asyncio` module for an asynchronous wait: see `asyncio.create_subprocess_exec`.
>
> > *Changed in version 3.3: timeout was added.*

Popen.**communicate**(*input=None, timeout=None*)

> Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the `returncode` attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.
>
> `communicate()` returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.
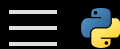
If the process does not terminate after *timeout* seconds, a `TimeoutExpired` exception will be raised. Catching this exception and retrying communication will not lose any output.

The child process is not killed if the timeout expires, so in order to cleanup properly a well-behaved application should kill the child process and finish communication:

```python
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

> **Note:**   The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

Changed in version 3.3: *timeout* was added.

Popen.**send_signal**(*signal*)

Sends the signal *signal* to the child.

Do nothing if the process completed.

> **Note:**   On Windows, SIGTERM is an alias for `terminate()`. CTRL_C_EVENT and CTRL_BREAK_EVENT can be sent to processes started with a *creationflags* parameter which includes `CREATE_NEW_PROCESS_GROUP`.

Popen.**terminate**()

Stop the child. On POSIX OSs the method sends `SIGTERM` to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

Popen.**kill**()

Kills the child. On POSIX OSs the function sends SIGKILL to the child. On Windows `kill()` is an alias for `terminate()`.

The following attributes are also set by the class for you to access. Reassigning them to new values is unsupported:
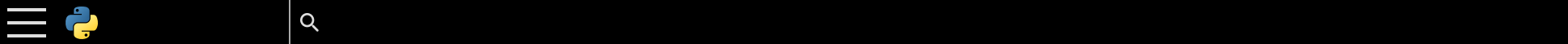
> *New in version 3.3.*

Popen.**stdin**

If the *stdin* argument was `PIPE`, this attribute is a writeable stream object as returned by `open()`. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdin* argument was not `PIPE`, this attribute is `None`.

Popen.**stdout**

If the *stdout* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stdout* argument was not `PIPE`, this attribute is `None`.

Popen.**stderr**

If the *stderr* argument was `PIPE`, this attribute is a readable stream object as returned by `open()`. Reading from the stream provides error output from the child process. If the *encoding* or *errors* arguments were specified or the *text* or *universal_newlines* argument was `True`, the stream is a text stream, otherwise it is a byte stream. If the *stderr* argument was not `PIPE`, this attribute is `None`.

> **Warning:**   Use `communicate()` rather than `.stdin.write`, `.stdout.read` or `.stderr.read` to avoid deadlocks due to any of the other OS pipe buffers filling up and blocking the child process.

Popen.**pid**

The process ID of the child process.

Note that if you set the *shell* argument to `True`, this is the process ID of the spawned shell.

Popen.**returncode**

The child return code. Initially `None`, `returncode` is set by a call to the `poll()`, `wait()`, or `communicate()` methods if they detect that the process has terminated.

A `None` value indicates that the process hadn't yet terminated at the time of the last method call.

A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).
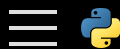
The STARTUPINFO class and following constants are only available on Windows.

class subprocess.**STARTUPINFO**(**, dwFlags=0, hStdInput=None, hStdOutput=None, hStdError=None, wShowWindow=0, LpAttributeList=None*)

> Partial support of the Windows STARTUPINFO structure is used for Popen creation. The following attributes can be set by passing them as keyword-only arguments.

> | *Changed in version 3.7:* Keyword-only argument support was added.

> **dwFlags**

>> A bit field that determines whether certain STARTUPINFO attributes are used when the process creates a window.

>> ```
>> si = subprocess.STARTUPINFO()
>> si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
>> ```

> **hStdInput**

>> If dwFlags specifies STARTF_USESTDHANDLES, this attribute is the standard input handle for the process. If STARTF_USESTDHANDLES is not specified, the default for standard input is the keyboard buffer.

> **hStdOutput**

>> If dwFlags specifies STARTF_USESTDHANDLES, this attribute is the standard output handle for the process. Otherwise, this attribute is ignored and the default for standard output is the console window's buffer.
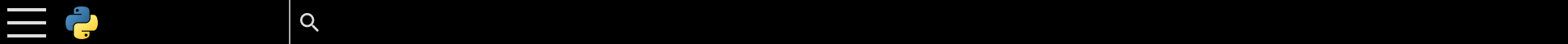
> **hStdError**

>> If dwFlags specifies STARTF_USESTDHANDLES, this attribute is the standard error handle for the process. Otherwise, this attribute is ignored and the default for standard error is the console window's buffer.

> **wShowWindow**

>> If dwFlags specifies STARTF_USESHOWWINDOW, this attribute can be any of the values that can be specified in the nCmdShow parameter for the ShowWindow function, except for SW_SHOWDEFAULT. Otherwise, this attribute is ignored.

>> SW_HIDE is provided for this attribute. It is used when Popen is called with shell=True.

A dictionary of additional attributes for process creation as given in `STARTUPINFOEX`, see `UpdateProcThreadAttribute`.

Supported attributes:

handle_list

Sequence of handles that will be inherited. *close_fds* must be true if non-empty.

The handles must be temporarily made inheritable by `os.set_handle_inheritable()` when passed to the `Popen` constructor, else `OSError` will be raised with Windows error `ERROR_INVALID_PARAMETER` (87).

> **Warning:**   In a multithreaded process, use caution to avoid leaking handles that are marked inheritable when combining this feature with concurrent calls to other process creation functions that inherit all handles such as `os.system()`. This also applies to standard handle redirection, which temporarily creates inheritable handles.

New in version 3.7.

Windows Constants

The `subprocess` module exposes the following constants.

`subprocess.`**`STD_INPUT_HANDLE`**

The standard input device. Initially, this is the console input buffer, `CONIN$`.

`subprocess.`**`STD_OUTPUT_HANDLE`**

The standard output device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.`**`STD_ERROR_HANDLE`**

The standard error device. Initially, this is the active console screen buffer, `CONOUT$`.

`subprocess.`**`SW_HIDE`**

Hides the window. Another window will be activated.

`subprocess.`**`STARTF_USESTDHANDLES`**

Specifies that the `STARTUPINFO.hStdInput`, `STARTUPINFO.hStdOutput`, and `STARTUPINFO.hStdError` attributes contain additional information.

subprocess.**CREATE_NEW_CONSOLE**

    The new process has a new console, instead of inheriting its parent's console (the default).

subprocess.**CREATE_NEW_PROCESS_GROUP**

    A `Popen` `creationflags` parameter to specify that a new process group will be created. This flag is necessary for using `os.kill()` on the subprocess.

    This flag is ignored if `CREATE_NEW_CONSOLE` is specified.

subprocess.**ABOVE_NORMAL_PRIORITY_CLASS**

    A `Popen` `creationflags` parameter to specify that a new process will have an above average priority.

      New in version 3.7.

subprocess.**BELOW_NORMAL_PRIORITY_CLASS**

    A `Popen` `creationflags` parameter to specify that a new process will have a below average priority.

      New in version 3.7.

subprocess.**HIGH_PRIORITY_CLASS**

    A `Popen` `creationflags` parameter to specify that a new process will have a high priority.

      New in version 3.7.

subprocess.**IDLE_PRIORITY_CLASS**

    A `Popen` `creationflags` parameter to specify that a new process will have an idle (lowest) priority.
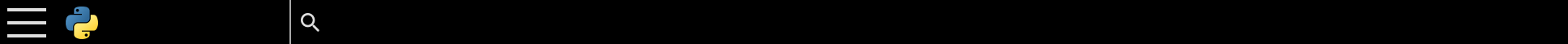
      New in version 3.7.

subprocess.**NORMAL_PRIORITY_CLASS**

    A `Popen` `creationflags` parameter to specify that a new process will have an normal priority. (default)

      New in version 3.7.

subprocess.**REALTIME_PRIORITY_CLASS**

class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

> *New in version 3.7.*

subprocess.**CREATE_NO_WINDOW**

> A `Popen` `creationflags` parameter to specify that a new process will not create a window.

> *New in version 3.7.*

subprocess.**DETACHED_PROCESS**

> A `Popen` `creationflags` parameter to specify that a new process will not inherit its parent's console. This value cannot be used with CREATE_NEW_CONSOLE.

> *New in version 3.7.*

subprocess.**CREATE_DEFAULT_ERROR_MODE**

> A `Popen` `creationflags` parameter to specify that a new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled.

> *New in version 3.7.*

subprocess.**CREATE_BREAKAWAY_FROM_JOB**

> A `Popen` `creationflags` parameter to specify that a new process is not associated with the job.

> *New in version 3.7.*

Older high-level API

Prior to Python 3.5, these three functions comprised the high level API to subprocess. You can now use `run()` in many cases, but lots of existing code calls these functions.

subprocess.**call**(*args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None, **other_popen_kwargs*)

> Run the command described by *args*. Wait for command to complete, then return the `returncode` attribute.

> Code needing to capture stdout or stderr should use `run()` instead:

To suppress stdout or stderr, supply a value of `DEVNULL`.

The arguments shown above are merely some common ones. The full function signature is the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

> **Note:**   Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

> *Changed in version 3.3:* *timeout* was added.

> *Changed in version 3.12:* Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

subprocess.**check_call**(*args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None, **other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute. If `check_call()` was unable to start the process it will propagate the exception that was raised.

Code needing to capture stdout or stderr should use `run()` instead:

```
run(..., check=True)
```

To suppress stdout or stderr, supply a value of `DEVNULL`.

The arguments shown above are merely some common ones. The full function signature is the same as that of the `Popen` constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

> **Note:**   Do not use `stdout=PIPE` or `stderr=PIPE` with this function. The child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from.

> *Changed in version 3.3:* *timeout* was added.

subprocess.**check_output**(*args, *, stdin=None, stderr=None, shell=False, cwd=None, encoding=None, errors=None, universal_newlines=None, timeout=None, text=None, **other_popen_kwargs*)

Run command with arguments and return its output.

If the return code was non-zero it raises a `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute and any output in the `output` attribute.

This is equivalent to:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely some common ones. The full function signature is largely the same as that of `run()` - most arguments are passed directly through to that interface. One API deviation from `run()` behavior exists: passing `input=None` will behave the same as `input=b''` (or `input=''`, depending on other arguments) rather than using the parent's standard input file handle.

By default, this function will return the data as encoded bytes. The actual encoding of the output data may depend on the command being invoked, so the decoding to text will often need to be handled at the application level.

This behaviour may be overridden by setting *text*, *encoding*, *errors*, or *universal_newlines* to `True` as described in [Frequently Used Arguments](#) and `run()`.

To also capture standard error in the result, use `stderr=subprocess.STDOUT`:
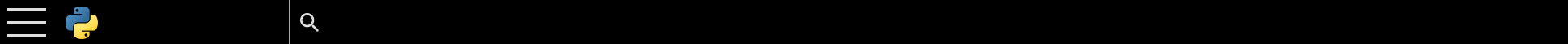
```
>>> subprocess.check_output(
...         "ls non_existent_file; exit 0",
...         stderr=subprocess.STDOUT,
...         shell=True)
'ls: non_existent_file: No such file or directory\n'
```

New in version 3.1.

Changed in version 3.3: *timeout* was added.

Changed in version 3.4: Support for the *input* keyword argument was added.

New in version 3.7: *text* was added as a more readable alias for *universal_newlines*.

Changed in version 3.12: Changed Windows shell search order for `shell=True`. The current directory and `%PATH%` are replaced with `%COMSPEC%` and `%SystemRoot%\System32\cmd.exe`. As a result, dropping a malicious program named `cmd.exe` into a current directory no longer works.

Replacing Older Functions with the [subprocess](#) Module

In this section, "a becomes b" means that b can be used as a replacement for a.

> **Note:** All "a" functions in this section fail (more or less) silently if the executed program cannot be found; the "b" replacements raise `OSError` instead.
>
> In addition, the replacements using `check_output()` will fail with a `CalledProcessError` if the requested operation produces a non-zero return code. The output is still available as the `output` attribute of the raised exception.

In the following examples, we assume that the relevant functions have already been imported from the [subprocess](#) module.

Replacing **/bin/sh** shell command substitution

```
output=$(mycmd myarg)
```
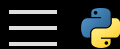
becomes:

```
output = check_output(["mycmd", "myarg"])
```

Replacing shell pipeline

```
output=$(dmesg | grep hda)
```

becomes:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

Alternatively, for trusted input, the shell's own pipeline support may still be used directly:

```
output=$(dmesg | grep hda)
```

becomes:

```
output = check_output("dmesg | grep hda", shell=True)
```

Replacing os.system()

```
sts = os.system("mycmd" + " myarg")
# becomes
retcode = call("mycmd" + " myarg", shell=True)
```

Notes:

- Calling the program through the shell is usually not required.
- The call() return value is encoded differently to that of os.system().
- The os.system() function ignores SIGINT and SIGQUIT signals while the command is running, but the caller must do this separately when using the subprocess module.

A more realistic example would look like this:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

Replacing the os.spawn family

P_NOWAIT example:

```
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT example:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector example:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment example:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

Replacing os.popen(), os.popen2(), os.popen3()

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

Replacing functions from the popen2 module

> **Note:**   If the cmd argument to popen2 functions is a string, the command is executed through /bin/sh. If it is a list, the command is directly executed.

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

popen2.Popen3 and popen2.Popen4 basically work as subprocess.Popen, except that:

- Popen raises an exception if the execution fails.
- The *capturestderr* argument is replaced with the *stderr* argument.
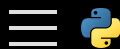
Python versions.

Legacy Shell Invocation Functions

This module also provides the following legacy functions from the 2.x `commands` module. These operations implicitly invoke the system shell and none of the guarantees described above regarding security and exception handling consistency are valid for these functions.

subprocess.**getstatusoutput**(*cmd, *, encoding=None, errors=None*)

> Return `(exitcode, output)` of executing *cmd* in a shell.
>
> Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(exitcode, output)`. *encoding* and *errors* are used to decode output; see the notes on [Frequently Used Arguments](#) for more details.
>
> A trailing newline is stripped from the output. The exit code for the command can be interpreted as the return code of subprocess. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

> [Availability](#): Unix, Windows.
>
> > *Changed in version 3.3.4:* Windows support was added.
> >
> > The function now returns (exitcode, output) instead of (status, output) as it did in Python 3.3.3 and earlier. exitcode has the same value as [returncode](#).
>
> > *Changed in version 3.11:* Added the *encoding* and *errors* parameters.

subprocess.**getoutput**(*cmd, *, encoding=None, errors=None*)

> Return output (stdout and stderr) of executing *cmd* in a shell.

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

[Availability](): Unix, Windows.

> *Changed in version 3.3.4:* Windows support added

> *Changed in version 3.11:* Added the *encoding* and *errors* parameters.

Notes

Converting an argument sequence to a string on Windows

On Windows, an *args* sequence is converted to a string that can be parsed using the following rules (which correspond to the rules used by the MS C runtime):

1. Arguments are delimited by white space, which is either a space or a tab.
2. A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
3. A double quotation mark preceded by a backslash is interpreted as a literal double quotation mark.
4. Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
5. If backslashes immediately precede a double quotation mark, every pair of backslashes is interpreted as a literal backslash. If the number of backslashes is odd, the last backslash escapes the next double quotation mark as described in rule 3.

> **See also:**
>
> [shlex]()
>   Module which provides function to parse and escape command lines.

Disabling use of `vfork()` or `posix_spawn()`

On Linux, [subprocess]() defaults to using the `vfork()` system call internally when it is safe to do so rather than `fork()`. This greatly improves performance.

```
subprocess._USE_VFORK = False   # See CPython issue gh-NNNNNN.
```

Setting this has no impact on use of `posix_spawn()` which could use `vfork()` internally within its libc implementation. There is a similar `subprocess._USE_POSIX_SPAWN` attribute if you need to prevent use of that.

```
subprocess._USE_POSIX_SPAWN = False   # See CPython issue gh-NNNNNN.
```

It is safe to set these to false on any Python version. They will have no effect on older versions when unsupported. Do not assume the attributes are available to read. Despite their names, a true value does not indicate that the corresponding function will be used, only that it may be.

Please file issues any time you have to use these private knobs with a way to reproduce the issue you were seeing. Link to that issue from a comment in your code.

> *New in version 3.8:* `_USE_POSIX_SPAWN`

> *New in version 3.11:* `_USE_VFORK`