17.5. `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

17.5.1. 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.

The `run()` function was added in Python 3.5; if you need to retain compatibility with older versions, see the Older high-level API section.

subprocess. **run**(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*)

> 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 - apart from *timeout*, *input* and *check*, all the arguments to this function are passed through to that interface.
>
> This does not capture stdout or stderr by default. To do so, pass `PIPE` for the *stdout* and/or *stderr* arguments.
>
> The *timeout* argument is passed 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 *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 *universal_newlines* is true. When used, the internal `Popen` object is automatically created with `stdin=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 *universal_newlines* 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. Otherwise, file objects are opened in binary mode.
>
> 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"], stdout=subprocess.PIPE)
> 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')
> ```
>
> *New in version 3.5.*
>
> *Changed in version 3.6:* Added *encoding* and *errors* parameters

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 or errors. `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 or errors. `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.

New in version 3.3.

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`.

stdout

Alias for output, for symmetry with `stderr`.

stderr

Stderr output of the child process if it was captured by `run()`. Otherwise, `None`.

New in version 3.3.

Changed in version 3.5: stdout and stderr attributes added

exception subprocess.**CalledProcessError**

Subclass of `SubprocessError`, raised when a process run by `check_call()` or `check_output()` 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

17.5.1.1. 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 `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be `STDOUT`, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

> If *encoding* or *errors* are specified, or *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.

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

> **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.

17.5.1.2. 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.

class `subprocess.` **Popen**(*args, bufsize=-1, executable=None, stdin=None, stdout=None, stderr=None, preexec_fn=None, close_fds=True, shell=False, cwd=None, env=None, universal_newlines=False, startupinfo=None, creationflags=0, restore_signals=True, start_new_session=False, pass_fds=(), *, encoding=None, errors=None*)

> Execute a child program in a new process. On POSIX, the class uses `os.execvp()`-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. 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.

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:** `shlex.split()` can be useful when determining the correct tokenization for *args*, especially in complex cases:
>
> ```
> >>> import shlex, subprocess
> >>> command_line = input()
> /bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
> >>> args = shlex.split(command_line)
> >>> print(args)
> ['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '$MONEY'"]
> >>> p = subprocess.Popen(args) # Success!
> ```
>
> 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.

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 `universal_newlines=True` i.e., in a text mode)
- any other positive value means use a buffer of approximately that size
- negative bufsize (the default) means the system default of io.DEFAULT_BUFFER_SIZE will be used.

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`.

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, `DEVNULL`, an existing file descriptor (a positive integer), an existing file object, and `None`. `PIPE` indicates that a new pipe to the child should be created. `DEVNULL` indicates that the special file `os.devnull` will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. 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. If you must use it, keep it trivial! Minimize the number of libraries you call into.

> **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* parameter can take the place of a previously common use of *preexec_fn* to call os.setsid() in the child.

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. (POSIX only). The default varies by platform: Always true on POSIX. On Windows it is true when *stdin*/*stdout*/*stderr* are `None`, false otherwise. On Windows, if *close_fds* is true then no handles will be inherited by the child process. Note that on Windows, you cannot set *close_fds* to true and also redirect the standard handles by setting *stdin*, *stdout* or *stderr*.

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

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)

New 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 `str` and path-like object. In particular, 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.

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. (POSIX only)

Changed in version 3.2: start_new_session was added.

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.

> **Note:**   If specified, *env* must provide any variables required for the program to execute. On Windows, in order to run a side-by-side assembly the specified *env* **must** include a valid `SystemRoot`.

If *encoding* or *errors* are specified, 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. If *universal_newlines* is `True`, they are opened in text mode with default encoding. Otherwise, they are opened as binary streams.

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

If given, *startupinfo* will be a `STARTUPINFO` object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be `CREATE_NEW_CONSOLE` or `CREATE_NEW_PROCESS_GROUP`. (Windows only)

Popen objects are supported as context managers via the `with` statement: on exit, standard file descriptors are closed, and the process is waited for.

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

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.

17.5.1.3. Exceptions

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called `child_traceback`, which is a string containing traceback information from the child's point of view.

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.

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 `call()` 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.

17.5.2. Security Considerations

Unlike some other popen functions, this implementation will never implicitly 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 all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities.

When using `shell=True`, the `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

17.5.3. Popen Objects

Instances of the `Popen` class have the following methods:

Popen.**poll**()
> Check if child process has terminated. Set and return `returncode` attribute.

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:** 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.
>
> *Deprecated since version 3.4:* Do not use the *endtime* parameter. It is was unintentionally exposed in 3.3 but was left undocumented as it was intended to be private for internal use. Use *timeout* instead.

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. 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.
>
> Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.
>
> 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:

```
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.
>
> > **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 available:

Popen.**args**

>   The *args* argument as it was passed to `Popen` – a sequence of program arguments or else a single string.
>
>   *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 *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 *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 *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, set by `poll()` and `wait()` (and indirectly by `communicate()`). A `None` value indicates that the process hasn't terminated yet.
>
>   A negative value `-N` indicates that the child was terminated by signal `N` (POSIX only).

17.5.4. Windows Popen Helpers

The `STARTUPINFO` class and following constants are only available on Windows.

class subprocess.**STARTUPINFO**

>   Partial support of the Windows STARTUPINFO structure is used for `Popen` creation.
>
>   **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`.

17.5.4.1. 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.**STARTF_USESHOWWINDOW**

>   Specifies that the STARTUPINFO.wShowWindow attribute contains 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.

17.5.5. 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*, *timeout=None*)

>   Run the command described by *args*. Wait for command to complete, then return the returncode attribute.
>
>   This is equivalent to:

```
run(...).returncode
```

>   (except that the *input* and *check* parameters are not supported)
>
>   The arguments shown above are merely the most common ones. The full function signature is largely 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_call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *timeout=None*)

>   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.
>
>   This is equivalent to:

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

>   (except that the *input* parameter is not supported)
>
>   The arguments shown above are merely the most common ones. The full function signature is largely 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*, *encoding=None*, *errors=None*, *universal_newlines=False*, *timeout=None*)

>   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 the most common ones. The full function signature is largely the same as that of `run()` - most arguments are passed directly through to that interface. However, explicitly passing `input=None` to inherit the parent's standard input file handle is not supported.

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 *universal_newlines* to `True` as described above in Frequently Used Arguments.

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.

17.5.6. 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.

17.5.6.1. Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

becomes:

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

17.5.6.2. 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]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

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)
```

17.5.6.3. Replacing `os.system()`

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

Notes:

- Calling the program through the shell is usually not required.

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)
```

17.5.6.4. Replacing the `os.spawn` family

P_NOWAIT example:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
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"})
```

17.5.6.5. 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)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
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")
```

17.5.6.6. 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.
- `stdin=PIPE` and `stdout=PIPE` must be specified.
- popen2 closes all file descriptors by default, but you have to specify `close_fds=True` with `Popen` to guarantee this behavior on all platforms or past Python versions.

17.5.7. 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*)

Return `(status, output)` of executing *cmd* in a shell.

Execute the string *cmd* in a shell with `Popen.check_output()` and return a 2-tuple `(status, output)`. The locale encoding is used; see the notes on Frequently Used Arguments for more details.

A trailing newline is stripped from the output. The exit status for the command can be interpreted according to the rules for the C function `wait()`. Example:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(256, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(256, 'sh: /bin/junk: not found')
```

Availability: POSIX & Windows

Changed in version 3.3.4: Windows support added

subprocess.**getoutput**(*cmd*)

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

Like `getstatusoutput()`, except the exit status is ignored and the return value is a string containing the command's output. Example:

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

Availability: POSIX & Windows

Changed in version 3.3.4: Windows support added

17.5.8. Notes

17.5.8.1. 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.