18.1. `subprocess` — Subprocess management¶

New in version 2.4.

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 other, older modules and functions, such as:

```
os.system
os.spawn*
os.popen*
popen2.*
commands.*
```

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

18.1.1. Using the subprocess Module¶

This module defines one class called `Popen`:

class `subprocess.Popen`(*args*, *bufsize=0*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=False*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=False*, *startupinfo=None*, *creationflags=0*)¶

Arguments are:

args should be a string, or a sequence of program arguments. The program to execute is normally the first item in the args sequence or the string if a string is given, but can be explicitly set by using the *executable* argument. When *executable* is given, the first item in the args sequence is still treated by most programs as the command name, which can then be different from the actual executable name. On Unix, it becomes the display name for the executing program in utilities such as **ps**.

On Unix, with *shell=False* (default): In this case, the Popen class uses `os.execvp()` to execute the child program. *args* should normally be a sequence. If a string is specified for *args*, it will be used as the name or path of the program to execute; this will only work if the program is being given no arguments.

Note

`shlex.split()` can be useful when determining the correct tokenization for *args*, especially in complex cases:

```
>>> import shlex, subprocess
>>> command_line = raw_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 Unix, with *shell=True*: If args is a string, it specifies the command string 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: the `Popen` class uses CreateProcess() to execute the child program, which operates on strings. If *args* is a sequence, it will be converted to a string using the `list2cmdline()` method. Please note that not all MS Windows applications interpret the command line the same way: `list2cmdline()` is designed for applications using the same rules as the MS C runtime.

bufsize, if given, has the same meaning as the corresponding argument to the built-in open() function: `0` means unbuffered, `1` means line buffered, any other positive value means use a buffer of (approximately) that size. A negative *bufsize* means to use the system default, which usually means fully buffered. The default value for *bufsize* is `0` (unbuffered).

The *executable* argument specifies the program to execute. It is very seldom needed: Usually, the program to execute is defined by the *args* argument. If `shell=True`, the *executable* argument specifies which shell to use. On Unix, the default shell is `/bin/sh`. On Windows, the default shell is specified by the **COMSPEC** environment variable. The only reason you would need to specify `shell=True` on Windows is where the command you wish to execute is actually built in to the shell, eg `dir`, `copy`. You don't need `shell=True` to run a batch file, nor to run a console-based executable.

stdin, *stdout* and *stderr* specify the executed programs' standard input, standard output and standard error file handles, respectively. Valid values are `PIPE`, 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. With `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. (Unix only)

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. (Unix only). Or, 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*.

If *shell* is `True`, the specified command will be executed through the shell.

If *cwd* is not `None`, the child's current directory will be changed to *cwd* before it is executed. Note that this directory is not considered when searching the executable, so you can't specify the program's path relative to *cwd*.

If *env* is not `None`, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process' environment, which is the default behavior.

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 *universal_newlines* is `True`, the file objects stdout and stderr are opened as text files, but lines may be terminated by any of `'\n'`, the Unix end-of-line convention, `'\r'`, the old Macintosh convention or `'\r\n'`, the Windows convention. All of these external representations are seen as `'\n'` by the Python program.

Note

This feature is only available if Python is built with universal newline support (the default). Also, the newlines attribute of the file objects `stdout`, `stdin` and `stderr` are not updated by the communicate() method.

The *startupinfo* and *creationflags*, if given, will be passed to the underlying CreateProcess() function. They can specify things such as appearance of the main window and priority for the new process. (Windows only)

`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.

`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.

18.1.1.1. Convenience Functions¶

This module also defines two shortcut functions:

`subprocess.call`(**popenargs*, ***kwargs*)¶

Run command with arguments. Wait for command to complete, then return the `returncode` attribute.

The arguments are the same as for the Popen constructor. Example:

```
retcode = call(["ls", "-l"])
```

`subprocess.check_call`(**popenargs*, ***kwargs*)¶

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute.

The arguments are the same as for the Popen constructor. Example:

```
check_call(["ls", "-l"])
```

New in version 2.5.

18.1.1.2. 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 childs 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() will raise `CalledProcessError`, if the called process returns a non-zero return code.

18.1.1.3. Security¶

Unlike some other popen functions, this implementation will never call /bin/sh implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes.

18.1.2. 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()`¶

Wait for child process to terminate. Set and return `returncode` attribute.

Warning

This will deadlock if the child process generates enough output to a stdout or stderr pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use `communicate()` to avoid that.

`Popen.communicate(`*input=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 a string to be sent to the child process, or `None`, if no data should be sent to the child.

`communicate()` returns a tuple `(stdoutdata, stderrdata)`.

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.

Note

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

`Popen.send_signal(`*signal*`)`¶

Sends the signal *signal* to the child.

Note

On Windows only SIGTERM is supported so far. It's an alias for `terminate()`.

New in version 2.6.

`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.

New in version 2.6.

`Popen.kill()`¶

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

New in version 2.6.

The following attributes are also available:

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.stdin`¶

If the *stdin* argument was `PIPE`, this attribute is a file object that provides input to the child process. Otherwise, it is `None`.

`Popen.stdout`¶

If the *stdout* argument was `PIPE`, this attribute is a file object that provides output from the child process. Otherwise, it is `None`.

`Popen.stderr`¶

If the *stderr* argument was `PIPE`, this attribute is a file object that provides error output from the child process. Otherwise, it is `None`.

`Popen.pid`¶

The process ID of the child process.

`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` (Unix only).

18.1.3. Replacing Older Functions with the subprocess Module¶

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

Note

All functions in this section fail (more or less) silently if the executed program cannot be found; this module raises an `OSError` exception.

In the following examples, we assume that the subprocess module is imported with "from subprocess import *".

18.1.3.1. Replacing /bin/sh shell backquote¶

```
output=`mycmd myarg`
==>
output = Popen(["mycmd", "myarg"], stdout=PIPE).communicate()[0]
```

18.1.3.2. Replacing shell pipeline¶

```
output=`dmesg | grep hda`
==>
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
output = p2.communicate()[0]
```

18.1.3.3. Replacing `os.system()`¶

```
sts = os.system("mycmd" + " myarg")
==>
p = Popen("mycmd" + " myarg", shell=True)
sts = os.waitpid(p.pid, 0)[1]
```

Notes:

- Calling the program through the shell is usually not required.
- It's easier to look at the `returncode` attribute than the exit status.

A more realistic example would look like this:

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

18.1.3.4. 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"})
```

18.1.3.5. Replacing `os.popen()`, `os.popen2()`, `os.popen3()`¶

```
pipe = os.popen("cmd", 'r', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdout=PIPE).stdout
```

```
pipe = os.popen("cmd", 'w', bufsize)
==>
pipe = Popen("cmd", shell=True, bufsize=bufsize, stdin=PIPE).stdin
```

```
(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)
```

On Unix, os.popen2, os.popen3 and os.popen4 also accept a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

```
(child_stdin, child_stdout) = os.popen2(["/bin/ls", "-l"], mode,
                                        bufsize)
==>
p = Popen(["/bin/ls", "-l"], bufsize=bufsize, stdin=PIPE, stdout=PIPE)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

Return code handling translates as follows:

```
pipe = os.popen("cmd", 'w')
...
rc = pipe.close()
if rc != None and rc % 256:
   print "There were some errors"
```

```
==>
process = Popen("cmd", 'w', shell=True, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
   print "There were some errors"
```

18.1.3.6. Replacing functions from the popen2 module¶

```
(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)
```

On Unix, popen2 also accepts a sequence as the command to execute, in which case arguments will be passed directly to the program without shell intervention. This usage can be replaced as follows:

```
(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`.

Table Of Contents