17.1. `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 older modules and functions:

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

Information about how this module can be used to replace the older functions can be found in the subprocess-replacements section.

See also: POSIX users (Linux, BSD, etc.) are strongly encouraged to install and use the much more recent subprocess32 module instead of the version included with python 2.7. It is a drop in replacement with better behavior in many situations.

PEP 324 – PEP proposing the subprocess module

17.1.1. Using the `subprocess` Module

The recommended way to launch subprocesses is to use the following convenience functions. For more advanced use cases when these do not meet your needs, use the underlying `Popen` interface.

`subprocess.`**`call`**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*)

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

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature

is the same as that of the `Popen` constructor - this functions passes all supplied arguments directly through to that interface.

Examples:

```
>>> subprocess.call(["ls", "-l"])
0

>>> subprocess.call("exit 1", shell=True)
1
```

Warning: Using `shell=True` can be a security hazard. See the warning under Frequently Used Arguments for details.

Note: Do not use `stdout=PIPE` or `stderr=PIPE` with this function as that can deadlock based on the child process output volume. Use `Popen` with the `communicate()` method when you need pipes.

`subprocess.`**`check_call`**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*)

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.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is the same as that of the `Popen` constructor - this functions passes all supplied arguments directly through to that interface.

Examples:

```
>>> subprocess.check_call(["ls", "-l"])
0

>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
   ...
subprocess.CalledProcessError: Command 'exit 1' returned non-ze
```

New in version 2.5.

Warning: Using `shell=True` can be a security hazard. See the warning under Frequently Used Arguments for details.

Note: Do not use `stdout=PIPE` or `stderr=PIPE` with this function as that can deadlock based on the child process output volume. Use `Popen` with the `communicate()` method when you need pipes.

`subprocess.`**`check_output`**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *universal_newlines=False*)

Run command with arguments and return its output as a byte string.

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.

The arguments shown above are merely the most common ones, described below in Frequently Used Arguments (hence the slightly odd notation in the abbreviated signature). The full function signature is largely the same as that of the `Popen` constructor, except that *stdout* is not permitted as it is used internally. All other supplied arguments are passed directly through to the `Popen` constructor.

Examples:

```
>>> subprocess.check_output(["echo", "Hello World!"])
'Hello World!\n'

>>> subprocess.check_output("exit 1", shell=True)
Traceback (most recent call last):
   ...
subprocess.CalledProcessError: Command 'exit 1' returned non-ze
```

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 2.7.

Warning: Using `shell=True` can be a security hazard. See the warning under Frequently Used Arguments for details.

Note: Do not use `stderr=PIPE` with this function as that can deadlock based on the child process error volume. Use `Popen` with the `communicate()` method when you need a stderr pipe.

`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.

exception `subprocess.`**`CalledProcessError`**

Exception raised when a process run by `check_call()` or `check_output()` returns a non-zero exit status.

`returncode`

Exit status of the child process.

`cmd`

Command that was used to spawn the child process.

`output`

Output of the child process if this exception is raised by `check_output()`. Otherwise, `None`.

17.1.1.1. 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`, 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 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.

When *stdout* or *stderr* are pipes and *universal_newlines* is `True` then all line endings will be converted to `'\n'` as described for the universal newlines `'U'` mode argument to `open()`.

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`).

Warning: Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of `shell=True` is **strongly discouraged** in

cases where the command string is constructed from external input:

```
>>> from subprocess import call
>>> filename = input("What file would you like to disp
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This
```

`shell=False` disables all shell based features, but does not suffer from this vulnerability; see the Note in the `Popen` constructor documentation for helpful hints in getting `shell=False` to work.

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

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

17.1.1.2. 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=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*)

Execute a child program in a new process. On Unix, 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 Unix, 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 = raw_input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "ec
>>> args = shlex.split(command_line)
>>> print args
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.
>>> 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 Unix 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.

Warning: Passing `shell=True` can be a security hazard if combined with untrusted input. See the warning under Frequently Used Arguments for details.

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).

Note: If you experience performance issues, it is recommended that you try to enable buffering by setting *bufsize* to either -1 or a large enough positive value (such as 4096).

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 Unix, the *args* name becomes the display name for the executable in utilities such as **ps**. If `shell=True`, on Unix 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`, 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 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 *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 *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 in universal newlines mode. 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.

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)

17.1.1.3. 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.

17.1.1.4. Security

Unlike some other popen functions, this implementation will never call a system shell implicitly. This means that all characters, including shell metacharacters, can safely be passed to child processes. Obviously, if the shell is invoked explicitly, then it is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately.

17.1.2. 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 when using `stdout=PIPE` and/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 `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, 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*.

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.

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` (Unix only).

17.1.3. 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.S
```

`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`.

17.1.3.1. 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).

  This flag is always set when `Popen` is created with `shell=True`.

`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.

17.1.4. 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.

17.1.4.1. Replacing /bin/sh shell backquote

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

becomes:

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

17.1.4.2. 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)
```

17.1.4.3. Replacing `os.system()`

```
status = os.system("mycmd" + " myarg")
# becomes
status = subprocess.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 >>sys.stderr, "Child was terminated by signal", -retc
    else:
        print >>sys.stderr, "Child returned", retcode
except OSError as e:
    print >>sys.stderr, "Execution failed:", e
```

17.1.4.4. 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"})
```

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

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

```
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=PI
(child_stdin, child_stdout) = (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", shell=True, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print "There were some errors"
```

17.1.4.6. Replacing functions from the `popen2` module

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize,
==>
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"], bu
                                            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`.

17.1.5. Notes

17.1.5.1. 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.