subprocess — Spawning Additional Processes

Purpose: Start and communicate with additional processes.

The subprocess module supports three APIs for working with processes. The run() function, added in Python 3.5, is a high-level API for running a process and optionally collecting its output. The functions call(), check_call(), and check_output() are the former high-level API, carried over from Python 2. They are still supported and widely used in existing programs. The class Popen is a low-level API used to build the other APIs and useful for more complex process interactions. The constructor for Popen takes arguments to set up the new process so the parent can communicate with it via pipes. It provides all of the functionality of the other modules and functions it replaces, and more. The API is consistent for all uses, and many of the extra steps of overhead needed (such as closing extra file descriptors and ensuring the pipes are closed) are “built in” instead of being handled by the application code separately.

The subprocess module is intended to replace functions such as os.system(), os.spawnv(), the variations of popen() in the os and popen2 modules, as well as the commands() module. To make it easier to compare subprocess with those other modules, many of the examples in this section re-create the ones used for os and popen2.

Note

The API for working on Unix and Windows is roughly the same, but the underlying implementation is different because of the difference in process models in the operating systems. All of the examples shown here were tested on Mac OS X. Behavior on a non-Unix OS may vary.

Running External Command

To run an external command without interacting with it in the same way as os.system(), use the run() function.

```
# subprocess_os_system.py

import subprocess

completed = subprocess.run(['ls', '-1'])
print('returncode:', completed.returncode)
```

The command line arguments are passed as a list of strings, which avoids the need for escaping quotes or other special characters that might be interpreted by the shell. run() returns a CompletedProcess instance, with information about the process like the exit code and output.

```
$ python3 subprocess_os_system.py

index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_output_error_trap_output.py
subprocess_os_system.py
subprocess_pipes.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_run_check.py
subprocess_run_output.py
subprocess_run_output_error.py
subprocess_run_output_error_suppress.py
subprocess_run_output_error_trap.py
subprocess_shell_variables.py
subprocess_signal_parent_shell.py
subprocess_signal_setpgrp.py
returncode: 0
```

Setting the `shell` argument to a true value causes `subprocess` to spawn an intermediate shell process which then runs the command. The default is to run the command directly.

```
# subprocess_shell_variables.py

import subprocess

completed = subprocess.run('echo $HOME', shell=True)
print('returncode:', completed.returncode)
```

Using an intermediate shell means that variables, glob patterns, and other special shell features in the command string are processed before the command is run.

```
$ python3 subprocess_shell_variables.py

/Users/dhellmann
returncode: 0
```

Note

Using `run()` without passing check=True is equivalent to using `call()`, which only returned the exit code from the process.

Error Handling

The `returncode` attribute of the `CompletedProcess` is the exit code of the program. The caller is responsible for interpreting it to detect errors. If the check argument to `run()` is True, the exit code is checked and if it indicates an error happened then a `CalledProcessError` exception is raised.

```
# subprocess_run_check.py

import subprocess

try:
    subprocess.run(['false'], check=True)
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
```

The `false` command always exits with a non-zero status code, which `run()` interprets as an error.

```
$ python3 subprocess_run_check.py

ERROR: Command '['false']' returned non-zero exit status 1
```

Note

Passing check=True to `run()` makes it equivalent to using `check_call()`.

Capturing Output

The standard input and output channels for the process started by `run()` are bound to the parent's input and output. That means the calling program cannot capture the output of the command. Pass PIPE for the `stdout` and `stderr` arguments to capture the output for later processing.

```
# subprocess_run_output.py

import subprocess

completed = subprocess.run(
    ['ls', '-1'],
    stdout=subprocess.PIPE,
)
print('returncode:', completed.returncode)
```

```
print('Have {} bytes in stdout:\n{}'.format(
    len(completed.stdout),
    completed.stdout.decode('utf-8'))
)
```

The `ls -1` command runs successfully, so the text it prints to standard output is captured and returned.

```
$ python3 subprocess_run_output.py

returncode: 0
Have 522 bytes in stdout:
index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_output_error_trap_output.py
subprocess_os_system.py
subprocess_pipes.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_run_check.py
subprocess_run_output.py
subprocess_run_output_error.py
subprocess_run_output_error_suppress.py
subprocess_run_output_error_trap.py
subprocess_shell_variables.py
subprocess_signal_parent_shell.py
subprocess_signal_setpgrp.py
```

Note

Passing check=True and setting `stdout` to PIPE is equivalent to using `check_output()`.

The next example runs a series of commands in a sub-shell. Messages are sent to standard output and standard error before the commands exit with an error code.

```
# subprocess_run_output_error.py

import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        check=True,
        shell=True,
        stdout=subprocess.PIPE,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('Have {} bytes in stdout: {!r}'.format(
        len(completed.stdout),
        completed.stdout.decode('utf-8'))
    )
```

The message to standard error is printed to the console, but the message to standard output is hidden.

```
$ python3 subprocess_run_output_error.py

to stderr
ERROR: Command 'echo to stdout; echo to stderr 1>&2; exit 1'
returned non-zero exit status 1
```

To prevent error messages from commands run through `run()` from being written to the console, set the `stderr` parameter to the constant `PIPE`.

```
# subprocess_run_output_error_trap.py

import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('Have {} bytes in stdout: {!r}'.format(
        len(completed.stdout),
        completed.stdout.decode('utf-8'))
    )
    print('Have {} bytes in stderr: {!r}'.format(
        len(completed.stderr),
        completed.stderr.decode('utf-8'))
    )
```

This example does not set `check=True` so the output of the command is captured and printed.

```
$ python3 subprocess_run_output_error_trap.py

returncode: 1
Have 10 bytes in stdout: 'to stdout\n'
Have 10 bytes in stderr: 'to stderr\n'
```

To capture error messages when using `check_output()`, set `stderr` to `STDOUT`, and the messages will be merged with the rest of the output from the command.

```
# subprocess_check_output_error_trap_output.py

import subprocess

try:
    output = subprocess.check_output(
        'echo to stdout; echo to stderr 1>&2',
        shell=True,
        stderr=subprocess.STDOUT,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('Have {} bytes in output: {!r}'.format(
        len(output),
        output.decode('utf-8'))
    )
```

The order of output may vary, depending on how buffering is applied to the standard output stream and how much data is being printed.

```
$ python3 subprocess_check_output_error_trap_output.py

Have 20 bytes in output: 'to stdout\nto stderr\n'
```

Suppressing Output

For cases where the output should not be shown or captured, use `DEVNULL` to suppress an output stream. This example suppresses both the standard output and error streams.

```
# subprocess_run_output_error_suppress.py
```

```
# subprocess_run_output_error_suppress.py

import subprocess

try:
    completed = subprocess.run(
        'echo to stdout; echo to stderr 1>&2; exit 1',
        shell=True,
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL,
    )
except subprocess.CalledProcessError as err:
    print('ERROR:', err)
else:
    print('returncode:', completed.returncode)
    print('stdout is {!r}'.format(completed.stdout))
    print('stderr is {!r}'.format(completed.stderr))
```

The name DEVNULL comes from the Unix special device file, /dev/null, which responds with end-of-file when opened for reading and receives but ignores any amount of input when writing.

```
$ python3 subprocess_run_output_error_suppress.py

returncode: 1
stdout is None
stderr is None
```

Working with Pipes Directly

The functions run(), call(), check_call(), and check_output() are wrappers around the Popen class. Using Popen directly gives more control over how the command is run, and how its input and output streams are processed. For example, by passing different arguments for stdin, stdout, and stderr it is possible to mimic the variations of os.popen().

One-way Communication With a Process

To run a process and read all of its output, set the stdout value to PIPE and call communicate().

```
# subprocess_popen_read.py

import subprocess

print('read:')
proc = subprocess.Popen(
    ['echo', '"to stdout"'],
    stdout=subprocess.PIPE,
)
stdout_value = proc.communicate()[0].decode('utf-8')
print('stdout:', repr(stdout_value))
```

This is similar to the way popen() works, except that the reading is managed internally by the Popen instance.

```
$ python3 subprocess_popen_read.py

read:
stdout: '"to stdout"\n'
```

To set up a pipe to allow the calling program to write data to it, set stdin to PIPE.

```
# subprocess_popen_write.py

import subprocess

print('write:')
proc = subprocess.Popen(
    ['cat', '-'],
    stdin=subprocess.PIPE,
)
proc.communicate('stdin: to stdin\n'.encode('utf-8'))
```

To send data to the standard input channel of the process one time, pass the data to `communicate()`. This is similar to using `popen()` with mode `'w'`.

```
$ python3 -u subprocess_popen_write.py

write:
stdin: to stdin
```

Bi-directional Communication With a Process

To set up the Popen instance for reading and writing at the same time, use a combination of the previous techniques.

```python
# subprocess_popen2.py

import subprocess

print('popen2:')

proc = subprocess.Popen(
    ['cat', '-'],
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value = proc.communicate(msg)[0].decode('utf-8')
print('pass through:', repr(stdout_value))
```

This sets up the pipe to mimic `popen2()`.

```
$ python3 -u subprocess_popen2.py

popen2:
pass through: 'through stdin to stdout'
```

Capturing Error Output

It is also possible watch both of the streams for `stdout` and `stderr`, as with `popen3()`.

```python
# subprocess_popen3.py

import subprocess

print('popen3:')
proc = subprocess.Popen(
    'cat -; echo "to stderr" 1>&2',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.PIPE,
)
msg = 'through stdin to stdout'.encode('utf-8')
stdout_value, stderr_value = proc.communicate(msg)
print('pass through:', repr(stdout_value.decode('utf-8')))
print('stderr      :', repr(stderr_value.decode('utf-8')))
```

Reading from `stderr` works the same as with `stdout`. Passing `PIPE` tells Popen to attach to the channel, and `communicate()` reads all of the data from it before returning.

```
$ python3 -u subprocess_popen3.py

popen3:
pass through: 'through stdin to stdout'
stderr      : 'to stderr\n'
```

Combining Regular and Error Output

To direct the error output from the process to its standard output channel, use `STDOUT` for stderr instead of `PIPE`.

To direct the error output from the process to its standard output channel, use STDOUT for stderr instead of PIPE.

```
# subprocess_popen4.py

import subprocess

print('popen4:')
proc = subprocess.Popen(
    'cat -; echo "to stderr" 1>&2',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
    stderr=subprocess.STDOUT,
)
msg = 'through stdin to stdout\n'.encode('utf-8')
stdout_value, stderr_value = proc.communicate(msg)
print('combined output:', repr(stdout_value.decode('utf-8')))
print('stderr value   :', repr(stderr_value))
```

Combining the output in this way is similar to how popen4() works.

```
$ python3 -u subprocess_popen4.py

popen4:
combined output: 'through stdin to stdout\nto stderr\n'
stderr value   : None
```

Connecting Segments of a Pipe

Multiple commands can be connected into a *pipeline*, similar to the way the Unix shell works, by creating separate Popen instances and chaining their inputs and outputs together. The stdout attribute of one Popen instance is used as the stdin argument for the next in the pipeline, instead of the constant PIPE. The output is read from the stdout handle for the final command in the pipeline.

```
# subprocess_pipes.py

import subprocess

cat = subprocess.Popen(
    ['cat', 'index.rst'],
    stdout=subprocess.PIPE,
)

grep = subprocess.Popen(
    ['grep', '.. literalinclude::'],
    stdin=cat.stdout,
    stdout=subprocess.PIPE,
)

cut = subprocess.Popen(
    ['cut', '-f', '3', '-d:'],
    stdin=grep.stdout,
    stdout=subprocess.PIPE,
)

end_of_pipe = cut.stdout

print('Included files:')
for line in end_of_pipe:
    print(line.decode('utf-8').strip())
```

The example reproduces the command line:

```
$ cat index.rst | grep ".. literalinclude" | cut -f 3 -d:
```

The pipeline reads the reStructuredText source file for this section and finds all of the lines that include other files, then prints the names of the files being included.

```
$ python3 -u subprocess_pipes.py

Included files:
subprocess_os_system.py
subprocess_shell_variables.py
subprocess_run_check.py
subprocess_run_output.py
subprocess_run_output_error.py
subprocess_run_output_error_trap.py
subprocess_check_output_error_trap_output.py
subprocess_run_output_error_suppress.py
subprocess_popen_read.py
subprocess_popen_write.py
subprocess_popen2.py
subprocess_popen3.py
subprocess_popen4.py
subprocess_pipes.py
repeater.py
interaction.py
signal_child.py
signal_parent.py
subprocess_signal_parent_shell.py
subprocess_signal_setpgrp.py
```

Interacting with Another Command

All of the previous examples assume a limited amount of interaction. The `communicate()` method reads all of the output and waits for child process to exit before returning. It is also possible to write to and read from the individual pipe handles used by the Popen instance incrementally, as the program runs. A simple echo program that reads from standard input and writes to standard output illustrates this technique.

The script `repeater.py` is used as the child process in the next example. It reads from `stdin` and writes the values to `stdout`, one line at a time until there is no more input. It also writes a message to `stderr` when it starts and stops, showing the lifetime of the child process.

```
# repeater.py

import sys

sys.stderr.write('repeater.py: starting\n')
sys.stderr.flush()

while True:
    next_line = sys.stdin.readline()
    sys.stderr.flush()
    if not next_line:
        break
    sys.stdout.write(next_line)
    sys.stdout.flush()

sys.stderr.write('repeater.py: exiting\n')
sys.stderr.flush()
```

The next interaction example uses the `stdin` and `stdout` file handles owned by the Popen instance in different ways. In the first example, a sequence of five numbers are written to `stdin` of the process, and after each write the next line of output is read back. In the second example, the same five numbers are written but the output is read all at once using `communicate()`.

```
# interaction.py

import io
import subprocess

print('One line at a time:')
proc = subprocess.Popen(
    'python3 repeater.py',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
stdin = io.TextIOWrapper(
```

```
stdin = io.TextIOWrapper(
    proc.stdin,
    encoding='utf-8',
    line_buffering=True,  # send data on newline
)
stdout = io.TextIOWrapper(
    proc.stdout,
    encoding='utf-8',
)
for i in range(5):
    line = '{}\n'.format(i)
    stdin.write(line)
    output = stdout.readline()
    print(output.rstrip())
remainder = proc.communicate()[0].decode('utf-8')
print(remainder)

print()
print('All output at once:')
proc = subprocess.Popen(
    'python3 repeater.py',
    shell=True,
    stdin=subprocess.PIPE,
    stdout=subprocess.PIPE,
)
stdin = io.TextIOWrapper(
    proc.stdin,
    encoding='utf-8',
)
for i in range(5):
    line = '{}\n'.format(i)
    stdin.write(line)
stdin.flush()

output = proc.communicate()[0].decode('utf-8')
print(output)
```

The "`repeater.py: exiting`" lines come at different points in the output for each loop style.

```
$ python3 -u interaction.py

One line at a time:
repeater.py: starting
0
1
2
3
4
repeater.py: exiting


All output at once:
repeater.py: starting
repeater.py: exiting
0
1
2
3
4
```

Signaling Between Processes

The process management examples for the os module include a demonstration of signaling between processes using `os.fork()` and `os.kill()`. Since each Popen instance provides a *pid* attribute with the process id of the child process, it is possible to do something similar with `subprocess`. The next example combines two scripts. This child process sets up a signal handler for the USR signal.

```
# signal_child.py

import os
```

```
import signal
import time
import sys

pid = os.getpid()
received = False


def signal_usr1(signum, frame):
    "Callback invoked when a signal is received"
    global received
    received = True
    print('CHILD {:>6}: Received USR1'.format(pid))
    sys.stdout.flush()


print('CHILD {:>6}: Setting up signal handler'.format(pid))
sys.stdout.flush()
signal.signal(signal.SIGUSR1, signal_usr1)
print('CHILD {:>6}: Pausing to wait for signal'.format(pid))
sys.stdout.flush()
time.sleep(3)

if not received:
    print('CHILD {:>6}: Never received signal'.format(pid))
```

This script runs as the parent process. It starts `signal_child.py`, then sends the USR1 signal.

```
# signal_parent.py

import os
import signal
import subprocess
import time
import sys

proc = subprocess.Popen(['python3', 'signal_child.py'])
print('PARENT      : Pausing before sending signal...')
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling child')
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)
```

The output is:

```
$ python3 signal_parent.py

PARENT      : Pausing before sending signal...
CHILD  26976: Setting up signal handler
CHILD  26976: Pausing to wait for signal
PARENT      : Signaling child
CHILD  26976: Received USR1
```

Process Groups / Sessions

If the process created by Popen spawns sub-processes, those children will not receive any signals sent to the parent. That means when using the `shell` argument to Popen it will be difficult to cause the command started in the shell to terminate by sending SIGINT or SIGTERM.

```
# subprocess_signal_parent_shell.py

import os
import signal
import subprocess
import tempfile
import time
import sys
```

```
script = '''#!/bin/sh
echo "Shell script in process $$"
set -x
python3 signal_child.py
'''
script_file = tempfile.NamedTemporaryFile('wt')
script_file.write(script)
script_file.flush()

proc = subprocess.Popen(['sh', script_file.name])
print('PARENT      : Pausing before signaling {}...'.format(
    proc.pid))
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling child {}'.format(proc.pid))
sys.stdout.flush()
os.kill(proc.pid, signal.SIGUSR1)
time.sleep(3)
```

The pid used to send the signal does not match the pid of the child of the shell script waiting for the signal, because in this example there are three separate processes interacting:

1. The program `subprocess_signal_parent_shell.py`
2. The shell process running the script created by the main python program
3. The program `signal_child.py`

```
$ python3 subprocess_signal_parent_shell.py

PARENT      : Pausing before signaling 26984...
Shell script in process 26984
+ python3 signal_child.py
CHILD  26985: Setting up signal handler
CHILD  26985: Pausing to wait for signal
PARENT      : Signaling child 26984
CHILD  26985: Never received signal
```

To send signals to descendants without knowing their process id, use a *process group* to associate the children so they can be signaled together. The process group is created with `os.setpgrp()`, which sets process group id to the process id of the current process. All child processes inherit their process group from their parent, and since it should only be set in the shell created by Popen and its descendants, `os.setpgrp()` should not be called in the same process where the Popen is created. Instead, the function is passed to Popen as the `preexec_fn` argument so it is run after the `fork()` inside the new process, before it uses `exec()` to run the shell. To signal the entire process group, use `os.killpg()` with the `pid` value from the Popen instance.

```
# subprocess_signal_setpgrp.py

import os
import signal
import subprocess
import tempfile
import time
import sys


def show_setting_prgrp():
    print('Calling os.setpgrp() from {}'.format(os.getpid()))
    os.setpgrp()
    print('Process group is now {}'.format(os.getpgrp()))
    sys.stdout.flush()


script = '''#!/bin/sh
echo "Shell script in process $$"
set -x
python3 signal_child.py
'''
script_file = tempfile.NamedTemporaryFile('wt')
script_file.write(script)
script_file.flush()

proc = subprocess.Popen(
```

```
proc = subprocess.Popen(
    ['sh', script_file.name],
    preexec_fn=show_setting_prgrp,
)
print('PARENT      : Pausing before signaling {}...'.format(
    proc.pid))
sys.stdout.flush()
time.sleep(1)
print('PARENT      : Signaling process group {}'.format(
    proc.pid))
sys.stdout.flush()
os.killpg(proc.pid, signal.SIGUSR1)
time.sleep(3)
```

The sequence of events is

1. The parent program instantiates Popen.
2. The Popen instance forks a new process.
3. The new process runs `os.setpgrp()`.
4. The new process runs `exec()` to start the shell.
5. The shell runs the shell script.
6. The shell script forks again and that process execs Python.
7. Python runs `signal_child.py`.
8. The parent program signals the process group using the pid of the shell.
9. The shell and Python processes receive the signal.
10. The shell ignores the signal.
11. The Python process running `signal_child.py` invokes the signal handler.

```
$ python3 subprocess_signal_setpgrp.py

Calling os.setpgrp() from 75636
Process group is now 75636
PARENT      : Pausing before signaling 75636...
Shell script in process 75636
+ python3 signal_child.py
CHILD  75637: Setting up signal handler
CHILD  75637: Pausing to wait for signal
PARENT      : Signaling process group 75636
CHILD  75637: Received USR1
```

See also

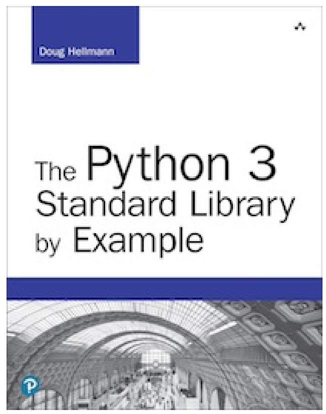
- Standard library documentation for subprocess
- os – Although `subprocess` replaces many of them, the functions for working with processes found in the os module are still widely used in existing code.
- UNIX Signals and Process Groups – A good description of Unix signaling and how process groups work.
- signal – More details about using the `signal` module.
- Advanced Programming in the UNIX(R) Environment – Covers working with multiple processes, such as handling signals, closing duplicated file descriptors, etc.
- `pipes` – Unix shell command pipeline templates in the standard library.

Quick Links

This page was last updated 2018-03-18.

Navigation

Concurrency with Processes, Threads, and Coroutines
signal — Asynchronous System Events



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example