pymotw.com

subprocess – Work with additional processes

18-23 minutes

Purpose:	Spawn and communicate with additional processes.
Available In:	2.4 and later

The `subprocess` module provides a consistent interface to creating and working with additional processes. It offers a higher-level interface than some of the other available modules, and is intended to replace functions such as `os.system()`, `os.spawn*()`, `os.popen*()`, `popen2.*()` and `commands.*()`. To make it easier to compare `subprocess` with those other modules, many of the examples here re-create the ones used for `os` and `popen`.

The `subprocess` module defines one class, `Popen` and a few wrapper functions that use that class. 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.

Note

The API is roughly the same, but the underlying implementation is

slightly different between Unix and Windows. All of the examples shown here were tested on Mac OS X. Behavior on a non-Unix OS will vary.

Running External Command¶

To run an external command without interacting with it, such as one would do with *os.system()*, Use the `call()` function.

```
import subprocess

# Simple command
subprocess.call(['ls', '-1'], shell=True)
```

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.

```
$ python subprocess_os_system.py

__init__.py
index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_call.py
subprocess_check_output.py
subprocess_check_output_error.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_shell_variables.py
```

```
subprocess_signal_parent_shell.py
subprocess_signal_setsid.py
```

Setting the *shell* argument to a true value causes `subprocess` to spawn an intermediate shell process, and tell it to run the command. The default is to run the command directly.

```
import subprocess

# Command with shell expansion
subprocess.call('echo $HOME', shell=True)
```

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.

```
$ python subprocess_shell_variables.py

/Users/dhellmann
```

Error Handling¶

The return value from `call()` is the exit code of the program. The caller is responsible for interpreting it to detect errors. The `check_call()` function works like `call()` except that the exit code is checked, and if it indicates an error happened then a `CalledProcessError` exception is raised.

```
import subprocess

subprocess.check_call(['false'])
```

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

```
$ python subprocess_check_call.py

Traceback (most recent call last):
  File "subprocess_check_call.py", line 11, in
<module>
```

```
    subprocess.check_call(['false'])
  File "/Library/Frameworks/Python.framework
/Versions/2.7/lib/python2.
7/subprocess.py", line 511, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['false']'
returned non-zero e
xit status 1
```

Capturing Output¶

The standard input and output channels for the process started by `call()` are bound to the parent's input and output. That means the calling programm cannot capture the output of the command. Use `check_output()` to capture the output for later processing.

```
import subprocess

output = subprocess.check_output(['ls', '-1'])
print 'Have %d bytes in output' % len(output)
print output
```

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

```
$ python subprocess_check_output.py

Have 462 bytes in output
__init__.py
index.rst
interaction.py
repeater.py
signal_child.py
signal_parent.py
subprocess_check_call.py
subprocess_check_output.py
```

```
subprocess_check_output_error.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_shell_variables.py
subprocess_signal_parent_shell.py
subprocess_signal_setsid.py
```

This script runs a series of commands in a subshell. Messages are sent to standard output and standard error before the commands exit with an error code.

```
import subprocess

output = subprocess.check_output(
    'echo to stdout; echo to stderr 1>&2; exit 1',
    shell=True,
    )
print 'Have %d bytes in output' % len(output)
print output
```

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

```
$ python subprocess_check_output_error.py

to stderr
Traceback (most recent call last):
  File "subprocess_check_output_error.py", line
14, in <module>
    shell=True,
```

```
  File "/Library/Frameworks/Python.framework
/Versions/2.7/lib/python2.
7/subprocess.py", line 544, in check_output
    raise CalledProcessError(retcode, cmd,
output=output)
subprocess.CalledProcessError: 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 `check_output()` from being written to the console, set the *stderr* parameter to the constant `STDOUT`.

```
import subprocess

output = subprocess.check_output(
    'echo to stdout; echo to stderr 1>&2; exit 1',
    shell=True,
    stderr=subprocess.STDOUT,
    )
print 'Have %d bytes in output' % len(output)
print output
```

Now the error and standard output channels are merged together so if the command prints error messages, they are captured and not sent to the console.

```
$ python
subprocess_check_output_error_trap_output.py

Traceback (most recent call last):
  File
"subprocess_check_output_error_trap_output.py",
line 15, in <mo
dule>
    stderr=subprocess.STDOUT,
```

```
  File "/Library/Frameworks/Python.framework
/Versions/2.7/lib/python2.
7/subprocess.py", line 544, in check_output
    raise CalledProcessError(retcode, cmd,
output=output)
subprocess.CalledProcessError: Command 'echo to
stdout; echo to stderr
 1>&2; exit 1' returned non-zero exit status 1
```

Working with Pipes Directly¶

By passing different arguments for *stdin*, *stdout*, and *stderr* it is possible to mimic the variations of `os.popen()`.

popen¶

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

```
import subprocess

print '\nread:'
proc = subprocess.Popen(['echo', '"to stdout"'],
                        stdout=subprocess.PIPE,
                        )
stdout_value = proc.communicate()[0]
print '\tstdout:', repr(stdout_value)
```

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

```
$ python 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`.

```
import subprocess

print '\nwrite:'
proc = subprocess.Popen(['cat', '-'],
                        stdin=subprocess.PIPE,
                        )
proc.communicate('\tstdin: to stdin\n')
```

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'`.

```
$ python -u subprocess_popen_write.py


write:
        stdin: to stdin
```

popen2¶

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

```
import subprocess

print '\npopen2:'

proc = subprocess.Popen(['cat', '-'],
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        )
stdout_value = proc.communicate('through stdin to
stdout')[0]
print '\tpass through:', repr(stdout_value)
```

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

```
$ python -u subprocess_popen2.py
```

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

popen3¶

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

```
import subprocess

print '\npopen3:'
proc = subprocess.Popen('cat -; echo "to stderr"
1>&2',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        )
stdout_value, stderr_value =
proc.communicate('through stdin to stdout')
print '\tpass through:', repr(stdout_value)
print '\tstderr      :', repr(stderr_value)
```

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.

```
$ python -u subprocess_popen3.py


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

popen4¶

To direct the error output from the process to its standard output

channel, use `STDOUT` for *stderr* instead of `PIPE`.

```
import subprocess

print '\npopen4:'
proc = subprocess.Popen('cat -; echo "to stderr"
1>&2',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        stderr=subprocess.STDOUT,
                        )
stdout_value, stderr_value =
proc.communicate('through stdin to stdout\n')
print '\tcombined output:', repr(stdout_value)
print '\tstderr value   :', repr(stderr_value)
```

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

```
$ python -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.

```
import subprocess
```

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

grep = subprocess.Popen(['grep', '.. include::'],
                        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 '\t', line.strip()
```

This example reproduces the command line `cat index.rst | grep ".. include" | cut -f 3 -d:`, which reads the reStructuredText source file for this section and finds all of the lines that include other files, then prints only the filenames.

```
$ python -u subprocess_pipes.py

Included files:
        subprocess_os_system.py
        subprocess_shell_variables.py
        subprocess_check_call.py
        subprocess_check_output.py
        subprocess_check_output_error.py

subprocess_check_output_error_trap_output.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_setsid.py
```

Interacting with Another Command¶

All of the above 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. A simple echo program that reads from standard input and writes to standard output illustrates this:

```
import sys

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

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

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

The script, `repeater.py`, writes to stderr when it starts and stops. That information can be used to show the lifetime of the child process.

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 10 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 10 numbers are written but the output is read all at once using `communicate()`.

```
import subprocess

print 'One line at a time:'
proc = subprocess.Popen('python repeater.py',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        )
for i in range(10):
    proc.stdin.write('%d\n' % i)
    output = proc.stdout.readline()
    print output.rstrip()
remainder = proc.communicate()[0]
print remainder

print
print 'All output at once:'
proc = subprocess.Popen('python repeater.py',
                        shell=True,
                        stdin=subprocess.PIPE,
                        stdout=subprocess.PIPE,
                        )
for i in range(10):
    proc.stdin.write('%d\n' % i)
```

```
output = proc.communicate()[0]
print output
```

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

```
$ python -u interaction.py

One line at a time:
repeater.py: starting
0
1
2
3
4
5
6
7
8
9
repeater.py: exiting


All output at once:
repeater.py: starting
repeater.py: exiting
0
1
2
3
4
5
6
7
8
9
```

Signaling Between Processes¶

The `os` examples 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`. For example, using this script for the child process to be executed by the parent process

```
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 %6s: Received USR1' % pid
    sys.stdout.flush()

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

if not received:
    print 'CHILD %6s: Never received signal' % pid
```

combined with this parent process

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

proc = subprocess.Popen(['python',
'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:

```
$ python signal_parent.py

PARENT      : Pausing before sending signal...
CHILD  14756: Setting up signal handler
CHILD  14756: Pausing to wait for signal
PARENT      : Signaling child
CHILD  14756: Received USR1
```

Process Groups / Sessions¶

Because of the way the process tree works under Unix, if the process created by `Popen` spawns sub-processes, those children will not receive any signals sent to the parent. That means, for example, it will be difficult to cause them to terminate by sending `SIGINT` or `SIGTERM`.

```
import os
import signal
import subprocess
import tempfile
```

```
import time
import sys

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

proc = subprocess.Popen(['sh', script_file.name],
close_fds=True)
print 'PARENT      : Pausing before sending signal
to child %s...' % proc.pid
sys.stdout.flush()
time.sleep(1)
print 'PARENT      : Signaling child %s' %
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. `subprocess_signal_parent_shell.py`
2. The Unix shell process running the script created by the main python program.
3. `signal_child.py`

```
$ python subprocess_signal_parent_shell.py

PARENT      : Pausing before sending signal to
```

```
child 14759...
Shell script in process 14759
+ python signal_child.py
CHILD  14760: Setting up signal handler
CHILD  14760: Pausing to wait for signal
PARENT      : Signaling child 14759
CHILD  14760: Never received signal
```

The solution to this problem is to use a *process group* to associate the children so they can be signaled together. The process group is created with `os.setsid()`, setting the "session id" to the process id of the current process. All child processes inherit the session id, and since it should only be set set in the shell created by `Popen` and its descendants, `os.setsid()` should not be called in the parent process. 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.

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

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

proc = subprocess.Popen(['sh', script_file.name],
                        close_fds=True,
```

```
                           preexec_fn=os.setsid,
                           )
print 'PARENT      : Pausing before sending signal
to child %s...' % proc.pid
sys.stdout.flush()
time.sleep(1)
print 'PARENT      : Signaling process group %s' %
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.setsid()`.
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. The shell ignores it. Python invokes the signal handler.

To signal the entire process group, use `os.killpg()` with the pid value from the `Popen` instance.

```
$ python subprocess_signal_setsid.py

PARENT      : Pausing before sending signal to
child 14763...
Shell script in process 14763
+ python signal_child.py
```

```
CHILD  14764: Setting up signal handler
CHILD  14764: Pausing to wait for signal
PARENT      : Signaling process group 14763
CHILD  14764: Received USR1
```

See also

subprocess
: Standard library documentation for this module.

os
: Although many are deprecated, 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.

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.