Shell Command Documentation

Release 0.1

Nick Coghlan

Nov 05, 2017

Contents

The standard libary's `subprocess` modules provides a convenient way to invoke arbitrary subprocesses. However, it is currently written primarily from an application developer's point of view: it views invocation of the system shell as something risky that leaves you open to shell injection attacks, rather than the normal, unexceptional operation it is when using Python to automate system administration tasks.

This module aims to take over where `subprocess` leaves off, providing convenient, low-level access to the system shell, that automatically handles filenames and paths containing whitespace, as well as protecting naive code from shell injection vulnerabilities.

Significantly, it allows system administrators to divide responsibility appropriately, using Python for its superior data structures and flow control syntax, while using the underlying system shell normally for command invocation and pipeline manipulation.

A couple of basic examples:

```
>>> from shell_command import shell_call
>>> shell_call("ls *.py")
setup.py  shell_command.py  test_shell_command.py
0
>>> shell_call("ls -l *.py")
-rw-r--r-- 1 ncoghlan ncoghlan  391 2011-12-11 12:07 setup.py
-rw-r--r-- 1 ncoghlan ncoghlan 7855 2011-12-11 16:16 shell_command.py
-rwxr-xr-x 1 ncoghlan ncoghlan 8463 2011-12-11 16:17 test_shell_command.py
0
```

Now with some string interpolation (protected from shell injection attacks by default - use `!u` to override):

```
>>> from shell_command import shell_call
>>> shell_call("ls {}", "*.py")
ls: cannot access *.py: No such file or directory
2
>>> shell_call("ls {!u}", "*.py")
setup.py  shell_command.py  test_shell_command.py
0
```

And a slightly more complex example:

```
>>> from shell_command import shell_output, iter_shell_output
>>> print(shell_output("ls -l *.py"))
-rw-r--r-- 1 ncoghlan ncoghlan  391 2011-12-11 12:07 setup.py
-rw-r--r-- 1 ncoghlan ncoghlan 7855 2011-12-11 16:16 shell_command.py
-rwxr-xr-x 1 ncoghlan ncoghlan 8463 2011-12-11 16:17 test_shell_command.py
>>> for line in iter_shell_output("ls -l *.py | tee {}", "example file.txt"):
...     print(line)
...
-rw-r--r-- 1 ncoghlan ncoghlan  391 2011-12-11 12:07 setup.py
-rw-r--r-- 1 ncoghlan ncoghlan 7855 2011-12-11 16:16 shell_command.py
-rwxr-xr-x 1 ncoghlan ncoghlan 8463 2011-12-11 16:17 test_shell_command.py

 >>> print(open("example file.txt").read())
-rw-r--r-- 1 ncoghlan ncoghlan  391 2011-12-11 12:07 setup.py
-rw-r--r-- 1 ncoghlan ncoghlan 7855 2011-12-11 16:16 shell_command.py
-rwxr-xr-x 1 ncoghlan ncoghlan 8463 2011-12-11 16:17 test_shell_command.py
```

CHAPTER 1

String Interpolation

This module uses a custom string interpolation mechanism based on `str.format()`. By default, all interpolated arguments are coerced to strings and quoted to escape any whitespace and shell metacharacters. This quoting can be bypassed with the `!u` conversion specifier as shown in the examples above.

CHAPTER 2

Convenience API

The module level convenience API consists of four functions:

shell_call (**args*, ***kwds*)
A subprocess.call() variant for shell command invocation.

args and *kwds* are both passed though to the string formatting call. Refer to *String Interpolation* for details of the implicit quoting behaviour.

check_shell_call (**args*, ***kwds*)
A subprocess.check_call() variant for shell command invocation.

args and *kwds* are both passed though to the string formatting call. Refer to *String Interpolation* for details of the implicit quoting behaviour.

shell_output (**args*, ***kwds*)
A subprocess.check_output() variant for shell command invocation.

args and *kwds* are both passed though to the string formatting call. Refer to *String Interpolation* for details of the implicit quoting behaviour.

For a successful call, a trailing newline (if any) will be removed from the result.

Use shell redirection (2>&1) to capture stderr in addition to stdout.

As with subprocess.check_output(), this returns encoded bytes by default in Python 3. Passing `universal_newlines=True` in the constructor will also automatically decode the output to text with the UTF-8 codec. Alternatively, the result may be explicitly decoded after the call.

iter_shell_output (**args*, ***kwds*)
An alternative to shell_output() that yields output data as it becomes available.

args and *kwds* are both passed though to the string formatting call. Refer to *String Interpolation* for details of the implicit quoting behaviour.

Since lines are made available as they are produced, the final line will still contain its terminating newline (if any).

This operation relies on the use of `select.select()` on subrocess pipes, and hence is known to fail on Windows.

CHAPTER 3

ShellCommand

The *ShellCommand* class implements the actual functionality of the module. By creating instances of this class directly, it is possible to override the default arguments to subprocess.Popen used by the convenience functions.

class ShellCommand(*command*, ***subprocess_kwds*)

ShellCommand accepts a command string and Popen constructor arguments.

When initialised with an existing ShellCommand object, a new copy is made with the original Popen arguments updated with any new arguments.

Method arguments are interpolated into the command string using str.format() style processing. All method arguments are coerced to strings and escaped using shlex.quote() by default, use the custom conversion specifier "!u" (for "unquoted") or any of the standard conversion specifiers (such as "!s") to bypass this quoting process.

As brace characters ('{' and '}') in the command string are used to indicate interpolated fields, they must either be included in an interpolated value or else doubled (i.e. '{{' and '}}') in the format string in order to be passed to the underlying shell.

The "shell" argument to Popen is enabled be default, but this can be overridden by explicitly setting it to False. In Python 3, the "universal_newlines" option is also enabled by default.

check_shell_call(**args*, ***kwds*)

A subprocess.check_call() variant for shell command invocation.

Refer to ShellCommand for details of the implicit quoting behaviour.

format(**args*, ***kwds*)

A str.format() variant for shell command interpolation.

Refer to ShellCommand for details of the implicit quoting behaviour.

format_map(*mapping*)

A str.format_map() variant for shell command interpolation.

Refer to ShellCommand for details of the implicit quoting behaviour.

iter_shell_output(**args*, ***kwds*)

An alternative to shell_output() that yields output data as it becomes available.

Since lines are made available as they are produced, the final line will still contain its terminating newline (if any).

This operation relies on the use of select.select() on subrocess pipes, and hence is known to fail on Windows.

shell_call (**args*, ***kwds*)
A subprocess.call() variant for shell command invocation.

Refer to ShellCommand for details of the implicit quoting behaviour.

shell_output (**args*, ***kwds*)
A subprocess.check_output() variant for shell command invocation.

Refer to ShellCommand for details of the implicit quoting behaviour.

Use shell redirection (2>&1) to capture stderr in addition to stdout A trailing newline (if any) will be removed from the result

As with subprocess.check_output(), this returns encoded bytes by default in Python 3. Passing "universal_newlines=True" in the constructor will also automatically decode the output to text with the UTF-8 codec. Alternatively, the result may be explicitly decoded after the call.

CHAPTER 4

Obtaining the Module

This module can be installed directly from the Python Package Index with pip:

```
pip install shell_command
```

Alternatively, you can download and unpack it manually from the shell_command PyPI page.

There are no operating system or distribution specific versions of this module - it is a pure Python module that should work on all platforms (aside from :func:`iter_shell_output` being known not to work on Windows).

Supported Python versions are 2.7 and 3.2+.

4.1 Development and Support

Shell Command is developed and maintained on BitBucket. Problems and suggested improvements can be posted to the issue tracker.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index