shell Documentation

Release 1.0.1

Daniel Lindsley

Jun 08, 2017

Contents

“”“A better way to run shell commands in Python.”“”

Built because every time I go to use subprocess, I spend more time in the docs & futzing around than actually implementing what I’m trying to get done.

CHAPTER 1

Philosophy

- Makes running commands more natural
- Assumes you care about the output/errors by default
- Covers the 80% case of running commands
- A nicer API
- Works on Linux/OS X (untested on Windows but might work?)

Contents:

shell Tutorial

If you've ever tried to run a shell command in Python, you're likely unhappy about it. The `subprocess` module, while a huge & consistent step forward over the previous ways Python shelled out, has a rather painful interface. If you're like me, you spent more time in the docs than you did writing working code.

`shell` tries to fix this, by glossing over the warts in the `subprocess` API & making running commands *easy*.

Installation

If you're developing in Python, you ought to be using pip. Installing (from your terminal) looks like:

```
$ pip install shell
```

Quickstart

For the impatient:

```
>>> from shell import shell
>>> ls = shell('ls')
>>> for file in ls.output():
...     print file
'another.txt'

# Or if you need more control, the same code can be stated as...
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('ls')
>>> for file in sh.output():
...     print file
'another.txt'
```

Getting Started

Importing

The first thing you'll need to do is import `shell`. You can either use the easy functional version:

```
>>> from shell import shell
```

Or the class-based & extensible version:

```
>>> from shell import Shell
```

Your First Command

Running a basic command is simple. Simply hand the command you'd use at the terminal off to `shell`:

```
>>> from shell import shell
>>> shell('touch hello_world.txt')

# The class-based variant.
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('touch hello_world.txt')
```

You should now have a `hello_world.txt` file created in your current directory.

Reading Output

By default, `shell` captures output/errors from the command being run. You can read the output & errors like so:

```
>>> from shell import shell
>>> sh = shell('ls /tmp')
# Your output from these calls will vary...
>>> sh.output()
[
    'hello.txt',
    'world.py',
]
>>> sh.errors()
```

```
[]

# The class-based variant.
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('ls /tmp')
>>> sh.output()
[
    'hello.txt',
    'world.py',
]
>>> sh.errors()
[]
```

You can also look at what the process ID was & the return code.:

```
>>> sh.pid
15172
>>> sh.code
0
```

Getting a `0` from `sh.code` means a process finished sucessfully. Higher integer return values generally mean there was an error.

Interactive

If the command is interactive, you can send it input as well.:

```
>>> from shell import shell
>>> sh = shell('cat -u', has_input=True)
>>> sh.write('Hello, world!')
>>> sh.output()
[
    'Hello, world!'
]

# The class-based variant.
>>> from shell import Shell
>>> sh = Shell(has_input=True)
>>> sh.run('cat -u')
>>> sh.write('Hello, world!')
>>> sh.output()
[
    'Hello, world!'
]
```

Warning: You get one shot at sending input, after which the command will finish. Using `shell` for advanced, multi-prompt shell commands is likely is not a good option.

Failing Fast

You can have non-zero exit codes propagate as exceptions:

```
>>> from shell import shell
>>> shell('ls /not/a/real/place', die=True)
    Traceback (most recent call last):
    ...
    shell.CommandError: Command exited with code 1
>>> import sys
>>> from shell import CommandError
>>> try:
>>>     shell('ls /also/definitely/fake', die=True)
>>> except CommandError, e:
>>>     print e.stderr
>>>     sys.exit(e.code)
ls: /also/definitely/fake: No such file or directory
$ echo $?
1
```

Chaining

You can also chain calls together, if that suits you.:

```
>>> from shell import shell
>>> shell('cat -u', has_input=True).write('Hello, world!').output()
[
    'Hello, world!'
]

# The class-based variant.
>>> from shell import Shell
>>> Shell(has_input=True).run('cat -u').write('Hello, world!').output()
[
    'Hello, world!'
]
```

Ignoring Large Output

By default, `shell` captures all output/errors. If you have a command that generates a large volume of output that you don't care about, you can ignore it like so.:

```
>>> from shell import shell
>>> sh = shell('run_intensive_command -v', record_output=False, record_errors=False)
>>> sh.code
0

# The class-based variant.
>>> from shell import Shell
>>> sh = Shell(record_output=False, record_errors=False)
>>> sh.run('run_intensive_command -v')
>>> sh.code
0
```

What Now?

If you need more advanced functionality, subclassing the `Shell` class is the best place to start.

You can find more details about it in the *Shell API*.

Shell API

shell

shell

A better way to run shell commands in Python.

If you just need to quickly run a command, you can use the `shell` shortcut function:

```
>>> from shell import shell
>>> ls = shell('ls')
>>> for file in ls.output():
...     print file
'another.txt'
```

If you need to extend the behavior, you can also use the `Shell` object:

```
>>> from shell import Shell
>>> sh = Shell(has_input=True)
>>> cat = sh.run('cat -u')
>>> cat.write('Hello, world!')
>>> cat.output()
['Hello, world!']
```

exception `shell.`**`CommandError`**(*message*, *code*, *stderr*)
Thrown when a command fails.

exception `shell.`**`MissingCommandException`**
Thrown when no command was setup.

class `shell.`**`Shell`**(*has_input=False*, *record_output=True*, *record_errors=True*, *strip_empty=True*, *die=False*, *verbose=False*)
Handles executing commands & recording output.

Optionally accepts a `has_input` parameter, which should be a boolean. If set to `True`, the command will wait to execute until you call the `Shell.write` method & send input. (Default: `False`)

Optionally accepts a `record_output` parameter, which should be a boolean. If set to `True`, the stdout from the command will be recorded. (Default: `True`)

Optionally accepts a `record_errors` parameter, which should be a boolean. If set to `True`, the stderr from the command will be recorded. (Default: `True`)

Optionally accepts a `strip_empty` parameter, which should be a boolean. If set to `True`, only non-empty lines from `Shell.output` or `Shell.errors` will be returned. (Default: `True`)

Optionally accepts a `die` parameter, which should be a boolean. If set to `True`, raises a CommandError if the command exits with a non-zero return code. (Default: `False`)

Optionally accepts a `verbose` parameter, which should be a boolean. If set to `True`, prints stdout to stdout and stderr to stderr as execution happens. (Default: `False`)

`errors`(*raw=False*)
Returns the errors from running a command.

Optionally accepts a `raw` parameter, which should be a boolean. If `raw` is set to `False`, you get an array of lines of errors. If `raw` is set to `True`, the raw string of errors is returned. (Default: `False`)

Example:

```
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('ls /there-s-no-way-anyone/has/this/directory/please')
>>> sh.errors()
[
    'ls /there-s-no-way-anyone/has/this/directory/please: No such file or␣
↪directory'
]
```

kill()

Kills a given process.

Example:

```
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('some_long_running_thing')
>>> sh.kill()
```

output(*raw=False*)

Returns the output from running a command.

Optionally accepts a `raw` parameter, which should be a boolean. If `raw` is set to `False`, you get an array of lines of output. If `raw` is set to `True`, the raw string of output is returned. (Default: `False`)

Example:

```
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('ls ~')
>>> sh.output()
[
    'hello.txt',
    'world.txt',
]
```

run(*command*)

Runs a given command.

Requires a `command` parameter should be either a string command (easier) or an array of arguments to send as the command (if you know what you're doing).

Returns the `Shell` instance.

Example:

```
>>> from shell import Shell
>>> sh = Shell()
>>> sh.run('ls -alh')
```

write(*the_input*)

If you're working with an interactive process, sends that input to the process.

This needs to be used in conjunction with the `has_input=True` parameter.

Requires a `the_input` parameter, which should be a string of the input to send to the command.

Returns the `Shell` instance.

Example:

```
>>> from shell import Shell
>>> sh = Shell(has_input=True)
>>> sh.run('cat -u')
>>> sh.write('Hello world!')
```

exception `shell.`**`ShellException`**

The base exception for all shell-related errors.

`shell.`**`shell`**(*command*, *has_input=False*, *record_output=True*, *record_errors=True*, *strip_empty=True*, *die=False*, *verbose=False*)

A convenient shortcut for running commands.

Requires a `command` parameter should be either a string command (easier) or an array of arguments to send as the command (if you know what you're doing).

Optionally accepts a `has_input` parameter, which should be a boolean. If set to `True`, the command will wait to execute until you call the `Shell.write` method & send input. (Default: `False`)

Optionally accepts a `record_output` parameter, which should be a boolean. If set to `True`, the stdout from the command will be recorded. (Default: `True`)

Optionally accepts a `record_errors` parameter, which should be a boolean. If set to `True`, the stderr from the command will be recorded. (Default: `True`)

Optionally accepts a `strip_empty` parameter, which should be a boolean. If set to `True`, only non-empty lines from `Shell.output` or `Shell.errors` will be returned. (Default: `True`)

Optionally accepts a `die` parameter, which should be a boolean. If set to `True`, raises a CommandError if the command exits with a non-zero return code. (Default: `False`)

Optionally accepts a `verbose` parameter, which should be a boolean. If set to `True`, prints stdout to stdout and stderr to stderr as execution happens. (Default: `False`)

Returns the `Shell` instance, which has been run with the given command.

Example:

```
>>> from shell import shell
>>> sh = shell('ls -alh *py')
>>> sh.output()
['hello.py', 'world.py']
```

Testing shell

`shell` maintains 100% passing tests at all times. That said, there are undoubtedly bugs or odd configurations it doesn't cover.

Setup

Getting setup to run tests (Python 2) looks like:

```
$ git clone https://github.com/toastdriven/shell
$ cd shell
$ virtualenv env
```

```
$ . env/bin/activate
$ pip install mock==1.0.1
$ pip install nose==1.3.0
```

Once that's setup, setting up for Python 3 looks like:

```
$ virtualenv -p python3 env3
$ . env3/bin/activate
$ pip install mock==1.0.1
$ pip install nose==1.3.0
```

Running the tests

To run the tests, run the following:

```
$ nosetests -s tests.py
```

Contributing

To contribute to `shell`, it must meet the following criteria:

- Has a failing test case (see `tests.py` & testing) without the fix
- Has a fix that matches existing style
- Has docstrings
- Adds to the documentation if the change is user-facing
- Is BSD-compatibly licensed

Please create fork on Github, clone your fork, create a new branch, make your changes on that branch, push it back to Github & open a pull request.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index