# Testing shell commands from Python

Janneke van der Zwaan Follow

Dec 12, 2018 · 3 min read

What to do with these shells?

How do you test shell commands? Recently, I came across several cases where I wanted to run shell commands for testing, but couldn’t find a tutorial about how to do it from Python. After a lot of Googling, I found a solution that worked for me, and maybe it works for you too!

## Why test from Python?

You could use dedicated tools to test shell commands. Why choose Python over those? If you are working on a Python package, it makes sense to use Python, because Python already includes robust test functionality that is easy to integrate with other tools. Testing shell commands from Python allows you to harness those facilities and prevents you from having to keep track of tests in different places. Plus, if you are already familiar with writing tests in Python, writing tests for shell commands becomes a breeze.

## Use cases

For the Netherlands eScience Center Python package template, I wanted tests to verify that the generated package can be installed, that the tests can be run, and that the documentation can be generated without errors. My Python text processing package nlppln contains CWL specifications of text mining tools, that can be validated by running them using a command line tool called `cwltool`. Another use case would be testing your package's console scripts (although in this case it might be more convenient to use a package for creating command line interfaces that comes with built-in testing functionality, such as Click).

## The sh package

You can run shell commands from Python using the subprocess module from the Python standard library. However, using this module is a hassle, because you have to do all the error handling yourself. Sh is a Python package that takes care of all that and allows you to run shell commands using a single line of code. If you want to run `python setup.py install`, all you have to do is:

```
import sh

sh.python(['setup.py', 'install'])
```

If you want to run `foo` and it is installed on your system, you can just do `sh.foo()`.

## Writing a test

So how can we use this for testing? Let's look at an example. For the Python template, I want to test whether a project generated from the cookiecutter template can be installed without errors. The goal of the template is to help users write high quality code with less effort, and having an installable empty project is a good first step. The code for the test that tests the installation is:

```
import pytest
import os
import sh

def test_install(cookies):
  # generate a temporary project using the cookiecutter
  # cookies fixture
  project = cookies.bake()

  # remember the directory where tests should be run from
  cwd = os.getcwd()
  # change directories to the generated project directory
  # (the installation command must be run from here)
  os.chdir(str(project.project))

  try:
    # run the shell command
    sh.python(['setup.py', 'install'])
  except sh.ErrorReturnCode as e:
    # print the error, so we know what went wrong
    print(e)
    # make sure the test fails
    pytest.fail(e)
  finally:
    # always change directories to the test directory
    os.chdir(cwd)
```

That is all there is to it!

## More examples

Of course there is a lot more you can do, e.g., checking whether files exist after running a shell command, or verifying the contents of generated files. What use cases can you come up with?

## On Windows: use subprocess

Sh does not work on Windows. If you need to test shell commands on Windows, you are stuck with subprocess. Provenance tracking package recipy contains some nice examples of tests using subprocess that might help you on your way.

Thanks to Carlos Martinez-Ortiz and Patrick Bos.

Python Testing Command Line Pytest Tutorial