ScriptTest Documentation

Release 1.3

Individual Contributors

May 29, 2016

Contents

CHAPTER 1

scripttest – test command-line scripts

Helpers for testing command-line scripts

1.1 Module Contents

class scripttest.**TestFileEnvironment**(*base_path=None*, *template_path=None*, *environ=None*, *cwd=None*, *start_clear=True*, *ignore_paths=None*, *ignore_hidden=True*, *capture_temp=False*, *assert_no_temp=False*, *split_cmd=True*)

This represents an environment in which files will be written, and scripts will be run.

__init__(*base_path=None*, *template_path=None*, *environ=None*, *cwd=None*, *start_clear=True*, *ignore_paths=None*, *ignore_hidden=True*, *capture_temp=False*, *assert_no_temp=False*, *split_cmd=True*)

Creates an environment. `base_path` is used as the current working directory, and generally where changes are looked for. If not given, it will be the directory of the calling script plus `test-output/`.

`template_path` is the directory to look for *template* files, which are files you'll explicitly add to the environment. This is done with `.writefile()`.

`environ` is the operating system environment, `os.environ` if not given.

`cwd` is the working directory, `base_path` by default.

If `start_clear` is true (default) then the `base_path` will be cleared (all files deleted) when an instance is created. You can also use `.clear()` to clear the files.

`ignore_paths` is a set of specific filenames that should be ignored when created in the environment. `ignore_hidden` means, if true (default) that filenames and directories starting with `'.'` will be ignored.

`capture_temp` will put temporary files inside the environment (using `$TMPDIR`). You can then assert that no temporary files are left using `.assert_no_temp()`.

assert_no_temp()

If you use `capture_temp` then you can use this to make sure no files have been left in the temporary directory

clear(*force=False*)

Delete all the files in the base directory.

run(*script*, **args*, ***kw*)

Run the command, with the given arguments. The `script` argument can have space-separated arguments, or you can use the positional arguments.

Keywords allowed are:

expect_error: (default False) Don't raise an exception in case of errors

expect_stderr: (default expect_error) Don't raise an exception if anything is printed to stderr

stdin: (default "") Input to the script

cwd: (default self.cwd) The working directory to run in (default base_path)

quiet: (default False) When there's an error (return code != 0), do not print stdout/stderr

Returns a ProcResult object.

writefile(*path*, *content=None*, *frompath=None*)
Write a file to the given path. If content is given then that text is written, otherwise the file in frompath is used. frompath is relative to self.template_path

1.1.1 Objects that are returned

These objects are returned when you use env.run(...). The *ProcResult* object is returned, and it has .files_updated, .files_created, and .files_deleted which are dictionaries of *FoundFile* and *FoundDir*. The files in .files_deleted represent the pre-deletion state of the file; the other files represent the state of the files after the command is run.

and .files_deleted. These objects dictionary

class scripttest.**ProcResult**(*test_env*, *args*, *stdin*, *stdout*, *stderr*, *returncode*, *files_before*, *files_after*)
Represents the results of running a command in TestFileEnvironment.

Attributes to pay particular attention to:

stdout, stderr: What is produced on those streams.

returncode: The return code of the script.

files_created, files_deleted, files_updated: Dictionaries mapping filenames (relative to the base_path) to FoundFile or FoundDir objects.

class scripttest.**FoundFile**(*base_path*, *path*)
Represents a single file found as the result of a command.

Has attributes:

path: The path of the file, relative to the base_path

full: The full path

bytes: The contents of the file.

stat: The results of os.stat. Also mtime and size contain the .st_mtime and .st_size of the stat.

mtime: The modification time of the file.

size: The size (in bytes) of the file.

You may use the in operator with these objects (tested against the contents of the file), and the .mustcontain() method.

class scripttest.**FoundDir**(*base_path*, *path*)
Represents a directory created by a command.

CHAPTER 2

License

CHAPTER 3

News

3.1 1.3

- Use CRC32 to protect against a race condition where if a run took less than 1 second updates files would not appear to be updated.

3.2 1.2

- Python 3 support (thanks Marc Abramowitz!)

3.3 1.1.1

- Python 3 fixes

3.4 1.1

- Python 3 compatibility, from Hugo Tavares
- More Windows fixes, from Hugo Tavares

3.5 1.0.4

- Windows fixes (thanks Dave Abrahams); including an option for more careful string splitting (useful when testing a script with a space in the path), and more careful handling of environmental variables.

3.6 1.0.3

- Added a `capture_temp` argument to *`scripttest.TestFileEnvironment`* and `env.assert_no_temp()` to test that no temporary files are left over.

3.7 1.0.2

- Fixed regression with `FoundDir.invalid`

3.8 1.0.1

- Windows fix for cleaning up scratch files more reliably
- Allow spaces in the `script` name, e.g., `C:/program files/some-script` (but you must use multiple arguments to `env.run(script, more_args)`).
- Remove the resolution of scripts to an absolute path (just allow the OS to do this).
- Don't fail if there is an invalid symlink

3.9 1.0

- `env.run()` now takes a keyword argument `quiet`. If quiet is false, then if there is any error (return code != 0, or stderr output) the complete output of the script will be printed.
- ScriptTest puts a marker file in scratch directories it deletes, so that if you point it at a directory not created by ScriptTest it will raise an error. Without this, unwitting developers could point ScriptTest at the project directory, which would cause the entire project directory to be wiped.
- ProcResults now no longer print the absolute path of the script (which is often system dependent, and so not good for doctests).
- Added `scripttest.ProcResults.wildcard_matches()` which returns file objects based on a wildcard expression.

3.10 0.9

Initial release

Contents

CHAPTER 4

Status & License

ScriptTest is an extraction of `paste.fixture.TestFileEnvironment` from the Paste project. It was originally written to test Paste Script.

It is licensed under an MIT-style permissive license.

Discussion happens on the Paste mailing list, and bugs should go in the Github issue list.

It is available on pypi or in a git repository. You can get a checkout with:

```
$ git clone https://github.com/pypa/scripttest.git
```

CHAPTER 5

Purpose & Introduction

This library helps you test command-line scripts. It runs a script and watches the output, looks for non-zero exit codes, output on stderr, and any files created, deleted, or modified.

To start you instantiate `TestFileEnvironment`, which is the context in which all your scripts are run. You give it a base directory (typically a scratch directory), or if you don't it will guess `call_module_dir/test-output/`. Example:

```
>>> from scripttest import TestFileEnvironment
>>> env = TestFileEnvironment('./test-output')
```

Note: Everything in `./test-output` will be deleted every test run. To make sure you don't point at an important directory, the scratch directory must be created by ScriptTest (a hidden file is written by ScriptTest to confirm that it created the directory). If the directory already exists, you must delete it manually.

Then you run scripts with `env.run(script, arg1, arg2, ...)`:

```
>>> print(env.run('echo', 'hey'))
Script result: echo hey
-- stdout: --------------------
hey
```

There's several keyword arguments you can use with `env.run()`:

`expect_error`: (default False) Don't raise an exception in case of errors

`expect_stderr`: (default `expect_error`) Don't raise an exception if anything is printed to stderr

`stdin`: (default `""`) Input to the script

`cwd`: (default `self.cwd`) The working directory to run in (default `base_dir`)

As you can see from the options, if the script indicates anything error-like it is, by default, turned into an exception. This of course includes a non-zero response code. Also any output on stderr also counts as an error (unless turned off with `expect_stderr=True`).

The object you get back from a run represents what happened during the script. It has a useful `str()` (as you can see in the previous example) that shows a summary and can be useful in a doctest. It also has several useful attributes:

`stdout`, `stderr`: What is produced on those streams

`returncode`: The return code of the script.

`files_created`, `files_deleted`, `files_updated`: Dictionaries mapping filenames (relative to the `base_dir`) to FoundFile or FoundDir objects.

Of course by default `stderr` must be empty, and `returncode` must be zero, since anything else would be considered an error.

Of particular interest are the dictionaries `files_created`, etc. These show just what files were handled by the script. Each dictionary points to another helper object for inspecting the files (`.files_deleted` contains the files as they existed *before* the script ran).

Each file or directory object has useful attributes:

`path`: The path of the file, relative to the `base_path`

`full`: The full path

`stat`: The results of `os.stat`. Also `mtime` and `size` contain the `.st_mtime` and `st_size` of the stat. (Directories have no `size`)

`bytes`: The contents of the file (does not apply to directories).

`file`, `dir`: `file` is true for files, `dir` is true for directories.

You may use the `in` operator with the file objects (tested against the contents of the file), and the `.mustcontain()` method, where `file.mustcontain('a', 'b')` means `assert 'a' in file; assert 'b' in file`.

S

Symbols

A

C

F

P

R

S

T

W