stestr Documentation

Release trunk

Matthew Treinish

Feb 15, 2019

Contents

Contents:

CHAPTER 1

stestr

You can see the full rendered docs at: http://stestr.readthedocs.io/en/latest/

1.1 Overview

stestr is parallel Python test runner designed to execute unittest test suites using multiple processes to split up execution of a test suite. It also will store a history of all test runs to help in debugging failures and optimizing the scheduler to improve speed. To accomplish this goal it uses the subunit protocol to facilitate streaming and storing results from multiple workers.

stestr originally started as a fork of the testrepository project. But, instead of being an interface for any test runner that used subunit, like testrepository, stestr concentrated on being a dedicated test runner for python projects. While stestr was originally forked from testrepository it is not backwards compatible with testrepository. At a high level the basic concepts of operation are shared between the two projects but the actual usage is not exactly the same.

1.2 Installing stestr

stestr is available via pypi, so all you need to do is run:

```
pip install -U stestr
```

to get stestr on your system. If you need to use a development version of stestr you can clone the repo and install it locally with:

```
git clone https://github.com/mtreinish/stestr.git && pip install -e stestr
```

which will install stestr in your python environment in editable mode for local development

1.3 Using stestr

After you install stestr to use it to run tests is pretty straightforward. The first thing you'll want to do is create a `.stestr.conf` file for your project. This file is used to tell stestr where to find tests and basic information about how tests are run. A basic minimal example of the contents of this is:

```
[DEFAULT]
test_path=./project_source_dir/tests
```

which just tells stestr the relative path for the directory to use for test discovery. This is the same as `--start-directory` in the standard unittest discovery.

After this file is created you should be all set to start using stestr to run tests. To run tests just use:

```
stestr run
```

it will first create a results repository at `.stestr/` in the current working directory and then execute all the tests found by test discovery. If you're just running a single test (or module) and want to avoid the overhead of doing test discovery you can use the `--no-discover`/`-n` option to specify that test.

For all the details on these commands and more thorough explanation of options see the stestr manual: https://stestr.readthedocs.io/en/latest/MANUAL.html

1.4 Migrating from testrepository

If you have a project that is already using testrepository stestr's source repo contains a helper script for migrating your repo to use stestr. This script just creates a `.stestr.conf` file from a `.testr.conf` file. (assuming it uses a standard subunit.run test command format) To run this from your project repo just call:

```
$STESTR_SOURCE_DIR/tools/testr_to_stestr.py
```

and you'll have a `.stestr.conf` created.

1.5 Building a manpage

The stestr manual has been formatted so that it renders well as html and as a manpage. The html output and is autogenerated and published to: https://stestr.readthedocs.io/en/latest/MANUAL.html but the manpage has to be generated by hand. To do this you have to manually run sphinx-build with the manpage builder. This has been automated in a small script that should be run from the root of the stestr repository:

```
tools/build_manpage.sh
```

which will generate the troff file in doc/build/man/stestr.1 which is ready to be packaged and or put in your system's man pages.

CHAPTER 2

stestr user manual

2.1 Usage

A parallel Python test runner built around subunit

```
stestr
    [--version]
    [-v | -q]
    [--log-file LOG_FILE]
    [--debug]
    [--user-config USER_CONFIG]
    [-d HERE]
    [--config CONFIG]
    [--repo-type {file,sql}]
    [--repo-url REPO_URL]
    [--test-path TEST_PATH]
    [--top-dir TOP_DIR]
    [--group-regex GROUP_REGEX]
    [--parallel-class]
```

--version
show program's version number and exit

-v, --verbose
Increase verbosity of output. Can be repeated.

-q, --quiet
Suppress output except warnings and errors.

--log-file <LOG_FILE>
Specify a file to log output. Disabled by default.

--debug
Show tracebacks on errors.

--user-config <USER_CONFIG>
An optional path to a default user config file if one is not specified ~/.stestr.yaml and ~/.config/stestr.yaml will be tried in that order

-d <HERE>, **--here** <HERE>
Set the directory or url that a command should run from. This affects all default path lookups but does not affect paths supplied to the command.

--config <CONFIG>, **-c** <CONFIG>
Set a stestr config file to use with this command. If one isn't specified then .stestr.conf in the directory that a command is running from is used

--repo-type <REPO_TYPE>, **-r** <REPO_TYPE>
Select the repo backend to use

--repo-url <REPO_URL>, **-u** <REPO_URL>
Set the repo url to use. An acceptable value for this depends on the repository type used.

--test-path <TEST_PATH>, **-t** <TEST_PATH>
Set the test path to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.

--top-dir <TOP_DIR>
Set the top dir to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.

--group-regex <GROUP_REGEX>, **--group_regex** <GROUP_REGEX>, **-g** <GROUP_REGEX>
Set a group regex to use for grouping tests together in the stestr scheduler. If both this and the corresponding config file option are set this value will be used.

--parallel-class, **-p**
Set the flag to group tests by class. NOTE: This flag takes priority over the *–group-regex* option even if it's set.

2.1.1 failing

Show the current failures known by the repository

```
stestr failing [--subunit] [--list]
```

--subunit
Show output as a subunit stream.

--list
Show only a list of failing tests.

This command is provided by the stestr plugin.

2.1.2 init

Create a new repository.

```
stestr init
```

This command is provided by the stestr plugin.

2.1.3 last

Show the last run loaded into a repository.

Failing tests are shown on the console and a summary of the run is printed at the end.

Without –subunit, the process exit code will be non-zero if the test run was not successful. With –subunit, the process exit code is non-zero if the subunit stream could not be generated successfully.

```
stestr last
    [--subunit]
    [--no-subunit-trace]
    [--force-subunit-trace]
    [--color]
    [--suppress-attachments]
```

--subunit
Show output as a subunit stream.

--no-subunit-trace
Disable output with the subunit-trace output filter

--force-subunit-trace
Force subunit-trace output regardless of anyother options or config settings

--color
Enable color output in the subunit-trace output, if subunit-trace output is enabled. (this is the default). If subunit-trace is disable this does nothing.

--suppress-attachments
If set do not print stdout or stderr attachment contents on a successful test execution

This command is provided by the stestr plugin.

2.1.4 list

List the tests for a project. You can use a filter just like with the run command to see exactly what tests match

```
stestr list
    [--blacklist-file BLACKLIST_FILE]
    [--whitelist-file WHITELIST_FILE]
    [--black-regex BLACK_REGEX]
    [filters [filters ...]]
```

--blacklist-file <BLACKLIST_FILE>, **-b** <BLACKLIST_FILE>
Path to a blacklist file, this file contains a separate regex exclude on each newline

--whitelist-file <WHITELIST_FILE>, **-w** <WHITELIST_FILE>
Path to a whitelist file, this file contains a separate regex on each newline.

--black-regex <BLACK_REGEX>, **-B** <BLACK_REGEX>
Test rejection regex. If a test cases name matches on re.search() operation , it will be removed from the final test list. Effectively the black-regexp is added to black regexp list, but you do need to edit a file. The black filtering happens after the initial white selection, which by default is everything.

filters
A list of string regex filters to initially apply on the test list. Tests that match any of the regexes will be used. (assuming any other filtering specified also uses it)

This command is provided by the stestr plugin.

2.1.5 load

Load a subunit stream into a repository.

Failing tests are shown on the console and a summary of the stream is printed at the end.

```
stestr load
    [--partial]
    [--force-init]
    [--subunit]
    [--id ID]
    [--subunit-trace]
    [--color]
    [--abbreviate]
    [--suppress-attachments]
    [files [files ...]]
```

--partial
DEPRECATED: The stream being loaded was a partial run. This option is deprecated and no does anything. It will be removed in the future

--force-init
Initialise the repository if it does not exist already

--subunit
Display results in subunit format.

--id <ID>, **-i** <ID>
Append the stream into an existing entry in the repository

--subunit-trace
Display the loaded stream through the subunit-trace output filter

--color
Enable color output in the subunit-trace output, if subunit-trace output is enabled. If subunit-trace is disable this does nothing.

--abbreviate
Print one character status for each test

--suppress-attachments
If set do not print stdout or stderr attachment contents on a successful test execution

files
The subunit v2 stream files to load into the repository

This command is provided by the stestr plugin.

2.1.6 run

Run the tests for a project and store them into the repository.

```
stestr run
    [--failing]
    [--serial]
    [--concurrency CONCURRENCY]
    [--load-list LOAD_LIST]
    [--partial]
    [--subunit]
```

(continues on next page)

(continued from previous page)

```
    [--until-failure]
    [--analyze-isolation]
    [--isolated]
    [--worker-file WORKER_PATH]
    [--blacklist-file BLACKLIST_FILE]
    [--whitelist-file WHITELIST_FILE]
    [--black-regex BLACK_REGEX]
    [--no-discover TEST_ID]
    [--random]
    [--combine]
    [--no-subunit-trace]
    [--force-subunit-trace]
    [--color]
    [--slowest]
    [--abbreviate]
    [--suppress-attachments]
    [filters [filters ...]]
```

--failing
Run only tests known to be failing.

--serial
Run tests in a serial process.

--concurrency <CONCURRENCY>
How many processes to use. The default (0) autodetects your CPU count.

--load-list <LOAD_LIST>
Only run tests listed in the named file.

--partial
DEPRECATED: Only some tests will be run. Implied by –failing. This option is deprecated and no longer does anything. It will be removed in the future

--subunit
Display results in subunit format.

--until-failure
Repeat the run again and again until failure occurs.

--analyze-isolation
Search the last test run for 2-test test isolation interactions.

--isolated
Run each test id in a separate test runner.

--worker-file <WORKER_PATH>
Optional path of a manual worker grouping file to use for the run

--blacklist-file <BLACKLIST_FILE>, **-b** <BLACKLIST_FILE>
Path to a blacklist file, this file contains a separate regex exclude on each newline

--whitelist-file <WHITELIST_FILE>, **-w** <WHITELIST_FILE>
Path to a whitelist file, this file contains a separate regex on each newline.

--black-regex <BLACK_REGEX>, **-B** <BLACK_REGEX>
Test rejection regex. If a test cases name matches on re.search() operation , it will be removed from the final test list. Effectively the black-regexp is added to black regexp list, but you do need to edit a file. The black filtering happens after the initial white selection, which by default is everything.

--no-discover TEST_ID, **-n** TEST_ID
: Takes in a single test to bypasses test discover and just execute the test specified. A file may be used in place of a test name.

--random, -r
: Randomize the test order after they are partitioned into separate workers

--combine
: Combine the results from the test run with the last run in the repository

--no-subunit-trace
: Disable the default subunit-trace output filter

--force-subunit-trace
: Force subunit-trace output regardless of anyother options or config settings

--color
: Enable color output in the subunit-trace output, if subunit-trace output is enabled. (this is the default). If subunit-trace is disable this does nothing.

--slowest
: After the test run, print the slowest tests.

--abbreviate
: Print one character status for each test

--suppress-attachments
: If set do not print stdout or stderr attachment contents on a successful test execution

filters
: A list of string regex filters to initially apply on the test list. Tests that match any of the regexes will be used. (assuming any other filtering specified also uses it)

This command is provided by the stestr plugin.

2.1.7 slowest

Show the slowest tests from the last test run.

This command shows a table, with the longest running tests at the top.

```
stestr slowest [--all]
```

--all
: Show timing for all tests.

This command is provided by the stestr plugin.

2.2 Overview

stestr is an application for running and tracking test results. Any test run that can be represented as a subunit stream can be inserted into a repository. However, the test running mechanism assumes python is being used. It is originally forked from the testrepository project so the usage is similar.

A typical basic example workflow is:

```
# Create a store to manage test results in.
$ stestr init
# Do a test run
$ stestr run
```

Most commands in testr have comprehensive online help, and the commands:

```
$ stestr --help
$ stestr [command] --help
```

Will be useful to explore the system.

2.3 Configuration

To configure stestr for a project you can write a stestr configuration file. This lets you set basic information about how tests are run for a project. By default the config file needs to be `.stestr.conf` in the same directory that stestr is run from, normally the root of a project's repository. However, the `--config`/`-c` CLI argument can specify an alternate path for it.

The 2 most important options in the stestr config file are `test_path` and `top_dir`. These 2 options are used to set the unittest discovery options for stestr. (test_path is the same as `--start-directory` and top_dir is the same as `--top-level-directory` in the doc) Only test_path is a required field in the config file, if top_dir is not specified it defaults to `./`. It's also worth noting that shell variables for these 2 config options (and only these 2 options) are expanded on platforms that have a shell. This enables you to have conditional discovery paths based on your environment.

For example, having a config file like:

```
[DEFAULT]
test_path=${TEST_PATH:-./foo/tests}
```

will let you override the discovery start path using the TEST_PATH environment variable.

A full example config file is:

```
[DEFAULT]
test_path=./project/tests
top_dir=./
group_regex=([^\.]*\.)*
```

The `group_regex` option is used to specify is used to provide a scheduler hint for how tests should be divided between test runners. See the *Grouping Tests* section for more information on how this works. You can also specify the `parallel_class=True` instead of group_regex to group tests in the stestr scheduler together by class. Since this is a common use case this enables that without needing to memorize the complicated regex for `group_regex` to do this.

There is also an option to specify all the options in the config file via the CLI. This way you can run stestr directly without having to write a config file and manually specify the test_path like above with the `--test-path`/`-t` CLI argument.

2.4 Running tests

To run tests the `stestr run` command is used. By default this will run all tests discovered using the discovery parameters in the stestr config file.

If you'd like to avoid the overhead of test discovery and just manually execute a single test (test class, or module) you can do this using the `--no-discover`/`-n` option. For example:

```
$ stestr run --no-discover project.tests.test_foo.TestFoo
```

you can also give it a file path and stestr will convert that to the proper python path under the covers. (assuming your project don't manually mess with import paths) For example:

```
$ stestr run --no-discover project/tests/test_foo.py
```

will also bypass discovery and directly call subunit.run on the module specified.

Additionally you can specify a specific class or method within that file using `::` to specify a class and method. For example:

```
$ stestr run --no-discover project/tests/test_foo.py::TestFoo::test_method
```

will skip discovery and directly call subunit.run on the test method in the specified test class.

2.5 Test Selection

Arguments passed to `stestr run` are used to filter test ids that will be run. stestr will perform unittest discovery to get a list of all test ids and then apply each argument as a regex filter. Tests that match any of the given filters will be run. For example, if you called `stestr run foo bar` this will only run the tests that have a regex match with foo **or** a regex match with bar.

stestr allows you do to do simple test exclusion via passing a rejection/black regexp:

```
$ stestr run --black-regex 'slow_tests|bad_tests'
```

stestr also allow you to combine these arguments:

```
$ stestr run --black-regex 'slow_tests|bad_tests' ui\.interface
```

Here first we selected all tests which matches to `ui\.interface`, then we are dropping all test which matches `slow_tests|bad_tests` from the final list.

stestr also allows you to specify a blacklist file to define a set of regexes to exclude. You can specify a blacklist file with the `--blacklist-file`/`-b` option, for example:

```
$ stestr run --blacklist-file $path_to_file
```

The format for the file is line separated regex, with `#` used to signify the start of a comment on a line. For example:

```
# Blacklist File
^regex1 # Excludes these tests
.*regex2 # exclude those tests
```

The regexp used in the blacklist file or passed as argument, will be used to drop tests from the initial selection list. It will generate a list which will exclude any tests matching `^regex1` or `.*regex2`. If a blacklist file is used in conjunction with the normal filters then the regex filters passed in as an argument regex will be used for the initial test selection, and the exclusion regexes from the blacklist file on top of that.

The dual of the blacklist file is the whitelist file which will include any tests matching the regexes in the file. You can specify the path to the file with `--whitelist-file`/`-w`, for example:

```
$ stestr run --whitelist-file $path_to_file
```

The format for the file is more or less identical to the blacklist file:

```
# Whitelist File
^regex1 # Include these tests
.*regex2 # include those tests
```

However, instead of excluding the matches it will include them.

It's also worth noting that you can use the test list option to dry run any selection arguments you are using. You just need to use `stestr list` with your selection options to do this, for example:

```
$ stestr list 'regex3.*' --blacklist-file blacklist.txt
```

This will list all the tests which will be run by stestr using that combination of arguments.

2.6 Adjusting test run output

By default the `stestr run` command uses an output filter called subunit-trace. (as does the `stestr last` command) This displays the tests as they are finished executing, as well as their worker and status. It also prints aggregate numbers about the run at the end. You can read more about subunit-trace in the module doc: *Subunit Trace*.

However, the test run output is configurable, you can disable this output with the `--no-subunit-trace` flag which will be completely silent except for any failures it encounters. There is also the `--color` flag which will enable colorization with subunit-trace output. If you prefer to deal with the raw subunit yourself and run your own output rendering or filtering you can use the `--subunit` flag to output the result stream as raw subunit v2.

There is also an `--abbreviate` flag available, when this is used a single character is printed for each test as it is executed. A `.` is printed for a successful test, a `F` for a failed test, and a `S` for a skipped test.

In the default subunit-trace output any captured output to stdout and stderr is printed after test execution, for both successful and failed tests. However, in some cases printing these attachments on a successful tests is not the preferred behavior. You can use the `--suppress-attachments` flag to disable printing stdout or stderr attachments for successful tests.

2.7 Combining Test Results

There is sometimes a use case for running a single test suite split between multiple invocations of the stestr run command. For example, running a subset of tests with a different concurrency. In these cases you can use the `--combine` flag on `stestr run`. When this flag is specified stestr will append the subunit stream from the test run into the most recent entry in the repository.

Alternatively, you can manually load the test results from a subunit stream into an existing test result in the repository using the `--id`/`-i` flag on the `stestr load` command. This will append the results from the input subunit stream to the specified id.

2.8 Running previously failed tests

`stestr run` also enables you to run just the tests that failed in the previous run. To do this you can use the `--failing` argument.

A common workflow using this is:

1. Run tests (and some fail):

```
$ stestr run
```

2. Fix currently broken tests - repeat until there are no failures:

```
$ stestr run --failing
```

3. Do a full run to find anything that regressed during the reduction process:

```
$ stestr run
```

Another common use case is repeating a failure that occurred on a remote machine (e.g. during a jenkins test run). There are a few common ways to do approach this.

Firstly, if you have a subunit stream from the run you can just load it:

```
$ stestr load < failing-stream
```

and then run the tests which failed from that loaded run:

```
$ stestr run --failing
```

If using a file type repository (which is the default) the streams generated by test runs are in the repository path, which defaults to *.stestr/* in the working directory, and stores the stream in a file named for their run id - e.g. .stestr/0 is the first run.

Note: For right now these files are stored in the subunit v1 format, but all of the stestr commands, including load, only work with the subunit v2 format. This can be converted using the **subunit-1to2** tool in the python-subunit package.

If you have access to the remote machine you can also get the subunit stream by running:

```
$ stestr last --subunit > failing-stream
```

This is often a bit easier than trying to manually pull the stream file out of the .stestr directory. (also it will be in the subunit v2 format already)

If you do not have a stream or access to the machine you may be able to use a list file. If you can get a file that contains one test id per line, you can run the named tests like this:

```
$ stestr run --load-list FILENAME
```

This can also be useful when dealing with sporadically failing tests, or tests that only fail in combination with some other test - you can bisect the tests that were run to get smaller and smaller (or larger and larger) test subsets until the error is pinpointed.

`stestr run --until-failure` will run your test suite again and again and again stopping only when interrupted or a failure occurs. This is useful for repeating timing-related test failures.

2.9 Listing tests

To see a list of tests found by stestr you can use the `stestr list` command. This will list all tests found by discovery.

You can also use this to see what tests will be run by a given stestr run command. For instance, the tests that `stestr run myfilter` will run are shown by `stestr list myfilter`. As with the run command, arguments to list are used to regex filter the tests.

2.10 Parallel testing

stestr lets you run tests in parallel by default. So, it actually does this by def:

```
$ stestr run
```

This will first list the tests, partition the tests into one partition per CPU on the machine, and then invoke multiple test runners at the same time, with each test runner getting one partition. Currently the partitioning algorithm is simple round-robin for tests that stestr has not seen run before, and equal-time buckets for tests that stestr has seen run.

To determine how many CPUs are present in the machine, stestr will use the multiprocessing Python module On operating systems where this is not implemented, or if you need to control the number of workers that are used, the `--concurrency` option will let you do so:

```
$ stestr run --concurrency=2
```

When running tests in parallel, stestr adds a tag for each test to the subunit stream to show which worker executed that test. The tags are of the form `worker-%d` and are usually used to reproduce test isolation failures, where knowing exactly what test ran on a given worker is important. The %d that is substituted in is the partition number of tests from the test run - all tests in a single run with the same worker-N ran in the same test runner instance.

To find out which slave a failing test ran on just look at the 'tags' line in its test error:

```
======================================================================
label: testrepository.tests.ui.TestDemo.test_methodname
tags: foo worker-0
----------------------------------------------------------------------
error text
```

And then find tests with that tag:

```
$ stestr last --subunit | subunit-filter -s --xfail --with-tag=worker-3 | subunit-ls >
↪ slave-3.list
```

2.11 Grouping Tests

In certain scenarios you may want to group tests of a certain type together so that they will be run by the same worker process. The `group_regex` option in the stestr config file permits this. When set, tests are grouped by the entire matching portion of the regex. The match must begin at the start of the string. Tests with no match are not grouped.

For example, setting the following option in the stestr config file will group tests in the same class together (the last '.' splits the class and test method):

```
group_regex=([^\.]+\.)+
```

However, because grouping tests at the class level is a common use case there is also a config option, `parallel_class`, to do this. For example, you can use:

```
parallel_class=True
```

and it will group tests in the same class together.

Note: This `parallel_class` option takes priority over the `group_regex` option. And if both on the CLI and in the config are set, we use the option on the CLI not in a config file. For example, `--group-regex` on the CLI and `parallel-class` in a config file are set, `--group-regex` is higer priority than `parallel-class` in this case.

2.12 Test Scheduling

By default stestr schedules the tests by first checking if there is any historical timing data on any tests. It then sorts the tests by that timing data loops over the tests in order and adds one to each worker that it will launch. For tests without timing data, the same is done, except the tests are in alphabetical order instead of based on timing data. If a group regex is used the same algorithm is used with groups instead of individual tests.

However there are options to adjust how stestr will schedule tests. The primary option to do this is to manually schedule all the tests run. To do this use the `--worker-file` option for stestr run. This takes a path to a yaml file that instructs stestr how to run tests. It is formatted as a list of dicts with a single element each with a list describing the tests to run on each worker. For example:

```
- worker:
  - regex 1

- worker:
  - regex 2
  - regex 3
```

would create 2 workers. The first would run all tests that match regex 1, and the second would run all tests that match regex 2 or regex 3. In addition if you need to mix manual scheduling and the standard scheduling mechanisms you can accomplish this with the `concurrency` field on a worker in the yaml. For example, building on the previous example:

```
- worker:
  - regex 1

- worker:
  - regex 2
  - regex 3

- worker:
  - regex 4
  concurrency: 3
```

In this case the tests that match regex 4 will be run against 3 workers and the tests will be partitioned across those workers with the normal scheduler. This includes respecting the other scheduler options, like `group_regex` or `--random`.

There is also an option on `stestr run`, `--random`/`-r` to randomize the order of tests as they are passed to the workers. This is useful in certain use cases, especially when you want to test isolation between test cases.

2.13 User Config Files

If you prefer to have a different default output or setting for a particular command stestr enables you to write a user config file to overide the defaults for some options on some commands. By default stestr will look for this config file in `~/.stestr.yaml` and `~/.config/stestr.yaml` in that order. You can also specify the path to a config file with the `--user-config` parameter.

The config file is a yaml file that has a top level key for the command and then a sub key for each option. For an example, a fully populated config file that changes the default on all available options in the config file is:

```
run:
  concurrency: 42 # This can be any integer value >= 0
  random: True
  no-subunit-trace: True
  color: True
  abbreviate: True
  slowest: True
  suppress-attachments: True
failing:
  list: True
last:
  no-subunit-trace: True
  color: True
  suppress-attachments: True
load:
  force-init: True
  subunit-trace: True
  color: True
  abbreviate: True
  suppress-attachments: True
```

If you choose to use a user config file you can specify any subset of the options and commands you choose.

2.14 Automated test isolation bisection

As mentioned above, its possible to manually analyze test isolation issues by interrogating the repository for which tests ran on which worker, and then creating a list file with those tests, re-running only half of them, checking the error still happens, rinse and repeat.

However that is tedious. stestr can perform this analysis for you:

```
$ stestr run --analyze-isolation
```

will perform that analysis for you. The process is:

1. The last run in the repository is used as a basis for analysing against - tests are only cross checked against tests run in the same worker in that run. This means that failures accrued from several different runs would not be processed with the right basis tests - you should do a full test run to seed your repository. This can be local, or just stestr load a full run from your Jenkins or other remote run environment.
2. Each test that is currently listed as a failure is run in a test process given just that id to run.
3. Tests that fail are excluded from analysis - they are broken on their own.
4. The remaining failures are then individually analysed one by one.
5. For each failing, it gets run in one work along with the first 1/2 of the tests that were previously run prior to it.

6. If the test now passes, that set of prior tests are discarded, and the other half of the tests is promoted to be the full list. If the test fails then other other half of the tests are discarded and the current set promoted.
7. Go back to running the failing test along with 1/2 of the current list of priors unless the list only has 1 test in it. If the failing test still failed with that test, we have found the isolation issue. If it did not then either the isolation issue is racy, or it is a 3-or-more test isolation issue. Neither of those cases are automated today.

2.15 Forcing isolation

Sometimes it is useful to force a separate test runner instance for each test executed. The `--isolated` flag will cause stestr to execute a separate runner per test:

```
$ stestr run --isolated
```

In this mode stestr first determines tests to run (either automatically listed, using the failing set, or a user supplied load-list), and then spawns one test runner per test it runs. To avoid cross-test-runner interactions concurrency is disabled in this mode. `--analyze-isolation` supersedes `--isolated` if they are both supplied.

2.16 Repositories

stestr uses a data repository to keep track of test previous test runs. There are different backend types that each offer different advantages. There are currently 2 repository types to choose from, **file** and **sql**.

You can choose which repository type you want with the `--repo-type`/`-r` cli flag. **file** is the current default.

You can also specify an alternative repository with the `--repo-url`/`-u` cli flags. The default value for a **file** repository type is to use the directory: `$CWD/.stestr`. For a **sql** repository type is to use a sqlite database located at: `$CWD/.stestr.sqlite`.

Note: Make sure you put these flags before the cli subcommand

Note: Different repository types that use local storage will conflict with each other in the same directory. If you initialize one repository type and then try to use another in the same directory, it will not work.

2.16.1 File

The default stestr repository type has a very simple disk structure. It contains the following files:

- format: This file identifies the precise layout of the repository, in case future changes are needed.
- next-stream: This file contains the serial number to be used when adding another stream to the repository.
- failing: This file is a stream containing just the known failing tests. It is updated whenever a new stream is added to the repository, so that it only references known failing tests.
- #N - all the streams inserted in the repository are given a serial number.

2.16.2 SQL

This is an experimental repository backend, that is based on the *subunit2sql* library. It's currently still under development and should be considered experimental for the time being. Eventually it'll replace the File repository type

---

Note: The sql repository type requirements are not installed by default. They are listed under the 'sql' setuptools extras. You can install them with pip by running: `pip install 'stestr[sql]'`

CHAPTER 3

Development Guidelines for stestr

3.1 Coding style

PEP-8 is used for changes. We enforce running flake8 prior to landing any commits.

3.2 Testing and QA

For stestr please add tests where possible. There is no requirement for one test per change (because somethings are much harder to automatically test than the benefit from such tests). But, if unit testing is reasonable it will be expected to be present before it can merge.

3.3 Running the tests

Generally just `tox` is all that is needed to run all the tests. However if dropping into pdb, it is currently more convenient to use `python -m testtools.run testrepository.tests.test_suite`.

CHAPTER 4

Internal Architecture

This document is an attempt to explain at a high level how stestr is constructed. It'll likely go stale quickly as the code changes, but hopefully it'll be a useful starting point for new developers to understand how the stestr is built. Full API documentation can be found at *Internal API Reference*. It's also worth noting that any explanation of workflow or internal operation is not necessarily an exact call path, but instead just a high level explanation of how the components operate.

4.1 Basic Structure

At a high level there are a couple different major components to stestr: the repository, and the cli layer.

The repository is how stestr stores all results from test runs and the source of any data needed by any stestr operations that require past runs. There are actually multiple repository types which are different implementations of an abstract API. Right now there is only one complete implementation, the file repository type, which is useful in practice but that may not be the case in the future.

The CLI layer is where the different stestr commands are defined and provides the command line interface for performing the different stestr operations.

4.2 CLI Layer

The CLI layer is built using the cliff.command module. The stestr.cli module defines a basic interface using cliff. Each subcommand has its own module in stestr.commands and has 3 required functions to work properly:

1. get_parser(prog_name)
2. get_description()
3. take_action(parsed_args)

NOTE: To keep the api compatibility in stestr.commands, we still have each subcommands there.

4.2.1 get_parser(prog_name)

This function is used to define subcommand arguments. It has a single argparse parser object passed into it. The intent of this function is to have any command specific arguments defined on the provided parser object by calling parser.add_argument() for each argument.

4.2.2 get_description()

The intent of this function is to return an command specific help information. It is expected to return a string that will be used when the subcommand is defined in argparse and will be displayed before the arguments when `--help` is used on the subcommand.

4.2.3 take_action(parsed_args)

This is where the real work for the command is performed. This is the function that is called when the command is executed. This function is called being wrapped by sys.exit() so an integer return is expected that will be used for the command's return code. The arguments input parsed_args is the argparse.Namespace object from the parsed CLI options.

4.3 Operations for Running Tests

The basic flow when stestr run is called at a high level is fairly straight forward. In the default case when run is called the first operation performed is unittest discovery (via `subunit.run --discover`) which is used to get a complete list of tests present. This list is then filtered by any user provided selection mechanisms. (for example a cli regex filter) This is used to select which tests the user actually intends to run. For more details on test selection see: *Test Selection Module* which defines the functions which are used to actually perform the filtering.

Once there is complete list of tests that will be run the list gets passed to the scheduler/partitioner. The scheduler takes the list of tests and splits it into N groups where N is the concurrency that stestr will use to run tests. If there is any timing data available in the repository from previous runs this is used by the scheduler to try balancing the test load between the workers. For the full details on how the partitioning is performed see: *The Scheduler Module*.

With the tests split into multiple groups for each worker process we're ready to start executing the tests. Each group of tests is used to launch a subunit.run worker subprocess. As the name implies this is a test runner that emits a subunit stream to stdout. These stdout streams are combined in real time and stored in the repository at the end of the run (using the load command). The combined stream is also used for the CLI output either in a summary view or with a real time subunit output (which is enabled with the `--subunit` argument)

CHAPTER 5

Internal API Reference

This document serves as a reference for the python API used in stestr. It should serve as a guide for both internal and external use of stestr components via python. The majority of the contents here are built from internal docstrings in the actual code.

5.1 Repository

5.1.1 Abstract Repository Class

Storage of test results.

A Repository provides storage and indexing of results.

The AbstractRepository class defines the contract to which any Repository implementation must adhere.

The stestr.repository.file module (see: *File Repository Type* is the usual repository that will be used. The stestr.repository.memory module (see: *Memory Repository Type*) provides a memory only repository useful for internal testing.

Repositories are identified by their URL, and new ones are made by calling the initialize function in the appropriate repository module.

class stestr.repository.abstract.**AbstractRepository**

The base class for Repository implementations.

There are no interesting attributes or methods as yet.

count()

Return the number of test runs this repository has stored.

Return count The count of test runs stored in the repository.

get_failing()

Get a TestRun that contains all of and only current failing tests.

Returns a TestRun.

get_inserter(*partial=False*, *run_id=None*)
Get an inserter that will insert a test run into the repository.

Repository implementations should implement _get_inserter.

get_inserter() does not add timing data to streams: it should be provided by the caller of get_inserter (e.g. commands.load).

Parameters **partial** – DEPREACTED: If True, the stream being inserted only executed some tests rather than all the projects tests. This option is deprecated and no longer does anything. It will be removed in the future.

Return an inserter Inserters meet the extended TestResult protocol that testtools 0.9.2 and above offer. The startTestRun and stopTestRun methods in particular must be called.

get_latest_run()
Return the latest run.

Equivalent to get_test_run(latest_id()).

get_test_ids(*run_id*)
Return the test ids from the specified run.

Parameters **run_id** – the id of the test run to query.

Returns a list of test ids for the tests that were part of the specified test run.

get_test_run(*run_id*)
Retrieve a TestRun object for run_id.

Parameters **run_id** – The test run id to retrieve.

Returns A TestRun object.

get_test_times(*test_ids*)
Retrieve estimated times for the tests test_ids.

Parameters **test_ids** – The test ids to query for timing data.

Returns A dict with two keys: 'known' and 'unknown'. The unknown key contains a set with the test ids that did run. The known key contains a dict mapping test ids to time in seconds.

latest_id()
Return the run id for the most recently inserted test run.

class stestr.repository.abstract.**AbstractRepositoryFactory**
Interface for making or opening repositories.

initialise(*url*)
Create a repository at URL.

Call on the class of the repository you wish to create.

open(*url*)
Open the repository at url.

Raise RepositoryNotFound if there is no repository at the given url.

class stestr.repository.abstract.**AbstractTestRun**
A test run that has been stored in a repository.

Should implement the StreamResult protocol as well as the stestr specific methods documented here.

get_id()
Get the id of the test run.

Sometimes test runs will not have an id, e.g. test runs for 'failing'. In that case, this should return None.

get_subunit_stream()
Get a subunit stream for this test run.

get_test()
Get a testtools.TestCase-like object that can be run.

Returns A TestCase like object which can be run to get the individual tests reported to a testtools.StreamResult/TestResult. (Clients of repository should provide an ExtendedToStreamDecorator decorator to permit either API to be used).

exception stestr.repository.abstract.**RepositoryNotFound**(*url*)
Raised when we try to open a repository that isn't there.

5.1.2 File Repository Type

Persistent storage of test results.

class stestr.repository.file.**Repository**(*base*)
Disk based storage of test results.

This repository stores each stream it receives as a file in a directory. Indices are then built on top of this basic store.

This particular disk layout is subject to change at any time, as its primarily a bootstrapping exercise at this point. Any changes made are likely to have an automatic upgrade process.

count()
Return the number of test runs this repository has stored.

Return count The count of test runs stored in the repository.

get_failing()
Get a TestRun that contains all of and only current failing tests.

Returns a TestRun.

get_test_run(*run_id*)
Retrieve a TestRun object for run_id.

Parameters **run_id** – The test run id to retrieve.

Returns A TestRun object.

latest_id()
Return the run id for the most recently inserted test run.

class stestr.repository.file.**RepositoryFactory**

initialise(*url*)
Create a repository at url/path.

open(*url*)
Open the repository at url.

Raise RepositoryNotFound if there is no repository at the given url.

5.1.3 Memory Repository Type

In memory storage of test results.

class stestr.repository.memory.**Repository**
In memory storage of test results.

count()
Return the number of test runs this repository has stored.

Return count The count of test runs stored in the repository.

get_failing()
Get a TestRun that contains all of and only current failing tests.

Returns a TestRun.

get_test_run(*run_id*)
Retrieve a TestRun object for run_id.

Parameters **run_id** – The test run id to retrieve.

Returns A TestRun object.

latest_id()
Return the run id for the most recently inserted test run.

class stestr.repository.memory.**RepositoryFactory**
A factory that can initialise and open memory repositories.

This is used for testing where a repository may be created and later opened, but tests should not see each others repositories.

initialise(*url*)
Create a repository at URL.

Call on the class of the repository you wish to create.

open(*url*)
Open the repository at url.

Raise RepositoryNotFound if there is no repository at the given url.

5.1.4 SQL Repository Type

5.2 Commands

These modules are used for the operation of all the various subcommands in stestr. As of the 1.0.0 release each of these commands should be considered a stable interface that can be relied on externally.

Each command module conforms to a basic format that is based on the cliff framework. The basic structure for these modules is the following three functions in each class:

```
def get_description():
    """This function returns a string that is used for the subcommand help"""
    help_str = "A descriptive help string about the command"
    return help_str

def get_parser(prog_name):
    """This function takes a parser and any subcommand arguments are defined
```

(continues on next page)

(continued from previous page)

```
        here"""
    parser.add_argument(...)

def take_action(parsed_args):
    """This is where the real work for the command is performed. This is the function
       that is called when the command is executed. This function is called being
       wrapped by sys.exit() so an integer return is expected that will be used
       for the command's return code. The arguments input parsed_args is the
       argparse.Namespace object from the parsed CLI options."""
    return call_foo(...)
```

The command class will not work if all 3 of these function are not defined. However, to make the commands externally consumable each module also contains another public function which performs the real work for the command. Each one of these functions has a defined stable Python API signature with args and kwargs so that people can easily call the functions from other python programs. This function is what can be expected to be used outside of stestr as the stable interface. All the stable functions can be imported the command module directly:

```
from stestr import command

def my_list():
    command.list_command(...)
```

5.2.1 stestr Commands

stestr.commands.__init__.**failing_command**(*repo_type='file'*, *repo_url=None*, *list_tests=False*, *subunit=False*, *stdout=<open file '<stdout>', mode 'w'>*)

Print the failing tests from the most recent run in the repository

This function will print to STDOUT whether there are any tests that failed in the last run. It optionally will print the test_ids for the failing tests if `list_tests` is true. If `subunit` is true a subunit stream with just the failed tests will be printed to STDOUT.

Note this function depends on the cwd for the repository if *repo_type* is set to file and *repo_url* is not specified it will use the repository located at CWD/.stestr

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **list_test** (*bool*) – Show only a list of failing tests.
- **subunit** (*bool*) – Show output as a subunit stream.
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

stestr.commands.__init__.**init_command**(*repo_type='file'*, *repo_url=None*, *stdout=<open file '<stdout>', mode 'w'>*)

Initialize a new repository

This function will create initialize a new repostiory if one does not exist. If one exists the command will fail.

Note this function depends on the cwd for the repository if *repo_type* is set to file and *repo_url* is not specified it will use the repository located at CWD/.stestr

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

stestr.commands.__init__.**last_command**(*repo_type='file'*, *repo_url=None*, *subunit_out=False*, *pretty_out=True*, *color=False*, *stdout=<open file '<stdout>'*, *mode 'w'>*, *suppress_attachments=False*)

Show the last run loaded into a a repository

This function will print the results from the last run in the repository to STDOUT. It can optionally print the subunit stream for the last run to STDOUT if the `subunit` option is set to true.

Note this function depends on the cwd for the repository if *repo_type* is set to file and *repo_url* is not specified it will use the repository located at CWD/.stestr

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **subunit_out** (*bool*) – Show output as a subunit stream.
- **pretty_out** – Use the subunit-trace output filter.
- **color** – Enable colorized output with the subunit-trace output filter.
- **subunit** (*bool*) – Show output as a subunit stream.
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout
- **suppress_attachments** (*bool*) – When set true attachments subunit_trace will not print attachments on successful test execution.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

stestr.commands.__init__.**list_command**(*config='.stestr.conf'*, *repo_type='file'*, *repo_url=None*, *test_path=None*, *top_dir=None*, *group_regex=None*, *blacklist_file=None*, *whitelist_file=None*, *black_regex=None*, *filters=None*, *stdout=<open file '<stdout>'*, *mode 'w'>*)

Print a list of test_ids for a project

This function will print the test_ids for tests in a project. You can filter the output just like with the run command to see exactly what will be run.

Parameters

- **config** (*str*) – The path to the stestr config file. Must be a string.
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **test_path** (*str*) – Set the test path to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.

- **top_dir** (*str*) – The top dir to use for unittest discovery. This takes precedence over the value in the config file. (if one is present in the config file)
- **group_regex** (*str*) – Set a group regex to use for grouping tests together in the stestr scheduler. If both this and the corresponding config file option are set this value will be used.
- **blacklist_file** (*str*) – Path to a blacklist file, this file contains a separate regex exclude on each newline.
- **whitelist_file** (*str*) – Path to a whitelist file, this file contains a separate regex on each newline.
- **black_regex** (*str*) – Test rejection regex. If a test cases name matches on re.search() operation, it will be removed from the final test list.
- **filters** (*list*) – A list of string regex filters to initially apply on the test list. Tests that match any of the regexes will be used. (assuming any other filtering specified also uses it)
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout

stestr.commands.__init__.**load_command**(*force_init=False*, *in_streams=None*, *partial=False*, *subunit_out=False*, *repo_type='file'*, *repo_url=None*, *run_id=None*, *streams=None*, *pretty_out=False*, *color=False*, *stdout=<open file '<stdout>'*, *mode 'w'>*, *abbreviate=False*, *suppress_attachments=False*, *serial=False*)

Load subunit streams into a repository

This function will load subunit streams into the repository. It will output to STDOUT the results from the input stream. Internally this is used by the run command to both output the results as well as store the result in the repository.

Parameters

- **force_init** (*bool*) – Initialize the specifiedrepository if it hasn't been created.
- **in_streams** (*list*) – A list of file objects that will be saved into the repository
- **partial** (*bool*) – DEPRECATED: Specify the input is a partial stream. This option is deprecated and no longer does anything. It will be removed in the future.
- **subunit_out** (*bool*) – Output the subunit stream to stdout
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **run_id** – The optional run id to save the subunit stream to.
- **streams** (*list*) – A list of file paths to read for the input streams.
- **pretty_out** (*bool*) – Use the subunit-trace output filter for the loaded stream.
- **color** (*bool*) – Enabled colorized subunit-trace output
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout
- **abbreviate** (*bool*) – Use abbreviated output if set true
- **suppress_attachments** (*bool*) – When set true attachments subunit_trace will not print attachments on successful test execution.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

stestr.commands.__init__.**run_command**(*config='.stestr.conf'*, *repo_type='file'*, *repo_url=None*, *test_path=None*, *top_dir=None*, *group_regex=None*, *failing=False*, *serial=False*, *concurrency=0*, *load_list=None*, *partial=False*, *subunit_out=False*, *until_failure=False*, *analyze_isolation=False*, *isolated=False*, *worker_path=None*, *blacklist_file=None*, *whitelist_file=None*, *black_regex=None*, *no_discover=False*, *random=False*, *combine=False*, *filters=None*, *pretty_out=True*, *color=False*, *stdout=<open file '<stdout>', mode 'w'>*, *abbreviate=False*, *suppress_attachments=False*)

Function to execute the run command

This function implements the run command. It will run the tests specified in the parameters based on the provided config file and/or arguments specified in the way specified by the arguments. The results will be printed to STDOUT and loaded into the repository.

Parameters

- **config** (*str*) – The path to the stestr config file. Must be a string.
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **test_path** (*str*) – Set the test path to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.
- **top_dir** (*str*) – The top dir to use for unittest discovery. This takes precedence over the value in the config file. (if one is present in the config file)
- **group_regex** (*str*) – Set a group regex to use for grouping tests together in the stestr scheduler. If both this and the corresponding config file option are set this value will be used.
- **failing** (*bool*) – Run only tests known to be failing.
- **serial** (*bool*) – Run tests serially
- **concurrency** (*int*) – "How many processes to use. The default (0) autodetects your CPU count and uses that.
- **load_list** (*str*) – The path to a list of test_ids. If specified only tests listed in the named file will be run.
- **partial** (*bool*) – DEPRECATED: Only some tests will be run. Implied by *–failing*. This flag is deprecated because and doesn't do anything it will be removed in a future release.
- **subunit_out** (*bool*) – Display results in subunit format.
- **until_failure** (*bool*) – Repeat the run again and again until failure occurs.
- **analyze_isolation** (*bool*) – Search the last test run for 2-test test isolation interactions.
- **isolated** (*bool*) – Run each test id in a separate test runner.
- **worker_path** (*str*) – Optional path of a manual worker grouping file to use for the run.
- **blacklist_file** (*str*) – Path to a blacklist file, this file contains a separate regex exclude on each newline.
- **whitelist_file** (*str*) – Path to a whitelist file, this file contains a separate regex on each newline.

- **black_regex** (*str*) – Test rejection regex. If a test cases name matches on re.search() operation, it will be removed from the final test list.
- **no_discover** (*str*) – Takes in a single test_id to bypasses test discover and just execute the test specified. A file name may be used in place of a test name.
- **random** (*bool*) – Randomize the test order after they are partitioned into separate workers
- **combine** (*bool*) – Combine the results from the test run with the last run in the repository
- **filters** (*list*) – A list of string regex filters to initially apply on the test list. Tests that match any of the regexes will be used. (assuming any other filtering specified also uses it)
- **pretty_out** (*bool*) – Use the subunit-trace output filter
- **color** (*bool*) – Enable colorized output in subunit-trace
- **stdout** (*file*) – The file object to write all output to. By default this is sys.stdout
- **abbreviate** (*bool*) – Use abbreviated output if set true
- **suppress_attachments** (*bool*) – When set true attachments subunit_trace will not print attachments on successful test execution.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

stestr.commands.__init__.**slowest_command**(*repo_type='file'*, *repo_url=None*, *show_all=False*, *stdout=<open file '<stdout>', mode 'w'>*)

Print the slowest times from the last run in the repository

This function will print to STDOUT the 10 slowests tests in the last run. Optionally, using the `show_all` argument, it will print all the tests, instead of just 10. sorted by time.

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **show_all** (*bool*) – Show timing for all tests.
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.2.2 stestr failing Command

Show the current failures in the repository.

class stestr.commands.failing.**Failing**(*app*, *app_args*, *cmd_name=None*)

get_description()

Return the command description.

The default is to use the first line of the class' docstring as the description. Set the `_description` class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with `_description` set to a string wrapped with a gettext translation marker.

get_parser(*prog_name*)
Return an `argparse.ArgumentParser`.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

`stestr.commands.failing.`**failing**(*repo_type='file'*, *repo_url=None*, *list_tests=False*, *subunit=False*, *stdout=<open file '<stdout>', mode 'w'>*)
Print the failing tests from the most recent run in the repository

This function will print to STDOUT whether there are any tests that failed in the last run. It optionally will print the test_ids for the failing tests if `list_tests` is true. If `subunit` is true a subunit stream with just the failed tests will be printed to STDOUT.

Note this function depends on the cwd for the repository if *repo_type* is set to file and *repo_url* is not specified it will use the repository located at CWD/.stestr

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **list_test** (*bool*) – Show only a list of failing tests.
- **subunit** (*bool*) – Show output as a subunit stream.
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.2.3 stestr init Command

Initialise a new repository.

class `stestr.commands.init.`**Init**(*app*, *app_args*, *cmd_name=None*)

get_description()
Return the command description.

The default is to use the first line of the class' docstring as the description. Set the `_description` class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with `_description` set to a string wrapped with a gettext translation marker.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

`stestr.commands.init.`**init**(*repo_type='file'*, *repo_url=None*, *stdout=<open file '<stdout>', mode 'w'>*)
Initialize a new repository

This function will create initialize a new repostiory if one does not exist. If one exists the command will fail.

Note this function depends on the cwd for the repository if *repo_type* is set to file and *repo_url* is not specified it will use the repository located at CWD/.stestr

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.2.4 stestr last Command

Show the last run loaded into a repository.

class stestr.commands.last.**Last**(*app*, *app_args*, *cmd_name=None*)

get_description()
Return the command description.

The default is to use the first line of the class' docstring as the description. Set the _description class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with _description set to a string wrapped with a gettext translation marker.

get_parser(*prog_name*)
Return an argparse.ArgumentParser.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

stestr.commands.last.**last**(*repo_type='file'*, *repo_url=None*, *subunit_out=False*, *pretty_out=True*, *color=False*, *stdout=<open file '<stdout>', mode 'w'>*, *suppress_attachments=False*)
Show the last run loaded into a a repository

This function will print the results from the last run in the repository to STDOUT. It can optionally print the subunit stream for the last run to STDOUT if the subunit option is set to true.

Note this function depends on the cwd for the repository if *repo_type* is set to file and *repo_url* is not specified it will use the repository located at CWD/.stestr

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **subunit_out** (*bool*) – Show output as a subunit stream.
- **pretty_out** – Use the subunit-trace output filter.
- **color** – Enable colorized output with the subunit-trace output filter.
- **subunit** (*bool*) – Show output as a subunit stream.
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout
- **suppress_attachments** (*bool*) – When set true attachments subunit_trace will not print attachments on successful test execution.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.2.5 stestr list Command

List the tests from a project and show them.

class stestr.commands.list.**List**(*app*, *app_args*, *cmd_name=None*)

get_description()
Return the command description.

The default is to use the first line of the class' docstring as the description. Set the _description class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with _description set to a string wrapped with a gettext translation marker.

get_parser(*prog_name*)
Return an argparse.ArgumentParser.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

stestr.commands.list.**list_command**(*config='.stestr.conf'*, *repo_type='file'*, *repo_url=None*, *test_path=None*, *top_dir=None*, *group_regex=None*, *blacklist_file=None*, *whitelist_file=None*, *black_regex=None*, *filters=None*, *stdout=<open file '<stdout>', mode 'w'>*)

Print a list of test_ids for a project

This function will print the test_ids for tests in a project. You can filter the output just like with the run command to see exactly what will be run.

Parameters

- **config** (*str*) – The path to the stestr config file. Must be a string.
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **test_path** (*str*) – Set the test path to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.
- **top_dir** (*str*) – The top dir to use for unittest discovery. This takes precedence over the value in the config file. (if one is present in the config file)
- **group_regex** (*str*) – Set a group regex to use for grouping tests together in the stestr scheduler. If both this and the corresponding config file option are set this value will be used.
- **blacklist_file** (*str*) – Path to a blacklist file, this file contains a separate regex exclude on each newline.
- **whitelist_file** (*str*) – Path to a whitelist file, this file contains a separate regex on each newline.
- **black_regex** (*str*) – Test rejection regex. If a test cases name matches on re.search() operation, it will be removed from the final test list.
- **filters** (*list*) – A list of string regex filters to initially apply on the test list. Tests that match any of the regexes will be used. (assuming any other filtering specified also uses it)
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout

5.2.6 stestr load Command

Load data into a repository.

class stestr.commands.load.**Load**(*app*, *app_args*, *cmd_name=None*)

get_description()
Return the command description.

The default is to use the first line of the class' docstring as the description. Set the _description class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with _description set to a string wrapped with a gettext translation marker.

get_parser(*prog_name*)
Return an argparse.ArgumentParser.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

stestr.commands.load.**load**(*force_init=False*, *in_streams=None*, *partial=False*, *subunit_out=False*, *repo_type='file'*, *repo_url=None*, *run_id=None*, *streams=None*, *pretty_out=False*, *color=False*, *stdout=<open file '<stdout>'*, *mode 'w'>*, *abbreviate=False*, *suppress_attachments=False*, *serial=False*)
Load subunit streams into a repository

This function will load subunit streams into the repository. It will output to STDOUT the results from the input stream. Internally this is used by the run command to both output the results as well as store the result in the repository.

Parameters

- **force_init** (*bool*) – Initialize the specifiedrepository if it hasn't been created.
- **in_streams** (*list*) – A list of file objects that will be saved into the repository
- **partial** (*bool*) – DEPRECATED: Specify the input is a partial stream. This option is deprecated and no longer does anything. It will be removed in the future.
- **subunit_out** (*bool*) – Output the subunit stream to stdout
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **run_id** – The optional run id to save the subunit stream to.
- **streams** (*list*) – A list of file paths to read for the input streams.
- **pretty_out** (*bool*) – Use the subunit-trace output filter for the loaded stream.
- **color** (*bool*) – Enabled colorized subunit-trace output
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout
- **abbreviate** (*bool*) – Use abbreviated output if set true
- **suppress_attachments** (*bool*) – When set true attachments subunit_trace will not print attachments on successful test execution.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.2.7 stestr run Command

Run a projects tests and load them into stestr.

class stestr.commands.run.**Run**(*app*, *app_args*, *cmd_name=None*)

get_description()
Return the command description.

The default is to use the first line of the class' docstring as the description. Set the _description class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with _description set to a string wrapped with a gettext translation marker.

get_parser(*prog_name*)
Return an argparse.ArgumentParser.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

stestr.commands.run.**run_command**(*config='.stestr.conf'*, *repo_type='file'*, *repo_url=None*, *test_path=None*, *top_dir=None*, *group_regex=None*, *failing=False*, *serial=False*, *concurrency=0*, *load_list=None*, *partial=False*, *subunit_out=False*, *until_failure=False*, *analyze_isolation=False*, *isolated=False*, *worker_path=None*, *blacklist_file=None*, *whitelist_file=None*, *black_regex=None*, *no_discover=False*, *random=False*, *combine=False*, *filters=None*, *pretty_out=True*, *color=False*, *stdout=<open file '<stdout>'*, *mode 'w'>*, *abbreviate=False*, *suppress_attachments=False*)
Function to execute the run command

This function implements the run command. It will run the tests specified in the parameters based on the provided config file and/or arguments specified in the way specified by the arguments. The results will be printed to STDOUT and loaded into the repository.

Parameters

- **config** (*str*) – The path to the stestr config file. Must be a string.
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **test_path** (*str*) – Set the test path to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.
- **top_dir** (*str*) – The top dir to use for unittest discovery. This takes precedence over the value in the config file. (if one is present in the config file)
- **group_regex** (*str*) – Set a group regex to use for grouping tests together in the stestr scheduler. If both this and the corresponding config file option are set this value will be used.
- **failing** (*bool*) – Run only tests known to be failing.
- **serial** (*bool*) – Run tests serially
- **concurrency** (*int*) – "How many processes to use. The default (0) autodetects your CPU count and uses that.

- **load_list** (*str*) – The path to a list of test_ids. If specified only tests listed in the named file will be run.
- **partial** (*bool*) – DEPRECATED: Only some tests will be run. Implied by *–failing*. This flag is deprecated because and doesn't do anything it will be removed in a future release.
- **subunit_out** (*bool*) – Display results in subunit format.
- **until_failure** (*bool*) – Repeat the run again and again until failure occurs.
- **analyze_isolation** (*bool*) – Search the last test run for 2-test test isolation interactions.
- **isolated** (*bool*) – Run each test id in a separate test runner.
- **worker_path** (*str*) – Optional path of a manual worker grouping file to use for the run.
- **blacklist_file** (*str*) – Path to a blacklist file, this file contains a separate regex exclude on each newline.
- **whitelist_file** (*str*) – Path to a whitelist file, this file contains a separate regex on each newline.
- **black_regex** (*str*) – Test rejection regex. If a test cases name matches on re.search() operation, it will be removed from the final test list.
- **no_discover** (*str*) – Takes in a single test_id to bypasses test discover and just execute the test specified. A file name may be used in place of a test name.
- **random** (*bool*) – Randomize the test order after they are partitioned into separate workers
- **combine** (*bool*) – Combine the results from the test run with the last run in the repository
- **filters** (*list*) – A list of string regex filters to initially apply on the test list. Tests that match any of the regexes will be used. (assuming any other filtering specified also uses it)
- **pretty_out** (*bool*) – Use the subunit-trace output filter
- **color** (*bool*) – Enable colorized output in subunit-trace
- **stdout** (*file*) – The file object to write all output to. By default this is sys.stdout
- **abbreviate** (*bool*) – Use abbreviated output if set true
- **suppress_attachments** (*bool*) – When set true attachments subunit_trace will not print attachments on successful test execution.

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.2.8 stestr slowest Command

Show the longest running tests in the repository.

class stestr.commands.slowest.**Slowest**(*app*, *app_args*, *cmd_name=None*)

get_description()

Return the command description.

The default is to use the first line of the class' docstring as the description. Set the `_description` class attribute to a one-line description of a command to use a different value. This is useful for enabling translations, for example, with `_description` set to a string wrapped with a gettext translation marker.

get_parser(*prog_name*)
Return an `argparse.ArgumentParser`.

take_action(*parsed_args*)
Override to do something useful.

The returned value will be returned by the program.

`stestr.commands.slowest.`**slowest**(*repo_type='file'*, *repo_url=None*, *show_all=False*, *stdout=<open file '<stdout>', mode 'w'>*)
Print the slowest times from the last run in the repository

This function will print to STDOUT the 10 slowests tests in the last run. Optionally, using the `show_all` argument, it will print all the tests, instead of just 10. sorted by time.

Parameters

- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **show_all** (*bool*) – Show timing for all tests.
- **stdout** (*file*) – The output file to write all output to. By default this is sys.stdout

Return return_code The exit code for the command. 0 for success and > 0 for failures.

Return type int

5.3 Internal APIs

The modules in this list do not necessarily have any external api contract, they are intended for internal use inside of stestr. If anything in these provides a stable contract and is intended for usage outside of stestr it will be noted in the api doc.

5.3.1 Configuration File Module

This module is used to deal with anything related to the stestr config file. This includes actually parsing it, and also dealing with interpreting some of it's contents (like generating a test_processor based on a config file's contents).

class `stestr.config_file.`**TestrConf**(*config_file*)
Create a TestrConf object to represent a specified config file

This class is used to represent an stestr config file. It

Parameters **config_file** (*str*) – The path to the config file to use

get_run_command(*test_ids=None*, *regexes=None*, *test_path=None*, *top_dir=None*, *group_regex=None*, *repo_type='file'*, *repo_url=None*, *serial=False*, *worker_path=None*, *concurrency=0*, *blacklist_file=None*, *whitelist_file=None*, *black_regex=None*, *randomize=False*, *parallel_class=None*)
Get a test_processor.TestProcessorFixture for this config file

Any parameters about running tests will be used for initialize the output fixture so the settings are correct when that fixture is used to run tests. Parameters will take precedence over values in the config file.

Parameters

- **options** – A argparse Namespace object of the cli options that were used in the invocation of the original CLI command that needs a TestProcessorFixture

- **test_ids** (*list*) – an optional list of test_ids to use when running tests
- **regexes** (*list*) – an optional list of regex strings to use for filtering the tests to run. See the test_filters parameter in TestProcessorFixture to see how this is used.
- **test_path** (*str*) – Set the test path to use for unittest discovery. If both this and the corresponding config file option are set, this value will be used.
- **top_dir** (*str*) – The top dir to use for unittest discovery. This takes precedence over the value in the config file. (if one is present in the config file)
- **group_regex** (*str*) – Set a group regex to use for grouping tests together in the stestr scheduler. If both this and the corresponding config file option are set this value will be used.
- **repo_type** (*str*) – This is the type of repository to use. Valid choices are 'file' and 'sql'.
- **repo_url** (*str*) – The url of the repository to use.
- **serial** (*bool*) – If tests are run from the returned fixture, they will be run serially
- **worker_path** (*str*) – Optional path of a manual worker grouping file to use for the run.
- **concurrency** (*int*) – How many processes to use. The default (0) autodetects your CPU count and uses that.
- **blacklist_file** (*str*) – Path to a blacklist file, this file contains a separate regex exclude on each newline.
- **whitelist_file** (*str*) – Path to a whitelist file, this file contains a separate regex on each newline.
- **black_regex** (*str*) – Test rejection regex. If a test cases name matches on re.search() operation, it will be removed from the final test list.
- **randomize** (*bool*) – Randomize the test order after they are partitioned into separate workers
- **parallel_class** (*bool*) – Set the flag to group tests together in the stestr scheduler by class. If both this and the corresponding config file option which includes *group-regex* are set, this value will be used.

Returns a TestProcessorFixture object for the specified config file and any arguments passed into this function

Return type *test_processor.TestProcessorFixture*

5.3.2 Test Selection Module

This module is used to deal with anything related to test selection in stestr.

stestr.selection.**construct_list**(*test_ids*, *blacklist_file=None*, *whitelist_file=None*, *regexes=None*, *black_regex=None*)

Filters the discovered test cases

Parameters

- **test_ids** (*list*) – The set of test_ids to be filtered
- **blacklist_file** (*str*) – The path to a blacklist file
- **whitelist_file** (*str*) – The path to a whitelist file

- **regexes** (*list*) – A list of regex filters to apply to the test_ids. The output will contain any test_ids which have a re.search() match for any of the regexes in this list. If this is None all test_ids will be returned
- **black_regex** (*str*) –

Returns iterable of strings. The strings are full test_ids

Return type set

stestr.selection.**filter_tests**(*filters*, *test_ids*)
Filter test_ids by the test_filters.

Parameters

- **filters** (*list*) – A list of regex filters to apply to the test_ids. The output will contain any test_ids which have a re.search() match for any of the regexes in this list. If this is None all test_ids will be returned
- **test_ids** (*list*) – A list of test_ids that will be filtered

Returns A list of test ids.

5.3.3 The Scheduler Module

This module is used to deal with anything related to test scheduling/partitioning in stestr.

stestr.scheduler.**generate_worker_partitions**(*ids*, *worker_path*, *repository=None*, *group_callback=None*, *randomize=False*)
Parse a worker yaml file and generate test groups

Parameters

- **ids** (*list*) – A list of test ids too be partitioned
- **worker_path** (*path*) – The path to a worker file
- **repository** – A repository object that will be used for looking up timing data. This is optional, and also will only be used for scheduling if there is a count field on a worker.
- **group_callback** – A callback function that is used as a scheduler hint to group test_ids together and treat them as a single unit for scheduling. This function expects a single test_id parameter and it will return a group identifier. Tests_ids that have the same group identifier will be kept on the same worker. This is optional and also will only be used for scheduling if there is a count field on a worker.
- **randomize** (*bool*) – If true each partition's test order will be randomized. This is optional and also will only be used for scheduling if there is a count field on a worker.

Returns A list where each element is a distinct subset of test_ids.

stestr.scheduler.**local_concurrency**()
Get the number of available CPUs on the system.

Returns An int for the number of cpus. Or None if it couldn't be found

stestr.scheduler.**partition_tests**(*test_ids*, *concurrency*, *repository*, *group_callback*, *randomize=False*)
Partition test_ids by concurrency.

Test durations from the repository are used to get partitions which have roughly the same expected runtime. New tests - those with no recorded duration - are allocated in round-robin fashion to the partitions created using test durations.

Parameters

- **test_ids** (*list*) – The list of test_ids to be partitioned
- **concurrency** (*int*) – The concurrency that will be used for running the tests. This is the number of partitions that test_ids will be split into.
- **repository** – A repository object that
- **group_callback** – A callback function that is used as a scheduler hint to group test_ids together and treat them as a single unit for scheduling. This function expects a single test_id parameter and it will return a group identifier. Tests_ids that have the same group identifier will be kept on the same worker.
- **randomize** (*bool*) – If true each partition's test order will be randomized

Returns A list where each element is a distinct subset of test_ids, and the union of all the elements is equal to set(test_ids).

5.3.4 The Output Module

This module provides functions for dealing with any output from stestr. This mostly just means helper functions to properly write output to stdout (or another file) Any function or class in this module that has a docstring is a stable interface and should be backwards compatible.

class stestr.output.**ReturnCodeToSubunit**(*process*)

Converts a process return code to a subunit error on the process stdout.

The ReturnCodeToSubunit object behaves as a read-only stream, supplying the read, readline and readlines methods. If the process exits non-zero a synthetic test is added to the output, making the error accessible to subunit stream consumers. If the process closes its stdout and then does not terminate, reading from the ReturnCodeToSubunit stream will hang.

Parameters **process** – A subprocess.Popen object that is generating subunit.

stestr.output.**output_summary**(*successful*, *tests*, *tests_delta*, *time*, *time_delta*, *values*, *output=<open file '<stdout>', mode 'w'>*)

Display a summary view for the test run.

Parameters

- **successful** (*bool*) – Was the test run successful
- **tests** (*int*) – The number of tests that ran
- **tests_delta** (*int*) – The change in the number of tests that ran since the last run
- **time** (*float*) – The number of seconds that it took for the run to execute
- **time_delta** (*float*) – The change in run time since the last run
- **values** – A list of sets that are used for a breakdown of statuses other than success. Each set is in the format: (status, number of tests, change in number of tests).
- **output** – The output file object to use. This defaults to stdout

stestr.output.**output_table**(*table*, *output=<open file '<stdout>', mode 'w'>*)

Display a table of information.

Parameters

- **table** – A list of sets representing each row in the table. Each element in the set represents a column in the table.

- **output** – The output file object to write the table to. By default this is sys.stdout

stestr.output.**output_tests**(*tests*, *output=<open file '<stdout>', mode 'w'>*)
Display a list of tests.

Parameters

- **tests** – A list of test objects to output
- **output** – The output file object to write the list to. By default this is sys.stdout

5.3.5 Test Processor Module

This module contains the definition of the `TestProcessorFixture` fixture class. This fixture is used for handling the actual spawning of worker processes for running tests, or listing tests. It is constructed as a fixture to handle the lifecycle of the test id list files which are used to pass test ids to the workers processes running the tests.

In the normal workflow a `TestProcessorFixture` get's returned by the *Configuration File Module*'s `get_run_command()` function. The config file parses the config file and the cli options to create a `TestProcessorFixture` with the correct options. This Fixture then gets returned to the CLI commands to enable them to run the commands.

The `TestProcessorFixture` class is written to be fairly generic in the command it's executing. This is an artifact of being forked from testrepository where the test command is defined in the configuration file. In stestr the command is hard coded `stestr.config_file` module so this extra flexibility isn't really needed.

API Reference

class stestr.test_processor.**TestProcessorFixture**(*test_ids*, *cmd_template*, *listopt*, *idoption*, *repository*, *parallel=True*, *listpath=None*, *test_filters=None*, *group_callback=None*, *serial=False*, *worker_path=None*, *concurrency=0*, *black-list_file=None*, *black_regex=None*, *whitelist_file=None*, *random-ize=False*)
Write a temporary file to disk with test ids in it.

The TestProcessorFixture is used to handle the lifecycle of running the subunit.run commands. A fixture is used for this class to handle the temporary list files creation.

Parameters

- **test_ids** – The test_ids to use. May be None indicating that no ids are known and they should be discovered by listing or configuration if they must be known to run tests. Test ids are needed to run tests when filtering or partitioning is needed: if the run concurrency is > 1 partitioning is needed, and filtering is needed if the user has passed in filters.
- **cmd_template** – string to be used for the command that will be filled out with the IDFILE when it is created.
- **listopt** – Option to substitute into LISTOPT to cause test listing to take place.
- **idoption** – Option to substitute into cmd when supplying any test ids.
- **repository** – The repository to query for test times, if needed.
- **parallel** – If not True, prohibit parallel use : used to implement –parallel run recursively.

- **listpath** – The file listing path to use. If None, a unique path is created.
- **test_filters** – An optional list of test filters to apply. Each filter should be a string suitable for passing to re.compile. Filters are applied using search() rather than match(), so if anchoring is needed it should be included in the regex. The test ids used for executing are the union of all the individual filters: to take the intersection instead, craft a single regex that matches all your criteria. Filters are automatically applied by run_tests(), or can be applied by calling filter_tests(test_ids).
- **group_callback** – If supplied, should be a function that accepts a test id and returns a group id. A group id is an arbitrary value used as a dictionary key in the scheduler. All test ids with the same group id are scheduled onto the same backend test process.
- **serial** (*bool*) – Run tests serially
- **worker_path** (*path*) – Optional path of a manual worker grouping file to use for the run
- **concurrency** (*int*) – How many processes to use. The default (0) autodetects your CPU count and uses that.
- **blacklist_file** (*path*) – Path to a blacklist file, this file contains a separate regex exclude on each newline.
- **whitelist_file** (*path*) – Path to a whitelist file, this file contains a separate regex on each newline.
- **randomize** (*boolean*) – Randomize the test order after they are partitioned into separate workers

list_tests()
List the tests returned by list_cmd.

Returns A list of test ids.

run_tests()
Run the tests defined by the command

Returns A list of spawned processes.

setUp()
Prepare the Fixture for use.

This should not be overridden. Concrete fixtures should implement _setUp. Overriding of setUp is still supported, just not recommended.

After setUp has completed, the fixture will have one or more attributes which can be used (these depend totally on the concrete subclass).

Raises MultipleExceptions if _setUp fails. The last exception captured within the MultipleExceptions will be a SetupError exception.

Returns None.

Changed in 1.3 The recommendation to override setUp has been reversed - before 1.3, setUp() should be overridden, now it should not be.

Changed in 1.3.1 BaseException is now caught, and only subclasses of Exception are wrapped in MultipleExceptions.

5.3.6 Subunit Trace

Trace a subunit stream in reasonable detail and high accuracy.

stestr.subunit_trace.**cleanup_test_name**(*name*, *strip_tags=True*, *strip_scenarios=False*)
Clean up the test name for display.

By default we strip out the tags in the test because they don't help us in identifying the test that is run to it's result.

Make it possible to strip out the testscenarios information (not to be confused with tempest scenarios) however that's often needed to indentify generated negative tests.

stestr.subunit_trace.**find_worker**(*test*)
Get the worker number.

If there are no workers because we aren't in a concurrent environment, assume the worker number is 0.

stestr.subunit_trace.**print_attachments**(*stream*, *test*, *all_channels=False*)
Print out subunit attachments.

Print out subunit attachments that contain content. This runs in 2 modes, one for successes where we print out just stdout and stderr, and an override that dumps all the attachments.

stestr.subunit_trace.**print_fails**(*stream*)
Print summary failure report.

Currently unused, however there remains debate on inline vs. at end reporting, so leave the utility function for later use.

CHAPTER 6

Indices and tables

- genindex

Python Module Index

S

Index

Symbols

A

C

F

G

I

T