
Slash Documentation

Release 1.5.2.dev23

Rotem Yaari

Apr 25, 2018

Contents

| | | |
|----------|---|-----------|
| 1 | What is Slash? | 1 |
| 2 | Diving in | 3 |
| 2.1 | As a Test Author | 3 |
| 2.2 | As a Framework Developer | 3 |
| 3 | Table Of Contents | 5 |
| 3.1 | Getting Started with Slash | 5 |
| 3.2 | Running Tests | 10 |
| 3.3 | Test Parametrization | 12 |
| 3.4 | Test Tags | 15 |
| 3.5 | Test Fixtures | 16 |
| 3.6 | Assertions, Exceptions and Errors | 21 |
| 3.7 | Warnings | 26 |
| 3.8 | Customizing and Extending Slash | 26 |
| 3.9 | Configuration | 31 |
| 3.10 | Logging | 42 |
| 3.11 | Saving Test Details | 44 |
| 3.12 | Hooks | 45 |
| 3.13 | Plugins | 49 |
| 3.14 | Built-in Plugins | 53 |
| 3.15 | Slash Internals | 54 |
| 3.16 | Misc. Features | 55 |
| 3.17 | Advanced Use Cases | 56 |
| 3.18 | Cookbook | 58 |
| 3.19 | FAQ | 59 |
| 3.20 | API Documentation | 60 |
| 3.21 | Changelog | 69 |
| 3.22 | Development | 79 |
| 3.23 | Contributors | 79 |
| 3.24 | Unit Testing Slash | 79 |
| 3.25 | Parallel Test Execution | 81 |
| 4 | Indices and tables | 83 |
| | Python Module Index | 85 |

CHAPTER 1

What is Slash?

Slash is a testing framework written in Python. Unlike many other testing frameworks out there, Slash focuses on building in-house testing solutions for large projects. It provides facilities and best practices for testing complete products, and not only unit tests for individual modules.

Slash provides several key features:

- A solid execution model based on fixtures, test factories and tests. This provides you with the flexibility you need to express your testing logic.
- Easy ways for extending the core functionality, adding more to the global execution environment and controlling how your tests interact with it.
- A rich configuration mechanism, helping you setting up your environment parameters and their various flavours.
- A plugin architecture, greatly simplifying adding extra functionality to your framework.

2.1 As a Test Author

If you only want to write tests for running with Slash, you should head first to the *Writing Tests* section which should help you get started.

2.2 As a Framework Developer

If you are looking to integrate Slash into your testing ecosystem, or want to learn how to extend its functionality and adapt it to specific purposes, head to the *Customizing and Extending Slash* section.

3.1 Getting Started with Slash

3.1.1 Writing Tests

Slash loads and runs tests from Python files. To get started, let's create an example test file and name it `test_addition.py`:

```
# test_addition.py

import slash

def test_addition():
    pass
```

As you can see in the above example, Slash can load tests written as functions. Similarly to `unittest` and `py.test`, only functions starting with the prefix `test_` are assumed to be runnable tests.

3.1.2 Running Tests

Once we have our file written, we can run it using `slash run`:

```
$ slash run test_addition.py
```

There's a lot to cover regarding `slash run`, and we will get to it *soon enough*. For now all we have to know is that it finds, loads and runs the tests in the files or directories we provide, and reports the result.

A single run of `slash run` is called a *session*. A session contains tests that were run in its duration.

Debugging

You can debug failing tests using the `--pdb` flag, which automatically runs the best available debugger on exceptions. You can also filter the exceptions which run the debugger by using `--pdb-filter` in addition to the `--pdb` flag.

See also:

Handling and Debugging Exceptions

3.1.3 Assertions and Errors

Tests don't do much without making sure things are like they expect. Slash borrows the awesome technology behind `py.test`, allowing us to just write assert statements where we want to test conditions of all sorts:

```
# test_addition.py

def test_addition():
    assert 2 + 2 == 4
```

Slash also analyzes assertions using assertion rewriting borrowed from the [pytest project](#), so you can get more details as for what exactly failed.

See also:

errors

3.1.4 Test Parameters

Slash tests can be easily parametrized, iterating parameter values and creating separate cases for each value:

```
@slash.parametrize('x', [1, 2, 3])
def test_something(x):
    # use x here
```

For boolean values, a shortcut exists for toggling between True and False:

```
@slash.parameters.toggle('with_power_operator')
def test_power_of_two(with_power_operator):
    num = 2
    if with_power_operator:
        result = num ** 2
    else:
        result = num * num
    assert result == 4
```

See also:

Test Parametrization

3.1.5 Logging

Testing complete products usually means you may not have a second chance to reproduce an issue. This is why Slash puts a strong emphasis on logging, managing log files and directories, and fine tuning your logging setup.

Slash uses [Logbook](#) for logging. It has many advantages over Python's own logging package, and is much more flexible.

Slash exposes a global logger intended for tests, which is recommended for use in simple logging tasks:

```
import slash

def test_1():
    slash.logger.debug("Hello!")
```

Console Log

By default logs above **WARNING** get emitted to the console when `slash run` is executed. You can use `-v/-q` to increase/decrease console verbosity accordingly.

Saving Logs to Files

By default logs are not saved anywhere. This is easily changed with the `-l` flag to `slash run`. Point this flag to a directory, and Slash will organize logs inside, in subdirectories according to the session and test run (e.g. `/path/to/logdir/<session id>/<test id>/debug.log`).

See also:

Logging

3.1.6 Cleanups

Slash provides a facility for cleanups. These get called whenever a test finishes, successfully or not. Adding cleanups is done with `slash.add_cleanup()`:

```
def test_product_power_on_sequence():
    product = ...
    product.plug_to_outlet()
    slash.add_cleanup(product.plug_out_of_outlet)
    product.press_power()
    slash.add_cleanup(product.wait_until_off)
    slash.add_cleanup(product.press_power)
    slash.add_cleanup(product.pack_for_shipping, success_only=True)
    product.wait_until_on()
```

Note: When a test is interrupted, most likely due to a `KeyboardInterrupt`, cleanups are not called unless added with the `critical` keyword argument. This is in order to save time during interruption handling. See *interruptions*.

Note: A cleanup added with `success_only=True` will be called only if the test ends successfully

Cleanups also receive an optional `scope` parameter, which can be either `'session'`, `'module'` or `'test'` (the default). The `scope` parameter controls *when* the cleanup should take place. *Session* cleanups happen at the end of the test session, *module* cleanups happen before Slash switches between test files during execution and *test* cleanups happen at the end of the test which added the cleanup callback.

3.1.7 Skips

In some case you want to skip certain methods. This is done by raising the *SkipTest* exception, or by simply calling *slash.skip_test()* function:

```
def test_microwave_has_supercool_feature():
    if microwave.model() == "Microtech Shitbox":
        slash.skip_test("Microwave model too old")
```

Slash also provides *slash.skipped()*, which is a decorator to skip specific tests:

```
@slash.skipped("reason")
def test_1():
    # ...

@slash.skipped # no reason
def test_2():
    # ...
```

In some cases you may want to register a custom exception to be recognized as a skip. You can do this by registering your exception type first with *slash.register_skip_exception()*.

3.1.8 Requirements

In many cases you want to depend in our test on a certain precondition in order to run. Requirements provide an explicit way of stating those requirements. Use *slash.requires()* to specify requirements:

```
def is_some_condition_met():
    return True

@slash.requires(is_some_condition_met)
def test_something():
    ...
```

Requirements are stronger than skips, since they can be reported separately and imply a basic precondition that is not met in the current testing environment.

slash.requires can receive either:

1. A boolean value (useful for computing on import-time)
2. A function returning a boolean value, to be called when loading tests
3. A function returning a tuple of (boolean, message) - the message being the description of the unmet requirements when *False* is returned

When a requirement fails, the test is skipped without even being started, and appears in the eventual console summary along with the unmet requirements. If you want to control the message shown if the requirement is not met, you can pass the message parameter:

```
@slash.requires(is_some_condition_met, message='My condition is not met!')
def test_something():
    ...
```

Note: Requirements are evaluated during the load phase of the tests, so they are usually checked before any test started running. This means that if you're relying on a transient state that can be altered by other tests, you have to use

skips instead. Requirements are useful for checking environmental constraints that are unlikely to change as a result of the session being run.

3.1.9 Storing Additional Test Details

It is possible for a test to store some objects that may help investigation in cause of failure.

This is possible using the `slash.set_test_detail()` method. This method accepts a hashable key object and a printable object. In case the test fails, the stored objects will be printed in the test summary:

```
def test_one():
    slash.set_test_detail('log', '/var/log/foo.log')
    slash.set_error("Some condition is not met!")

def test_two():
    # Every test has its own unique storage, so it's possible to use the same key in
    ↪ multiple tests
    slash.set_test_detail('log', '/var/log/bar.log')
```

In this case we probably won't see the details of `test_two`, as it should finish successfully.

`slash.set_test_detail(key, value)`

Store an object providing additional information about the current running test in a certain key. Each test has its own storage.

Parameters

- **key** – a hashable object
- **value** – can be either an object or a string representing additional details

3.1.10 Global State

Slash maintains a set of globals for convenience. The most useful one is `slash.g`, which is an attribute holder that can be used to hold environment objects set up by plugins or hooks for use in tests.

3.1.11 Misc. Utilities

Repeating Tests

Use the `slash.repeat()` decorator to make a test repeat several times:

```
@slash.repeat(5)
def test_probabilistic():
    assert still_works()
```

Note: You can also use the `--repeat-each=X` argument to `slash run`, causing it to repeat each test being loaded a specified amount of times, or `--repeat-all=X` to repeat the entire suite several times

3.2 Running Tests

The main front-end for Slash is the `slash run` utility, invoked from the command line. It has several interesting options worth mentioning.

By default, it receives the path to load and run tests from:

```
$ slash run /path/to/tests
```

3.2.1 Verbosity

Verbosity is increased with `-v` and decreased with `-q`. Those can be specified multiple times.

In addition to the verbosity itself, tracebacks which are displayed at the session summary can be controlled via the `--tb` flag, specifying the verbosity level of the tracebacks. 0 means no tracebacks, while 5 means the highest detail available.

See also:

Logging

3.2.2 Loading Tests from Files

You can also read tests from file or files which contain paths to run. Whitespaces and lines beginning with a comment `#` will be ignored:

```
$ slash run -f file1.txt -f file2.txt
```

Lines in suite files can optionally contain filters and repeat directive.

Filter allows restricting the tests actually loaded from them:

```
# my_suite_file.txt
# this is the first test file
/path/to/tests.py
# when running the following file, tests with "dangerous" in their name will not be
↳loaded
/path/to/other_tests.py # filter: not dangerous
```

See also:

The filter syntax is exactly like `-k` described below

Repeat allows to repeat a line:

```
# my_suite_file.txt
# the next line will be repeated twice
/path/to/other_tests.py # repeat: 2
# you can use filter and repeat together
/path/to/other_tests.py # filter: not dangerous, repeat: 2
```

3.2.3 Debugging & Failures

Debugging is done with `--pdb`, which invokes the best debugger available.

Stopping at the first unsuccessful test is done with the `-x` flag.

See also:

Handling and Debugging Exceptions

3.2.4 Including and Excluding Tests

The `-k` flag to `slash run` is a versatile way to include or exclude tests. Provide it with a substring to only run tests containing the substring in their names:

```
$ slash run -k substr /path/to/tests
```

Use `not X` to exclude any test containing `X` in their names:

```
$ slash run -k 'not failing_' /path/to/tests
```

Or use a more complex expression involving `or` and `and`:

```
$ slash run -k 'not failing_ and components' /path/to/tests
```

The above will run all tests with `components` in their name, but without `failing_` in it.

3.2.5 Overriding Configuration

The `-o` flag enables us to override specific paths in the configuration, properly converting their respective types:

```
$ slash run -o path.to.config.value=20 ...
```

See also:

configuration

3.2.6 Running Interactively

As a part of the development cycle, it is often useful or even necessary to run your infrastructure in interactive mode. This allows users to experiment with your framework and learn how to use it.

Slash supports running interactively out-of-the-box, using the `-i` flag to `slash run`:

```
$ slash run -i
```

This will invoke an interactive IPython shell, initialized with your project's environment (and, of course, a valid Slash session).

By default, the namespace in which the interactive test runs contains all content of the `slash.g` global container. You can disable this behavior by setting `interactive.expose_g_globals` to `False`.

See also:

Controlling Interactive Namespaces from Plugins

3.2.7 Resuming Previous Sessions

When you run a session that fails, Slash automatically saves the tests intended to be run for later reference. For quickly retrying a previously failed session, skipping tests which had already passed, you can use `slash resume`:

```
$ slash resume -vv <session id>
```

This command receives all flags which can be passed to `slash run`, but receives an id of a previously run session for resuming.

3.2.8 Rerunning Previous Sessions

You can rerun all the tests of a previous session, given the session's tests were reported. This might be helpful when reproducing a run of specific worker, for example. You can use `slash rerun`:

```
$ slash rerun -vv <session id>
```

This command receives all flags which can be passed to `slash run`, but receives an id of a previously run session for rerunning.

3.3 Test Parametrization

3.3.1 Using `slash.parametrize`

Use the `slash.parametrize()` decorator to multiply a test function for different parameter values:

```
@slash.parametrize('x', [1, 2, 3])
def test_something(x):
    pass
```

The above example will yield 3 test cases, one for each value of `x`. Slash also supports parametrizing the `before` and `after` methods of test classes, thus multiplying each case by several possible setups:

```
class SomeTest(Test):
    @slash.parametrize('x', [1, 2, 3])
    def before(self, x):
        # ...

    @slash.parametrize('y', [4, 5, 6])
    def test(self, y):
        # ...

    @slash.parametrize('z', [7, 8, 9])
    def after(self, z):
        # ...
```

The above will yield 27 different runnable tests, one for each cartesian product of the `before`, `test` and `after` possible parameter values.

This also works across inheritance. Each base class can parametrize its `before` or `after` methods, multiplying the number of variations actually run accordingly. Calls to `super` are handled automatically in this case:

```
class BaseTest(Test):

    @slash.parametrize('base_parameter', [1, 2, 3])
    def before(self, base_parameter):
        # ....
```

(continues on next page)

(continued from previous page)

```
class DerivedTest(BaseTest):

    @slash.parametrize('derived_parameter', [4, 5, 6])
    def before(self, derived_parameter):
        super(DerivedTest, self).before() # note that base parameters aren't
        ↪ specified here
        # .....
```

3.3.2 More Parametrization Shortcuts

In addition to `slash.parametrize()`, Slash also supports `slash.parameters.toggle` as a shortcut for toggling a boolean flag in two separate cases:

```
@slash.parameters.toggle('with_safety_switch')
def test_operation(with_safety_switch):
    ...
```

Another useful shortcut is `slash.parameters.iterate`, which is an alternative way to specify parametrizations:

```
@slash.parameters.iterate(x=[1, 2, 3], y=[4, 5, 6])
def test_something(x, y):
    ...
```

3.3.3 Specifying Multiple Arguments at Once

You can specify dependent parameters in a way that forces them to receive related values, instead of a simple cartesian product:

```
@slash.parametrize(('fruit', 'color'), [('apple', 'red'), ('apple', 'green'), ('banana
    ↪', 'yellow')])
def test_fruits(fruit, color):
    ... # <-- this never gets a yellow apple
```

3.3.4 Labeling Parameters

By default, parameters are being designated by their ordinal number, starting with zero. This means that the following test:

```
@slash.parametrize('param', [Object1(), Object2()])
def test_something(param):
    ...
```

This will generate tests named `test_something(param=param0)` and `test_something(param=param1)`. This is not very useful for most cases – as the tests should be indicative of their respective parametrization flavors.

To cope with this, Slash supports *parametrization labels*. This can be done as follows:

```
@slash.parametrize('param', [
    slash.param('first', Object1()),
    slash.param('second', Object2()),
])
def test_something(param):
    ...
```

The above will generate tests named `test_something(param=first)` and `test_something(param=second)`, which, given descriptive labels, should differentiate the cases more clearly.

The labeling mechanism has a second possible syntactic shortcut, for developers preferring the value to appear first:

```
@slash.parametrize('param', [
    Object1() // slash.param('first'),
    Object2() // slash.param('second'),
])
def test_something(param):
    ...
```

The two forms are functionally equivalent.

Note: Label names are limited to 30 characters, and are under the same naming constraints as Python variables. This is intentional, and is intended to avoid abuse and keep labels concise.

3.3.5 Excluding Parameter Values

You can easily skip specific values from parametrizations in tests through `slash.exclude`:

```
import slash

SUPPORTED_SIZES = [10, 15, 20, 25]

@slash.parametrize('size', SUPPORTED_SIZES)
@slash.exclude('size', [10, 20])
def test_size(size): # <-- will be skipped for sizes 10 and 20
    ...
```

This also works for parameters of fixtures (for more information about fixtures see [the fixtures chapter](#))

```
import slash

SUPPORTED_SIZES = [10, 15, 20, 25]

@slash.exclude('car.size', [10, 20])
def test_car(car):
    ...

@slash.parametrize('size', SUPPORTED_SIZES)
@slash.fixture
def car(size): # <-- will be skipped for sizes 10 and 20
    ...
```

3.4 Test Tags

3.4.1 Tagging Tests

Slash supports organizing tests by tagging them. This is done using the `slash.tag()` decorator:

```
@slash.tag('dangerous')
def test_something():
    ...
```

You can also have tag decorators prepared in advance for simpler usage:

```
dangerous = slash.tag('dangerous')
...

@dangerous
def test_something():
    ...
```

Tags can also have values:

```
@slash.tag('covers', 'requirement_1294')
def test_something():
    ...
```

3.4.2 Filtering Tests by Tags

When running tests you can select by tags using the `-k` flag. A simple case would be matching a tag substring (the same way the test name is matched):

```
$ slash run tests -k dangerous
```

This would work, but will also select tests whose names contain the word ‘dangerous’. Prefix the argument with `tag:` to only match tags:

```
$ slash run tests -k tag:dangerous
```

Combined with the regular behavior of `-k` this yields a powerful filter:

```
$ slash run tests -k 'microwave and power and not tag:dangerous'
```

Filtering by value is also supported:

```
$ slash run test -k covers=requirement_1294
```

Or:

```
$ slash run test -k tag:covers=requirement_1294
```

3.5 Test Fixtures

Slash includes a powerful mechanism for parametrizing and composing tests, called *fixtures*. This feature resembles, and was greatly inspired by, the feature of the same name in **py.test**.

To demonstrate this feature we will use *test functions*, but it also applies to test methods just the same.

3.5.1 What is a Fixture?

A *fixture* refers to a certain piece of setup or data that your test requires in order to run. It generally does not refer to the test itself, but the base on which the test builds to carry out its work.

Slash represents fixtures in the form of arguments to your test function, thus denoting that your test function needs this fixture in order to run:

```
def test_microwave_turns_on(microwave):
    microwave.turn_on()
    assert microwave.get_state() == STATE_ON
```

So far so good, but what exactly is *microwave*? Where does it come from?

The answer is that Slash is responsible of looking up needed fixtures for each test being run. Each function is examined, and telling by its arguments, Slash goes ahead and looks for a fixture definition called *microwave*.

3.5.2 The Fixture Definition

The fixture definition is where the logic of your fixture goes. Let's write the following somewhere in your file:

```
import slash

...

@slash.fixture
def microwave():
    # initialization of the actual microwave instance
    return Microwave(...)
```

In addition to the test file itself, you can also put your fixtures in a file called *slashconf.py*, and put it in your test directory. Multiple such files can exist, and a test automatically “inherits” fixtures from the entire directory hierarchy above it.

3.5.3 Fixture Cleanups

You can control what happens when the lifetime of your fixture ends. By default, this happens at the end of each test that requested your fixture. To do this, add an argument for your fixture called *this*, and call its *add_cleanup* method with your cleanup callback:

```
@slash.fixture
def microwave(this):
    returned = Microwave()
    this.add_cleanup(returned.turn_off)
    return returned
```

Note: This variable is also available globally while computing each fixture as the `slash.context.fixture` global variable.

3.5.4 Opting Out of Fixtures

In some cases you may want to turn off Slash's automatic deduction of parameters as fixtures. For instance in the following case you want to explicitly call a version of a base class's `before` method:

```
>>> class BaseTest(slash.Test):
...     def before(self, param):
...         self._construct_case_with(param)

>>> class DerivedTest(BaseTest):
...     @slash.parametrize('x', [1, 2, 3])
...     def before(self, x):
...         param_value = self._compute_param(x)
...         super(DerivedTest, self).before(x)
```

This case would fail to load, since Slash will assume `param` is a fixture name and will not find such a fixture to use. The solution is to use `slash.nofixtures()` on the parent class's `before` method to mark that `param` is *not* a fixture name:

```
>>> class BaseTest(slash.Test):
...     @slash.nofixtures
...     def before(self, param):
...         self._construct_case_with(param)
```

3.5.5 Fixture Needing Other Fixtures

A fixture can depend on other fixtures just like a test depends on the fixture itself, for instance, here is a fixture for a heating plate, which depends on the type of microwave we're testing:

```
@slash.fixture
def heating_plate(microwave):
    return get_appropriate_heating_plate_for(microwave)
```

Slash takes care of spanning the fixture dependency graph and filling in the values in the proper order. If a certain fixture is needed in multiple places in a single test execution, it is guaranteed to return the same value:

```
def test_heating_plate_usage(microwave, heating_plate):
    # we can be sure that heating_plate matches the microwave,
    # since `microwave` will return the same value for the test
    # and for the fixture
```

3.5.6 Fixture Parametrization

Fixtures become interesting when you parametrize them. This enables composing many variants of tests with a very little amount of effort. Let's say we have many kinds of microwaves, we can easily parametrize the microwave class:

```
@slash.fixture
@slash.parametrize('microwave_class', [SimpleMicrowave, AdvancedMicrowave]):
def microwave(microwave_class, this):
    returned = microwave_class()
    this.add_cleanup(returned.turn_off)
    return returned
```

Now that we have a parametrized fixture, Slash takes care of multiplying the test cases that rely on it automatically. The single test we wrote in the beginning will now cause two actual test cases to be loaded and run – one with a simple microwave and one with an advanced microwave.

As you add more parametrizations into dependent fixtures in the dependency graph, the actual number of cases being run eventually multiplies in a cartesian manner.

3.5.7 Fixture Requirements

It is possible to specify requirements for fixture functions, very much like *test requirements*. Fixtures for which requirements are not met will prevent their dependent tests from being run, being skipped instead:

```
@slash.fixture
@slash.requires(condition, 'Requires a specific flag')
def some_fixture():
    ...
```

See also:

Requirements

3.5.8 Fixture Scopes

By default, a fixture “lives” through only a single test at a time. This means that:

1. The fixture function will be called again for each new test needing the fixture
2. If any cleanups exist, they will be called at the end of each test needing the fixture.

We say that fixtures, by default, have a **scope of a single test**, or *test scope*.

Slash also supports *session* and *module* scoped fixtures. *Session fixtures* live from the moment of their activation until the end of the test session, while *module fixtures* live until the last test of the module that needed them finished execution. Specifying the scope is rather straightforward:

```
@slash.fixture(scope='session')
def some_session_fixture(this):
    @this.add_cleanup
    def cleanup():
        print('Hurray! the session has ended')

@slash.fixture(scope='module')
def some_module_fixture(this):
    @this.add_cleanup
    def cleanup():
        print('Hurray! We are finished with this module')
```

Test Start/End for Widely Scoped Fixtures

When a fixture is scoped wider than a single test, it is useful to add custom callbacks to the fixtures to be called when a test starts or ends. This is done via the `this.test_start` and `this.test_end` callbacks, which are specific to the current fixture.

```
@slash.fixture(scope='module')
def background_process(this):
    process = SomeComplexBackgroundProcess()

    @this.test_start
    def on_test_start():
        process.make_sure_still_running()

    @this.test_end
    def on_test_end():
        process.make_sure_no_errors()

    process.start()

    this.add_cleanup(process.stop)
```

Note: Exceptions propagating out of the `test_start` or `test_end` hooks will fail the test, possibly preventing it from starting properly

3.5.9 Autouse Fixtures

You can also “force” a fixture to be used, even if it is not required by any function argument. For instance, this example creates a temporary directory that is deleted at the end of the session:

```
@slash.fixture(autouse=True, scope='session')
def temp_dir():
    """Create a temporary directory"""
    directory = '/some/directory'
    os.makedirs(directory)

    @this.add_cleanup
    def cleanup():
        shutil.rmtree(directory)
```

3.5.10 Aliasing Fixtures

In some cases you may want to name your fixtures descriptively, e.g.:

```
@slash.fixture
def microwave_with_up_to_date_firmware(microwave):
    microwave.update_firmware()
    return microwave
```

Although this is a very nice practice, it makes tests clumsy and verbose:

```
def test_turning_off(microwave_with_up_to_date_firmware):
    microwave_with_up_to_date_firmware.turn_off()
    assert microwave_with_up_to_date_firmware.is_off()
    microwave_with_up_to_date_firmware.turn_on()
```

Fortunately, Slash allows you to *alias* fixtures, using the `slash.use()` shortcut:

```
def test_turning_off(m: slash.use('microwave_with_up_to_date_firmware')):
    m.turn_off()
    assert m.is_off()
    m.turn_on()
```

Note: Fixture aliases require Python 3.x, as they rely on [function argument annotation](#)

3.5.11 Misc. Utilities

Yielding Fixtures

Fixtures defined as generators are automatically detected by Slash. In this mode, the fixture is run as a generator, with the yielded value acting as the fixture value. Code after the yield is treated as cleanup code (similar to using `this.add_cleanup()`):

```
@slash.fixture
def microwave(model_name):
    m = Microwave(model_name)
    yield m
    m.turn_off()
```

Generator Fixtures

`slash.generator_fixture()` is a shortcut for a fixture returning a single parametrization:

```
@slash.generator_fixture
def model_types():
    for model_config in all_model_configs:
        if model_config.supported:
            yield model_config.type
```

In general, this form:

```
@slash.generator_fixture
def fixture():
    yield from x
```

is equivalent to this form:

```
@slash.fixture
@slash.parametrize('param', x)
def fixture(param):
    return param
```


3.5.12 Listing Available Fixtures

Slash can be invoked with the `list` command and the `--only-fixtures` flag, which takes a path to a testing directory. This command gets the available fixtures for the specified testing directory:

```
$ slash list --only-fixtures path/to/tests
```

temp_dir Create a temporary directory

Source: path/to/tests/utilities.py:8

3.6 Assertions, Exceptions and Errors

3.6.1 Assertions

Assertions are the bread and butter of tests. They ensure constraints are held and that conditions are met:

```
# test_addition.py

def test_addition(self):
    assert 2 + 2 == 4
```

When assertions fail, the assertion rewriting code Slash uses will help you understand what exactly happened. This also applies for much more complex expressions:

```
...
assert f(g(x)) == g(f(x + 1))
...
```

When the above assertion fails, for instance, you can expect an elaborate output like the following:

```
>      assert f(g(x)) == g(f(x + 1))
F      AssertionError: assert 1 == 2
+ where 1 = <function f at 0x10b10f848>(1)
+   where 1 = <function g at 0x10b10f8c0>(1)
+ and     2 = <function g at 0x10b10f8c0>(2)
+   where 2 = <function f at 0x10b10f848>((1 + 1))
```

Note: The assertion rewriting code is provided by `dessert`, which is a direct port of the code that powers `pytest`. All credit goes to Holger Krekel and his fellow devs for this masterpiece.

Note: By default, even asserts with accompanied messages will emit introspection information. This can be overridden through the `run.message_assertion_introspection` configuration flag.

New in version 1.3.0.

More Assertion Utilities

One case that is not easily covered by the `assert` statement is asserting Exception raises. This is easily done with `slash.assert_raises()`:

```
with slash.assert_raises(SomeException) as caught:
    some_func()

assert caught.exception.param == 'some_value'
```

`slash.assert_raises()` will raise `ExpectedExceptionNotCaught` exception in case the expected exception was not raised:

```
>>> with slash.assert_raises(Exception) as caught:
...     pass
Traceback (most recent call last):
...
ExpectedExceptionNotCaught: ...
```

In a case where the test author wants to allow a specific exception but not to enforce its propagation (e.g. allowing a timing issue to be present), `slash.allowing_exceptions()` can be used.

```
>>> with slash.allowing_exceptions(Exception) as caught:
...     pass
```

You also have `slash.assert_almost_equal()` to test for near equality:

```
slash.assert_almost_equal(1.001, 1, max_delta=0.1)
```

Note: `slash.assert_raises()` and `slash.allowing_exceptions()` interacts with `handling_exceptions()` - exceptions anticipated by `assert_raises` or `allowing_exceptions` will be ignored by `handling_exceptions`.

3.6.2 Errors

Any exception which is not an assertion is considered an ‘error’, or in other words, an unexpected error, failing the test. Like many other testing frameworks Slash distinguishes failures from errors, the first being anticipated while the latter being unpredictable. For most cases this distinction is not really important, but exists nonetheless.

Any exceptions thrown from a test will be added to the test result as an error, thus marking the test as ‘error’.

3.6.3 Interruptions

Usually when a user hits Ctrl+C this means he wants to terminate the running program as quickly as possible without corruption or undefined state. Slash treats `KeyboardInterrupt` a bit differently than other exceptions, and tries to quit as quickly as possible when they are encountered.

Note: `KeyboardInterrupt` also causes regular cleanups to be skipped. You can set critical cleanups to be carried out on both cases, as described in the [relevant section](#).

3.6.4 Explicitly Adding Errors and Failures

Sometimes you would like to report errors and failures in mid-test without failing it immediately (letting it run to the end). This is good when you want to collect all possible failures before officially quitting, and this is more helpful for reporting.

This is possible using the `slash.add_error()` and `slash.add_failure()` methods. They can accept strings (messages) or actual objects to be kept for reporting. It is also possible to add more than one failure or error for each test.

```
class MyTest(slash.Test):

    def test(self):
        if not some_condition():
            slash.add_error("Some condition is not met!")

        # code keeps running here...
```

`slash.add_error(msg=None, frame_correction=0, exc_info=None)`
Adds an error to the current test result

Parameters

- **msg** – can be either an object or a string representing a message
- **frame_correction** – when delegating `add_error` from another function, specifies the amount of frames to skip to reach the actual cause of the added error
- **exc_info** – (optional) - the `exc_info` tuple of the exception being recorded

`slash.add_failure(msg=None, frame_correction=0, exc_info=None)`
Adds a failure to the current test result

Parameters

- **msg** – can be either an object or a string representing a message
- **frame_correction** – when delegating `add_failure` from another function, specifies the amount of frames to skip to reach the actual cause of the added failure

3.6.5 Handling and Debugging Exceptions

Exceptions are an important part of the testing workflow. They happen all the time – whether they indicate a test lifetime event or an actual error condition. Exceptions need to be debugged, handled, responded to, and sometimes with delicate logic of what to do when.

You can enter a debugger when exceptions occur via the `--pdb` flag. Slash will attempt to invoke `puddb` or `ipdb` if you have them installed, but will revert to the default `pdb` if they are not present.

Note that the hooks named `exception_caught_after_debugger`, and `exception_caught_before_debugger` handle exception cases. It is important to plan your hook call-backs and decide which of these two hooks should call them, since a debugger might stall for a long time until a user notices it.

Exception Handling Context

Exceptions can occur in many places, both in tests and in surrounding infrastructure. In many cases you want to give Slash the first opportunity to handle an exception before it propagates. For instance, assume you have the following code:

```
def test_function():
    func1()

def func1():
```

(continues on next page)

(continued from previous page)

```
with some_cleanup_context():
    func2()

def func2():
    do_something_that_can_fail()
```

In the above code, if `do_something_that_can_fail` raises an exception, and assuming you're running slash with `--pdb`, you will indeed be thrown into a debugger. However, the end consequence will not be what you expect, since `some_cleanup_context` will have already been left, meaning any cleanups it performs on exit take place *before* the debugger is entered. This is because the exception handling code Slash uses kicks in only after the exception propagates out of the test function.

In order to give Slash a chance to handle the exception closer to where it originates, Slash provides a special context, `slash.exception_handling.handling_exceptions()`. The purpose of this context is to give your infrastructure a chance to handle an erroneous case as close as possible to its occurrence:

```
def func1():
    with some_cleanup_context(), slash.handle_exceptions_context():
        func2()
```

the `handling_exceptions` context can be safely nested – once an exception is handled, it is appropriately marked, so the outer contexts will skip handling it:

```
from slash.exception_handling import handling_exceptions

def some_function():
    with handling_exceptions():
        do_something_that_might_fail()

with handling_exceptions():
    some_function()
```

Note: `handling_exceptions` will ignore exceptions currently anticipated by `assert_raises()`. This is desired since these exceptions are an expected flow and not an actual error that needs to be handled. These exceptions will be simply propagated upward without any handling or marking of any kind.

Exception Marks

The exception handling context relies on a convenience mechanism for marking exceptions.

Marks with Special Meanings

- `mark_exception_fatal()`: See *below*.
- `noswallow()`: See *below*.
- `inhibit_unhandled_exception_traceback()`: See *below*.

Fatal Exceptions

Slash supports marking special exceptions as *fatal*, causing the immediate stop of the session in which they occur. This is useful if your project has certain types of failures which are considered important enough to halt everything for

investigation.

Fatal exceptions can be added in two ways. Either via marking explicitly with `mark_exception_fatal()`:

```
...
raise slash.exception_handling.mark_exception_fatal(Exception('something'))
```

Or, when adding errors explicitly, via the `mark_fatal` method:

```
slash.add_error("some error condition detected!").mark_fatal()
```

Note: The second form, using `add_error` will not stop immediately since it does not raise an exception. It is your responsibility to avoid any further actions which might tamper with your setup or your session state.

Exception Swallowing

Slash provides a convenience context for swallowing exceptions in various places, `get_exception_swallowing_context()`. This is useful in case you want to write infrastructure code that should not collapse your session execution if it fails. Use cases for this feature:

1. Reporting results to external services, which might be unavailable at times
2. Automatic issue reporting to bug trackers
3. Experimental features that you want to test, but don't want to disrupt the general execution of your test suites.

Swallowed exceptions get reported to log as debug logs, and assuming the `sentry.dsn` configuration path is set, also get reported to `sentry`:

```
def attempt_to_upload_logs():
    with slash.get_exception_swallowing_context():
        ...
```

You can force certain exceptions through by using the `noswallow()` or `disable_exception_swallowing` functions:

```
from slash.exception_handling import (
    noswallow,
    disable_exception_swallowing,
)

def func1():
    raise noswallow(Exception("CRITICAL!"))

def func2():
    e = Exception("CRITICAL!")
    disable_exception_swallowing(e)
    raise e

@disable_exception_swallowing
def func3():
    raise Exception("CRITICAL!")
```

Console Traceback of Unhandled Exceptions

Exceptions thrown from hooks and plugins *outside of running tests* normally cause emitting full traceback to the console. In some cases, you would like to use these errors to denote usage errors or specific known erroneous conditions (e.g. missing configuration or conflicting usages). In these cases you can mark your exceptions to inhibit a full traceback:

```
from slash.exception_handling import inhibit_unhandled_exception_traceback
...
raise inhibit_unhandled_exception_traceback(Exception('Some Error'))
```

New in version 1.3.0.

3.7 Warnings

In many cases test executions succeed, but warnings are emitted. These warnings can mean a lot of things, and in some cases even invalidate the success of the test completely.

3.7.1 Warning Capture

Slash collects warnings emitted throughout the session in the form of either *warning logs* or the *native warnings mechanism*. The warnings are recorded in the `session.warnings` (instance of `warnings.SessionWarnings`) component, and cause the `warning_added` hook to be fired.

3.7.2 Filtering Warnings

By default all native warnings are captured. In cases where you want to silence specific warnings, you can use the `slash.ignore_warnings()` function to handle them.

For example, you may want to include code in your project's `.slashrc` as follows:

```
@slash.hooks.configure.register
def configure_warnings():
    slash.ignore_warnings(category=DeprecationWarning, filename='/some/bad/file.py')
```

3.8 Customizing and Extending Slash

This section describes how to tailor Slash to your needs. We'll walk through the process in baby steps, each time adding a small piece of functionality. If you want to start by looking at the finished example, you can skip and see it [here](#).

3.8.1 Customization Basics

`.slashrc`

In order to customize Slash we have to write code that will be executed when Slash loads. Slash offers an easy way to do this – by placing a file named `.slashrc` in your project's root directory. This file is loaded as a regular Python file, so we will write regular Python code in it.

Note: The `.slashrc` file location is read from the configuration (*run.project_customization_file_path*). However since it is ready before the command-line parsing phase, it cannot be specified using `-o`.

Hooks and Plugins

When our `.slashrc` file is loaded we have only one shot to install and configure all the customizations we need for the entire session. Slash supports two facilities that can be used together for this task, as we'll see shortly.

Hooks are a collection of callbacks that any code can register, thus getting notified when certain events take place. They also support receiving arguments, often detailing what exactly happened.

Plugins are a mechanism for loading pieces of code conditionally, and are *described in detail in the relevant section*. For now it is sufficient to say that plugins are classes deriving from `slash.plugins.PluginInterface`, and that can be activated upon request. Once activated, methods defined on the plugin which correspond to names of known hooks get registered on those hooks automatically.

3.8.2 1. Customizing Using Plain Hooks

Our first step is customizing the logging facility to our needs. We are going to implement two requirements:

1. Have logging always turned on in a fixed location (Say `~/slash_logs`)
2. Collect execution logs at the end of each session, and copy them to a central location (Say `/remote/path`).

The first requirement is simple - it is done by modifying the global Slash configuration:

```
# file: .slashrc
import os
import slash

slash.config.root.log.root = os.path.expanduser('~slash_logs')
```

Note: Don't be confused about `slash.config.root.log.root` above. `slash.config.root` is used to access the root of the configuration, while `log.root` is the name of the configuration value that controls the log location.

See also:

Configuration

The second requirement requires us to do something when the session ends. This is where **hooks** come in. It allows us to register a callback function to be called when the session ends.

Slash uses *gossip* to implement hooks, so we can simply use *gossip.register* to register our callback:

```
import gossip
import shutil

...
@gossip.register('slash.session_end')
def collect_logs():
    shutil.copytree(...)
```

Now we need to supply arguments to `copytree`. We want to copy only the directory of the current session, into a destination directory also specific to this session. How do we do this? The important information can be extracted from `slash.session`, which is a proxy to the current object representing the session:

```
...
@gossip.register('slash.session_end')
def collect_logs():
    shutil.copytree(
        slash.session.logging.session_log_path,
        os.path.join('/remote/path', slash.session.id))
```

See also:

Hooks, Slash Internals

3.8.3 2. Organizing Customizations in Plugins

Suppose you want to make the log collection behavior optional. Our previous implementation registered the callback immediately, meaning you had no control over whether or not it takes place. Optional customizations are best made optional through organizing them in plugins.

Information on plugins in Slash can be found in *Plugins*, but for now it is enough to mention that plugins are classes deriving from `slash.plugins.PluginInterface`. Plugins can be *installed* and *activated*. Installing a plugin makes it available for activation (but does little else), while activating it actually makes it kick into action. Let's write a plugin that performs the log collection for us:

```
...
class LogCollectionPlugin(slash.plugins.PluginInterface):

    def get_name(self):
        return 'logcollector'

    def session_end(self):
        shutil.copytree(
            slash.session.logging.session_log_path,
            os.path.join('/remote/path', slash.session.id))

collector_plugin = LogCollectionPlugin()
plugins.manager.install(collector_plugin)
```

The above class inherits from `slash.plugins.PluginInterface` - this is the base class for implementing plugins. We then call `slash.plugins.PluginManager.install()` to *install* our plugin. Note that at this point the plugin is not activated.

Once the plugin is installed, you can pass `--with-logcollector` to actually activate the plugin. More on that soon.

The `get_name` method is required for any plugin you implement for slash, and it should return the name of the plugin. This is where the `logcollector` in `--with-logcollector` comes from.

The second method, `session_end`, is the heart of how the plugin works. When a plugin is activated, methods defined on it automatically get registered to the respective hooks with the same name. This means that upon activation of the plugin, our collection code will be called when the session ends..

Activating by Default

In some cases you want to activate the plugin by default, which is easily done with the `slash.plugins.PluginManager.activate()`:

```
...
slash.plugins.manager.activate(collector_plugin)
```

Note: You can also just pass `activate=True` in the call to `install`

Once the plugin is enabled by default, you can correspondingly disable it using `--without-logcollector` as a parameter to `slash run`.

See also:

[Plugins](#)

3.8.4 3. Passing Command-Line Arguments to Plugins

In the real world, you want to test integrated products. These are often physical devices or services running on external machines, sometimes even officially called *devices under test*. We would like to pass the target device IP address as a parameter to our test environment. The easiest way to do this is by writing a plugin that adds command-line options:

```
...
@slash.plugins.active
class ProductTestingPlugin(slash.plugins.PluginInterface):

    def get_name(self):
        return 'your product'

    def configure_argument_parser(self, parser):
        parser.add_argument('-t', '--target',
                            help='ip address of the target to test')

    def configure_from_parsed_args(self, args):
        self.target_address = args.target

    def session_start(self):
        slash.g.target = Target(self.target_address)
```

First, we use `slash.plugins.active()` decorator here as a shorthand. See [Plugins](#) for more information.

Second, we use two new plugin methods here - `configure_argument_parser` and `configure_from_parsed_args`. These are called on every activated plugin to give it a chance to control how the commandline is processed. The parser and args passed are the same as if you were using `argparse` directly.

Note that we separate the stages of obtaining the address from actually initializing the target object. This is to postpone the heavier code to the actual beginning of the testing session. The `session_start` hook helps us with that - it is called after the argument parsing part.

Another thing to note here is the use of `slash.g`. This is a convenient location for shared global state in your environment, and is documented in [Global State](#). In short we can conclude with the fact that this object will be available to all test under `slash.g.target`, as a global setup.

3.8.5 4. Configuration Extensions

Slash supports a hierarchical configuration facility, described in *the relevant documentation section*. In some cases you might want to parametrize your extensions to allow the user to control its behavior. For instance let's add an option to specify a timeout for the target's API:

```
...
@slash.plugins.activate
class ProductTestingPlugin(slash.plugins.PluginInterface):
    ...
    def get_name(self):
        return 'your product'

    def get_default_config(self):
        return {'api_timeout_seconds': 50}

    ...
    def session_start(self):
        slash.g.target = Target(
            self.target_address,
            timeout=slash.config.root.plugin_config.your_product.api_timeout_seconds)
```

We use the `slash.plugins.PluginInterface.activate()` method to control what happens when our plugin is **activated**. Note that this happens very early in the execution phase - even before tests are loaded to be executed.

In the `activate` method we use the **extend** capability of Slash's configuration to append configuration paths to it. Then in `session_start` we use the value off the configuration to initialize our target.

The user can now easily modify these values from the command-line using the `-o` flag to `slash run`:

```
$ slash run ... -o product.api_timeout_seconds=100 ./
```

3.8.6 Complete Example

Below is the final code for the `.slashrc` file for our project:

```
import os
import shutil

import slash

slash.config.root.log.root = os.path.expanduser('~/.slash_logs')

@slash.plugins.activate
class LogCollectionPlugin(slash.plugins.PluginInterface):

    def get_name(self):
        return 'logcollector'

    def session_end(self):
        shutil.copytree(
            slash.session.logging.session_log_path,
            os.path.join('/remote/path', slash.session.id))
```

(continues on next page)

(continued from previous page)

```
@slash.plugins.active
class ProductTestingPlugin(slash.plugins.PluginInterface):

    def get_name(self):
        return 'your product'

    def get_default_config(self):
        return {'api_timeout_seconds': 50}

    def configure_argument_parser(self, parser):
        parser.add_argument('-t', '--target',
                            help='ip address of the target to test')

    def configure_from_parsed_args(self, args):
        self.target_address = args.target

    def session_start(self):
        slash.g.target = Target(
            self.target_address, timeout=slash.config.root.plugin_config.your_product.
            ↪api_timeout_seconds)
```

3.9 Configuration

Slash uses a hierarchical configuration structure provided by [Confetti](#). The configuration values are addressed by their full path (e.g. `debug.enabled`, meaning the value called ‘enabled’ under the branch ‘debug’).

Note: You can inspect the current paths, defaults and docs for Slash’s configuration via the `slash list-config` command from your shell

Several ways exist to modify configuration values.

3.9.1 Overriding Configuration Values via Command-Line

When running tests via `slash run`, you can use the `-o` flag to override configuration values:

```
$ slash run -o hooks.swallow_exceptions=yes ...
```

Note: Configuration values get automatically converted to their respective types. More specifically, boolean values also recognize `yes` and `no` as valid values.

3.9.2 Customization Files

There are several locations in which you can store files that are to be automatically executed by Slash when it runs. These files can contain code that overrides configuration values:

slashrc file If the file `~/ .slash/slashrc` (See [run.user_customization_file_path](#)) exists, it is loaded and executed as a regular Python file by Slash on startup.

SLASH_USER_SETTINGS If an environment variable named `SLASH_USER_SETTINGS` exists, the file path it points to will be loaded *instead* of the `slashrc` file.

SLASH_SETTINGS If an environment variable named `SLASH_SETTINGS` exists, it is assumed to point at a file path or URL to load as a regular Python file on startup.

Each of these files can contain code which, among other things, can modify Slash's configuration. The configuration object is located in `slash.config`, and modified through `slash.config.root` as follows:

```
# ~/.slash/slashrc contents
import slash

slash.config.root.debug.enabled = False
```

List of Available Configuration Values

`plugins.search_paths`

Default: [] List of paths in which to search for plugin modules

`debug.debugger`

Default: None

`debug.enabled`

Default: False Enter pdb on failures and errors

`debug.debug_hook_handlers`

Default: False Enter pdb also for every exception encountered in a hook/callback. Only relevant when debugging is enabled

`debug.filter_strings`

Default: [] A string filter, selecting if to enter pdb

`debug.debug_skips`

Default: False Enter pdb also for SkipTest exceptions

`parallel.worker_connect_timeout`

Default: 10 timeout for each worker to connect

parallel.num_workers

Default: 0 Parallel execution

parallel.parent_session_id

Default: None parent session id

parallel.server_port

Default: 0 Server port

parallel.workers_error_dir

Default: None workers error directory

parallel.server_addr

Default: localhost Server address

parallel.communication_timeout_secs

Default: 60 timeout of worker in seconds

parallel.no_request_timeout

Default: 20 timeout for server not getting requests

parallel.worker_id

Default: None Worker_id

parallel.worker_error_file

Default: errors-worker worker error filename template

plugin_config.coverage.report_type

Default: html Coverage report format

plugin_config.coverage.config_filename

Default: False Coverage configuration file

plugin_config.coverage.report

Default: True

plugin_config.coverage.append

Default: False Append coverage data to existing file

plugin_config.coverage.sources

Default: [] Modules or packages for which to track coverage

plugin_config.xunit.filename

Default: testsuite.xml Name of XML xUnit file to create

plugin_config.notifications.notify_on_pdb

Default: True

plugin_config.notifications.nma.enabled

Default: True

plugin_config.notifications.nma.api_key

Default: None

plugin_config.notifications.pushbullet.enabled

Default: True

plugin_config.notifications.pushbullet.api_key

Default: None

plugin_config.notifications.notification_threshold

Default: 5

plugin_config.notifications.pushbullet_api_key

Default: None

`plugin_config.notifications.email.cc_list`

Default: []

`plugin_config.notifications.email.to_list`

Default: []

`plugin_config.notifications.email.smtp_server`

Default: None

`plugin_config.notifications.email.from_email`

Default: Slash <noreply@getslash.github.io>

`plugin_config.notifications.email.enabled`

Default: False

`plugin_config.notifications.nma_api_key`

Default: None

`plugin_config.notifications.prowl_api_key`

Default: None

`plugin_config.notifications.notify_only_on_failures`

Default: False

`plugin_config.notifications.slack.from_user`

Default: slash-bot

`plugin_config.notifications.slack.channel`

Default: None

`plugin_config.notifications.slack.url`

Default: None

plugin_config.notifications.slack.enabled

Default: False

plugin_config.notifications.prowl.enabled

Default: True

plugin_config.notifications.prowl.api_key

Default: None

sentry.dsn

Default: None Possible DSN for a sentry service to log swallowed exceptions. See <http://getsentry.com> for details

interactive.expose_g_globals

Default: True When False, slash.g won't be added to interactive test namespaces

run.default_sources

Default: [] Default tests to run assuming no other sources are given to the runner

run.project_name

Default: None

run.user_customization_file_path

Default: ~/.slash/slashrc

run.stop_on_error

Default: False Stop execution when a test doesn't succeed

run.project_customization_file_path

Default: ./slashrc

run.session_state_path

Default: ~/.slash/last_session Where to keep last session serialized data

run.repeat_each

Default: 1 Repeat each test a specified amount of times

run.message_assertion_introspection

Default: True When False, failing assertions which have messages attached will not emit introspection info

run.repeat_all

Default: 1 Repeat all suite a specified amount of times

run.suite_files

Default: [] File(s) to be read for lists of tests to be run

run.resume_state_path

Default: ~/.slash/session_states Path to store or load session's resume data

run.dump_variation

Default: False Output the full variation structure before each test is run (mainly used for internal debugging)

run.filter_strings

Default: [] A string filter, selecting specific tests by string matching against their name

tmux.enabled

Default: False Run inside tmux

tmux.use_panes

Default: False In parallel mode, run children inside panes and not windows

log.root

Default: None Root directory for logs

log.colorize

Default: False Emit log colors to files

log.truncate_console_lines

Default: True truncate long log lines on the console

log.unittest_mode

Default: False Used during unit testing. Emit all logs to stderr as well as the log files

log.compression.use_rotating_raw_file

Default: False When compression is enabled, write also to uncompressed rotating log file

log.compression.enabled

Default: False Compress log files

log.compression.algorithm

Default: brotli Compression algorithm to use, either gzip or brotli

log.cleanup.enabled

Default: False

log.cleanup.keep_failed

Default: True

log.console_traceback_level

Default: 2 Detail level of tracebacks

log.show_manual_errors_tb

Default: True Show tracebacks for errors added via slash.add_error

log.highlights_subpath

Default: None If set, this path will be used to record highlights (eg. errors added) in the session and/or tests

log.subpath

Default: {context.session.id}/{context.test_id}/debug.log Path to write logs to under the root

log.last_session_dir_symlink

Default: None If set, specifies a symlink path to the last session log directory

log.console_theme.session-summary-success

Default: green/bold

log.console_theme.inline-file-end-skip

Default: yellow

log.console_theme.inline-file-end-fail

Default: red

log.console_theme.test-skip-message

Default: yellow

log.console_theme.tb-line-cause

Default: white

log.console_theme.tb-line

Default: black/bold

log.console_theme.inline-test-interrupted

Default: yellow

log.console_theme.test-additional-details-header

Default: black/bold

log.console_theme.test-additional-details

Default: black/bold

log.console_theme.tb-frame-location

Default: white/bold

`log.console_theme.session-summary-failure`

Default: red/bold

`log.console_theme.error-separator-dash`

Default: red

`log.console_theme.tb-error-message`

Default: red/bold

`log.console_theme.inline-error`

Default: red

`log.console_theme.tb-test-line`

Default: red/bold

`log.console_theme.test-error-header`

Default: white

`log.console_theme.fancy-message`

Default: yellow/bold

`log.console_theme.tb-error`

Default: red/bold

`log.console_theme.error-cause-marker`

Default: white/bold

`log.console_theme.frame-local-varname`

Default: yellow/bold

`log.console_theme.inline-file-end-success`

Default: green

log.traceback_variables

Default: False Logs values of variables in traceback frames for added errors

log.errors_subpath

Default: None Depreacted - Use 'highlights_subpath' config instead

log.console_level

Default: 13

log.format

Default: None Format of the log line, as passed on to logbook. None will use the default format

log.last_session_symlink

Default: None If set, specifies a symlink path to the last session log file in each run

log.last_failed_symlink

Default: None If set, specifies a symlink path to the last failed test log file

log.truncate_console_errors

Default: False If truncate_console_lines is set, also truncate long log lines, including and above the "error" level, on the console

log.color_console

Default: None

log.unified_session_log

Default: False Make the session log file contain all logs, including from tests

log.session_subpath

Default: {context.session.id}/session.log

log.silence_loggers

Default: [] Logger names to silence

log.console_format

Default: None Optional format to be used for console output. Defaults to the regular format

log.last_test_symlink

Default: None If set, specifies a symlink path to the last test log file in each run

log.localtime

Default: False Use local time for logging. If False, will use UTC

log.core_log_level

Default: 13 Minimal level of slash log messages to show

3.10 Logging

As mentioned in *the introductory section*, logging in Slash is done by Logbook. The path to which logs are written is controlled with the `-l` flag and console verbosity is controlled with `-v/-q`. Below are some more advanced topics which may be relevant for extending Slash’s behavior.

3.10.1 Controlling Console Colors

Console logs are colorized according to their level by default. This is done using Logbook’s colorizing handler. In some cases you might want logs from specific sources to get colored differently. This is done using `slash.log.set_log_color()`:

```
>>> import slash.log
>>> import logbook
>>> slash.log.set_log_color('my_logger_name', logbook.NOTICE, 'red')
```

Note: Available colors are taken from **logbook**. Options are “black”, “darkred”, “darkgreen”, “brown”, “darkblue”, “purple”, “teal”, “lightgray”, “darkgray”, “red”, “green”, “yellow”, “blue”, “fuchsia”, “turquoise”, “white”

Note: You can also colorize log fiels by setting the `log.colorize` configuration variable to True

3.10.2 Controlling the Log Subdir Template

The filenames created under the root are controlled with the `log.subpath` config variable, which can be also a format string receiving the `context` variable from slash (e.g. `sessions/{context.session.id}/{context.test.id}/logfile.log`).

Test Ordinals

You can use `slash.core.metadata.Metadata.test_index0` to include an ordinal prefix in log directories, for example setting `log.subpath` to:

```
{context.session.id}/{context.test.__slash__.test_index0:03}-{context.test.id}.log
```

Timestamps

The current timestamp can also be used when formatting log paths. This is useful if you want to create log directories named according to the current date/time:

```
logs/{timestamp:%Y%m%d-%H%M%S}.log
```

The Session Log

Another important config path is `log.session_subpath`. In this subpath, a special log file will be kept logging all records that get emitted when there's no active test found. This can happen between tests or on session start/end.

The session log, by default, does not contain logs from tests, as they are redirected to test log files. However, setting the `log.unified_session_log` to `True` will cause the session log to contain *all* logs from all tests.

The Highlights Log

Slash allows you to configure a separate log file to receive “highlight” logs from your sessions. This isn't necessarily related to the log level, as any log emitted can be marked as a “highlight”. This is particularly useful if you have infrequent operations that you'd like to track and skim occasionally.

To configure a log location for your highlight logs, set the `log.highlights_subpath` configuration path. To emit a highlight log, just pass `{ 'highlight': True }` to the required log's extra dict:

```
slash.logger.info("hey", extra={"highlight": True})
```

Tip: The `log.highlights_subpath` configuration path is treated just like other logging subpaths, and thus supports all substitutions and formatting mentioned above

Note: All errors emitted in a session are automatically added to the highlights log

3.10.3 Last Log Symlinks

Slash can be instructed to maintain a symlink to recent logs. This is useful to quickly find the last test executed and dive into its logs.

- To make slash store a symlink to the last session log file, use `log.last_session_symlink`
- To make slash store a symlink to the last session log directory, use `log.last_session_dir_symlink`
- To make slash store a symlink to the last session log file, use `log.last_test_symlink`
- To make slash store a symlink to the last session log file, use `log.last_failed_symlink`

Both parameters are strings pointing to the symlink path. In case they are relative paths, they will be computed relative to the log root directory (see above).

The symlinks are updated at the beginning of each test run to point at the recent log directory.

3.10.4 Silencing Logs

In certain cases you can silence specific loggers from the logging output. This is done with the *log.silence_loggers* config path:

```
slash run -i -o "log.silence_loggers=['a','b']"
```

3.10.5 Changing Formats

The *log.format* config path controls the log line format used by slash:

```
$ slash run -o log.format="{record.time:%Y%m%d}]- {record.message}" ...
```

3.11 Saving Test Details

Slash supports saving additional data about test runs, by attaching this data to the global result object.

3.11.1 Test Details

Test details can be thought of as an arbitrary dictionary of values, keeping important information about the session that can be later browsed by reporting tools or plugins.

To set a detail, just use `result.details.set`, accessible through Slash's global context:

```
def test_steering_wheel(car):
    mileage = car.get_mileage()
    slash.context.result.details.set('mileage', mileage)
```

3.11.2 Test Facts

Facts are very similar to details but they are intended for a more strict set of values, serving as a basis for coverage matrices.

For instance, a test reporting tool might want to aggregate many test results and see which ones succeeded on model A of the product, and which on model B.

To set facts, use `result.facts` just like the details feature:

```
def test_steering_wheel(car):
    slash.context.result.facts.set('is_van', car.is_van())
```

Note: facts also trigger the *fact_set* hook when set

Note: The distinction of when to use details and when to use facts is up for the user and/or the plugins that consume that information

3.12 Hooks

Slash leverages the [gossip library](#) to implement hooks. Hooks are endpoints to which you can register callbacks to be called in specific points in a test session lifetime.

All built-in hooks are members of the `slash` gossip group. As a convenience, the hook objects are all kept as globals in the `slash.hooks` module.

The `slash` gossip group is set to be both strict (See [Gossip strict registrations](#)) and has exception policy set to `RaiseDefer` (See [Gossip error handling](#)).

3.12.1 Registering Hooks

Hooks can be registered through `slash.hooks`:

```
import slash

@slash.hooks.session_start.register
def handler():
    print("Session has started: ", slash.context.session)
```

Which is roughly equivalent to:

```
import gossip

@gossip.register("slash.session_start")
def handler():
    print("Session has started: ", slash.context.session)
```

3.12.2 Hook Errors

By default, exceptions propagate from hooks and on to the test, but first all hooks are attempted. In some cases though you may want to debug the exception close to its raising point. Setting `debug.debug_hook_handlers` to `True` will cause the debugger to be triggered as soon as the hook dispatcher encounters the exception. This is done via [gossip's error handling mechanism](#).

3.12.3 Hooks and Plugins

Hooks are especially useful in conjunction with [Plugins](#). By default, plugin method names correspond to hook names on which they are automatically registered upon activation.

See also:

[Plugins](#)

3.12.4 Advanced Usage

You may want to further customize hook behavior in your project. Mose of these customizations are available through `gossip`.

See also:

[Advanced Usage In Gossip](#)

See also:

[Hook Dependencies](#)

3.12.5 Available Hooks

The following hooks are available from the `slash.hooks` module:

`slash.hooks.after_session_end`

Called right after `session_end` hook

`slash.hooks.after_session_start`

Second entry point for session start, useful for plugins relying on other plugins' `session_start` routine

`slash.hooks.before_interactive_shell(namespace)`

Called before starting interactive shell

`slash.hooks.before_session_cleanup`

Called right before session cleanup begins

`slash.hooks.before_session_start`

Entry point which is called before `session_start`, useful for configuring plugins and other global resources

`slash.hooks.before_test_cleanups`

Called right before a test cleanups are executed

`slash.hooks.before_worker_start(worker_config)`

Called in parallel execution mode, before the parent starts the child worker

`slash.hooks.configure`

Configuration hook that happens during commandline parsing, and before plugins are activated. It is a convenient point to override plugin activation settings

slash.hooks.entering_debugger(exc_info)

Called right before entering debugger

slash.hooks.error_added(error, result)

Called when an error is added to a result (either test result or global)

slash.hooks.exception_caught_after_debugger

Called whenever an exception is caught, and a debugger has already been run

slash.hooks.exception_caught_before_debugger

Called whenever an exception is caught, but a debugger hasn't been entered yet

slash.hooks.fact_set(name, value)

Called when a fact is set for a test

slash.hooks.interruption_added(result, exception)

Called when an exception is encountered that triggers test or session interruption

slash.hooks.log_file_closed(path, result)

Called right after a log file was closed

slash.hooks.prepare_notification(message)

Called with a message object prior to it being sent via the notifications plugin (if enabled)

slash.hooks.result_summary

Called at the end of the execution, when printing results

slash.hooks.session_end

Called right before the session ends, regardless of the reason for termination

slash.hooks.session_interrupt

Called when the session is interrupted unexpectedly

slash.hooks.session_start

Called right after session starts

slash.hooks.test_avoided(reason)

Called when a test is skipped completely (not even started)

slash.hooks.test_distributed(test_logical_id, worker_session_id)

Called in parallel mode, after the parent sent a test to child)

slash.hooks.test_end

Called right before a test ends, regardless of the reason for termination

slash.hooks.test_error

Called on test error

slash.hooks.test_failure

Called on test failure

slash.hooks.test_interrupt

Called when a test is interrupted by a KeyboardInterrupt or other similar means

slash.hooks.test_skip(reason)

Called on test skip

slash.hooks.test_start

Called right after a test starts

slash.hooks.test_success

Called on test success

slash.hooks.tests_loaded(tests)

Called when Slash finishes loading a batch of tests for execution (not necessarily all tests)

slash.hooks.warning_added(warning)

Called when a warning is captured by Slash

slash.hooks.worker_connected(session_id)

Called on new worker startup

3.13 Plugins

Plugins are a comfortable way of extending Slash’s behavior. They are objects inheriting from a *common base class* that can be activated to modify or what happens in select point of the infrastructure.

3.13.1 The Plugin Interface

Plugins have several special methods that can be overridden, like *get_name* or *configure_argument_parser*. Except for these methods and the ones documented, each public method (i.e. a method not beginning with an underscore) must correspond to a *slash hook* by name.

The name of the plugin should be returned by *get_name*. This name should be unique, and not shared by any other plugin.

3.13.2 Plugin Discovery

Plugins can be loaded from multiple locations.

Search Paths

First, the paths in `plugins.search_paths` are searched for python files. For each file, a function called `install_plugins` is called (assuming it exists), and this gives the file a chance to install its plugins.

3.13.3 Plugin Installation

To install a plugin, use the *slash.plugins.manager.install* function, and pass it the plugin class that is being installed. Note that installed plugins are not active by default, and need to be explicitly activated (see below).

Only plugins that are *PluginInterface* derivative instances are accepted.

To uninstall plugins, you can use the *slash.plugins.manager.uninstall*.

Note: uninstalling plugins also deactivates them.

Internal Plugins

By default, plugins are considered “external”, meaning they were loaded by the user (either directly or indirectly). External plugins can be activated and deactivated through the command-line using `--with-<plugin name>` and `--without-<plugin name>`.

In some cases, though, you may want to install a plugin in a way that would not let the user disable it externally. Such plugins are considered “internal”, and cannot be deactivated through the command line.

You can install a plugin as an internal plugin by passing `internal=True` to the install function.

3.13.4 Plugin Activation

Plugins are activated via `slash.plugins.manager.activate` and deactivated via `slash.plugins.manager.deactivate`.

During the activation all hook methods get registered to their respective hooks, so any plugin containing an unknown hook will trigger an exception.

Note: by default, all method names in a plugin are assumed to belong to the *slash* gossip group. This means that the method `session_start` will register on `slash.session_start`. You can override this behavior by using `slash.plugins.registers_on()`:

```
from slash.plugins import registers_on

class MyPlugin(PluginInterface):
    @registers_on('some_hook')
    def func(self):
        ...
```

`registers_on(None)` has a special meaning - letting Slash know that this is not a hook entry point, but a private method belonging to the plugin class itself.

See also:

Hooks

Activating plugins from command-line is usually done with the `--with-` prefix. For example, to activate a plugin called `test-plugin`, you can pass `--with-test-plugin` when running `slash run`.

Also, since some plugins can be activated from other locations, you can also override and deactivate plugins using `--without-X` (e.g. `--without-test-plugin`).

Conditionally Registering Hooks

You can make the hook registration of a plugin *conditional*, meaning it should only happen if a boolean condition is `True`.

This can be used to create plugins that are compatible with multiple versions of Slash:

```
class MyPlugin(PluginInterface):
    ...
    @slash.plugins.register_if(int(slash.__version__.split('.')[0]) >= 1)
    def shiny_new_hook(self):
        ...
```

See also:

`slash.plugins.register_if()`

3.13.5 Plugin Command-Line Interaction

In many cases you would like to receive options from the command line. Plugins can implement the `configure_argument_parser` and the `configure_parsed_args` functions:

```
class ResultsReportingPlugin(PluginInterface):

    def configure_argument_parser(self, parser):
        parser.add_argument("--output-filename", help="File to write results to")

    def configure_from_parsed_args(self, args):
        self.output_filename = args.output_filename
```

3.13.6 Plugin Configuration

Plugins can override the `config` method to provide configuration to be placed under `plugin_config.<plugin name>`:

```
class LogCollectionPlugin(PluginInterface):

    def get_default_config(self):
        return {
            'log_destination': '/some/default/path'
        }
```

The configuration is then accessible with `get_current_config` property.

3.13.7 Plugin Examples

An example of a functioning plugin can be found in the *Customizing and Extending Slash* section.

3.13.8 Errors in Plugins

As more logic is added into plugins it becomes more likely for exceptions to occur when running their logic. As seen above, most of what plugins do is done by registering callbacks onto hooks. Any exception that escapes these registered functions will be handled the same way any exception in a hook function is handled, and this depends on the current exception swallowing configuration.

See also:

- *exception swallowing*
- *hooks documentation*

3.13.9 Plugin Dependencies

Slash supports defining dependencies between plugins, in a mechanism closely related to `gossip's hook dependencies`. The purpose of these dependencies is to make sure a certain hook registration in a specific plugin (or all such hooks for that matter) is called before or after equivalent hooks on other plugins.

Notable examples of why you might want this include, among many other cases:

- Plugins reporting test status needing a state computed by other plugins
- Error handling plugins wanting to be called first in certain events
- Log collection plugins wanting to be called only after all interesting code paths are logged

Defining Plugin Dependencies

Defining dependencies is done primarily with two decorators Slash provides: `@slash.plugins.needs` and `@slash.plugins.provides`. Both of these decorators use string identifiers to denote the dependencies used. These identifiers are arbitrary, and can be basically any string, as long as it matches between the dependent plugin and the providing plugin.

Several use cases exist:

Hook-Level Dependencies

Adding the `slash.plugins.needs` or `slash.plugins.provides` decorator to a specific hook method on a plugin indicates that we would like to depend on or be the dependency accordingly. For example:

```
class TestIdentificationPlugin(PluginInterface):

    @slash.plugins.provides('awesome_test_id')
    def test_start(self):
        slash.context.test.awesome_test_id = awesome_id_allocation_service()

class TestIdentificationLoggingPlugin(PluginInterface):

    @slash.plugins.needs('awesome_test_id')
    def test_start(self):
        slash.logger.debug('Test has started with the awesome id of {!r}', slash.
        ↪context.test.awesome_id)
```

In the above example, the `test_start` hook on `TestIdentificationLoggingPlugin` needs the `test_start` of `TestIdentificationPlugin` to be called first, and thus **requires** the `'awesome_test_id'` identifier which is provided by the latter.

Plugin-Level Dependencies

Much like hook-level dependencies, you can decorate the entire plugin with the `needs` and `provides` decorators, creating a dependency on all hooks provided by the plugin:

```
@slash.plugins.provides('awesome_test_id')
class TestIdentificationPlugin(PluginInterface):

    def test_start(self):
        slash.context.test.awesome_test_id = awesome_id_allocation_service()

@slash.plugins.needs('awesome_test_id')
class TestIdentificationLoggingPlugin(PluginInterface):

    def test_start(self):
        slash.logger.debug('Test has started with the awesome id of {!r}', slash.
        ↪context.test.awesome_id)
```

The above example is equivalent to the previous one, only now future hooks added to either of the plugins will automatically assume the same dependency specifications.

Note: You can use `provides` and `needs` in more complex cases, for example specifying needs on a specific hook in one plugin, where the entire other plugin is decorated with `provides` (at plugin-level).

Note: Plugin-level provides and needs also get transferred upon inheritance, automatically adding the dependency configuration to derived classes.

3.13.10 Plugin Manager

As mentioned above, the Plugin Manager provides API to activate (or deactivate) and install (or uninstall) plugins. Additionally, it provides access to instances of registered plugins by their name via `slash.plugins.manager.get_plugin`. This could be used to access plugin attributes whose modification (e.g. by fixtures) can alter the plugin's behavior.

3.13.11 Plugins and Parallel Runs

Not all plugins can support *parallel execution*, and for others implementing support for it can be much harder than supporting non-parallel runs alone.

To deal with this, in addition to possible mistakes or corruption caused by plugins incorrectly used in parallel mode, Slash requires each plugin to indicate whether or not it supports parallel execution. The assumption is that by default plugins do not support parallel runs at all.

To indicate that your plugin supports parallel execution, use the `plugins.parallel_mode` marker:

```
from slash.plugins import PluginInterface, parallel_mode

@parallel_mode('enabled')
class MyPlugin(PluginInterface):
    ...
```

`parallel_mode` supports the following modes:

- `disabled` - meaning the plugin does not support parallel execution at all. This is the default.
- `parent-only` - meaning the plugin supports parallel execution, but should be active only on the parent process.
- `child-only` - meaning the plugin should only be activated on worker/child processes executing the actual tests.
- `enabled` - meaning the plugin supports parallel execution, both on parent and child.

3.14 Built-in Plugins

Slash comes with pre-installed, built-in plugins that can be activated when needed.

3.14.1 Coverage

This plugin tracks and reports runtime code coverage during runs, and reports the results in various formats. It uses the Net Batchelder's [coverage package](#).

To use it, run Slash with `--with-coverage`, and optionally specify modules to cover:

```
$ slash run --with-coverage --cov mypackage --cov-report html
```

3.14.2 Notifications

The notifications plugin allows users to be notified when sessions end in various methods, or notification mediums.

To use it, run Slash with `--with-notifications`. Please notice that each notification type requires additional configuration values. You will also have to enable your desired backend with `--notify-<backend name>` (e.g. `--notify-email`)

For e-mail notification, you'll need to configure your SMTP server, and pass the recipients using `--email-to`:

```
$ slash run --notify-email --with-notifications -o plugin_config.notifications.email.  
→smtp_server='my-smtp-server.com --email-to youremail@company.com'
```

For using Slack notification, you should firstly configure [slack webhook integration](#). And run slash:

```
$ slash run --with-notifications -o plugin_config.notifications.slack.url='your-  
→webhook-inegration-url' -o plugin_config.notifications.slack.channel='@myslackuser'
```

Including Details in Notifications

You can include additional information in your notifications, which is then sent as a part of email messages you receive. This can be done with the `prepare_notification` hook:

```
@slash.hooks.prepare_notification.register  
def prepare_notification(message):  
    message.details_dict['additional_information'] = 'some information included'
```

3.14.3 XUnit

The xUnit plugin outputs an XML file when sessions finish running. The XML conforms to the xunit format, and thus can be read and processed by third party tools (like CI services, for example)

Use it by running with `--with-xunit` and by specifying the output filename with `--xunit-filename`:

```
$ slash run --with-xunit --xunit-filename xunit.xml
```

3.15 Slash Internals

3.15.1 The Result Object

Running tests store their results in `slash.core.result.Result` objects, accessible through `slash.context.result`.

In normal scenarios, tests are not supposed to directly interact with result objects, but in some cases it may come in handy.

A specific example of such cases is adding additional test details using `details``. These details are later displayed in the summary and other integrations:

```
def test_something(microwave):  
    slash.context.result.details.set('microwave_version', microwave.get_version())
```

See also:

details_

3.15.2 The Session Object

Tests are always run in a context, called a **session**. A session is used to identify the test execution process, giving it a unique id and collecting the entire state of the run.

The `Session` represents the current test execution session, and contains the various state elements needed to maintain it. Since sessions also contain test results and statuses, trying to run tests without an active session will fail.

The currently active session is accessible through `slash.session`:

```
from slash import session

print("The current session id is", session.id)
```

Note: Normally, you don't have to create slash sessions programmatically. Slash creates them for you when running tests. However, it is always possible to create sessions in an interpreter:

```
from slash import Session

...
with slash.Session() as s:
    ... # <--- in this context, s is the active session
```

3.15.3 Test Metadata

Each test being run contains the `__slash__` attribute, meant to store metadata about the test being run. The attribute is an instance of `slash.core.metadata.Metadata`.

Note: Slash does not save the actual test instance being run. This is important because in most cases dead tests contain reference to whole object graphs that need to be released to conserve memory. The only thing that is saved is the test metadata structure.

Test ID

Each test has a unique ID derived from the session id and the ordinal number of the test being run. This is saved as `test.__slash__.id` and can be used (through property) as `test.id`.

3.16 Misc. Features

3.16.1 Notifications

Slash provides an optional plugin for sending notifications at end of runs, via `--with-notifications`. It supports [NMA](#), [Prowl](#) and [Pushbullet](#).

To use it, specify either `plugins.notifications.prowl_api_key`, `plugins.notifications.nma_api_key` or `plugins.notifications.pushbullet_api_key` when running. For example:

```
slash run my_test.py --with-notifications -o plugins.notifications.nma_api_
↪key=XXXXXXXXXXXXXXXXXX
```

3.16.2 XUnit Export

Pass `--with-xunit`, `--xunit-filenam=PATH` to export results as xunit XMLs (useful for CI solutions and other consumers).

3.17 Advanced Use Cases

3.17.1 Customizing via Setuptools Entry Points

Slash can be customized globally, meaning anyone who will run `slash run` or similar commands will automatically get a customized version of Slash. This is not always what you want, but it may still come in handy.

To do this we write our own customization function (like we did in *the section about customization <customize>*):

```
def cool_customization_logic():
    ... # install plugins here, change configuration, etc...
```

To let slash load our customization on startup, we'll use a feature of `setuptools` called *entry points*. This lets us register specific functions in “slots”, to be read by other packages. We'll append the following to our `setup.py` file:

```
# setup.py

...
setup(...
    # ...
    entry_points = {
        "slash.site.customize": [
            "cool_customization_logic = my_package:cool_customization_logic"
        ]
    },
    # ...
)
```

Note: You can read more about setuptools entry points [here](#).

Now Slash will call our `customize` function when loading.

3.17.2 Loading and Running Tests in Code

Sometimes you would like to run a sequence of tests that you control in fine detail, like checking various properties of a test before it is being loaded and run. This can be done in many ways, but the easiest is to use the test loader explicitly.

Running your Tests

```
import slash
from slash.loader import Loader

if __name__ == "__main__":
    with slash.Session() as session:
        tests = Loader().get_runnables(["/my_path", ...])
        with session.get_started_context():
            slash.run_tests(tests)
```

The parameter given above to `slash.runner.run_tests()` is merely an iterator yielding runnable tests. You can interfere or skip specific tests quite easily:

```
import slash
...
def _filter_tests(iterator):
    for test in iterator:
        if "forbidden" in test.__slash__.file_path:
            continue
        yield test
...
slash.run_tests(_filter_tests(slash.loader.Loader().get_runnables(...)))
```

Analyzing Results

Once you run your tests, you can examine the results through `session.results`:

```
if not session.results.is_success(allow_skips=False):
    print('Some tests did not succeed')
```

Iterating over test results can be done with `slash.core.result.SessionResults.iter_test_results()`:

```
for result in session.results.iter_test_results():
    print('Result for', result.test_metadata.name)
    print(result.is_success())
    for error in result.get_errors():
        ...
```

For errors and failures, you can examine each of them using the methods and properties offered by `slash.core.error.Error`.

See also:

Test Metadata

See also:

Customizing and Extending Slash

3.17.3 Specifying Default Test Source for `slash run`

If you use `slash run` for running your tests, it is often useful to specify a default for the test path to run. This is useful if you want to provide a sane default running environment for your users via a `.slashrc` file. This can be done with the `run.default_sources` configuration option:

```
# ...
slash.config.root.run.default_sources = ["/my/default/path/to/tests"]
```

3.18 Cookbook

3.18.1 Execution

Controlling Test Execution Order

Slash offers a hook called `tests_loaded` which can be used, among else, to control the test execution order. Tests are sorted by a dedicated key in their metadata (a.k.a the `__slash__` attribute), which defaults to the discovery order. You can set your hook registration to modify the tests as you see fit, for instance to reverse test order:

```
@slash.hooks.tests_loaded.register
def tests_loaded(tests):
    for index, test in enumerate(reversed(tests)):
        test.__slash__.set_sort_key(index)
```

The above code is best placed in a `slashconf.py` file at the root of your test repository.

3.18.2 Interactive Tests

Controlling Interactive Namespaces from Plugins

You can customize the namespace available by default to interactive tests run with Slash (like `slash run -i`) using the special hook `slash.hooks.before_interactive_shell(namespace)`:

```
class MyPlugin(PluginInterface):
    ...
    def before_interactive_shell(self, namespace):
        namespace['lab_name'] = 'MicrowaveLab'
```

Now when running your session interactively you'll get:

```
$ slash run -i
In [1]: lab_name
Out[1]: 'MicrowaveLab'
```

3.18.3 Logging

Adding Multiple Log Files to a Single Test Result

Slash result objects contain the main path of the log file created by Slash (if logging is properly configured for the current run).

In some cases it may be desirable to include multiple log files for the current test. This can be useful, for example, if the current test runs additional tools or processes emitting additional logs:

```
import slash
import subprocess

def test_running_validation_tool():
    log_dir = slash.context.result.get_log_dir()
    log_file = os.path.join(log_dir, "tool.log")

    slash.context.result.add_extra_log_path(log_file)

    with open(os.path.join(log_dir, "tool.log"), "w") as logfile:
        res = subprocess.run(f'/bin/validation_tool -l {log_dir}', shell=True,
↪ stdout=logfile)
        res.check_returncode()
```

You can also configure extra session paths, for example from plugins:

```
class MyPlugin(slash.plugins.PluginInterface):

    def get_name(self):
        return "my plugin"

    def get_default_config(self):
        return {'extra_log_path': ''}

    def session_start(self):
        log_path = slash.config.root.plugin_config.my_plugin.extra_log_path
        if log_path:
            slash.context.session.results.global_result.add_extra_log_path(log_path)
```

3.19 FAQ

3.19.1 What is the Difference Between Slash and Pytest/Nose/Unittest?

We would first like to point out that both *Nose* and Python’s built-in `unittest` were built for building and running unit tests. `Unittest` provides a decent runner, whereas *Nose* is more of an evolved runner that supports plugins. Both try not to get involved too much in your project’s test code, and assume you are running `unittest`-based tests, or not far from it.

Pytest, on the other hand, took the next step - it’s not only a great test runner, but provides more utilities and infrastructure for your tests, like fixtures, parametrization etc. We personally love *pytest*, and use it to test *Slash* itself, as you can see from our code.

However, the main difference is in the project’s focus. *Pytest* was created as a successor to *nose/unittest*, and as such its primary focus tends to remain around unit tests. This implies certain defaults (like `stdout/stderr` capturing) and certain sets of features which are more likely to be implemented for it.

The main project for which we wrote *Slash* involved testing an external product. As such, it was less about maintaining individual state for each test and setting it up again later, and more about building a consistent state for the entire test session – syncing with the test “subject” before the first test, performing validations between tests, recycling objects and entities between tests etc. What was missing for us in *Pytest* became clear after a certain period of active development – *Pytest*, being focused around the tests being written, lacks (some) facilities to deal with everything around and between the tests.

One specific example for us was widely-scoped cleanups (like tests registering cleanups that are to happen at the end of the session or module) - in this case it was difficult to tie the error to the entity that created the cleanup logic. There

are more examples of how Slash focuses on the testing session itself and its extensibility - the concept of session errors is much better defined in Slash, it includes mechanisms for controlling plugin dependencies, multiple levels of customizations and a hierarchical configuration mechanism. There are also features that Slash provides that Pytest does not, like better logging control, advanced fixture parametrization, early-catch exception handling and more - with even more yet to be shipped.

Another difference is that while pytest can be loosely thought of as a tool, Slash can be thought of as a framework. It puts much more emphasis on letting you build on top of it, set up your environment and integrate with external services (we ourselves built Backslash as a centralized reporting solution for it, for instance). Slash eventually aims at helping you evolve your own testing solution with it.

In the end, the distinction isn't clear-cut though, and different people might find different tools better suited for them. This is great - having choice when it comes to which tool to use is a good thing, and we embrace this fact.

3.20 API Documentation

3.20.1 Testing Utilities

class `slash.Test` (*test_method_name, fixture_store, fixture_namespace, variation*)

This is a base class for implementing unittest-style test classes.

after ()

Gets called after each separate case from this test class executed, assuming `before()` was successful.

before ()

Gets called before each separate case generated from this test class

run ()

Warning: Not to be overridden

`slash.parametrize` (*parameter_name, values*)

Decorator to create multiple test cases out of a single function or module, where the cases vary by the value of *parameter_name*, as iterated through *values*.

`slash.core.fixtures.parameters.toggle` (*param_name*)

A shortcut for `slash.parametrize(param_name, [True, False])`

Note: Also available for import as `slash.parameters.toggle`

`slash.core.fixtures.parameters.iterate` (***kwargs*)

`slash.abstract_test_class` (*cls*)

Marks a class as **abstract**, thus meaning it is not to be run directly, but rather via a subclass.

3.20.2 Assertions

`slash.assert_raises` (*exception_class, msg=None*)

Ensures a subclass of **ARG1** leaves the wrapped context:


```
>>> with assert_raises(AttributeError):
...     raise AttributeError()
```

`slash.assert_almost_equal(a, b, delta=1e-08)`
 Asserts that $\text{abs}(a - b) \leq \text{delta}$

3.20.3 Cleanups

`slash.add_cleanup(self, _func, *args, **kwargs)`
 Adds a cleanup function to the cleanup stack. Cleanups are executed in a LIFO order.
 Positional arguments and keywords are passed to the cleanup function when called.

Parameters

- **critical** – If True, this cleanup will take place even when tests are interrupted by the user (Using Ctrl+C for instance)
- **success_only** – If True, execute this cleanup only if no errors are encountered
- **scope** – Scope at the end of which this cleanup will be executed
- **args** – positional arguments to pass to the cleanup function
- **kwargs** – keyword arguments to pass to the cleanup function

`slash.add_critical_cleanup(_func, *args, **kwargs)`
 Same as `add_cleanup()`, only the cleanup will be called even on interrupted tests

`slash.add_success_only_cleanup(_func, *args, **kwargs)`
 Same as `add_cleanup()`, only the cleanup will be called only if the test succeeds

3.20.4 Skips

class `slash.exceptions.SkipTest(reason='Test skipped')`
 This exception should be raised in order to interrupt the execution of the currently running test, marking it as skipped

`slash.skipped(thing, reason=None)`
 A decorator for skipping methods and classes

`slash.skip_test(*args)`
 Skips the current test execution by raising a `slash.exceptions.SkipTest` exception. It can optionally receive a reason argument.

`slash.register_skip_exception(exception_type)`
 Registers a custom exception type to be recognized a test skip. This makes the exception behave just as if the test called `skip_test`

Note: this must be called within an active session

3.20.5 Tags

`slash.tag(tag_name, tag_value=<NOTHING>)`
 Decorator for tagging tests

3.20.6 Fixtures

`slash.fixture` (*func=None, name=None, scope=None, autouse=False*)

`slash.yield_fixture` (*func=None, **kw*)

Builds a fixture out of a generator. The pre-yield part of the generator is used as the setup, where the yielded value becomes the fixture value. The post-yield part is added as a cleanup:

```
>>> @slash.yield_fixture
... def some_fixture(arg1, arg2):
...     m = Microwave()
...     m.turn_on(wait=True)
...     yield m
...     m.turn_off()
```

`slash.generator_fixture` (*func=None, **kw*)

A utility for generating parametrization values from a generator:

```
>>> @slash.generator_fixture
... def some_parameter():
...     yield first_value
...     yield second_value
```

Note: A generator parameter is a shortcut for a simple parametrized fixture, so the entire iteration is exhausted during test load time

`slash.nofixtures` ()

Marks the decorated function as opting out of automatic fixture deduction. Slash will not attempt to parse needed fixtures from its argument list

`slash.use` (*real_fixture_name*)

Allows tests to use fixtures under different names:

```
>>> def test_something(m: use('microwave')):
...     ...
```

For cosmetic purposes, you can also use `use` with attribute access:

```
>>> def test_something(m: use.microwave):
...     ...
```

3.20.7 Requirements

`slash.requires` (*req, message=None*)

A decorator specifying that the decorated tests requires a certain precondition in order to run

Parameters **req** – Either a function receiving no arguments and returning a boolean, or a boolean specifying whether or not the requirement is met

3.20.8 Warnings

class `slash.warnings.SessionWarnings`

Holds all warnings emitted during the session

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

`__iter__()`
Iterates through stored warnings

`__weakref__`
list of weak references to the object (if defined)

`slash.ignore_warnings` (*category=None, message=None, filename=None, lineno=None*)

Ignores warnings of specific origin (category/filename/lineno/message) during the session. Unlike Python’s default `warnings.filterwarnings`, the parameters are matched only if specified (not defaulting to “match all”). Message can also be a regular expression object compiled with `re.compile`.

`slash.ignore_warnings(category=CustomWarningCategory)`

3.20.9 Hooks

`slash.hooks.register` (*func*)

A shortcut for registering hook functions by their names

3.20.10 Plugins

`slash.plugins.active` (*plugin_class*)

Decorator for automatically installing and activating a plugin upon definition

`slash.plugins.parallel_mode` (*mode*)

Marks compatibility of a specific plugin to parallel execution.

Parameters *mode* – Can be either disabled, enabled, parent-only or child-only

`slash.plugins.registers_on` (*hook_name*)

Marks the decorated plugin method to register on a custom hook, rather than the method name in the ‘slash’ group, which is the default behavior for plugins

Specifying `registers_on(None)` means that this is not a hook entry point at all.

`slash.plugins.register_if` (*condition*)

Marks the decorated plugins method to only be registered if *condition* is `True`

`class slash.plugins.PluginInterface`

This class represents the base interface needed from plugin classes.

`activate()`
Called when the plugin is activated

`configure_argument_parser` (*parser*)
Gives a chance to the plugin to add options received from command-line

`configure_from_parsed_args` (*args*)
Called after successful parsing of command-line arguments

`current_config`
Returns configuration object for plugin

`deactivate()`
Called when the plugin is deactivated

Note: this method might not be called in practice, since it is not guaranteed that plugins are always deactivated upon process termination. The intention here is to make plugins friendlier to cases in which multiple sessions get established one after another, each with a different set of plugins.

get_config()

Use `get_default_config()` instead.

Deprecated since version 1.5.0.

get_default_config()

Optional: should return a dictionary or a confetti object which will be placed under `slash.config.plugin_config.<plugin_name>`

get_description()

Retrieves a quick description for this plugin, mostly used in command-line help or online documentation. It is not mandatory to override this method.

get_name()

Returns the name of the plugin class. This name is used to register, disable and address the plugin during runtime.

Note that the command-line switches (`--with-...`) are derived from this name.

Any implemented plugin must override this method.

class `slash.plugins.PluginManager`

activate (*plugin*)

Activates a plugin, registering its hook callbacks to their respective hooks.

Parameters **plugin** – either a plugin object or a plugin name

activate_later (*plugin*)

Adds a plugin to the set of plugins pending activation. It can be removed from the queue with `deactivate_later()`

See also:

`activate_pending_plugins()`

activate_pending_plugins()

Activates all plugins queued with `activate_later()`

deactivate (*plugin*)

Deactivates a plugin, unregistering all of its hook callbacks

Parameters **plugin** – either a plugin object or a plugin name

deactivate_later (*plugin*)

Removes a plugin from the set of plugins pending activation.

See also:

`activate_pending_plugins()`

discover()

Iterates over all search paths and loads plugins

get_active_plugins()

Returns a dict mapping plugin names to currently active plugins

get_future_active_plugins ()
Returns a dictionary of plugins intended to be active once the ‘pending activation’ mechanism is finished

get_installed_plugins (*include_internals=True*)
Returns a dict mapping plugin names to currently installed plugins

get_plugin (*plugin_name*)
Retrieves a registered plugin by name, or raises a LookupError

install (*plugin, activate=False, activate_later=False, is_internal=False*)
Installs a plugin object to the plugin mechanism. *plugin* must be an object deriving from *slash.plugins.PluginInterface*.

is_internal_plugin (*plugin*)
Returns whether installed plugin is internal plugin

uninstall (*plugin*)
Uninstalls a plugin

uninstall_all ()
Uninstalls all installed plugins

3.20.11 Logging

class *slash.log.ColorizedFileHandler* (*filename, mode='a', encoding=None, level=0, format_string=None, delay=False, filter=None, bubble=False*)

should_colorize (*record*)
Returns *True* if coloring should be applied to this record. The default implementation returns *True* if the stream is a tty. If we are executing on Windows, colorama must be installed.

class *slash.log.ColorizedHandlerMixin*

get_color (*record*)
Returns the color for this record.

class *slash.log.ConsoleHandler* (***kw*)

emit (*record*)
Emit the specified logging record. This should take the record and deliver it to wherever the handler sends formatted log records.

format (*record*)
Formats a record with the given formatter. If no formatter is set, the record message is returned. Generally speaking the return value is most likely a unicode string, but nothing in the handler interface requires a formatter to return a unicode string.

The combination of a handler and formatter might have the formatter return an XML element tree for example.

class *slash.log.RetainedLogHandler* (**args, **kwargs*)

A logbook handler that retains the emitted logs in order to flush them later to a handler.

This is useful to keep logs that are emitted during session configuration phase, and not lose them from the session log

emit (*record*)

Emit the specified logging record. This should take the record and deliver it to wherever the handler sends formatted log records.

class slash.log.**SessionLogging** (*session, console_stream=None*)

A context creator for logging within a session and its tests

session_log_path = **None**

contains the path for the session logs

test_log_path = **None**

contains the path for the current test logs

class slash.log.**SilencedLoggersHandler** (*silence_logger_names*)

should_handle (*record*)

Returns *True* if this handler wants to handle the record. The default implementation checks the level.

slash.log.**add_log_handler** (*handler*)

Adds a log handler to be entered for sessions and for tests

slash.log.**set_log_color** (*logger_name, level, color*)

Sets the color displayed in the console, according to the logger name and level

3.20.12 Exceptions

slash.exception_handling.**handling_exceptions** (*fake_traceback=True, **kwargs*)

Context manager handling exceptions that are raised within it

Parameters

- **passthrough_types** – a tuple specifying exception types to avoid handling, raising them immediately onward
- **swallow** – causes this context to swallow exceptions
- **swallow_types** – causes the context to swallow exceptions of, or derived from, the specified types
- **context** – An optional string describing the operation being wrapped. This will be emitted to the logs to simplify readability

Note: certain exceptions are never swallowed - most notably KeyboardInterrupt, SystemExit, and SkipTest

slash.**allowing_exceptions** (*exception_class, msg=None*)

Allow subclass of **ARG1** to be raised during context:

```
>>> with allowing_exceptions (AttributeError) :  
...     raise AttributeError()  
>>> with allowing_exceptions (AttributeError) :  
...     pass
```

slash.exception_handling.**mark_exception** (*e, name, value*)

Associates a mark with a given value to the exception *e*

slash.exception_handling.**get_exception_mark** (*e, name, default=None*)

Given an exception and a label name, get the value associated with that mark label. If the label does not exist on the specified exception, *default* is returned.

`slash.exception_handling.noswallow(exception)`
 Marks an exception to prevent swallowing by `slash.exception_handling.get_exception_swallowing_context()`, and returns it

`slash.exception_handling.mark_exception_fatal(exception)`
 Causes this exception to halt the execution of the entire run.

This is useful when detecting errors that need careful examination, thus preventing further tests from altering the test subject's state

`slash.exception_handling.get_exception_swallowing_context(report_to_sentry=True)`
 Returns a context under which all exceptions are swallowed (ignored)

`slash.exception_handling.inhibit_unhandled_exception_traceback(exception)`
 Causes this exception to inhibit console traceback

3.20.13 Misc. Utilities

`slash.repeat(num_repetitions)`
 Marks a test to be repeated multiple times when run

3.20.14 Internals

class `slash.core.session.Session` (*reporter=None, console_stream=None*)
 Represents a slash session

get_total_num_tests()
 Returns the total number of tests expected to run in this session

results = None
 an aggregate result summing all test results and the global result

`slash.runner.run_tests(iterable, stop_on_error=None)`
 Runs tests from an iterable using the current session

class `slash.core.metadata.Metadata` (*factory, test*)
 Class representing the metadata associated with a test object. Generally available as `test.__slash__`

address
 String identifying the test, to be used when logging or displaying results in the console generally it is composed of the file path and the address inside the file

address_in_file = None
 Address string to identify the test inside the file from which it was loaded

id = None
 The test's unique id

module_name = None
 The path to the file from which this test was loaded

test_index0 = None
 The index of the test in the current execution, 0-based

test_index1
 Same as `test_index0`, only 1-based

class `slash.core.error.Error` (*msg=None, exc_info=None, frame_correction=0*)

exception

Deprecated since version 1.2.3: Use `error.exception_str`

exception_attributes

Deprecated since version 1.5.0.

func_name

Function name from which the error was raised

get_detailed_traceback_str()

Returns a formatted traceback string for the exception caught

lineno

Line number from which the error was raised

mark_fatal()

Marks this error as fatal, causing session termination

class `slash.core.result.Result` (*test_metadata=None*)

Represents a single result for a test which was run

add_error (*e=None, frame_correction=0, exc_info=None, append=True*)

Adds a failure to the result

add_exception (*exc_info=None*)

Adds the currently active exception, assuming it wasn't already added to a result

add_extra_log_path (*path*)

Add additional log path. This path will be added to the list returns by `get_log_paths`

add_failure (*e=None, frame_correction=0, exc_info=None, append=True*)

Adds a failure to the result

data = None

dictionary to be use by tests and plugins to store result-related information for later analysis

details = None

a `slash.core.details.Details` instance for storing additional test details

get_additional_details

Deprecated since version 0.20.0: Use `result.details.all()`

get_errors()

Returns the list of errors recorded for this result

Returns a list of `slash.core.error.Error` objects

get_failures()

Returns the list of failures recorded for this result

Returns a list of `slash.core.error.Error` objects

get_log_dir()

Returns log's directory.

get_log_path()

Returns log path

get_log_paths()

Returns a list of all log paths

is_just_failure()

Indicates this is a pure failure, without errors involved


```

set_log_path (path)
    Set log path

set_test_detail (key, value)
    Adds a generic detail to this test result, which can be later inspected or used

class slash.core.result.SessionResults (session)

    current
        Obtains the currently running result, if exists
        Otherwise, returns the global result object

    get_result (test)
        Returns the result stored belonging to a given test

    has_fatal_errors ()
        Indicates whether any result has an error marked as fatal (causing the session to terminate)

    is_interrupted ()
        Indicates if this session was interrupted

    is_success (allow_skips=False)
        Indicates whether this run is successful

        Parameters allow_skips – Whether to consider skips as unsuccessful

    iter_all_errors ()
        Iterates over all results which have errors
        yields tuples of the form (result, errors_list)

    iter_all_failures ()
        Iterates over all results which have failures
        yields tuples of the form (result, failure_list)

    iter_all_results ()
        Iterates over all results, ending with the global result object

    iter_test_results ()
        Iterates over results belonging to tests

class slash.core.details.Details (set_callback=None)

    append (key, value)
        Appends a value to a list key, or creates it if needed

    set (key, value)
        Sets a specific detail (by name) to a specific value

```

3.21 Changelog

- [#782](#): Added new hooks: `before_session_cleanup`, `after_session_end`
- [#779](#): Added `config.root.run.project_name`, which can be configured to hold the name of the current project. It defaults to the name of the directory in which your project's `.slashrc` is located

- [#785](#): Plugins can now be marked to indicate whether or not they support parallel execution, using `slash.plugins.parallel_mode`. To avoid errors, Slash assumes that unmarked plugins do not support parallel execution.
- [#772](#): Fix handling exceptions which raised from `None` in interactive session
- [#783](#): Session errors in children are now handled and reported when running with parallel
- [#767](#): Fixed traceback variable capture for cases where `self=None`
- [#662](#): Change email notification icon based on session success status
- [#660](#): Add configuration for notifications plugin `--notify-only-on-failure`
- [#661](#): Support PDB notifications by notifications plugin
- [#675](#): Emit native python warnings for logbook warning level
- [#686](#): `assert_raises` raises `ExpectedExceptionNotCaught` if exception wasn't caught also allowing inspection of the expected exception object
- [#689](#): Added a new hook, `interruption_added`, for registering exceptions which cause test/session interruptions
- [#658](#): Deprecate `PluginInterface.get_config()` and rename it to `PluginInterface.get_default_config()`
- [#692](#): Enhance errors summary log to session highlights log (configuration changed: `log.errors_subpath` -> `log.highlights_subpath`)
- [#685](#): `use.X` is now a shortcut for `use('x')` for fixture annotations
- `:` Suite files can now have a `repeat: X` marker to make the test run multiple times (Thanks @pierreluctg!)
- `:` During the execution of `error_added` hooks, traceback frame objects now have `python_frame`, containing the original Pythonic frame that yielded them. Those are cleared soon after the hook is called.
- [#704](#): Error objects now have their respective `exc_info` attribute containing the exception info for the current info (if available). This deprecates the use of the `locals/globals` attributes on traceback frames.
- [#698](#): By setting `log.traceback_variables` to `True`, traceback variable values will now be written to the debug log upon failures/errors
- [#712](#): Added `--pdb-filter` - a new command-line flag that allows the user to enter pdb only on specific caught exceptions, based on pattern matching (similar to `-k`)
- `:` Support fixture keyword arguments for `generator_fixture`
- [#723](#): Add configuration for resume state path location
- [#711](#): Logs can now optionally be compressed on-the-fly through the `log.compression.enabled` configuration parameter
- `-X` can now be used to turn off stop-on-error behavior. Useful if you have it on by default through a configuration file
- [#719](#): Added `log.core_log_level`, allowing limiting the verbosity of logs initiated from Slash itself
- [#681](#): Added a new hook, `log_file_closed`, and added configuration `log.cleanup` to enable removing log files after they are closed
- [#702](#): Rename `log.traceback_level` to `log.console_traceback_level`
- [#740](#): `session.results.current` is now a complete synonym for `slash.context.result`
- `:` Add `slash rerun -` given a `session_id`, run all the tests of this session

- #747: `session.results.global_result.is_success()` now returns `False` if any test in the session isn't successful
- #755: `timestamp` can now be used when formatting log path names
- #757: `slash list tests` now accepts the `--warnings-as-errors` flag, making it treat warnings it encounters as errors
- #752: Added `slash.ignore_warnings` to filter unwanted warnings during sessions
- #664: Added `metadata.set_file_path`, allowing integrations to set a custom file path to be associated with a loaded test
- : Added a configuration option preventing `slash.g` from being available in interactive namespaces
- #697: Added `slash.before_interactive_shell` hook
- #590: Add support for labeling parametrization variations
- #679: Fix coloring console for non TTY stdout
- #684: Optimize test loading with `--repeat-each` and `--repeat-all`
- : Fix tests loading order for some FS types
- #671: Help for `slash resume` is now more helpful
- #710: Fix sorting when `repeat-all` option is use
- #669: Session-scoped fixtures now properly register cleanups on session scope as expected
- #714: Session cleanups now happen under the global result object
- #721: Add timeout to sending emails through SMTP
- #700: Fixed handling of non-exception errors in session scope
- #701: Fixed error in coverage reporter cleanup
- #665: Support overriding notifications plugin's `from_email` by configuration
- #668: Properly initialize colorama under Windows
- #670: Improve handling of interruption exceptions - custom interruption exceptions will now properly cause the session and test to trigger the `session_interrupt` and `test_interrupt` hooks. Unexpected exceptions like `SystemExit` from within tests are now also reported properly instead of silently ignored
- : Add `current_config` property to plugins
- : Restore default enabled state for Prowl/NMA/Pushbullet notifications
- : Add ability to include details in email notifications
- #647: Support installing plugins as "internal" – thus not letting users disable or enable them through the command line
- #647: Support internal plugins
- #651: Add `host_fqdn` and `host_name` attributes to session
- #662: Improve notifications plugin, add support for email notifications
- : Added new hook `prepare_notification` to process notifications before being sent by the notifications plugin
- : Assertions coming from plugins and modules loaded from the project's `.slashrc` now also have assertion rewriting introspection enabled
- #556: Long variable representations are now capped by default when distilling tracebacks

- : Slash now detects test functions being redefined, hiding previous tests, and warns about it
- : Added `session.results.has_fatal_errors` to check for fatal errors within a session
- #595: Add *allowing_exceptions* context letting tests allow specific exceptions in selective context
- #600: Use *vintage* package for deprecations
- #592: Added `exception_attributes` dict to Error objects
- : Added `SLASH_USER_SETTINGS=x` environment variable to give a possibility to override the user slashrc file
- #633: When using the *handling_exceptions*, it is now possible to obtain the exception object that was handled
- #635: `slash run` now supports `--force-color/--no-color` flags.
- #617: Support `inhibit_unhandled_exception_traceback`
- #642: Support multiple registrations on the same plugin method with `plugins.registers_on`
- #596: Slash now supports a flag to disable assertion introspection on assertions containing messages (`run.message_assertion_introspection`)
- #213: Added parallel execution capability (still considered experimental) - tests can be run in parallel by multiple subprocess “workers”. See [the documentation](#) for more information
- : Several Windows-specific fixes (thanks Pierre-Luc Tessier Gagné)
- : Honor `run.default_sources` configuration when using `slash list` (thanks Pierre-Luc Tessier Gagné)
- #606: Swallow python warnings during `ast.parse`
- : Add `exception_str` shortcut for future compatibility on error objects
- #580: `tests_loaded` hooks now get called with a list of tests including the interactive test if applicable
- #581: Fix `slash.exclude` to work across fixture namespaces
- #564: Fix test collection bug causing tests to not be loaded with some plugins
- #502: Added `session_interrupt` hook for when sessions are interrupted
- #511: Support adding external logs `Result.add_extra_log_path` which will be retrieved by `Result.get_log_paths()`
- #507: Test id can now be obtained via `slash.context.test.id`
- #467: Yield fixtures are now automatically detected by Slash – using `yield_fixture` explicitly is no longer required
- #497: Major overhaul of CLI mechanics – improve help message and usage, as well as cleaner error exits during the session configuration phase
- #519: Add `--no-output` flag for `slash list`
- #512: `slash list-config` now receives a path filter for config paths to display
- #513: Add deep parametrization info (including nested fixtures) to the metadata variation info
- #524: `slash list`, `slash list-config` and `slash list-plugins` now supports `--force-color/--no-color` flags. The default changed from colored to colored only for tty
- #476: `slash resume` was greatly improved, and can now also fetch resumed tests from a recorded session in Backslash, if its plugin is configured
- #544: Added `debug.debugger` configuration to enable specifying preferred debugger. You can now pass `-o debug.debugger=ipdb` to prefer ipdb over pudb, for example

- #508: Added optional `end_message` argument to `notify_if_slow_context`, allowing better verbosity of long operations
- #529: Switch to PBR
- #510: Explicitly fail fixtures which name is valid for tests (currently: `test_ prefix`)
- #516: Fire `test_interrupt` earlier and properly mark session as interrupted when a test is interrupted
- #490: Fixed behavior of plugin dependencies in cases involving mixed usage of plugin-level and hook-level dependencies
- #551: Fix stopping on error behavior when errors are reported on previous tests
- #466: Add `--relative-paths` flag to `slash list`
- #344: Exceptions recorded with `handling_exceptions` context now properly report the stack frames above the call
- : Added the `entering_debugger` hook to be called before actually entering a debugger
- #468: Slash now detects tests that accidentally contain `yield` statements and fails accordingly
- #461: `yield_fixture` now honors the `scope` argument
- #403: add `slash list-plugins` to show available plugins and related information
- #462: Add `log.errors_subpath` to enable log files only recording added errors and failures.
- #400: `slash.skipped` decorator is now implemented through the requirements mechanism. This saves a lot of time in unnecessary setup, and allows multiple skips to be assigned to a single test
- #384: Accumulate logs in the configuration phase of sessions and emit them to the session log. Until now this happened before logging gets configured so the logs would get lost
- #195: Added `this.test_start` and `this.test_end` to enable fixture-specific test start and end hooks while they're active
- #287: Add support for "facts" in test results, intended for coverage reports over relatively narrow sets of values (like OS, product configuration etc.)
- #352: Suite files can now contain filters on specific items via a comment beginning with `filter:`, e.g. `/path/to/test.py # filter: x and not y`
- #362: Add ability to intervene during test loading and change run order. This is done with a new `tests_loaded` hook and a new field in the test metadata controlling the sort order. See [the cookbook](#) for more details
- #359: Add trace logging of fixture values, including dependent fixtures
- #417: `add_error/add_failure` can now receive both message and `exc_info` information
- #484: `slash list` now indicates fixtures that are overriding outer fixtures (e.g. from `slashconf.py`)
- #369: Add `slash.exclude` to only skip specific parametrizations of a specific test or a dependent fixture. See [the cookbook section](#) for more details
- #485: xunit plugin now saves the run results even when the session doesn't end gracefully (Thanks @eplaut)
- #464: Fix `exc_info` leaks outside of `assert_raises` & `handling_exceptions`
- #477: Fix `assert_raises` with message for un-raised exceptions
- #479: When installing and activating plugins and activation fails due to incompatibility, the erroneous plugins are now automatically uninstalled
- #483: Properly handle possible exceptions when examining traceback object attributes

- #481: Fixed tuple parameters for fixtures
- #457: Fixed initialization order for *autouse* fixtures
- #464: Fix reraising behavior from `handling_exceptions`
- #412: Add `is_in_test_code` to `traceback` json
- #413: Test names inside files are now sorted
- #416: Add `--no-params` for “slash list”
- #427: Drop support for Python 2.6
- #428: Requirements using functions can now have these functions return tuples of (fulfilled, requirement_message) specifying the requirement message to display
- #430: Added coverage plugin to generate code coverage report at the end of the run (`--with-coverage`)
- #435: Added `swallow_types` argument to `exception_handling` context to enable selective swallowing of specific exceptions
- #436: `slash list` now fails by default if no tests are listed. This can be overridden by specifying `--allow-empty`
- #424: slash internal app context can now be instructed to avoid reporting to console (use `report=False`)
- #437: Added `test_avoided` hook to be called when tests are completely skipped (e.g. requirements)
- #423: Added support for generator fixtures
- #401: `session_end` no longer called on plugins when `session_start` isn’t called (e.g. due to errors with other plugins)
- #441: `variation` in test metadata now contains both `id` and `values`. The former is a unique identification of the test variation, whereas the latter contains the actual fixture/parameter values when the test is run
- #439: Added support `yield_fixture`
- #276: Added support for fixture aliases using `slash.use`
- #407: Added `--repeat-all` option for repeating the entire suite several times
- #397: Native Python warnings are now captured during testing sessions
- #446: Exception tracebacks now include instance attributes to make debugging easier
- #447: Added a more stable sorting logic for cartesian products of parametrizations
- #442: Prevent `session_end` from being called when `session_start` doesn’t complete successfully
- #432: Fixed a bug where session cleanups happened before `test_end` hooks are fired
- #434: Fixed a bug where class names were not deduced properly when loading tests
- #409: Improve session startup/shutdown logic to avoid several potentially invalid states
- #410: Fixed bug causing incorrect test frame highlighting in tracebacks
- #339: Errors in interactive session (but not ones originating from IPython input itself) are now recorded as test errors
- #379: Allow exception marks to be used on both exception classes and exception values
- #385: Add test details to xunit plugin output
- #386: Make slash list support `-f` and other configuration parameters
- #391: Add `result.details`, giving more options to adding/appending test details

- #395: Add `__slash__.variation`, enabling investigation of exact parametrization of tests
- #398: Allow specifying `exc_info` for `add_error`
- #381: `handling_exceptions` now doesn't handle exceptions which are currently expected by `assert_raises`
- #388: `-k` can now be specified multiple times, implying AND relationship
- #405: Add `--show-tags` flag to `slash list`
- #348: Color test code differently when displaying tracebacks
- #402: `TerminatedException` now causes interactive sessions to terminate
- #406: Fix error reporting for session scoped cleanups
- #408: Fix handling of cleanups registered from within cleanups
- : Minor fixes
- #390: Fix handling of `add_failure` and `add_error` with message strings in xunit plugin
- #389: Fix deduction of function names for parametrized tests
- #383: Fix fixture passing to `before` and `after`
- #376: Fix xunit bug when using skip decorators without reasons
- #374: Fix issue with xunit plugin
- #349: Plugin configuration is now installed in the installation phase, not activation phase
- #371: Add `warning_added` hook
- #366: Added `configure` hook which is called after command-line processing but before plugin activation
- #366: `--with-X` and `--without-X` don't immediately activate plugins, but rather use `activate_later` / `deactivate_later`
- #366: Added `activate_later` and `deactivate_later` to the plugin manager, allowing plugins to be collected into a 'pending activation' set, later activated with `activate_pending_plugins`
- #368: add `slash list-config` command
- #361: Demote slash logs to TRACE level
- #373: Fix test collection progress when outputting to non-ttys
- #372: Fixed logbook compatibility issue
- #350: Fixed scope mismatch bug when hooks raise exceptions
- #319: Add `class_name` metadata property for method tests
- #324: Add test for cleanups with fatal exceptions
- #240: Add support for test tags
- #332: Add ability to filter by test tags - you can now filter with `-k tag:sometag`, `-k sometag=2` and `-k "not sometag=3"`
- #333: Allow customization of console colors
- #337: Set `tb` level to 2 by default
- #233: `slash.parametrize`: allow argument tuples to be specified
- #279: Add option to silence manual `add_error` tracebacks (`-o show_manual_errors_tb=no`)

- #295: SIGTERM handling for stopping sessions gracefully
- #321: add `Error.mark_fatal()` to enable calls to `mark_fatal` right after `add_error`
- #335: Add ‘needs’ and ‘provides’ to plugins, to provide fine-grained flow control over plugin calling
- #347: Add `slash.context.fixture` to point at the ‘this’ variable of the currently computing fixture
- #320: Fix scope mechanism to allow cleanups to be added from `test_start` hooks
- #322: Fix behavior of skips thrown from cleanup callbacks
- #322: Refactored a great deal of the test running logic for easier maintenance and better solve some corner cases
- #329: `handling_exceptions(swallow=True)` now does not swallow `SkipTest` exceptions
- #341: Make sure tests are garbage collected after running
- #308: Support registering private methods in plugins using `registers_on`
- #311: Support plugin methods avoiding hook registrations with `registers_on(None)`
- #312: Add `before_session_start` hook
- #314: Added `Session.get_total_num_tests` for returning the number of tests expected to run in a session
- : fix strict import dependency
- #300: Add `log.unified_session_log` flag to make session log contain all logs from all tests
- #306: Allow class variables in plugins
- #307: Interactive test is now a first-class test and allows any operation that is allowed from within a regular test
- #271: Add `passthrough_types=TYPES` parameter to `handling_exceptions` context
- #275: Add `get_no_deprecations_context` to disable deprecation messages temporarily
- #274: Add optional separation between console log format and file log format
- #280: Add optional message argument to `assert_raises`
- #170: Add optional `scope` argument to `add_cleanup`, controlling when the cleanup should take place
- #267: Scoped cleanups: associate errors in cleanups to their respective result object. This means that errors can be added to tests after they finish from now on.
- #286: Better handling of unrun tests when using `x` or similar. Count of unrun tests is now reported instead of detailed console line for each unrun test.
- #282: Better handling of fixture dependency cycles
- #289: Added `get_config` optional method to plugins, allowing them to supplement configuration to `config.root.plugin_config.<plugin_name>`
- #288: Fixed accidental log file line truncation
- #285: Fixed representation of fixture values that should not be printable (strings with slashes, for instance)
- #270: Fixed handling of directory names and class/method names in suite files
- #264: Allow specifying location of `.slashrc` via configuration
- #263: Support writing colors to log files
- #257: `slash fixtures` is now `slash list`, and learned the ability to list both fixtures and tests
- : `start_interactive_shell` now automatically adds the contents of `slash.g` to the interactive namespace

- #268: Treat relative paths listed in suite files (-f) relative to the file's location
- #269: Add option to specify suite files within suite files
- : Slash now emits a console message when session_start handlers take too long
- #249: Added @slash.repeat decorator to repeat tests multiple times
- #140: Added --repeat-each command line argument to repeat each test multiple times
- #258: Added hooks.error_added, a hook that is called when an error is added to a test result or to a global result. Also works when errors are added after the test has ended.
- #261: Added a traceback to manually added errors (through slash.add_error and friends)
- : Add slash.session.reporter.report_fancy_message
- #177: Added 'slash fixtures' command line utility to list available fixtures
- #211: Added log.last_session_dir_symlink to create symlinks to log directory of the last run session
- #220: slash.add_cleanup no longer receives arbitrary positional args or keyword args. The old form is still allowed for now but issues a deprecation warning.
- #226: Implemented slash.hooks.before_test_cleanups.
- #189: add add_success_only_cleanup
- #196: Add 'slash version' to display current version
- #199: A separate configuration for traceback verbosity level (log.traceback_level, also controlled via --tb=[0-5])
- #203: Group result output by tests, not by error type
- #209: Test cleanups are now called before fixture cleanups
- #16: Added slash.requires decorator to formally specify test requirements
- #214: Added slash.nofixtures decorator to opt out of automatic fixture deduction.
- #204: Fixed a console formatting issue causing empty lines to be emitted without reason
- #198: fix test_methodname accidentally starting with a dot
- #183: Add slash.parameters.toggle as a shortcut for iterating [True, False]
- #179: Documentation overhaul
- #190: Support __slash__.test_index0 and __slash__.test_index1 for easier enumeration in logs
- #194: add assert_almost_equal
- #171: Add error times to console reports
- #160: Add option to serialize warnings to dicts
- : Log symlinks can now be relative paths (considered relative to the logging root directory)
- #162: Test loading and other setup operations now happen before session_start, causing faster failing on simple errors
- #163: Added -k for selecting tests by substrings
- #159: Add optional 'last failed' symlink to point to last failed test log
- #167: Fixed erroneous behavior in which skipped tasks after using -x caused log symlinks to move
- : removed the test contexts facility introduced in earlier versions. The implementation was partial and had serious drawbacks, and is inferior to fixtures.

- [#127](#): `py.test` style fixture support, major overhaul of tests and loading code.
- : Fixed error summary reporting
- [#144](#): Add option to colorize console logs in custom colors
- [#149](#): Make console logs interact nicely with the console reporter non-log output
- [#146](#): Add test id and error/failure enumeration in test details
- [#145](#): Add option to save symlinks to the last session log and last test log
- [#150](#): Add log links to results when reporting to console
- [#137](#): Fixed parameter iteration across inheritance trees
- : Renamed `debug_hooks` to `debug_hook_handlers`. Debugging hook handlers will only trigger for slash hooks.
- [#148](#): Detailed tracebacks now emitted to log file
- [#152](#): Truncate long log lines in the console output
- [#153](#): Report warnings at the end of sessions
- [#143](#): Use `gossip`'s internal handler exception hook to debug hook failures when `--pdb` is used
- [#142](#): Allow registering plugin methods on custom hooks
- : Overhaul the reporting mechanism, make output more similar to `py.test`'s, including better error reporting.
- [#128](#): Slash now loads tests eagerly, failing earlier for bad imports etc. This might change in the future to be an opt-out behavior (change back to lazy loading)
- [#129](#): Overhaul rerunning logic (now called 'resume')
- [#141](#): Add `slash.utils.deprecated` to mark internal facilities bound for removal
- [#138](#): Move to `gossip` as hook framework.
- : Added assertion introspection via AST rewrite, borrowed from `pytest`.
- [#132](#): Support for providing hook requirements to help resolving callback order (useful on initialization)
- [#115](#): Add `session.logging.extra_handlers` to enable adding custom handlers to tests and the session itself
- [#120](#): Support multiple exception types in `should.raise_exception`
- [#121](#): Support 'append' for CLI arguments deduced from config
- [#116](#): Support '-f' to specify one or more files containing lists of files to run
- [#114](#): Support for fatal exception marks
- [#113](#): Add option to debug hook exceptions (`-o debug.debug_hooks=yes`)
- [#19](#): Add ability to add non-exception errors and failures to test results
- [#96](#): Add option to specify logging format
- [#103](#): Add `context.test_filename`, `context.test_classname`, `context.test_methodname`
- [#75](#): Support matching by parameters in FQN, Support running specific or partial tests via FQN
- : Add `should.be_empty`, `should.not_be_empty`
- [#69](#): Move `slash.session` to `slash.core.session`. `slash.session` is now the session context proxy, as documented
- : Documentation additions and enhancements
- : Coverage via coveralls

- [#26](#): Support test rerunning via “slash rerun”
- [#72](#): Clarify errors in plugins section
- [#74](#): Enable local .slashrc file
- [#45](#): Add option for specifying default tests to run
- [#5](#): add_critical_cleanup for adding cleanups that are always called (even on interruptions)
- [#3](#): Handle KeyboardInterrupts (quit fast), added the test_interrupt hook
- [#48](#)., [#54](#): handle import errors and improve captured exceptions
- [:](#) Renamed slash.fixture to slash.g (fixture is an overloaded term that will maybe refer to test contexts down the road)
- [#40](#):: Added test context support - you can now decorate tests to provide externally implemented contexts for more flexible setups
- [#46](#):: Added plugin.activate() to provide plugins with the ability to control what happens upon activation

3.22 Development

Slash tries to bring a lot of features to the first releases. For starters, the very first usable version (0.0.1) aims at providing basic running support and most of the groundwork needed for the following milestones.

All changes are checked against [Travis](#). Before committing you should test against supported versions using `tox`, as it runs the same job being run by travis. For more information on Slash’s internal unit tests see [Unit Testing Slash](#).

Development takes place on [github](#). Feel free to open issues or pull requests, as a lot of the project’s success depends on your feedback!

I normally do my best to respond to issues and PRs as soon as possible (hopefully within one day). Don’t hesitate to ping me if you don’t hear from me - there’s a good chance I missed a notification or something similar.

3.23 Contributors

Special thanks go to these people for taking the time in improving Slash and providing feedback:

- Alon Horev ([@alinho](#))
- Omer Gertel
- Pierre-Luc Tessier Gagné

3.24 Unit Testing Slash

The following information is intended for anyone interested in developing Slash or adding new features, explaining how to effectively use the unit testing facilities used to test Slash itself.

3.24.1 The Suite Writer

The unit tests use a dedicated mechanism allowing creating a virtual test suite, and then easily writing it to a real directory, run it with Slash, and introspect the result.

The suite writer is available from `tests.utils.suite_writer`:

```
>>> from tests.utils.suite_writer import Suite
>>> suite = Suite()
```

Basic Usage

Add tests by calling `add_test()`. By default, this will pick a different test type (function/method) every time.

```
>>> for i in range(10):
...     test = suite.add_test()
```

The created **test object** is not an actual test that can be run by Slash – it is an object representing a future test to be created. The test can later be manipulated to perform certain actions when run or to expect things when run.

The simplest thing we can do is run the suite:

```
>>> summary = suite.run()
>>> len(summary.session.results)
10
>>> summary.ok()
True
```

We can, for example, make our test raise an exception, thus be considered an error:

```
>>> test.when_run.raise_exception()
```

Now let's run the suite again (it will commit itself to a new path so we can completely disregard the older session):

```
>>> summary = suite.run()
>>> summary.session.results.get_num_errors()
1
>>> summary.ok()
False
```

The suite writer already takes care of verifying that the errored test is actually reported as error and fails the run.

Adding Parameters

To test parametrization, the suite writer supports adding parameters and fixtures to test. First we will look at parameters (translating into `@slash.parametrize` calls):

```
>>> suite.clear()
>>> test = suite.add_test()
>>> p = test.add_parameter()
>>> len(p.values)
3
>>> suite.run().ok()
True
```

Adding Fixtures

Fixtures are slightly more complex, since they have to be added to a file first. You can create a fixture at the file level:

```
>>> suite.clear()
>>> test = suite.add_test()

>>> f = test.file.add_fixture()
>>> _ = test.depend_on_fixture(f)
>>> suite.run().ok()
True
```

Fixtures can also be added to the slashconf file:

```
>>> f = suite.slashconf.add_fixture()
```

Fixtures can depend on each other and be parametrized:

```
>>> suite.clear()
>>> f1 = suite.slashconf.add_fixture()
>>> test = suite.add_test()
>>> f2 = test.file.add_fixture()
>>> _ = f2.depend_on_fixture(f1)
>>> _ = test.depend_on_fixture(f2)
>>> p = f1.add_parameter()
>>> summary = suite.run()
>>> summary.ok()
True
>>> len(summary.session.results) == len(p.values)
True
```

You can also control the fixture scope:

```
>>> f = suite.slashconf.add_fixture(scope='module')
>>> _ = suite.add_test().depend_on_fixture(f)
>>> suite.run().ok()
True
```

And specify autouse (or implicit) fixtures:

```
>>> suite.clear()
>>> f = suite.slashconf.add_fixture(scope='module', autouse=True)
>>> t = suite.add_test()
>>> suite.run().ok()
True
```

3.25 Parallel Test Execution

By default, Slash runs tests sequentially through a single session process. However, it is also possible to use Slash to run tests in parallel. In this mode, slash will run a ‘parent’ session process that will be used to distribute the tests, and a number of child session processes that will receive the distributed tests and run them.

3.25.1 Running in Parallel Mode

In order to run tests in parallel, just add `--parallel` and the number of workers you want to start. For example:

```
$ slash run /path/to/tests --parallel 4
```

If, for instance, most of your tests are CPU-bound, it would make sense to run them like this:

```
$ slash run /path/to/tests --parallel $(nproc)
```

to use a single worker per CPU core.

Note: The parallel mechanism works by listening on a local TCP socket, to which the worker session processes connect and receive test descriptions via RPC. In case you want, you can control the address and/or port settings via the `--parallel-addr` and `--parallel-port` command-line arguments.

By default, only the parent session process outputs logs to the console. For a more controlled run you can use `tmux` to run your workers, so that you can examine their outputs:

```
$ slash run /path/to/tests --parallel 4 --tmux [--tmux-panes]
```

If `--tmux-panes` is specified, a new pane will be opened for every worker, letting it emit console output. Otherwise each worker will open a new window.

3.25.2 The Parallel Execution Mechanism

When running Slash in parallel mode, the main process starts a server and a number of workers as new processes. The server then waits until all the workers connect and start collecting tests. Only after all the workers connect and validate that all of them collected the same tests collection, the test execution will start:

- Each worker asks the master process for a test.
- The master process gives them one test to execute.
- The worker executes the test and reports the test's results to the parent.
- The worker asks for the next test and so on, until all tests are executed.
- The worker processes disconnect from the server, and the server terminates.

3.25.3 Worker session ids

Each worker will have a `session_id` that starts with the server's `session_id`, and ends with its `client_id`.

For example, if the server's `session_id` is `496272f0-66dd-11e7-a4f0-00505699924f_0` and there are 2 workers, their session ids will be:

- `496272f0-66dd-11e7-a4f0-00505699924f_1`
- `496272f0-66dd-11e7-a4f0-00505699924f_2`

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

S

`slash.core.session`, [67](#)
`slash.hooks`, [63](#)
`slash.log`, [65](#)
`slash.runner`, [67](#)

Symbols

`__init__()` (slash.warnings.SessionWarnings method), 62
`__iter__()` (slash.warnings.SessionWarnings method), 63
`__weakref__` (slash.warnings.SessionWarnings attribute), 63

A

`abstract_test_class()` (in module slash), 60
`activate()` (slash.plugins.PluginInterface method), 63
`activate()` (slash.plugins.PluginManager method), 64
`activate_later()` (slash.plugins.PluginManager method), 64
`activate_pending_plugins()`
 (slash.plugins.PluginManager method), 64
`active()` (in module slash.plugins), 63
`add_cleanup()` (in module slash), 61
`add_critical_cleanup()` (in module slash), 61
`add_error`, 22
`add_error()` (in module slash), 23
`add_error()` (slash.core.result.Result method), 68
`add_exception()` (slash.core.result.Result method), 68
`add_extra_log_path()` (slash.core.result.Result method), 68
`add_failure`, 22
`add_failure()` (in module slash), 23
`add_failure()` (slash.core.result.Result method), 68
`add_log_handler()` (in module slash.log), 66
`add_success_only_cleanup()` (in module slash), 61
`adding`
 errors, 22
 failures, 22
`address` (slash.core.metadata.Metadata attribute), 67
`address_in_file` (slash.core.metadata.Metadata attribute), 67
`after()` (slash.Test method), 60
`allowing_exceptions()` (in module slash), 66
`append()` (slash.core.details.Details method), 69
`assert_almost_equal()` (in module slash), 61
`assert_raises()` (in module slash), 60

B

`before()` (slash.Test method), 60

C

`ColorizedFileHandler` (class in slash.log), 65
`ColorizedHandlerMixin` (class in slash.log), 65
`configure_argument_parser()`
 (slash.plugins.PluginInterface method), 63
`configure_from_parsed_args()`
 (slash.plugins.PluginInterface method), 63
`console`
 log, 7
`ConsoleHandler` (class in slash.log), 65
`couple: metadata`
 test, 55
`couple: test`
 metadata, 55
`current` (slash.core.result.SessionResults attribute), 69
`current_config` (slash.plugins.PluginInterface attribute), 63

D

`data` (slash.core.result.Result attribute), 68
`deactivate()` (slash.plugins.PluginInterface method), 63
`deactivate()` (slash.plugins.PluginManager method), 64
`deactivate_later()` (slash.plugins.PluginManager method), 64
`debugging`, 23
 hooks, 45
`details`
 notification, 54
`Details` (class in slash.core.details), 69
`details` (slash.core.result.Result attribute), 68
`discover()` (slash.plugins.PluginManager method), 64

E

`emit()` (slash.log.ConsoleHandler method), 65
`emit()` (slash.log.RetainedLogHandler method), 65
`Error` (class in slash.core.error), 67

errors, [22](#)

 adding, [22](#)

errors in

 hooks, [45](#)

exception (slash.core.error.Error attribute), [67](#)

exception_attributes (slash.core.error.Error attribute), [68](#)

exceptions, [23](#)

exceptions in

 hooks, [45](#)

exclude, [14](#)

F

failures

 adding, [22](#)

fixture() (in module slash), [62](#)

fixtures

 requirements, [18](#)

format() (slash.log.ConsoleHandler method), [65](#)

func_name (slash.core.error.Error attribute), [68](#)

G

generator_fixture() (in module slash), [62](#)

get_active_plugins() (slash.plugins.PluginManager method), [64](#)

get_additional_details (slash.core.result.Result attribute), [68](#)

get_color() (slash.log.ColorizedHandlerMixin method), [65](#)

get_config() (slash.plugins.PluginInterface method), [64](#)

get_default_config() (slash.plugins.PluginInterface method), [64](#)

get_description() (slash.plugins.PluginInterface method), [64](#)

get_detailed_traceback_str() (slash.core.error.Error method), [68](#)

get_errors() (slash.core.result.Result method), [68](#)

get_exception_mark() (in module slash.exception_handling), [66](#)

get_exception_swallowing_context() (in module slash.exception_handling), [67](#)

get_failures() (slash.core.result.Result method), [68](#)

get_future_active_plugins() (slash.plugins.PluginManager method), [64](#)

get_installed_plugins() (slash.plugins.PluginManager method), [65](#)

get_log_dir() (slash.core.result.Result method), [68](#)

get_log_path() (slash.core.result.Result method), [68](#)

get_log_paths() (slash.core.result.Result method), [68](#)

get_name() (slash.plugins.PluginInterface method), [64](#)

get_plugin() (slash.plugins.PluginManager method), [65](#)

get_result() (slash.core.result.SessionResults method), [69](#)

get_total_num_tests() (slash.core.session.Session method), [67](#)

H

handling_exceptions() (in module slash.exception_handling), [66](#)

has_fatal_errors() (slash.core.result.SessionResults method), [69](#)

hooks

 debugging, [45](#)

 errors in, [45](#)

 exceptions in, [45](#)

I

id (slash.core.metadata.Metadata attribute), [67](#)

ignore_warnings() (in module slash), [63](#)

inhibit_unhandled_exception_traceback() (in module slash.exception_handling), [67](#)

install() (slash.plugins.PluginManager method), [65](#)

internal

 plugins, [49](#)

interrupting, [22](#)

is_internal_plugin() (slash.plugins.PluginManager method), [65](#)

is_interrupted() (slash.core.result.SessionResults method), [69](#)

is_just_failure() (slash.core.result.Result method), [68](#)

is_success() (slash.core.result.SessionResults method), [69](#)

iter_all_errors() (slash.core.result.SessionResults method), [69](#)

iter_all_failures() (slash.core.result.SessionResults method), [69](#)

iter_all_results() (slash.core.result.SessionResults method), [69](#)

iter_test_results() (slash.core.result.SessionResults method), [69](#)

iterate() (in module slash.core.fixtures.parameters), [60](#)

K

KeyboardInterrupt, [22](#)

L

lineno (slash.core.error.Error attribute), [68](#)

log
 console, [7](#)

M

mark_exception() (in module slash.exception_handling), [66](#)

mark_exception_fatal() (in module slash.exception_handling), [67](#)

mark_fatal() (slash.core.error.Error method), [68](#)

metadata, [55](#)

Metadata (class in slash.core.metadata), [67](#)

module_name (slash.core.metadata.Metadata attribute), [67](#)

N

nofixtures() (in module slash), 62
 noswallow() (in module slash.exception_handling), 66
 notification
 details, 54

P

parallel, 81
 plugins, 53
 test execution, 81
 parallel_mode() (in module slash.plugins), 63
 parametrize() (in module slash), 60
 PluginInterface (class in slash.plugins), 63
 PluginManager (class in slash.plugins), 64
 plugins, 49
 internal, 49
 parallel, 53

R

register() (in module slash.hooks), 63
 register_if() (in module slash.plugins), 63
 register_skip_exception() (in module slash), 61
 registers_on() (in module slash.plugins), 63
 repeat() (in module slash), 67
 requirements, 8
 fixtures, 18
 requires() (in module slash), 62
 Result (class in slash.core.result), 68
 results (slash.core.session.Session attribute), 67
 RetainedLogHandler (class in slash.log), 65
 run() (slash.Test method), 60
 run_tests() (in module slash.runner), 67

S

Session (class in slash.core.session), 67
 session_log_path (slash.log.SessionLogging attribute), 66
 SessionLogging (class in slash.log), 66
 SessionResults (class in slash.core.result), 69
 SessionWarnings (class in slash.warnings), 62
 set() (slash.core.details.Details method), 69
 set_log_color() (in module slash.log), 66
 set_log_path() (slash.core.result.Result method), 68
 set_test_detail() (in module slash), 9
 set_test_detail() (slash.core.result.Result method), 69
 should_colorize() (slash.log.ColorizedFileHandler method), 65
 should_handle() (slash.log.SilencedLoggersHandler method), 66
 SilencedLoggersHandler (class in slash.log), 66
 skip_test() (in module slash), 61
 skipped() (in module slash), 61
 skipping
 tests, 8

SkipTest (class in slash.exceptions), 61
 slash.core.session (module), 67
 slash.exclude, 14
 slash.hooks (module), 63
 slash.log (module), 65
 slash.runner (module), 67

T

tag() (in module slash), 61
 Test (class in slash), 60
 test execution
 parallel, 81
 test_index0 (slash.core.metadata.Metadata attribute), 67
 test_index1 (slash.core.metadata.Metadata attribute), 67
 test_log_path (slash.log.SessionLogging attribute), 66
 tests
 skipping, 8
 toggle() (in module slash.core.fixtures.parameters), 60

U

uninstall() (slash.plugins.PluginManager method), 65
 uninstall_all() (slash.plugins.PluginManager method), 65
 use() (in module slash), 62

Y

yield_fixture() (in module slash), 62