Slash Framework

QuanDA11

Contents

I. Running Tests

- The main front-end for Slash is the slash run utility, invoked from the command line.
- By default, it receives the path to load and run tests from:

slash run /path/to/tests

1. Verbosity:

- **Verbosity** refers to the amount of information or detail that is displayed during the execution of tests.
    - Increasing verbosity with " -v" – more information

slash run -v

    - Decreasing verbosity with "-q" – less information

slash run -q

```
D:\project\Slash>slash run -v test_parameters.py
2 tests collected
[2024-05-02 03:55:04] #1: test_parameters.py:test_power_of_two(with_power_operator=True)
[2024-05-02 03:55:04] #2: test_parameters.py:test_power_of_two(with_power_operator=False)
======================== Session ended. 2 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================

D:\project\Slash>slash run  test_parameters.py
2 tests collected
test_parameters.py ..  PASS
======================== Session ended. 2 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================

D:\project\Slash>slash run -q  test_parameters.py
2 tests collected
..
======================== Session ended. 2 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

- **Tracebacks**: Tracebacks are error messages that provide information about what went wrong when a test fails.
    - Setting traceback verbosity level with --tb: You can specify the verbosity level of tracebacks using the --tb flag followed by a number from 0 to 5.

slash run --tb=2

2. Loading Tests from Files

- **Loading tests:** You can also read tests from file or files which contain paths to run. Whitespaces and lines beginning with a comment # will be ignored:
    - -f <file>: Specifies a file containing paths to tests to be executed.
    - Comments (#) and whitespace in files are ignored.

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

```
D:\project\Slash>slash run -f my_suite_file.txt
3 tests collected
D:\project\Slash\test_addition.py .  PASS
D:\project\Slash\test_parameters.py ..  PASS
======================== Session ended. 3 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

- **Filtering Tests:** You can include a filter directive in the file to restrict which tests are actually loaded from that file. This is useful if you want to exclude certain tests based on specific criteria.
    - Use # filter: <filter_condition> to restrict which tests are loaded from a file based on a condition.

 # my_suite_file.txt

 # this is the first test file

/path/to/tests.py

/path/to/other_tests.py # filter: not dangerous

```
D:\project\Slash>slash run -v -f FilteringTests.txt
3 tests collected
[2024-05-02 04:12:40] #1: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=True)
[2024-05-02 04:12:40] #2: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=False)
[2024-05-02 04:12:40] #3: D:\project\Slash\test_addition.py:test_safe_addition
====================== Session ended. 3 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================

D:\project\Slash>slash run -v -f FilteringTests.txt
4 tests collected
[2024-05-02 04:12:58] #1: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=True)
[2024-05-02 04:12:58] #2: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=False)
[2024-05-02 04:12:58] #3: D:\project\Slash\test_addition.py:test_dangerous_addition
[2024-05-02 04:12:58] #4: D:\project\Slash\test_addition.py:test_safe_addition
====================== Session ended. 4 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

- **Repeating Lines:** You can include a repeat directive to repeat a line (path to a test) multiple times. This can be useful if you want to execute the same test(s) multiple times.
  - Use # repeat: <count> to repeat a line (path to a test) multiple times.

 # my_suite_file.txt

 # the next line will be repeated twice

/path/to/other_tests.py # repeat: 2

/path/to/other_tests.py # filter: not dangerous, repeat: 2

```
D:\project\Slash>slash run -v -f RepeatingLines.txt
4 tests collected
[2024-05-02 06:46:31] #1: D:\project\Slash\test_addition.py:test_dangerous_addition
[2024-05-02 06:46:31] #2: D:\project\Slash\test_addition.py:test_safe_addition
[2024-05-02 06:46:32] #3: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=True)
[2024-05-02 06:46:32] #4: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=False)
====================== Session ended. 4 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================

D:\project\Slash>slash run -v -f RepeatingLines.txt
8 tests collected
[2024-05-02 06:46:44] #1: D:\project\Slash\test_addition.py:test_dangerous_addition
[2024-05-02 06:46:44] #2: D:\project\Slash\test_addition.py:test_safe_addition
[2024-05-02 06:46:44] #3: D:\project\Slash\test_addition.py:test_dangerous_addition
[2024-05-02 06:46:44] #4: D:\project\Slash\test_addition.py:test_safe_addition
[2024-05-02 06:46:44] #5: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=True)
[2024-05-02 06:46:44] #6: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=False)
[2024-05-02 06:46:44] #7: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=True)
[2024-05-02 06:46:44] #8: D:\project\Slash\test_parameters.py:test_power_of_two(with_power_operator=False)
====================== Session ended. 8 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

3. Debugging & Failures:

- **Debugging with --pdb:** Debugging allows you to investigate and understand why a test is failing by inspecting variables, stepping through code, and identifying the source of the issue. Slash Framework provides a convenient way to debug tests using the --pdb flag, which invokes the Python debugger.
  - **Debugging mode:**
    - **“n” or next:** Execute the next line of code.
    - **“s” or step:** Step into the next function call.
    - **“c” or continue:** Continue execution until the next breakpoint or until the program finishes.

- **“r” or return:** Continue execution until the current function returns.
- **print function** to inspect the value.
- **“break” command:** set breakpoints at specific lines in your code

slash run –pdb

- **Stopping at the First Unsuccessful Test with -x:** Sometimes, you may want to stop test execution as soon as the first test failure occurs, especially during debugging or when you want to focus on fixing one issue at a time. Slash provides the -x flag for this purpose.

slash run -x

4. Including and Excluding Tests:

- The **-k flag** to slash run is a versatile way to include or exclude tests.
  - Use **-k <substring>** to include only tests containing the specified substring in their names.

slash run -k substr /path/to/tests

```
D:\project\Slash>slash run -v -k division test_including_excluding.py
1 tests collected
[2024-05-02 07:15:55] #1: test_including_excluding.py:test_division
======================== Session ended. 1 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

  - Use **-k 'not <substring>'** to exclude tests containing the specified substring in their names.

slash run -k “not failing_” /path/to/tests

```
D:\project\Slash>slash run -v -k "not failing_" test_including_excluding.py
7 tests collected
[2024-05-02 07:16:20] #1: test_including_excluding.py:test_addition
[2024-05-02 07:16:20] #2: test_including_excluding.py:test_component_one
[2024-05-02 07:16:20] #3: test_including_excluding.py:test_component_two
[2024-05-02 07:16:20] #4: test_including_excluding.py:test_division
[2024-05-02 07:16:20] #5: test_including_excluding.py:test_exponential
[2024-05-02 07:16:20] #6: test_including_excluding.py:test_multiplication
[2024-05-02 07:16:20] #7: test_including_excluding.py:test_subtraction
======================== Session ended. 7 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

  - You can use more complex expressions involving or and and to include or exclude tests based on multiple criteria.

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

```
D:\project\Slash>slash run -v -k "not failing_ and component" test_including_excluding.py
2 tests collected
[2024-05-02 07:17:12] #1: test_including_excluding.py:test_component_one
[2024-05-02 07:17:12] #2: test_including_excluding.py:test_component_two
======================== Session ended. 2 successful, 0 skipped, 0 failed, 0 erroneous. Total duration: 00:00:00 ========================
```

5. Overriding Configuration:

- The **-o flag** in Slash Framework allows you to override specific configuration settings by providing new values directly via the command line. This feature is useful when you want to customize certain configuration parameters without modifying the configuration files directly.

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

6. Running Interactively:

- **Running interactively** with Slash Framework allows developers to experiment with the testing framework and explore its features in real-time. It provides an interactive IPython shell initialized with your project's environment, allowing for interactive testing and exploration.
  - Use **the -i flag** with slash run to invoke an interactive IPython shell.

slash run -i

- **IPython's interactive features:**
  - **Tab Completion:** IPython provides tab completion for object attributes, module contents, and filenames.
  - **Object Introspection:** You can use the **? symbol** to access documentation and source code of objects.

slash.config?

  - **Executing Python Code Interactively:** You can execute Python code interactively and see the results immediately.
  - **Exit Interactive Mode:** use the exit command or press Ctrl + D.

7. Resuming Previous Sessions:

- The slash resume command in Slash Framework allows you to quickly retry a previously failed session, skipping tests that have already passed. It reruns all unsuccessful tests from the previous session in a new session, providing an efficient way to retry failed tests without rerunning the entire test suite.

slash resume -vv <session id>

- Get session id:

from slash import session

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

```
D:\project\Slash>slash resume -vv 7cab1801-085a-11ef-b8d0-dc41a9d9f679_0
The current session id is 8b34bfd0-085a-11ef-ade7-dc41a9d9f679_0
3 tests collected
[2024-05-02 08:04:06] #1: test_session.py:test_failing_addition
F: <Error: AssertionError: assert (2 + 3) == 10>, Test: <test_session.py:test_failing_addition>
[2024-05-02 08:04:06] #2: test_session.py:test_failing_operation
F: <Error: AssertionError: assert (2 * 3) == 5>, Test: <test_session.py:test_failing_operation>
[2024-05-02 08:04:06] #3: test_session.py:test_failing_subtraction
F: <Error: AssertionError: assert (2 - 3) == 5>, Test: <test_session.py:test_failing_subtraction>
============================================================ Session Summary ============================================================
== #1: test_session.py:test_failing_addition ==========================================================================================
 - 1/1 F (2024-05-02 15:04:06 +07:00):   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 D:\project\Slash\test_session.py:26
>    assert 2 + 3 == 10  # Intentional failure
F    AssertionError: assert (2 + 3) == 10
== #2: test_session.py:test_failing_operation =========================================================================================
 - 1/1 F (2024-05-02 15:04:06 +07:00):   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 D:\project\Slash\test_session.py:23
>    assert 2 * 3 == 5  # Intentional failure
F    AssertionError: assert (2 * 3) == 5
== #3: test_session.py:test_failing_subtraction =======================================================================================
 - 1/1 F (2024-05-02 15:04:06 +07:00):   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 D:\project\Slash\test_session.py:30
>    assert 2 - 3 == 5  # Intentional failure
F    AssertionError: assert (2 - 3) == 5
====================== Session ended. 0 successful, 0 skipped, 3 failed, 0 erroneous. Total duration: 00:00:00 =======================
```

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>

II. Test Parametrization:

1. Using slash.parametrize:

- **Basic usage:**
  - 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
```

- **Parametrizing Before, Test, and After methods:**
  - Slash also supports parametrizing the before and after methods of test classes, thus multiplying each case by several possible setups:

```
class SomeTest(slash.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):
        # ...
```

  - In this case, you're parametrizing not only the test method but also the before and after methods. This results in 27 distinct tests (3 x 3 x 3), representing all possible combinations of x, y, and z
- **Inheritance:**
  - This example demonstrates that parametrization can be used across inheritance. The DerivedTest class inherits from BaseTest and adds its

own parameterization. When you run a test from DerivedTest, it will run 9 tests (3 from BaseTest x 3 from DerivedTest). The call to super().before() ensures that the before method from the base class is also executed with its parameter values.

```
class BaseTest(slash.Test):

@slash.parametrize('base_parameter', [1, 2, 3])

def before(self, base_parameter):

# ....


class DerivedTest(BaseTest):

@slash.parametrize('derived_parameter', [4, 5, 6])

def before(self, derived_parameter):

super(DerivedTest, self).before()  # Automatically handles base parameters

# .....
```

2. More Parametrization Shortcuts:

- **slash.parameters.toggle:**
  - This shortcut is useful when you want to test a function or method with a boolean flag that can be toggled between two states.

```
@slash.parameters.toggle('with_safety_switch')

def test_operation(with_safety_switch):

if with_safety_switch:

# Execute operation with safety switch

pass

else:

# Execute operation without safety switch

pass
```

- **slash.parameters.iterate**:
  - This shortcut allows you to specify multiple parameter values in a more compact and readable format.

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

def test_something(x, y):

# Test logic using x and y

pass
```

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
```

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):

...


@slash.parametrize('param', [

Object1() // slash.param('first'),

Object2() // slash.param('second'),

])

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.

- **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.

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:

```
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

...
```

- Exclusions also work on sets of parameters:

```
import slash

SUPPORTED_SIZES = [10, 15, 20, 25]

@slash.exclude(('car.size', 'car.color'), [(10, 'red'), (20, 'blue')])

def test_car(car):

...

@slash.parametrize('size', SUPPORTED_SIZES)

@slash.parametrize('color', ['red', 'green', 'blue'])

@slash.fixture

def car(size, color): # <-- red cars of size 10 and blue cars of size 20 will be skipped
```

...

III. Test Tags:

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():

...

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 powrful 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

slash run test -k tag:covers=requirement_1294

IV. Test Fixtures:

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
```

- 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.**

2. The Fixture Definition:

- In software testing, **fixtures** are **pieces of setup code that prepare the environment for running tests.**
- They often include tasks like **creating necessary objects, initializing connections, or setting up configurations.**

```
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. 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.

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)

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
```

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 multiples in a cartesian manner.

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():
...
```

8. Fixture Scopes:

- By default, a fixture **"lives" through only a single test at a time**. This means that:

    - The fixture function **will be called again for each new test needing the fixture**
    - 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.**
- **Session fixtures** live from the moment of their activation until the end of the test session

@slash.fixture(scope='session')

def some_session_fixture(this):

@this.add_cleanup

def cleanup():

print('Hurray! the session has ended')

- **Module fixtures** live until the last test of the module that needed them finished execution.

@slash.fixture(scope='module')

def some_module_fixture(this):

@this.add_cleanup

def cleanup():

print('Hurray! We are finished with this module')

9. 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

10. 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)
```

11. The use_fixtures Decorator:

- In some cases, you **may want to use a certain fixture but don't need its return value.** In such cases, rather than using the fixture as an unused argument to your test function you can use the **use_fixtures decorator**.
- This decorator **receives a list of fixture names and indicates that the decorated test needs them to run:**

```
@slash.fixture()
def used_fixture1():
"""do something"""
pass
@slash.fixture()
def used_fixture2():
"""do another thing"""
pass
@slash.use_fixtures(["used_fixture1", "used_fixture2"])
def test_something():
```

pass

12. Aliasing Fixtures:

- 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()

13. 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

14. 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

V. Assertions, Exceptions and Errors:

1. Assertions:

- 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:

...

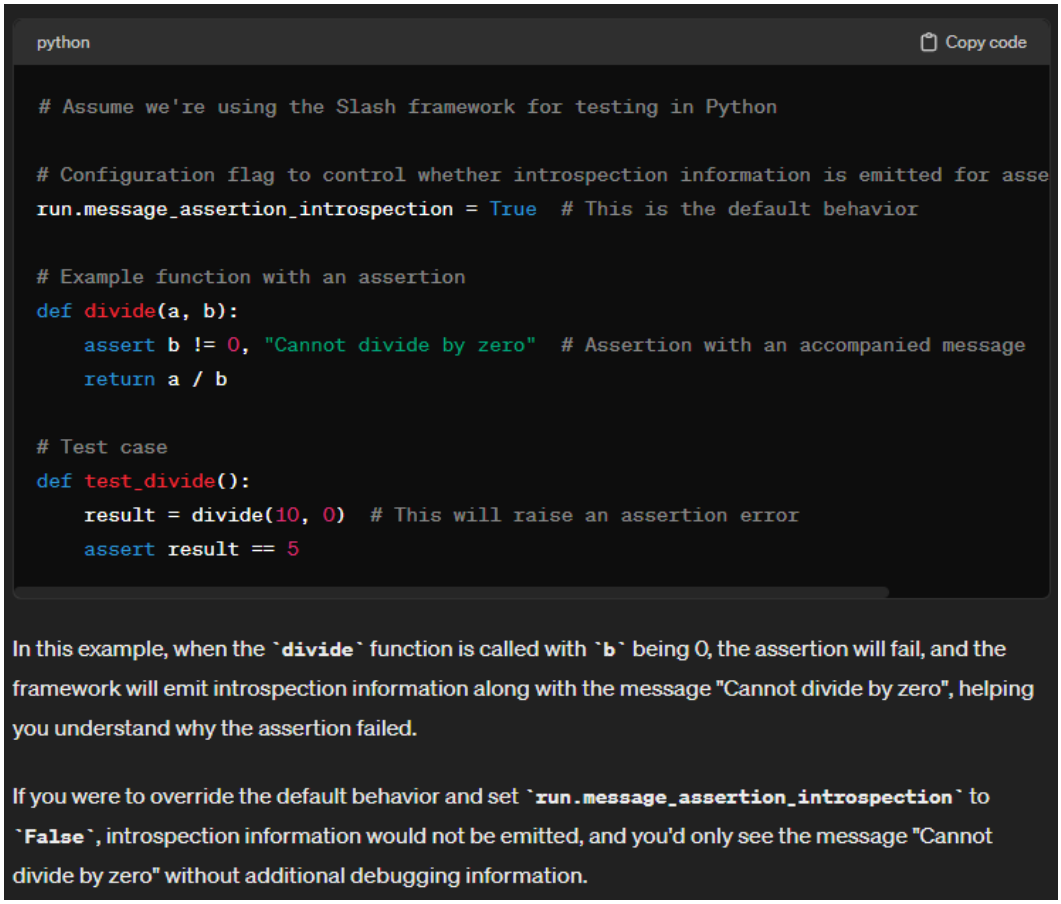
assert f(g(x)) == g(f(x + 1))

...

- 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:** By default, even asserts with accompanied messages will emit introspection information. This can be overriden through the run.message_assertion_introspection configuration flag.



2. 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: ...

- **slash.allowing_exceptions():** This utility allows you to **specify exceptions that are allowed to be raised within a block of code without causing the test to fail**. It's handy when you want to account for certain exceptions but don't want them to fail the test.

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

... pass

- **slash.assert_almost_equal():** This utility is used to **test for near equality between two values within a specified tolerance (max_delta).** It's often used when comparing floating-point numbers where exact equality might not be feasible due to precision issues.

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. 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 nontheless.
- **Any exceptions thrown from a test will be added to the test result as an error, thus marking the test as 'error'.**

4. 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**

5. 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.**

```python
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*) [source]

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*) [source]

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

6. 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 pudb 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 callbacks and decide which of these two hooks should call them, since a debugger might stall for a long time until a user notices it.

a) 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 oppurtunity to handle an exception before it propagates.**

def test_function():

func1()

def func1():

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 provices 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

1. **Exception Handling**: The primary purpose of `handling_exceptions` is to manage exceptions that occur within its context. When you use it as a context manager with a `with` statement, any exceptions that occur within that block of code will be caught and handled according to the parameters you provide.
2. **Parameters**:
   - `passthrough_types`: This parameter allows you to specify certain exception types that should not be handled by the context manager. Instead, they will be immediately raised onward.
   - `swallow`: If set to `True`, this parameter causes the context manager to swallow (suppress) exceptions that occur within its context, preventing them from propagating further.
   - `swallow_types`: Similar to `passthrough_types`, this parameter allows you to specify exception types that should be swallowed by the context manager.
   - `context`: An optional string parameter that describes the operation being wrapped by the context manager. This is useful for logging purposes to improve readability and understanding of the context in which exceptions occur.
3. **Note on Exceptions**: The documentation notes that certain exceptions, such as `KeyboardInterrupt`, `SystemExit`, and `SkipTest`, are never swallowed by the context manager. This means that these exceptions will still propagate outward and won't be caught or handled by `handling_exceptions`.

  - **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

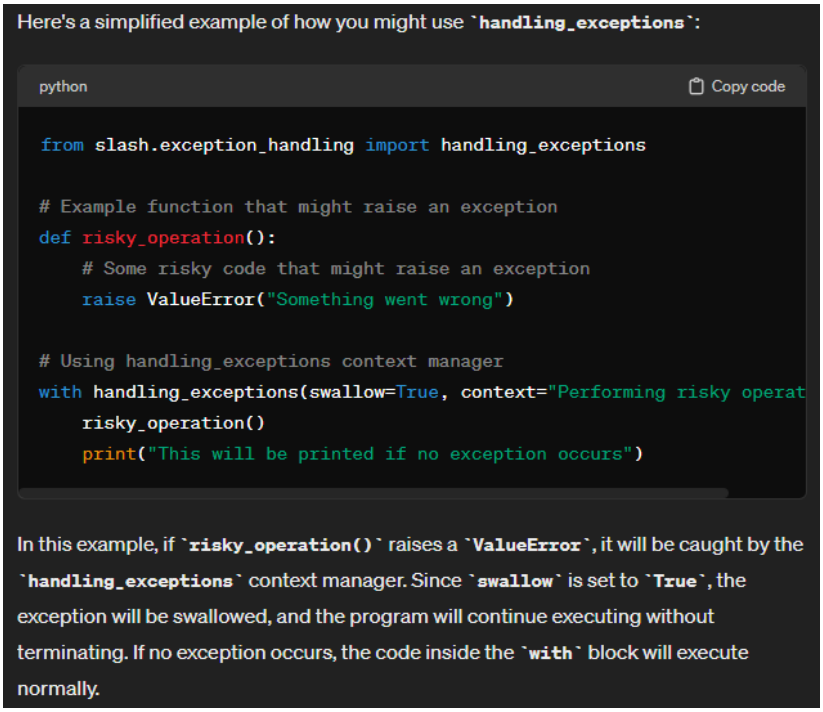
def some_function():

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.

b) Exception Marks:

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

i. 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:

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

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 reponsibility to avoid any further actions which might tamper with your setup or your session state.**

ii. 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:
  - **Reporting results to external services, which might be unavailable at times**
  - **Automatic issue reporting to bug trackers**
  - **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!")
```

iii. 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'))
```

VI. 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.**

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.**
  - **Cause the warning_added hook to be fired.**

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.**

3. Example:

- Slash.ignore_warnings():

```
@slash.hooks.configure.register

def configure_warnings():

    slash.ignore_warnings(category=DeprecationWarning, filename='/some/bad/file.py')
```

- For ignoring warnings in specific code-block:
- slash.ignored_warnings context: .. code-block:: python
- with slash.ignore_warnings(category=DeprecationWarning, filename='/some/bad/file.py'):
- ...

VII. Customizing and Extending Slash:

- This section describes **how to tailor Slash to your needs.**

1. Customization Basics:

a) .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.**

b) 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 activated upon request.**
    - **Once activated, methods defined on the plugin** which **correspond to names of known hooks get registered on those hooks automatically.**

2. Customizing Using Plain Hooks.

- Our first step is **customizing the logging facility to our needs.**
- We are going to implement **two requirements**:
    - Have **logging always turned on in a fixed location (Say ~/slash_logs)**
    - **Collect execution logs at the end of each session, and copy them to a central location (Say /remote/path).**
- **Steps:**
    - 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')
```

- 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))
```

  -

3. 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.**
- **Example:**
    - **Installation:**

```
...
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.plugin_manager.PluginManager.install()** to install our plugin.
        - Note that **at this point the plugin is not activated.**
    - **Run:**
        - Once the plugin is installed, you can **pass --with-logcollector to actually activate the plugin.**
        - **Active by default:**
            - In some cases you want to **activate the plugin by default,** which is easily done with the **slash.plugins.plugin_manager.PluginManager.activate():**

```
...
```

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

- You can also just **pass activate=True in the call to install**

```
slash.plugins.manager.install(collector_plugin, activate=True)
```

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

- **Specification:**
  - 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..**

4. 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.**
- Second, we use **two new plugin methods**
  - **configure_argument_parser**
  - **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.**

5. 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.active
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 ./
```

6. Complete Example:

```
import os
import shutil

 import slash

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


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

     def get_name(self):
```

```
19.         return 'logcollector'
20.
21.     def session_end(self):
22.         shutil.copytree(
23.             slash.session.logging.session_log_path,
24.             os.path.join('/remote/path', slash.session.id))
25.
26.
27. @slash.plugins.active
28. class ProductTestingPlugin(slash.plugins.PluginInterface):
29.
30.     def get_name(self):
31.         return 'your product'
32.
33.     def get_default_config(self):
34.         return {'api_timeout_seconds': 50}
35.
36.     def configure_argument_parser(self, parser):
37.         parser.add_argument('-t', '--target',
38.                             help='ip address of the target to
   test')
39.
40.     def configure_from_parsed_args(self, args):
41.         self.target_address = args.target
42.
43.     def session_start(self):
44.         slash.g.target = Target(
```

```
45.             self.target_address,
    timeout=slash.config.root.plugin_config.your_product.api_timeout_sec
    onds)
```

VIII. 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

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.**

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
```

3. List of Available Configuration Values:

- https://slash.readthedocs.io/en/master/configuration.html#list-of-available-configuration-values

IX. 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.
  - **Console verbosity** is controlled with **-v/-q.**

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**

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).

a) 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
```

b) 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
```

c) 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.**

d) 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. 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.

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']"
```

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}" ...
```

X. Saving Test Details:

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

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)
```

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**

XI. 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).

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)
```

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 encounteres the exception.** This is done via gossip's error handling mechanism.

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.**

4. Advanced Usage:

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

5. Available Hooks:

- https://slash.readthedocs.io/en/master/hooks.html#available-hooks

XII. 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.**

1. The Plugin Interface:

- Plugins have **several special methods that can be overriden**, 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.**

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. 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**.
- **Only plugins that are PluginInterface derivative instances are accepted.**
- To **uninstall plugins,** you can use the **slash.plugins.manager.uninstall (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.**

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.**

- **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):
        ...
```

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
```

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.

7. 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.**

8. 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.

a) 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.

b) 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.

c) 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).
- Plugin-level provides and needs also get **transferred upon inheritence, automatically adding the dependency configuration to derived classes.**

9. Plugin Manager:

- As mentioned above, the Plugin Manager provides API to activate (or deacativate) 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.

10. 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 r**equires 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.

XIII. Built-in Plugins:

1. Coverage:

- This plugins **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
```

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'
```

- **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. 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
```

4. Signal handling:

- The signal handling plugin **allows users to register handlers for OS signals**.
- By default, the plugin **registers the following handlers** (if supported by the OS):
    - 1. SIGUSR1 drops into debugger
    - 2. SIGUSR2 skips current test
- The plugin also s**upports registering additional signal handlers, as well as overriding the default ones,** by using the **register_handler function**

5. Linking to logs archived by a CI system:

- The CI links plugin **adds a web link to the test log file artifact archived by a CI system to the test's additional details store.**
- This creates a link to the test log file for each failing test in the test summary view, and also adds the link to each test's additional details table in the Backslash web interface.
- Use it by running with **--with-ci-links** and ensuring that the URL for the build is defined.
- **Note:** The plugin will not add any links to the additional details store if the build URL is not defined.
- By default the plugin retrieves the build URL from the BUILD_URL environment variable populated by the Jenkins CI system, but this can be changed by specifying a different environment variable in the **slash.config.root.plugin_config.ci_links.build_url_environment_variable configuration item.**
- The default template used to generate the link is **%(build_url)s/artifact/%(log_path)s, (where log_path is the log path generated by Slash for the test case),** but this can be changed by specifying a different template in the **slash.config.root.plugin_config.ci_links.link_template configuration item.**

XIV. Slash Internals:

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())
```

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. 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`.

XV. Misc. Features:

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_
```

XUnit Export

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

XVI. Advanced Use Cases:

1. Customizing via Setuptools Entry Points:

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.

2. Loading and Running Tests in Code:

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. 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"]
```

XVII. Cookbook:

https://slash.readthedocs.io/en/master/cookbook.html#

XVIII. Unit Testing Slash:

https://slash.readthedocs.io/en/master/unit_testing.html

XIX. Parallel Test Execution:

https://slash.readthedocs.io/en/master/parallel.html