Contexts Documentation

Release 0.11.2

Benjamin Hodgson

September 15, 2015

Contents

Dead simple descriptive testing for Python. No custom decorators, no context managers, no .feature files, no fuss.

CHAPTER 1

About

Contexts is a 'Context-Specification'-style test framework for Python 3.3 and above, inspired by C#'s Machine.Specifications. It aims to be flexible and extensible, and is appropriate for unit, integration and acceptance testing. Read more at the Huddle Dev Blog.

Test written with Contexts resemble the grammar of Given, When, Then-style specifications. Writing tests which read like user-centric sentences can encourage you to focus on the behaviour, not the implementation, of your code. Contexts takes cues from Behaviour Driven Development, but it aims to be useful for more than just acceptance testing.

CHAPTER 2

Quick start

Install Contexts and optionally colorama.

```
$ pip install contexts colorama
```

Create a file in the current directory called `test.py`. Put a class in that file; this will be your test. The class's name must include the word `When`.

Spread the *arrange, act, assert* steps of your test across three separate methods in the class; their names must match the *supported keywords*.

```
class WhenAddingTwoNumbers:
    def given_the_two_numbers(self):
        self.x = 4
        self.y = 2
    def when_i_add_them(self):
        self.result = self.x + self.y
    def it_should_produce_the_correct_sum(self):
        assert self.result == 6
```

Run the test!

```
$ run-contexts
.
----------------------------------------------------------------------
PASSED!
1 context, 1 assertion
(0.1 seconds)
```

CHAPTER 3

Table of contents

3.1 Guide

Contents

3.1.1 Command line usage

`run-contexts` will run all test files and folders in the current directory.

`run-contexts {filename}` will run the tests in the specified file.

`run-contexts {directory}` will run the tests in the specified directory (or package) and any subdirectories (or packages).

`run-contexts {filename}:{classname}` will run a single test. Example: `run-contexts test/my_test.py:MyTestClass`.

Note: Running a single class is currently not compatible with assertion rewriting.

`run-contexts` accepts the following command-line flags:

- `-h` or `--help`: Print a help message and exit.

- `--version`: Print information about which version of Contexts is installed
- `-s` or `--no-capture`: Don't capture stdout during tests. By default, Contexts will prevent stdout from being printed to the console unless a test fails. Use this option to disable this.
- `--teamcity`: Use when the tests are being run in TeamCity. Contexts tries to detect this automatically, but the flag is provided in case you have trouble.
- `-v` or `--verbose`: Report tests that pass as well as those that fail.
- `--no-colour`: Disable output colouring.
- `--no-random`: Disable test order randomisation. Note that, even with randomisation disabled, Contexts makes no promises about the order in which tests will be run.
- `--no-assert`: Disable assertion rewriting - don't try to add helpful messages to assertions made with the *assert* statement.
- `--xml`: Specify output file for a Jenkins-compatible XML test report
- `--filespec=<FILE>`: Path to a file which defines tests to run.

3.1.2 Test discovery

If a *module* contains the words **test** or **spec** in its name, Contexts will import it and run any tests therein. If a *folder* has the words 'test' or 'spec' in its name, Contexts will scan its contents for modules and subfolders matching this pattern. If a *package* has the words 'test' or 'spec' in its name, Contexts will import it, and scan the package's contents for modules and subfolders matching this pattern.

If a class has **spec** or **when** in the name, Contexts will treat it as a test case. Test classes can inherit from `object` - there's no need to subclass `TestCase` for Contexts to pick up your tests.

3.1.3 Defining tests

Contexts will instantiate and run each test class once. The methods on the class will be run in a certain order, depending on their names.

If a method in a test class has an ambiguous name (its name would place it in more than one of the categories below), Contexts will raise an exception.

By default, Contexts randomises the order in which test classes, and assertions within each class, will be run. It's therefore important to ensure that all your test cases are independent. Randomisation can be disabled by supplying the `--no-random` flag at the command line. This is not recommended (since you may inadvertently introduce coupling between your tests), and the order will still be arbitrary and liable to change between runs.

`given` - setting up

If the words **establish**, **context**, or **given** appear in a method, it will be run before the other test methods in the class. 'Establish' methods are typically used to build test data, instantiate the object under test, set up mock objects, write test files to the file system, and so on. The purpose of this is to put the system in a known state before the test begins.

The setup method is run *once for each test class*, to encourage you to ensure your assertions don't modify any state. Compare this with the xUnit style, wherein the setup is run before every test method.

There should be *one setup method per class* - Contexts will throw an exception if it finds more than one method that appears to be a setup method.

Contexts supports inheritance of setup methods. If a test class has a superclass, the parent's 'establish' method will be run before the child's. This allows you to share setup code between test classes. The superclass's setup will be run *even if it has the same name* as the subclass's setup method.

`when` - acting on the SUT

If the words 'because', 'when', 'since' or 'after' appear in the name of a method, it is treated as an 'action' method. Typically, this method will do nothing except call the function under test - after all, you've set up all your test data in the 'context' method.

The action method is run *once for each test class*, to encourage you to ensure your assertions don't modify any state.

There should be *one action method per class* - Contexts will throw an exception if it finds more than one method that appears to be a action method.

The 'because' method will be run immediately after the 'establish' method.

Inheritance of action methods is not supported. The 'because' method will only be run on the concrete class of the test object.

`should` - assertions

If a method contains the words **it**, **should**, **then**, **must** or **will** in its name, it is treated as an assertion. Assertion methods should be very granular - one assertion per method, if possible - and named to describe the *behaviour* you're trying to test (rather than details such as function names).

Assertions may be made using the `assert` statement, or any assertion library which raises *AssertionError* upon failure.

Each assertion method will be run once, after the 'because' method and before the 'cleanup' method (see below). Contexts makes no promises about the order in which assertions will be made, and the order may change between runs, so it's important to ensure that all the assertions on a given class are independent of one another.

If an assertion fails, all the remaining assertions will still be run, and Contexts will report precisely which ones failed. Contrast this with the xUnit testing style, wherein a failing assertion ends the test and any subsequent assertions will not be run.

Contexts supports testing with the `assert` statement. No one likes writing their own assertion messages (especially when you've just labelled the method name descriptively!), so Contexts tries to supply a useful message if you didn't add one yourself. This is achieved by metaprogramming - Contexts introspects the source code of your module while it's being imported, and modifies it to add assertion messages. If this behaviour freaks you out, you can disable it by supplying a `--no-assert` flag at the command line.

You can have as many assertion methods as you like on a single class.

`cleanup`

If the word **cleanup** appears in a method's name, it is treated as a tearing-down method, and run after all the assertions are finished. The cleanup method is guaranteed to be run, even if exceptions get raised in the setup, action or assertion methods.

Good tests should leave the world in the state in which they found it. Cleanup methods are therefore most commonly found in integration tests which modify the filesystem or database, or otherwise do IO in order to set up the test.

The cleanup method is run *once for each test class*, to encourage you to ensure your assertions don't modify any state. Compare this with the xUnit style, wherein the teardown is run after every test method.

There should be *one cleanup method per class* - Contexts will throw an exception if it finds more than one method that appears to be a cleanup method.

Contexts supports inheritance of cleanup methods. If a test class has a superclass, the parent's 'cleanup' method will be run after the child's. This allows you to share cleanup code between test classes. The superclass's cleanup will be run *even if it has the same name* as the subclass's setup method.

`examples` - triangulating

Contexts has support for 'examples' - sets of test data for which the whole test is expected to pass. Examples allow you to triangulate your tests very easily - if you need more test data, simply add a line to the 'examples' method.

If you define a *classmethod* with the words **examples** or **data** in its name, it is treated as a test-data-generating method. This method must return an iterable (you can use `yield`), and it will be called before testing begins.

For each example returned by the 'examples' method, the test class will be instantiated and run once. Test methods which accept one argument will have the current example passed into them. A method which accepts no arguments will be run normally. This allows you to take one of two approaches to testing using examples. You can accept the example once in the setup and set it as an attribute on *self*, or you can accept it into every test method.

Other methods

Other methods, which do not contain any of the keywords detailed above, are treated as normal instance methods. They can be called as usual by the other methods of the class.

Catching exceptions

Sometimes you need to assert that a given function call will raise a certain type of exception. You can catch and store an exception - to make assertions about it later - using Contexts's *catch* function.

`contexts.catch()` accepts a function, and runs it inside a `try` block. If an exception gets raised by the function, *catch* returns the exception. If no exception was raised, it returns `None`. Any additional arguments or keyword arguments to `catch` are forwarded to the function under test.

You'll typically see `catch` in a 'because' method. The caught exception generally gets saved as an instance attribute, and assertions are made about (for example) its type in assertion methods.

Debugging

It's often useful to be able to drop into a debugger at a set point in your test run. However, Contexts's default stdout-capturing behaviour can interfere with this. This can be disabled using `-s`/`--no-capture` at the command line. Also provided is a `set_trace()` convenience function - add the line `contexts.set_trace()` to your code to launch a debugger from that line connected to the *real* stdout.

Timing things

Sometimes you need to assert that an action is performant. Contexts provides a `time()` convenience function for this purpose.

`contexts.time()` measures the execution time of a function and returns the execution time as a float in seconds, by calling `time.time()` before and after running the function. The precision of `contexts.time()` on your platform therefore depends on the precision of `time.time()` on your platform.

Overriding name-based usage

Sometimes you need to name a test object in such a way that upsets the test runner. Such an example would be a setup method with the word 'it' in the name.

Contexts provides a built-in plugin which defines a set of decorators for overriding the way an object is named:

- `@setup` to mark setup methods
- `@action` to mark action methods
- `@assertion` to mark assertion methods
- `@teardown` to mark cleanup methods
- `@spec` or its alias `@context` to mark classes as tests

A brief example:

```
from contexts import setup
class WhenINameMethodsAmbiguously:
    @setup
    def establish_that_it_has_an_ambiguous_name(self):
        # this method has both 'establish' and 'it' in the name.
        # Contexts will have a hard time discerning its purpose
        # unless we mark it explicitly.
```

3.1.4 TeamCity

Contexts has support for running tests in TeamCity. `run-contexts` should automatically recognise when a build is being run by TeamCity. If you have problems, try invoking the test runner with a `--teamcity` flag.

Each assertion will be reported to TeamCity as a separate test, and each test file that gets run will be reported as a separate suite. Contexts reports failures to TeamCity along with any stack traces, and also captures and reports any activity on stdout and stderr.

3.2 List of keywords

Meaning	Keywords
Test folder	`test`, `spec`
Test file	`test`, `spec`
Test class	`test`, `spec`
Examples	`example`, `data`
Setup	`establish`, `context`, `given`
Action	`because`, `since`, `after`, `when`
Assertion	`it`, `should`, `must`, `will`, `then`
Cleanup	`cleanup`

3.3 Plugins

Contexts features an experimental 'plugin' interface for user-customisable behaviour.

Contents

3.3.1 The plugin interface

For documentation purposes, the full plugin interface is defined in *PluginInterface*. Currently, plugins are **required** to implement the *initialise()* method (to determine whether the plugin is active in the current test run). The rest of the plugin interface is optional.

Contexts's plugin support is implemented as an ordered list of plugin classes. Each time a plugin hook is called, each plugin is called in turn. Plugins which do not implement the hook are skipped. The *first* return value is used as the return value of the aggregated plugin calls - when a plugin returns a value from a hook, all the remaining plugins in the list are skipped. This means that a given plugin is able to override the behaviour of plugins which follow it in the list.

3.3.2 The plugin lifecycle

Because plugins can override one another, the ordering of the list matters. The interface defines a `classmethod` entitled *locate()*, which you can implement to insert your plugin before or after another plugin.

Plugins are given control over whether or not they appear in the list. All plugins **must** define an *initialise(args, environ)* method, and return either `True` or `False` to signal whether they want to appear in the list. You may also define a *setup_parser(parser)* method to modify the `argparse.ArgumentParser` instance that is used to parse command-line arguments.

Very occasionally, it is necessary for plugins to modify the behaviour of other plugin objects (see `FailuresOnly` for an example) - you can define a *request_plugins()* generator method to request the current instances of some other plugin classes from the test runner.

3.3.3 Progress notifications

There exist a number of plugin hooks which are called when progress through the test run reaches certain points. These methods include *test_run_started()*, *context_ended()*, *assertion_failed()*, and so on.

These hooks are typically used to report progress to the user. It's not recommended to return a value from these methods, unless you want to prevent other plugins from being told about the progress.

3.3.4 Identifying test objects

When the test runner sees a file, folder, class, or method, it queries the list of plugins to find out whether it should run it, using the *identify_folder()*, *identify_file()*, *identify_class()*, and *identify_method()* hooks. The expected return values from these methods are defined as constants in the *contexts.plugin_interface* module: *TEST_FILE*, *CONTEXT*, *ASSERTION* and so on.

After all the modules have been imported, the *process_module_list()* hook is called, which plugins can use to inject their own test modules, or remove modules that should not be run. There are also *process_class_list()* and *process_assertion_list()* hooks.

3.3.5 Other hooks

There are a few extra plugin hooks to override the way modules are imported (see `AssertionRewritingImporter` for an example) and to set the exit code for the process.

3.3.6 Registering a plugin

Once you've written your plugin, you can register it with Contexts using the `contexts.plugins` Setuptools entry point:

```
from setuptools import setup

setup(
    # ...
    entry_points = {
        'contexts.plugins': ['MyPluginClass = my_package.my_module:MyPluginClass']
    }
    # ...
)
```

3.3.7 The plugin API

class `contexts.plugin_interface.`**`PluginInterface`**

Defines the interface for plugins.

You do not need to inherit from this class in your own plugins. Just create a new class and implement *initialise* (and one or more other hooks).

classmethod `locate`()

Called before the plugin is instantiated, to determine where it should appear in the list of plugins. The ordering of this list matters. If a plugin returns a (non-`None`) value from a given method, plugins later in the list will not get called.

Plugins may return a 2-tuple of `(left, right)`. Here, `left` is a plugin class which this plugin wishes to *follow*, and `right` is a class the plugin wishes to *precede*. Either or both of the values may be *None*, to indicate that the plugin does not mind what it comes before or after, respectively. Returning `None` from this method is equivalent to returning `(None, None)`.

`setup_parser`(*parser*)

Called before command-line arguments are parsed.

Parameters **`parser`** – An instance of `argparse.ArgumentParser`. Plugins may mutate `parser` in order to set it up to expect the options the plugin needs to configure itself. See the standard library documentation for `argparse` for more information.

`initialise`(*args*, *environ*)

Called after command-line arguments are parsed.

Parameters

- **`args`** – The result of `ArgumentParser.parse_args`. (see the standard library documentation for `argparse` for more information).

- **environ** – The value of `os.environ`.

Returns A boolean. Returning `True` will cause the plugin to be added to the list of plugins for this test run. Returning `False` will prevent this.

request_plugins()
Called after all plugins have been initialised.

Plugins which need to modify the behaviour of other plugins may request instances of those plugins from the framework.

This must be a generator method. Yield an iterable of other plugin classes, and you will be sent a dictionary mapping those classes to the active instances of those plugins. Requested plugins that do not have an active instance will not be present in the dict.

test_run_started()
Called at the beginning of a test run.

test_run_ended()
Called at the end of a test run.

suite_started(*module*)
Called at the start of a test module.

Parameters **module** – The Python module (an instance of `ModuleType`) that is about to be run.

suite_ended(*module*)
Called at the end of a test module.

Parameters **module** – The Python module (an instance of `ModuleType`) that was just run.

test_class_started(*cls*)
Called when a test class begins its run.

A test class may contain one or more test contexts. (Test classes with examples will generally contain more than one.)

Parameters **cls** – The class object that is being run.

test_class_ended(*cls*)
Called when a test class ends its run.

Parameters **cls** – The class object that is being run.

test_class_errored(*cls*, *exception*)
Called when a test class unexpectedly errors.

Parameters

- **cls** – The class object that is being run.
- **exception** – The exception that got caused the error.

context_started(*cls*, *example*)
Called when a test context begins its run.

Parameters

- **cls** – The class object of the test being run.
- **example** – The current example, which may be *NO_EXAMPLE* if it is not a parametrised test.

context_ended(*cls*, *example*)
Called when a test context completes its run.

Parameters

- **cls** – The class object of the test being run.
- **example** – The current example, which may be *NO_EXAMPLE* if it is not a parametrised test.

context_errored(*cls*, *example*, *exception*)
Called when a test context (not an assertion) throws an exception.

Parameters

- **cls** – The class object of the test being run.
- **example** – The current example, which may be *NO_EXAMPLE* if it is not a parametrised test.
- **exception** – The exception that caused the error.

assertion_started(*func*)
Called when an assertion begins.

Parameters **func** – The assertion method being run.

assertion_passed(*func*)
Called when an assertion passes.

Parameters **func** – The assertion method being run.

assertion_errored(*func*, *exception*)
Called when an assertion throws an exception.

Parameters

- **func** – The assertion method being run.
- **exception** – The exception that caused the error.

assertion_failed(*func*, *exception*)
Called when an assertion throws an AssertionError.

Parameters

- **func** – The assertion method being run.
- **exception** – The exception that caused the failure.

unexpected_error(*exception*)
Called when an error occurs outside of a Context or Assertion.

Parameters **exception** – The exception that caused the failure.

get_object_to_run()
Called before the start of the test run, when the test runner wants to know what it should run.

This method should return one of:

- a class - the test runner will run the identified methods in this class.
- a file path as a string - the test runner will run the identified classes in this file.
- a folder path as a string - the test runner will run the identified files and subfolders in this folder.
- `None` - the plugin doesn't want to choose what to run.

identify_folder(*folder*)
Called when the test runner encounters a folder and wants to know if it should run the tests in that folder.

Parameters **str** (*folder*) – The full path of the folder which the test runner wants to be identified

This method should return one of:

- *TEST_FOLDER* - plugin wishes the folder to be treated as a test folder
- None - plugin does not wish to identify the folder (though other plugins may still cause it to be run)

identify_file (*file*)
Called when the test runner encounters a file and wants to know if it should run the tests in that file.

Parameters **str** (*file*) – The full path of the file which the test runner wants to be identified.

This method should return one of:

- *TEST_FILE* - plugin wishes the file to be imported and run as a test file
- None - plugin does not wish to identify the file (though other plugins may still cause it to be run)

identify_class (*cls*)
Called when the test runner encounters a class and wants to know if it should treat it as a test class.

Parameters **cls** – The class object which the test runner wants to be identified.

This method should return one of:

- *CONTEXT* - plugin wishes the class to be treated as a test class
- None - plugin does not wish to identify the class (though other plugins may still cause it to be run)

identify_method (*func*)
Called when the test runner encounters a method on a test class and wants to know if it should run the method.

When a test class has a superclass, all the superclass's methods will be passed in first.

Parameters **func** – The unbound method (or bound classmethod) which the test runner wants to be identified

This method should return one of:

- *EXAMPLES* - plugin wishes the method to be treated as an 'examples' method
- *SETUP* - plugin wishes the method to be treated as an 'establish' method
- *ACTION* - plugin wishes the method to be treated as a 'because'
- *ASSERTION* - plugin wishes the method to be treated as an assertion method
- *TEARDOWN* - plugin wishes the method to be treated as a teardown method
- None - plugin does not wish to identify the method (though other plugins may still cause it to be run)

process_module_list (*modules*)
A hook to change (or examine) the list of modules which will be run with the full list of found modules. Plugins may modify the list in-place by adding or removing modules.

Parameters **modules** – A list of types.ModuleType.

process_class_list(*module*, *classes*)
A hook to change (or examine) the list of classes found in a module. Plugins may modify the list in-place by adding or removing classes.

Parameters

- **module** – The Python module in which the classes were found (an instance of types.ModuleType).
- **classes** – A list of classes found in that module.

process_assertion_list(*cls*, *functions*)
A hook to change (or examine) the list of (unbound) assertion methods found in a class. Plugins may modify the list in-place by adding or removing functions.

Parameters

- **cls** – The test class in which the methods were found
- **functions** – A list of unbound assertion methods found in that class

import_module(*location*, *name*)
Called when the test runner needs to import a module.

Arguments: location: string. Path to the folder containing the module or package. name: string. Full name of the module, including dot-separated package names.

This method should return one of:

- **an imported module (an instance of types.ModuleType).** This may be a reference to an existing module, or a "fake" generated module.
- None, if the plugin is not able to import the module.

get_exit_code()
Called at the end of the test runner to obtain the exit code for the process.

This method should return one of:

- An integer
- None, if you do not want to override the default behaviour.

contexts.plugin_interface.**TEST_FOLDER**
Returned by plugins to indicate that a folder contains tests.

contexts.plugin_interface.**TEST_FILE**
Returned by plugins to indicate that a file contains tests.

contexts.plugin_interface.**CONTEXT**
Returned by plugins to indicate that a class is a test class.

contexts.plugin_interface.**EXAMPLES**
Returned by plugins to indicate that a method is an Examples method.

contexts.plugin_interface.**SETUP**
Returned by plugins to indicate that a method is a setup method.

contexts.plugin_interface.**ACTION**
Returned by plugins to indicate that a method is an action method.

contexts.plugin_interface.**ASSERTION**
Returned by plugins to indicate that a method is an assertion method.

contexts.plugin_interface.**TEARDOWN**
Returned by plugins to indicate that a method is a teardown method.

contexts.plugin_interface.**NO_EXAMPLE**
Passed to plugins when a class is not a parametrised test.

3.4 Code samples

3.4.1 A simple test case

Here's an example of a test case that the authors of Requests might have written, if they were using Contexts. See the *Guide* for details.

```
import requests

class WhenRequestingAResourceThatDoesNotExist:
    def establish_that_we_are_asking_for_a_made_up_resource(self):
        self.uri = "http://www.github.com/itdontexistman"
        self.session = requests.Session()

    def because_we_make_a_request(self):
        self.response = self.session.get(self.uri)

    def the_response_should_have_a_status_code_of_404(self):
        assert self.response.status_code == 404

    def the_response_should_have_an_HTML_content_type(self):
        assert self.response.headers['content-type'] == 'text/html'

    def cleanup_the_session(self):
        self.session.close()

if __name__ == '__main__':
    contexts.main()
```

3.4.2 Triangulation

Here's a brief example of Contexts's triangulation feature. We're asserting that the various different types of numbers in Python can all be multiplied by 0 to produce the expected result.

```
class WhenMultiplyingANumberByZero:
    @classmethod
    def examples_of_numbers(cls):
        yield 0
        yield -6
        yield 3
        yield 1.6
        yield 6 + 2j

    def because_we_multiply_by_0(self, example):
        self.result = example * 0

    def it_should_return_0(self):
        assert self.result == 0
```

If you yield tuples from the *examples* method, and you accept multiple arguments to the test methods, Contexts will unpack the tuple and pass it in as separate arguments.

```
class WhenMultiplyingTwoNumbers:
    @classmethod
    def examples_of_numbers_and_their_products(cls):
        yield 1, 12, 12
        yield -3.2, 2, -6.4
        yield 6 + 2j, 9, 54 + 18j

    def because_we_multiply_the_two(self, x, y, expected):
        self.result = x * y

    def it_should_equal_what_we_expected(self, x, y, expected):
        assert self.result == expected
```

If you accept only one argument to a test method, but you yield tuples, Contexts will not unpack the tuple.

```
class WhenIYieldTuples:
    @classmethod
    def examples(cls):
        yield 'abc', 123
        yield [], {}

    def it_should_give_me_tuples(self, example):
        assert isinstance(example, tuple)
```

- genindex
- modindex
- search

C

A

C

E

G

I

L

N

P

R

S

T

U