contextional Documentation

Release 1.6.3

Chris NeJame

Nov 28, 2017

Contents

A context-based functional testing tool for Python

CHAPTER 1

Overview

1.1 Why Use Contextional?

Short answer

- fast and easy to use
- works with `unittest`, `unittest2`, `nose`, and `pytest`
- deterministic fixture and test order without being tied to a class
- might significantly improve functional test suite run time
- much better test output organization and readability (doesn't even need to be used for functional tests)

Long answer

If you use the standard `unittest`/`unittest2` libraries to write functional tests, you're likely to have a very complicated test suite, as the only thing you should have in each test method should be a single assert, your test classes should each only pertain to one specific scenario, and the setups and teardowns for each test class should be completely isolated from and independant of all the other test classes.

The more complex the thing you're testing is, the more scenarios you'll have to cover, which means more test classes that you'll have to write. A properly made functional test suite for a decently complex product can easily take a very long time to run, as each test class will have to run every bit of setup required for it to run and then completely tear it all down once it's finished running all its tests.

It's not a best practice to have a test class try and rely on any previously run classes when writing tests in the traditional fashion, as the order they run in isn't deterministic.

That's where Contextional comes in.

Contextional uses a combination of `with` statements and decorators to let you quickly and easily define fixtures and tests with easy-to-read descriptions for each context and test. This is done to make sure each fixture and test happens in a logical and deterministic order and that the test output is organized and more idiomatic.

1.2 Installation

You can install Contextional through `pip` with:

```
$ pip install contextional
```

1.3 Quick Example

Code:

```
from contextional import GroupContextManager as GCM


with GCM("Main Group") as MG:

    @GCM.add_setup
    def setUp():
        GCM.value = 1

    @GCM.add_test("value is 1")
    def test(case):
        case.assertEqual(GCM.value, 1)

    with GCM.add_group("Child Group"):

        @GCM.add_setup
        def setUp():
            GCM.value += 1

        @GCM.add_test("value is now 2")
        def test(case):
            case.assertEqual(GCM.value, 2)

MG.create_tests()
```

Test output:

```
Main Group
  value is 1 ... ok
  Child Group
    value is now 2 ... ok
```

CHAPTER 2

Writing Tests

2.1 The Basics

Contextional utilizes contexts (`with` statements) so you can easily control how your tests and fixtures play out.

You should know that any code you've written along side your fixture and test definitions is getting run, so you can use this to your advantage to influence how your fixtures and tests get defined. But none of the tests or fixtures you defined will get used if you don't call `create_tests()` on the `Context` instance that you made (see below for example usage).

This might seem odd, but it's required for two reasons:

1. It allows you to create a `Context`, containing layers of tests and fixtures, that can be included in other `Context` objects, and it will only get run if `create_tests()` is called by those `Context` objects.
2. Contextional needs to modify the global namespace of the module you want the tests to be created in so that it can create the `unittest.TestCase` classes in it (depending on the Python implementation you're using, this may also require you to pass `globals()` to `create_tests()`).

TL;DR The examples below should answer most questions you have.

2.1.1 Order of Fixture Execution

2.1.1.1 Setups

When enterring a layer, all of the setups defined in that layer are executed, in the order that they were defined.

2.1.1.2 Test Setups

Test-level setups are executed before every test that is defined in the same layer as them.

2.1.1.3 Test Teardowns

Test-level teardowns are executed after every test that is defined in the same layer as them.

2.1.1.4 Teardowns

When exiting a layer (i.e. the tests defined in the layer and all of the layer's descendant layers have all been executed), all of the teardowns defined in that layer are executed, in the order that they were defined.

> **Warning:** Fixtures will only be executed for the layer that they are defined in.
>
> Test-level fixtures will only be used for the tests that were declared in the same layer that they were, and none of the tests in any descendant layers will execute them.
>
> Layer-level fixtures only execute once. Even if the layer that they were defined in has more than one child layer, they will still only be executed once.

> **Warning:** If a layer has no tests, nor does any of its descendant layers, then any fixtures declared in that layer will not be executed.

Note: It doesn't matter if you define fixtures before or after fixtures of different types, tests, or child layers. The order in which you define a fixture is only relevant when compared to other fixtures of the same type.

For example, you can define `setup A`, then `test 1`, and then `setup B`, but `setup A` will still be executed before `setup B`, and `setup B` will still be executed before `test 1`.

2.1.1.5 Alternate Explanation

Here's some pseudocode that might help clarify:

```
def run_layer_tests(layer):
    # layer setups
    for setup in layer.setups:
        setup()

    # tests
    for test in layer.tests:
        # test setups
        for test_setup in layer.test_setups:
            test_setup()

        test()

        # test teardowns
        for test_teardown in layer.test_teardowns:
            test_teardown()

    # child layers
    for child_layer in layer.children:
        run_layer_tests(child_layer)
```

```
    # layer teardowns
    for teardown in layer.teardowns:
        teardown()
```

2.1.2 Getting Started

2.1.2.1 Do I need to install anything?

Yes, you'll need to install `contextional` through `pip`, like so:

```
$ pip install contextional
```

2.1.2.2 Do I need to import anything?

Yes, but luckily, you only need to import `GCM` (`GcmMaker`), like this:

```
from contextional import GCM
```

2.1.2.3 How do I create the first layer?

To create the first layer, you'll just need to import `GCM` and use a `with` statement to create a `Context` instance while giving the first layer of that `Context` a description. That instance is what you'll be using to add fixtures, tests, and child layers.

Here's what this looks like:

```
from contextional import GCM


with GCM("First Layer") as FL:
```

Here we've given the first layer a description of "First Layer", and created a `Context` instance, `FL`, that we can use to add fixtures, tests, and child layers.

2.1.2.4 How do I add a test?

For that, you would use `GCM.add_test()`, which is a decorator that takes a single argument (the description of the test).

Here's what it will look like once we've added a test:

```
from contextional import GCM


with GCM("First Layer") as FL:

    @GCM.add_test("1 is True")
    def test(case):
        case.assertTrue(1)
```

2.1.2.5 How do I get the tests to run?

After you're done defining everything, you may have noticed that your tests didn't actually run. That's likely because you will need to have your `GCM` call `create_tests()`, and make sure you pass it `globals()` as an argument. This is what creates the stuff that your testing framework will actually use.

Here's what it looks like:

```
from contextional import GCM


with GCM("First Layer") as FL:

    @GCM.add_test("1 is True")
    def test(case):
        case.assertTrue(1)


FL.create_tests()
```

With that, you can just use your testing framework like you normally would, and it will automatically detect and run these tests (assuming it works with tests made with `unittest`). For example, if you use `nosetests` to run your tests, you can just run it like this:

```
$ nosetests -v
```

If you do that, the test output for this would look something like this:

```
First Layer
  1 is True ... ok
```

Note: If you experience any problems with `create_tests()`, try passing it `globals()` (i.e. `FL.create_tests(globals())`). It tries to automatically grab the namespace of the module it's called in, but, depending on the implementation of Python that you're using, it might have some issues, but this should resolve them.

Do I have to name the test "`test`"?

Not at all; you can name it whatever you want. I just find that giving it a name of `test` makes it easy and straightforward to read. But if you would prefer to name it something else, you absolutely can. The same goes for fixtures, as well.

In fact, every test and fixture that you define and decorate using the `GCM`'s decorator methods will not exist after the decorator is evaluated, as the decorator doesn't return a function to replace it. This was done intentionally so that nothing is leftover that could be found by the test discovery process that you wouldn't want to be found.

Does the test have to take an argument?

Nope. But if you have it take an argument, you can use that argument to access the `unittest.TestCase` assert method and any assert methods you provided with **:method:'.GCM.utilize_asserts'**.

Does the argument that the test takes have to be named "`case`"?

Nope. It can be named whatever you want. Naming it "`case`" is just a suggestion.

Can the test take more than one argument?

Nope. It's either no arguments, or one argument.

If you're looking to do something like paramaterized tests, Contextional does support it, but you'll have to go to the "Advanced" section below to find out more about how to do that.

2.1.2.6 How do I add fixtures (i.e. setups and teardowns)?

Adding fixtures is also done through a decorator. You have the following options for fixtures:

- `GCM.add_setup()`
- `GCM.add_test_setup()`
- `GCM.add_test_teardown()`
- `GCM.add_teardown()`

To add a setup for the layer, you would do something like this:

```
from contextional import GCM


with GCM("Main Group") as MG:

    @GCM.add_setup
    def setUp():
        GCM.value = 1

    @GCM.add_test("value is 1")
    def test(case):
        case.assertEqual(GCM.value, 1)


MG.create_tests()
```

And with that, our test output would look like this:

```
Main Group
    value is 1 ... ok
```

You would take the same approach for all the other fixture types.

Can I have a layer with fixtures, but no tests?

Yes, but in order for it to be used, that layer must have a descendant layer that has tests. Otherwise, it will be ignored.

Also, if the fixtures are test-level fixtures (i.e. test setups and test teardowns), then they will definitely not be used if there aren't any tests defined in the same layer.

Then can I at least have multiple fixtures of a given type in a single layer?

Yes, and this is actually recommended. It's good to break up the various steps of your setups/teardowns into individual functions as it compartmentalizes your code in the event that you want to make a change or have an error.

Can I have each of those fixtures spit out what they're doing while they do it?

You sure can.

Just like tests can be given a description, setups and teardowns can also be given a description that will be printed out as each one is run; and if it throws an error, you'll see that description in the error report.

Here's an example of how to give a fixture a description:

```
from contextional import GCM


with GCM("Main Group") as MG:

    @GCM.add_setup("do a thing")
    def setUp():
        GCM.value = 1

    @GCM.add_teardown("undo all the things")
    def setUp():
        del GCM.value

    with GCM.add_group("Child Group"):

        @GCM.add_setup("do another thing")
        def setUp():
            GCM.value += 1

        @GCM.add_teardown("undo that last thing")
        def setUp():
            GCM.value -= 1

        @GCM.add_test("value is 2")
        def test(case):
            case.assertEqual(GCM.value, 2)


MG.create_tests()
```

and that would output this:

```
Main Group
  # do a thing
  Child Group
    # do another thing
    value is 2 ... ok
    # undo that last thing
  # undo all the things
```

Do I have to give a description to every setup and teardown?

Nope.

Not everything needs a description, so if you don't give a fixture a description, it just won't show up in the test output.

However, if a fixture throws an error, a generic description of the fixture will be spat out to show where the error occured specifically. It would look something like this:

```
Main Group
  Child Group
    # setup (2/5) ERROR
    some test ... FAIL
  # teardown (1/1) ERROR
```

And you would see something similar in the error report.

The two numbers are 1) the 1-indexed position of the fixture and 2) the total number of fixtures of that type in that layer. So if you see `# setup (2/5) ERROR`, that means there were 5 setups total in that group, and the 2nd one threw an error.

2.1.2.7 How do I predefine a `Context` that I can use elsewhere?

Whenever you define a `Context`, it doesn't *need* to have its tests run. That only happens if you have it call `create_tests()`. However, you can create a `Context`, which contains layers of fixtures and tests, that you can include in any other `Context` at any point, and even use it multiple times in the same `Context`.

To do this, you just need to create a `Context` containing the layers and fixtures that you want to use elsewhere, using the exact same syntax that you would use with any other `Context`, but don't have it call `create_tests()`. Once you've done this, go to where you are creating the `Context` that you want to include your predefined `Context`, and have it call `includes()` at the point that you want it to include the predefined `Context`.

The process looks something like this:

```
from contextional import GCM


with GCM("Predefined Group") PG:

    @GCM.add_setup
    def setUp():
        GCM.value += 1

    @GCM.add_test("value is now 2")
    def test(case):
        case.assertEqual(GCM.value, 2)


with GCM("Main Group") as MG:

    @GCM.add_setup
    def setUp():
        GCM.value = 1

    @GCM.add_test("value is 1")
    def test(case):
        case.assertEqual(GCM.value, 1)
```

```
    GCM.includes(PG)


MG.create_tests()
```

The output for this would look something like this:

```
Main Group
  value is 1 ... ok
  Predefined Group
    value is now 2 ... ok
```

Can I use the predefined `Context` in more than one spot?

Yep!

Even multiple times in the same `Context`?

Yup!

What about in other modules than the one I created it in?

Absolutely!

The `Context` that you want to include in other any `Context` is just like any other object. Even though it was created using context managers, nothing really happens to it once the outermost context is exited. Because of this, all you need to do is import it in the module you want to use it.

So if you started it off by saying:

```
with GCM("Includable Group") as IG:
```

then you would only need to say this in the module that would use it:

```
from some.module import IG
```

What if I want to include the predefined tests, fixtures, and/or child groups from a `Context` alongside those from my current group?

You can just use `combine()`, then. It takes the tests, fixtures, and child groups of a `Context` and makes them part of the group you're merging them into, so they won't just be added as a child group.

This is useful if you know you are going to be using identical tests but on different things.

It looks something like this:

```
def multiplier(num_1, num_2):
    return num_1 * num_2


with GCM("value test") as vt:
```

```
    @GCM.add_test("value")
    def test(case):
        case.assertEqual(
            GCM.value,
            GCM.expected_value,
        )

with GCM("Main Group") as MG:

    with GCM.add_group("2 and 3"):

        @GCM.add_setup
        def setUp():
            GCM.value = multiplier(2, 3)
            GCM.expected_value = 6

        GCM.combine(vt)

    with GCM.add_group("3 and 5"):

        @GCM.add_setup
        def setUp():
            GCM.value = multiplier(3, 5)
            GCM.expected_value = 15

        GCM.combine(vt)
```

Output:

```
Main Group
  value is 1 ... ok
  Sub Group
    value is still 1 ... ok
```

2.1.2.8 How can my fixtures and tests use persistent resources as they run?

Normally, when working with `unittest.TestCase`, you could use class attributes in `setUpClass()` or `tearDownClass()` (i.e. `cls`), or instance attributes in `setUp()` or `tearDown()` (i.e. `self`) to give your fixtures and tests access to persistent resources.

To let you do something similar, Contextional uses some Python magic to let each `Context` access a shared, persistent namespace.

You may have noticed it in the examples above, but the shared, persistent namespace is accessed through `GCM`. Just access an attribute of `GCM` in any test or fixture, and as long as it isn't one of it's normal attributes, you'll be referencing a persistent namespace from one test/fixture to another.

2.1.2.9 Can I see a simple example to get me started?

Sure! Here you go:

```
from contextional import GCM


with GroupContextManager("Predefined Group") PG:
```

```
    @GCM.add_test("value is still 1")
    def test(case):
        case.assertEqual(GCM.value, 1)


with GCM("Main Group") as MG:

    @GCM.add_setup
    def setUp():
        GCM.value = 1

    @GCM.add_test("value is 1")
    def test(case):
        case.assertEqual(GCM.value, 1)

    GCM.includes(PG)

    with GCM.add_group("Child Group"):

        @GCM.add_setup
        def setUp():
            GCM.value += 1

        @GCM.add_test("value is now 2")
        def test(case):
            case.assertEqual(GCM.value, 2)

    with GCM.add_group("Another Child Group"):

        @GCM.add_setup
        def setUp():
            GCM.value += 1

        @GCM.add_test("value is now 3")
        def test(case):
            case.assertEqual(GCM.value, 3)


MG.create_tests()
```

That would output the following:

```
Main Group
  value is 1 ... ok
  Predefined Group
    value is still 1 ... ok
  Child Group
    value is now 2 ... ok
  Another Child Group
    value is now 3 ... ok
```

2.2 Advanced Usage

2.2.1 Metaprogramming

Contextional works by having you write normal code that, when evaluted during the test discovery process, creates objects containing information about the tests you want to create and how you want them to run. Those objects are later used to create the tests that actually get run by the testing framework (when you call `create_tests()`).

Because of this, you can use any tools at your disposal to help create those objects just like you would create any other object in a Python script.

For example, let's say you have a series of characters that you want to check exist in a larger string. You could quickly write all of those tests using a for loop:

```
from string import ascii_lowercase


main_string = "the quick brown fox jumped over the lazy dog"


with GCM(main_string) as MG:

    for letter in ascii_lowercase:

        @GCM.add_test("contains '{}'".format(letter))
        def test(case):
            case.assertIn(letter, main_string)
```

which would spit out something like this:

```
the quick brown fox jumped over the lazy dog
  contains 'a' ... ok
  contains 'b' ... ok
  contains 'c' ... ok
  ...
  contains 'x' ... ok
  contains 'y' ... ok
  contains 'z' ... ok
```

You can even do it for whole child groups:

```
from string import ascii_lowercase


main_string = "the quick brown fox jumped over the lazy dog"


with GCM(main_string) as MG:

    for letter in ascii_lowercase:

        with GCM.add_group("Letter: '{}'".format(letter.upper())):

            @GCM.add_test("is present")
            def test(case):
                case.assertIn(letter, main_string)
```

which would be something like this:

```
the quick brown fox jumped over the lazy dog
  Letter: 'A'
    is present ... ok
  Letter: 'B'
    is present ... ok
  Letter: 'C'
    is present ... ok
  ...
  Letter: 'X'
    is present ... ok
  Letter: 'Y'
    is present ... ok
  Letter: 'Z'
    is present ... ok
```

2.2.2 Parameterization

Contextional handles parameterization by allowing you to pass parameters to `add_group()`. If parameters are passed, Contextional will make one version of the parameterized group for each set of parameters, so if you have 5 sets of parameters for a group, 5 versions of that group will be created. All of the child groups of the parameterized group will be included in each version of the parameterized group.

2.2.2.1 How do I format the sets of parameters that I want to use?

There's actually 2 parts to this that you can utilize.

Collections of Sets (or "Collections of Collections")

First, is how you provide the collection of sets.

You can either put each set of parameters into a `set`/`list`/`tuple`, like so:

```
with GCM("Main Group") as MG:

    my_params = (
        (1, 3, 5),
        (2, 4, 6),
    )
    with GCM.add_group("Parameterized Group:", params=my_params):
```

or you can put them in a `Mapping` (e.g. a `dict`), like this:

```
with GCM("Main Group") as MG:

    my_params = {
        "odds": (1, 3, 5),
        "evens": (2, 4, 6),
    }
    with GCM.add_group("Parameterized Group:", params=my_params):
```

The difference, is in the test output. Each version of the parameterized group will need to distinguish itself from the other versions of itself so that someone reading the test output can more easily tell where a problem occured (if there was one). To do this, the description of the group is changed.

If the collection of sets used was a `set`/`list`/`tuple`, then the set of parameters itself will be appended to the group's normal description. In the example above, you would see the output look like this:

```
Main Group
  Parameterized Group: (1, 3, 5)
    ...
  Parameterized Group: (2, 4, 6)
    ...
```

If the collection of sets used was a `Mapping`, then the key for the set of parameters itself will be appended to the group's normal description. In the example above, you would see the output look like this:

```
Main Group
  Parameterized Group: odds
    ...
  Parameterized Group: evens
    ...
```

The Sets of Parameters Themselves

Regardless of what kind of collection the sets of parameters are put into together, the individual set of parameters that each version of the group uses will be unpacked and passed as arguments to each of the `setUp()` functions that were defined in the root layer of the parameterized group. Child groups of the parameterized group will not have the parameters passed to them. It is up to the `setUp()` function(s) of parameterized group to make sure their child groups can access the parameters, if needed.

If a set of parameters is a `set`/`list`/`tuple`, then it will be unpacked with a single `*`, so you can either have your `setUp()` functions catch them all with a `*args`, or just make sure they take the right number of ordered arguments. It will look something like this:

```
with GCM("Main Group") as MG:

    my_params = (
        (1, 3, 5),
        (2, 4, 6),
    )
    with GCM.add_group("Parameterized Group:", params=my_params):

        @GCM.add_setup
        def setUp(*args):
            # some code

        @GCM.add_setup
        def setUp(num_1, num_2, num_3):
            # some code
```

If a set of parameters is a `Mapping` (e.g. a `dict`), then it will be unpacked with a `**`, so your `setUp()` functions can catch them all with a `**kwargs`, or they can accept the appropriately name keyword arguments. That will look something like this:

```
with GCM("Main Group") as MG:

    my_params = (
        {
            "num_1": 1,
            "num_2": 3,
```

```
            "num_3": 5,
        },
        {
            "num_1": 2,
            "num_2": 4,
            "num_3": 6,
        },
    )
    with GCM.add_group("Parameterized Group:", params=my_params):

        @GCM.add_setup
        def setUp(**kwargs):
            # some code

        @GCM.add_setup
        def setUp(num_1, num_2, num_3):
            # some code
```

This allows you to set default values for your parameters, and control how much flexibility you want with your parameters.

CHAPTER 3

Reference

3.1 GroupContextManager

contextional.**GroupContextManager**

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

G