# unittest_expander Documentation

***Release 0.5.0.dev0***

**Jan Kaliszewski (zuo) and others**

**Sep 26, 2023**

# Contents

Welcome to the documentation for *unittest_expander* – a Python library that provides flexible and easy-to-use tools **to parametrize unit tests**, especially (but not limited to) those based on `unittest.TestCase`.

CHAPTER 1

# Getting Started

*unittest_expander* is a Python library that provides flexible and easy-to-use tools to parametrize your unit tests, especially (but not limited to) those based on *unittest.TestCase*.

The library is compatible with Python 3.11, 3.10, 3.9, 3.8, 3.7, 3.6 and 2.7, and does not depend on any external packages (i.e., uses only the Python standard library).

**Authors** Jan Kaliszewski (zuo) and others...

**License** MIT License

**Home Page** https://github.com/zuo/unittest_expander

**Documentation** https://unittest-expander.readthedocs.io/en/stable/

## 1.1 Installing

**The easiest way** to install the library is to execute (preferably in a virtualenv) the command:

```
python -m pip install unittest_expander
```

(note that you need network access to do it this way). If you do not have the *pip* tool installed – see: https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/

**Alternatively**, you can download the library source archive, unpack it, `cd` to the unpacked directory, and execute (preferably in a virtualenv) the following command:

```
python -m pip install .
```

Note: you may need to have administrator privileges if you do *not* operate in a *virtualenv*.

**It is also possible** to use the library without installing it: as its code is contained in a single file (`unittest_expander.py`), you can just copy it into your project.

## 1.2 Usage example

Consider the following **ugly** test:

```
import unittest

class Test(unittest.TestCase):
    def test_sum(self):
        for iterable, expected in [
            ([], 0),
            ([0], 0),
            ([3], 3),
            ([1, 3, 1], 5),
            (frozenset({1, 3}), 4),
            ({1:'a', 3:'b'}, 4),
        ]:
            self.assertEqual(sum(iterable), expected)
```

Is it cool? **Not at all!** So let's improve it:

```
import unittest
from unittest_expander import expand, foreach

@expand
class Test(unittest.TestCase):
    @foreach(
        ([], 0),
        ([0], 0),
        ([3], 3),
        ([1, 3, 1], 5),
        (frozenset({1, 3}), 4),
        ({1:'a', 3:'b'}, 4),
    )
    def test_sum(self, iterable, expected):
        self.assertEqual(sum(iterable), expected)
```

Now you have **6 distinct tests** (properly *isolated* and being always *reported as separate tests*), although they share the same test method source.

You may want to do the same in a bit more verbose and descriptive way:

```
import unittest
from unittest_expander import expand, foreach, param

@expand
class Test(unittest.TestCase):

    test_sum_params = [
        param([], expected=0).label('empty gives 0'),
        param([0], expected=0),
        param([3], expected=3),
        param([1, 3, 1], expected=5),
        param(frozenset({1, 3}), expected=4),
        param({1:'a', 3:'b'}, expected=4).label('even dict is ok'),
    ]

    @foreach(test_sum_params)
```

(continues on next page)

(continued from previous page)

```
    def test_sum(self, iterable, expected):
        self.assertEqual(sum(iterable), expected)
```

This is only a fraction of the possibilities *unittest_expander* offers to you.

You can **learn more** from the actual documentation of the module.

CHAPTER 2

# Narrative Documentation

*unittest_expander* is a Python library that provides flexible and easy-to-use tools to parametrize your unit tests, especially (but not limited to) those based on `unittest.TestCase` from the Python standard library.

The *`unittest_expander`* module provides the following tools:

- a test class decorator: *`expand()`*,
- a test method decorator: *`foreach()`*,
- and a few helpers: *`param`*, *`paramseq`* and *`current`*.

Let's see how to use them...

## 2.1 Basic use of `expand()` and `foreach()`

Assume we have a function that checks whether the given number is even or not:

```
>>> def is_even(n):
...     return n % 2 == 0
```

Of course, in the real world the code we write is usually more interesting... Anyway, most often we want to test how it works for different parameters. At the same time, it is not the best idea to test many cases in a loop within one test method – because of lack of test isolation (tests depend on other ones – they may inherit some state which can affect their results), less information on failures (a test failure or fixture-related error may prevent subsequent tests from being run), less clear fail messages (when you don't see at first glance which case is the actual culprit), etc.

So let's write our tests in a smarter way:

```
>>> import unittest
>>> from unittest_expander import expand, foreach
>>>
>>> @expand
... class Test_is_even(unittest.TestCase):
...
```

(continues on next page)

(continued from previous page)

```
...     @foreach(0, 2, -14)         # call variant #1: parameter collection
...     def test_even(self, n):     # specified using multiple arguments
...         self.assertTrue(is_even(n))
...
...     @foreach([-1, 17])          # call variant #2: parameter collection as
...     def test_odd(self, n):      # one argument being a container (e.g. list)
...         self.assertFalse(is_even(n))
...
...     # just to demonstrate that test cases are really isolated
...     def setUp(self):
...         sys.stdout.write(' [DEBUG: separate test setUp] ')
...         sys.stdout.flush()
```

As you see, it's fairly simple: you bind parameter collections to your test methods with the *foreach()* decorator and decorate the whole test class with the *expand()* decorator. The latter does the actual job, i.e., generates (and adds to the test class) parametrized versions of the test methods.

Let's run this stuff...

```
>>> # a helper function that will run tests in our examples --
>>> # NORMALLY YOU DON'T NEED IT, of course!
>>> import sys
>>> def run_tests(*test_classes):
...     suite = unittest.TestSuite(
...         unittest.TestLoader().loadTestsFromTestCase(cls)
...         for cls in test_classes)
...     unittest.TextTestRunner(stream=sys.stdout, verbosity=2).run(suite)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_even__<-14> ... [DEBUG: separate test setUp] ok
test_even__<0> ... [DEBUG: separate test setUp] ok
test_even__<2> ... [DEBUG: separate test setUp] ok
test_odd__<-1> ... [DEBUG: separate test setUp] ok
test_odd__<17> ... [DEBUG: separate test setUp] ok
...Ran 5 tests...
OK
```

To test our *is_even()* function we created two test methods – each accepting one parameter value.

Another approach could be to define a method that accepts a couple of arguments:

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         (-14, True),
...         (-1, False),
...         (0, True),
...         (2, True),
...         (17, False),
...     )
...     def test(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
```

(continues on next page)

(continued from previous page)

```
test__<-1,False> ... ok
test__<-14,True> ... ok
test__<0,True> ... ok
test__<17,False> ... ok
test__<2,True> ... ok
...Ran 5 tests...
OK
```

As you see, you can use a tuple to specify several parameter values for a test call.

## 2.2 More flexibility: `param`

Parameters can be specified in a more descriptive way – in particular, using also keyword arguments. It is possible when you use *param* objects instead of tuples:

```
>>> from unittest_expander import param
>>>
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         param(-14, expected=True),
...         param(-1, expected=False),
...         param(0, expected=True),
...         param(2, expected=True),
...         param(17, expected=False),
...     )
...     def test(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<-1,expected=False> ... ok
test__<-14,expected=True> ... ok
test__<0,expected=True> ... ok
test__<17,expected=False> ... ok
test__<2,expected=True> ... ok
...Ran 5 tests...
OK
```

Generated *labels* of our tests (attached to the names of the generated test methods) became less cryptic. But what to do if we need to label our parameters explicitly?

We can use the *label()* method of *param* objects:

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         param(sys.maxsize, expected=False).label('sys.maxsize'),
...         param(-sys.maxsize, expected=False).label('-sys.maxsize'),
...     )
...     def test(self, n, expected):
...         actual = is_even(n)
```

(continues on next page)

(continued from previous page)

```
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<-sys.maxsize> ... ok
test__<sys.maxsize> ... ok
...Ran 2 tests...
OK
```

Note that the *label* (either auto-generated from parameter values or explicitly specified with *param.label()*) is available within the test method definition as *current.label* (*current* is a special object *described later*):

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         param(sys.maxsize, expected=False).label('sys.maxsize'),
...         param(-sys.maxsize, expected=False).label('-sys.maxsize'),
...     )
...     def test(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert current.label in ('sys.maxsize', '-sys.maxsize')
...         sys.stdout.write(' [DEBUG: {!r}] '.format(current.label))
...         sys.stdout.flush()
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<-sys.maxsize> ... [DEBUG: '-sys.maxsize'] ok
test__<sys.maxsize> ... [DEBUG: 'sys.maxsize'] ok
...Ran 2 tests...
OK
```

## 2.3 Other ways to explicitly label your tests

You can also label particular tests by passing a dictionary directly into *foreach()*:

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach({
...         'non-integer': (1.2345, False),
...         'horrible abuse!': ('%s', False),
...     })
...     def test(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert current.label in ('non-integer', 'horrible abuse!')
...         sys.stdout.write(' [DEBUG: {!r}] '.format(current.label))
...         sys.stdout.flush()
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<horrible abuse!> ... [DEBUG: 'horrible abuse!'] ok
```

(continues on next page)

(continued from previous page)

```
test__<non-integer> ... [DEBUG: 'non-integer'] ok
...Ran 2 tests...
OK
```

. . . or just using keyword arguments:

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         noninteger=(1.2345, False),
...         horrible_abuse=('%s', False),
...     )
...     def test(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert current.label in ('noninteger', 'horrible_abuse')
...         sys.stdout.write(' [DEBUG: {!r}] '.format(current.label))
...         sys.stdout.flush()
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<horrible_abuse> ... [DEBUG: 'horrible_abuse'] ok
test__<noninteger> ... [DEBUG: 'noninteger'] ok
...Ran 2 tests...
OK
```

## 2.4 Smart parameter collection: `paramseq`

How to concatenate some separately created parameter collections?

Just transform them (or at least the first of them) into *`paramseq`* instances – and then add one to another (with the + operator):

```
>>> from unittest_expander import paramseq
>>>
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     basic_params1 = paramseq(   # init variant #1: several parameters
...         param(-14, expected=True),
...         param(-1, expected=False),
...     )
...     basic_params2 = paramseq([  # init variant #2: one parameter collection
...         param(0, expected=True).label('just zero, because why not?'),
...         param(2, expected=True),
...         param(17, expected=False),
...     ])
...     basic = basic_params1 + basic_params2
...
...     huge = paramseq({  # explicit labelling by passing a dict
...         'sys.maxsize': param(sys.maxsize, expected=False),
...         '-sys.maxsize': param(-sys.maxsize, expected=False),
...     })
```

(continues on next page)

(continued from previous page)

```
...
...     other = paramseq(
...         (-15, False),
...         param(15, expected=False),
...         # explicit labelling with keyword arguments:
...         noninteger=param(1.2345, expected=False),
...         horribleabuse=param('%s', expected=False),
...     )
...
...     just_dict = {
...         '18->True': (18, True),
...     }
...
...     just_list = [
...         param(12399999999999999, False),
...         param(n=12399999999999998, expected=True),
...     ]
...
...     # just add them one to another (producing a new paramseq)
...     all_params = basic + huge + other + just_dict + just_list
...
...     @foreach(all_params)
...     def test(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<-1,expected=False> ... ok
test__<-14,expected=True> ... ok
test__<-15,False> ... ok
test__<-sys.maxsize> ... ok
test__<15,expected=False> ... ok
test__<17,expected=False> ... ok
test__<18->True> ... ok
test__<2,expected=True> ... ok
test__<<12399999999...>,False> ... ok
test__<expected=True,n=<12399999999...>> ... ok
test__<horribleabuse> ... ok
test__<just zero, because why not?> ... ok
test__<noninteger> ... ok
test__<sys.maxsize> ... ok
...Ran 14 tests...
OK
```

**Note:** Parameter collections – being *sequences* (e.g., `list` instances), or *mappings* (e.g., `dict` instances), or *sets* (e.g., `set` or `frozenset` instances), or callable objects (see below...), or just ready *`paramseq`* instances – do not need to be created or bound within the test class body; you could, for example, import them from a separate module. Obviously, that makes data/code reuse and refactorization easier.

Also, note that the call signatures of *`foreach()`* and the *`paramseq`* constructor are identical: you pass in either exactly one positional argument which is a parameter collection, or several (more than one) positional and/or keyword arguments being *`param`* instances or tuples of parameter values, or singular parameter values.

**Note:** We said that a parameter collection can be a *sequence* (among others; see the note above). To be more precise: it can be a *sequence*, except that it *cannot be*: a `tuple`, a *text string* (`str` in Python 3, `str` or `unicode` in Python 2) or a Python 3 *binary string-like sequence* (`bytes` or `bytearray`).

A *paramseq* instance can also be created from a callable object (e.g., a function) that returns a *sequence* or another *iterable* object (e.g., a generator iterator):

```
>>> from random import randint
>>>
>>> @paramseq   # <- yes, used as a decorator
... def randomized(test_cls):
...     print('DEBUG: LO = {0.LO}; HI = {0.HI}'.format(test_cls))
...     print('----')
...     yield param(randint(test_cls.LO, test_cls.HI) * 2,
...                 expected=True).label('random even')
...     yield param(randint(test_cls.LO, test_cls.HI) * 2 + 1,
...                 expected=False).label('random odd')
...
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     LO = -100
...     HI = 100
...
...     @foreach(randomized)
...     def test_even(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
...     # reusing the same instance of paramseq to show that the underlying
...     # callable is called separately for each use of @foreach:
...     @foreach(randomized)
...     def test_not_even_when_incremented(self, n, expected):
...         actual = (not is_even(n + 1))
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
DEBUG: LO = -100; HI = 100
----
DEBUG: LO = -100; HI = 100
----
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_even__<random even> ... ok
test_even__<random odd> ... ok
test_not_even_when_incremented__<random even> ... ok
test_not_even_when_incremented__<random odd> ... ok
...Ran 4 tests...
OK
```

A callable object (such as the generator function in the example above) which is passed to the *paramseq*'s constructor (or directly to *foreach()*) should accept either no arguments or one positional argument – in the latter case the *test class* will be passed in.

**Note:** The callable object will be called – and its *iterable* result will be iterated over (consumed) – *when* the *expand()* decorator is being executed, *directly before* generating parametrized test methods.

What should also be emphasized is that those operations (the aforementioned call and iterating over its result) will be performed *separately* for each use of *foreach()* with our *paramseq* instance as its argument (or with another *paramseq* instance that includes our instance; see the following code snippet in which the `input_values_and_results` instance includes the previously created `randomized` instance).

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     LO = -999999
...     HI = 999999
...
...     # reusing the same, previously created, instance of paramseq
...     # (`randomized`) to show that the underlying callable will
...     # still be called separately for each use of @foreach...
...     input_values_and_results = randomized + [
...         param(-14, expected=True),
...         param(-1, expected=False),
...         param(0, expected=True),
...         param(2, expected=True),
...         param(17, expected=False),
...     ]
...
...     @foreach(input_values_and_results)
...     def test_even(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
...     @foreach(input_values_and_results)
...     def test_not_even_when_incremented(self, n, expected):
...         actual = (not is_even(n + 1))
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
DEBUG: LO = -999999; HI = 999999
----
DEBUG: LO = -999999; HI = 999999
----
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_even__<-1,expected=False> ... ok
test_even__<-14,expected=True> ... ok
test_even__<0,expected=True> ... ok
test_even__<17,expected=False> ... ok
test_even__<2,expected=True> ... ok
test_even__<random even> ... ok
test_even__<random odd> ... ok
test_not_even_when_incremented__<-1,expected=False> ... ok
test_not_even_when_incremented__<-14,expected=True> ... ok
test_not_even_when_incremented__<0,expected=True> ... ok
test_not_even_when_incremented__<17,expected=False> ... ok
test_not_even_when_incremented__<2,expected=True> ... ok
test_not_even_when_incremented__<random even> ... ok
test_not_even_when_incremented__<random odd> ... ok
...Ran 14 tests...
OK
```

## 2.5 Combining several `foreach()` to get Cartesian product

You can stack up two or more *foreach()* decorators to the same test method – to combine several parameter collections, obtaining the Cartesian product of them:

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     # one param collection (7 items)
...     @paramseq
...     def randomized():
...         yield param(randint(-(10 ** 6), 10 ** 6) * 2,
...                     expected=True).label('random even')
...         yield param(randint(-(10 ** 6), 10 ** 6) * 2 + 1,
...                     expected=False).label('random odd')
...     input_values_and_results = randomized + [  # (<- note the use of +)
...         param(-14, expected=True),
...         param(-1, expected=False),
...         param(0, expected=True),
...         param(2, expected=True),
...         param(17, expected=False),
...     ]
...
...     # another param collection (2 items)
...     input_types = dict(
...         integer=int,
...         floating=float,
...     )
...
...     # let's combine them (7 * 2 -> 14 parametrized tests)
...     @foreach(input_values_and_results)
...     @foreach(input_types)
...     def test(self, input_type, n, expected):
...         n = input_type(n)
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<floating, -1,expected=False> ... ok
test__<floating, -14,expected=True> ... ok
test__<floating, 0,expected=True> ... ok
test__<floating, 17,expected=False> ... ok
test__<floating, 2,expected=True> ... ok
test__<floating, random even> ... ok
test__<floating, random odd> ... ok
test__<integer, -1,expected=False> ... ok
test__<integer, -14,expected=True> ... ok
test__<integer, 0,expected=True> ... ok
test__<integer, 17,expected=False> ... ok
test__<integer, 2,expected=True> ... ok
test__<integer, random even> ... ok
test__<integer, random odd> ... ok
...Ran 14 tests...
OK
```

If parameters combined this way specify some conflicting keyword arguments, they are detected and an error is reported:

```
>>> params1 = [param(a=1, b=2, c=3)]
>>> params2 = [param(b=4, c=3, d=2)]
>>>
>>> @expand   # doctest: +ELLIPSIS
... class TestSomething(unittest.TestCase):
...
...     @foreach(params2)
...     @foreach(params1)
...     def test(self, **kw):
...         "something"
...
Traceback (most recent call last):
  ...
ValueError: conflicting keyword arguments: 'b', 'c'
```

## 2.6 Context-manager-based fixtures: `param.context()`

When dealing with resources managed with context managers, you can specify a *context manager factory* and its arguments using the *`context()`* method of a *`param`* object – then each call of the resultant parametrized test will be enclosed in a dedicated *context manager* instance (created by calling the *context manager factory* with the given arguments).

XXX TODO

```
>>> from tempfile import NamedTemporaryFile
>>>
>>> @expand
... class TestSaveLoad(unittest.TestCase):
...
...     params_with_contexts = [
...         param(save='', load='').context(NamedTemporaryFile, 'w+t'),
...         param(save='abc', load='abc').context(NamedTemporaryFile, 'w+t'),
...     ]
...
...     @foreach(params_with_contexts)
...     def test_save_load(self, save, load):
...         file = current.context_targets[0]
...         file.write(save)
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load)
...
...     # reusing the same params to show that a *new* context manager
...     # instance is created for each test call:
...     @foreach(params_with_contexts)
...     def test_save_load_with_spaces(self, save, load):
...         file = current.context_targets[0]
...         file.write(' ' + save + ' ')
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, ' ' + load + ' ')
...
>>> run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test_save_load__<load='',save=''> ... ok
test_save_load__<load='abc',save='abc'> ... ok
```

(continues on next page)

(continued from previous page)

```
test_save_load_with_spaces__<load='',save=''> ... ok
test_save_load_with_spaces__<load='abc',save='abc'> ... ok
...Ran 4 tests...
OK
>>>
>>> # repeating the tests to show that, really, a *new* context manager
... # instance is created for *each* test call:
... run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test_save_load__<load='',save=''> ... ok
test_save_load__<load='abc',save='abc'> ... ok
test_save_load_with_spaces__<load='',save=''> ... ok
test_save_load_with_spaces__<load='abc',save='abc'> ... ok
...Ran 4 tests...
OK
```

As you can see in the above example, a *sequence* of context manager *as-targets* (i.e., objects returned by context managers' `__enter__()`) is available within the test method definition as *`current.context_targets`* (*`current`* is a special object *described later*):

*`current.context_targets`* is a sequence because there can be more than one *context* per parameter collection's item, e.g.:

```
>>> import contextlib
>>> @contextlib.contextmanager
... def debug_cm(tag=None):
...     debug.append('enter' + (':{}'.format(tag) if tag else ''))
...     yield tag
...     debug.append('exit' + (':{}'.format(tag) if tag else ''))
...
>>> debug = []
>>>
>>> @expand
... class TestSaveLoad(unittest.TestCase):
...
...     params_with_contexts = [
...         (
...             param(save='', load='', expected_tag='FOO')
...               .context(NamedTemporaryFile, 'w+t')  # (outer one)
...               .context(debug_cm, tag='FOO')        # (inner one)
...         ),
...         (
...             param(save='abc', load='abc', expected_tag='BAR')
...               .context(NamedTemporaryFile, 'w+t')
...               .context(debug_cm, tag='BAR')
...         ),
...     ]
...
...     @foreach(params_with_contexts)
...     def test_save_load(self, save, load, expected_tag):
...         file, tag = current.context_targets
...         assert tag == expected_tag
...         file.write(save)
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load)
...         debug.append('test')
...
```

(continues on next page)

(continued from previous page)

```
>>> debug == []
True
>>> run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test_save_load__<expected_tag='BAR',load='abc',save='abc'> ... ok
test_save_load__<expected_tag='FOO',load='',save=''> ... ok
...Ran 2 tests...
OK
>>> debug == [
...     'enter:BAR', 'test', 'exit:BAR',
...     'enter:FOO', 'test', 'exit:FOO',
... ]
True
```

Contexts are properly handled (context managers' `__enter__()` and `__exit__()` are properly called...) also when errors occur (with some legitimate subtle reservations – see: *Contexts cannot suppress exceptions unless you enable that explicitly*):

```
>>> class ErrDebugCM(object):
...
...     def __init__(self, tag):
...         debug.append('init:' + tag)
...         self._tag = tag
...
...     def __enter__(self):
...         if self._tag.endswith('context-enter-error'):
...             debug.append('ERR-enter:' + self._tag)
...             raise RuntimeError('error in __enter__')
...         debug.append('enter:' + self._tag)
...         return self._tag
...
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         if exc_type is None:
...             if self._tag.endswith('context-exit-error'):
...                 debug.append('ERR-exit:' + self._tag)
...                 raise RuntimeError('error in __exit__')
...             debug.append('exit:' + self._tag)
...         else:
...             debug.append('ERR-exit:' + self._tag)
...
>>> debug = []
>>> err_params = [
...     (
...         param().label('no_error')
...                .context(ErrDebugCM, tag='outer')
...                .context(ErrDebugCM, tag='inner')
...     ),
...     (
...         param().label('test_fail')
...                .context(ErrDebugCM, tag='outer')
...                .context(ErrDebugCM, tag='inner')
...     ),
...     (
...         param().label('test_error')
...                .context(ErrDebugCM, tag='outer')
...                .context(ErrDebugCM, tag='inner')
...     ),
```

(continues on next page)

(continued from previous page)

```
...     (
...         param().label('inner_context_enter_error')
...                .context(ErrDebugCM, tag='outer')
...                .context(ErrDebugCM, tag='inner-context-enter-error')
...     ),
...     (
...         param().label('inner_context_exit_error')
...                .context(ErrDebugCM, tag='outer')
...                .context(ErrDebugCM, tag='inner-context-exit-error')
...     ),
...     (
...         param().label('outer_context_enter_error')
...                .context(ErrDebugCM, tag='outer-context-enter-error')
...                .context(ErrDebugCM, tag='inner')
...     ),
...     (
...         param().label('outer_context_exit_error')
...                .context(ErrDebugCM, tag='outer-context-exit-error')
...                .context(ErrDebugCM, tag='inner')
...     ),
... ]
>>>
>>> @expand
... class SillyTest(unittest.TestCase):
...
...     def setUp(self):
...         debug.append('setUp')
...
...     def tearDown(self):
...         debug.append('tearDown')
...
...     @foreach(err_params)
...     def test(self):
...         if current.label == 'test_fail':
...             debug.append('FAIL-test')
...             self.fail()
...         elif current.label == 'test_error':
...             debug.append('ERROR-test')
...             raise RuntimeError
...         else:
...             debug.append('test')
...
>>> run_tests(SillyTest)  # doctest: +ELLIPSIS
test__<inner_context_enter_error> ... ERROR
test__<inner_context_exit_error> ... ERROR
test__<no_error> ... ok
test__<outer_context_enter_error> ... ERROR
test__<outer_context_exit_error> ... ERROR
test__<test_error> ... ERROR
test__<test_fail> ... FAIL
...Ran 7 tests...
FAILED (failures=1, errors=5)
>>> debug == [
...     # inner_context_enter_error
...     'setUp',
...     'init:outer',
...     'enter:outer',
```

(continues on next page)

(continued from previous page)

```
...     'init:inner-context-enter-error',
...     'ERR-enter:inner-context-enter-error',
...     'ERR-exit:outer',
...     'tearDown',
...
...     # inner_context_exit_error
...     'setUp',
...     'init:outer',
...     'enter:outer',
...     'init:inner-context-exit-error',
...     'enter:inner-context-exit-error',
...     'test',
...     'ERR-exit:inner-context-exit-error',
...     'ERR-exit:outer',
...     'tearDown',
...
...     # no_error
...     'setUp',
...     'init:outer',
...     'enter:outer',
...     'init:inner',
...     'enter:inner',
...     'test',
...     'exit:inner',
...     'exit:outer',
...     'tearDown',
...
...     # outer_context_enter_error
...     'setUp',
...     'init:outer-context-enter-error',
...     'ERR-enter:outer-context-enter-error',
...     'tearDown',
...
...     # outer_context_exit_error
...     'setUp',
...     'init:outer-context-exit-error',
...     'enter:outer-context-exit-error',
...     'init:inner',
...     'enter:inner',
...     'test',
...     'exit:inner',
...     'ERR-exit:outer-context-exit-error',
...     'tearDown',
...
...     # test_error
...     'setUp',
...     'init:outer',
...     'enter:outer',
...     'init:inner',
...     'enter:inner',
...     'ERROR-test',
...     'ERR-exit:inner',
...     'ERR-exit:outer',
...     'tearDown',
...
...     # test_fail
...     'setUp',
```

(continues on next page)

(continued from previous page)

```
...     'init:outer',
...     'enter:outer',
...     'init:inner',
...     'enter:inner',
...     'FAIL-test',
...     'ERR-exit:inner',
...     'ERR-exit:outer',
...     'tearDown',
... ]
True
```

Note that contexts are handled *directly* before (by running `__enter__()`) and after (by running `__exit__()`) each relevant test method call, that is, *after* the `setUp()` and *before* `tearDown()` calls (if those `unittest.TestCase`-specific methods are engaged) – so `setUp()` and `tearDown()` are not affected by any errors related to those contexts.

Obviously, if an error in `setUp()` occurs then the test method is not called at all. Therefore, then, relevant context managers are not even created:

```
>>> def setUp(self):
...     debug.append('setUp')
...     raise ValueError
...
>>> SillyTest.setUp = setUp
>>> debug = []
>>> run_tests(SillyTest)  # doctest: +ELLIPSIS
test__<inner_context_enter_error> ... ERROR
test__<inner_context_exit_error> ... ERROR
test__<no_error> ... ERROR
test__<outer_context_enter_error> ... ERROR
test__<outer_context_exit_error> ... ERROR
test__<test_error> ... ERROR
test__<test_fail> ... ERROR
...Ran 7 tests...
FAILED (errors=7)
>>> debug == ['setUp', 'setUp', 'setUp', 'setUp', 'setUp', 'setUp', 'setUp']
True
```

## 2.7 Convenience shortcut: `paramseq.context()`

You can use the method *`paramseq.context()`* to apply the given context properties to *all* parameter items the *`paramseq`* instance aggregates:

```
>>> @contextlib.contextmanager
... def silly_cm():
...     yield 42
...
>>> @expand
... class TestSaveLoad(unittest.TestCase):
...
...     params_with_contexts = paramseq(
...         param(save='', load=''),
...         param(save='abc', load='abc'),
...     ).context(NamedTemporaryFile, 'w+t').context(silly_cm)
```

(continues on next page)

(continued from previous page)

```
...
...     @foreach(params_with_contexts)
...     def test_save_load(self, save, load):
...         file, silly_cm_target = current.context_targets
...         assert silly_cm_target == 42
...         file.write(save)
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load)
...
>>> run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test_save_load__<load='',save=''> ... ok
test_save_load__<load='abc',save='abc'> ... ok
...Ran 2 tests...
OK
```

It should be noted that *paramseq.context()* as well as *param.context()* and *param.label()* methods create new objects (respectively *paramseq* or *param* instances), *without* modifying the existing ones.

```
>>> pseq1 = paramseq(1, 2, 3)
>>> pseq2 = pseq1.context(open, '/etc/hostname', 'rb')
>>> isinstance(pseq1, paramseq) and isinstance(pseq2, paramseq)
True
>>> pseq1 is not pseq2
True
```

```
>>> p1 = param(1, 2, c=3)
>>> p2 = p1.context(open, '/etc/hostname', 'rb')
>>> p3 = p2.label('one with label')
>>> isinstance(p1, param) and isinstance(p2, param) and isinstance(p3, param)
True
>>> p1 is not p2
True
>>> p2 is not p3
True
>>> p3 is not p1
True
```

Generally, instances of these types (*param* and *paramseq*) should be considered immutable.

## 2.8 Contexts cannot suppress exceptions unless you enable that explicitly

The Python *context manager* protocol provides a way to suppress an exception occuring in the code enclosed by a context: the exception is *suppresed* (*not* propagated) if the context manager's `__exit__()` method returns a *true* value (such as `True`).

It does **not** apply to context managers declared with *param.context()* or *paramseq.context()*: if `__exit__()` of such a context manager returns a *true* value, it is ignored and the exception (if any) is propagated anyway. The rationale of this behavior is that silencing exceptions is generally not a good idea when dealing with testing (it could easily make your tests leaky and useless).

However, if you **really** need to allow your context manager to suppress exceptions, pass the keyword argument `_enable_exc_suppress_=True` (note the single underscores at the beginning and the end of its name) to the

*param.context()* or *paramseq.context()* method (and, of course, make the `__exit__()` context manager's method return a *true* value).

Here we pass `_enable_exc_suppress_=True` to *param.context()*:

```
>>> class SillySuppressingCM(object):
...     def __enter__(self): return self
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         if exc_type is not None:
...             debug.append('suppressing {}'.format(exc_type.__name__))
...         return True  # suppress any exception
...
>>> @expand
... class SillyExcTest(unittest.TestCase):
...
...     @foreach(
...         param(test_error=AssertionError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True),
...         param(test_error=KeyError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True),
...     )
...     def test_it(self, test_error):
...         debug.append('raising {}'.format(test_error.__name__))
...         raise test_error('ha!')
...
>>> debug = []
>>> run_tests(SillyExcTest)  # doctest: +ELLIPSIS
test_it__... ok
test_it__... ok
...Ran 2 tests...
OK
>>> debug == [
...     'raising AssertionError',
...     'suppressing AssertionError',
...     'raising KeyError',
...     'suppressing KeyError',
... ]
True
```

Here we pass `_enable_exc_suppress_=True` to *paramseq.context()*:

```
>>> my_params = paramseq(
...     AssertionError,
...     KeyError,
... ).context(SillySuppressingCM, _enable_exc_suppress_=True)
>>> @expand
... class SecondSillyExcTest(unittest.TestCase):
...
...     @foreach(my_params)
...     def test_it(self, test_error):
...         debug.append('raising {}'.format(test_error.__name__))
...         raise test_error('ha!')
...
>>> debug = []
>>> run_tests(SecondSillyExcTest)  # doctest: +ELLIPSIS
test_it__... ok
test_it__... ok
...Ran 2 tests...
```

(continues on next page)

(continued from previous page)

```
OK
>>> debug == [
...     'raising AssertionError',
...     'suppressing AssertionError',
...     'raising KeyError',
...     'suppressing KeyError',
... ]
True
```

Yet another example:

```
>>> class ErrorCM:
...     def __init__(self, error): self.error = error
...     def __enter__(self): return self
...     def __exit__(self, exc_type, exc_val, exc_tb):
...         if exc_type is not None:
...             debug.append('replacing {} with {}'.format(
...                 exc_type.__name__, self.error.__name__))
...         else:
...             debug.append('raising {}'.format(self.error.__name__))
...         raise self.error('argh!')
...
>>> @expand
... class AnotherSillyExcTest(unittest.TestCase):
...
...     @foreach([
...         param(test_error=AssertionError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True),
...         param(test_error=KeyError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=RuntimeError),
...         param(test_error=None)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True),
...         param(test_error=None)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=IndexError, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=TypeError, _enable_exc_suppress_=True),
...         param(test_error=OSError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=ValueError)
...             .context(ErrorCM, error=ZeroDivisionError),
...         param(test_error=UnboundLocalError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=ValueError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=ZeroDivisionError)
...             .context(SillySuppressingCM, _enable_exc_suppress_=True),
...     ])
...     def test_it(self, test_error):
...         if test_error is None:
...             debug.append('no error')
...         else:
...             debug.append('raising {}'.format(test_error.__name__))
...             raise test_error('ha!')
...
>>> debug = []
>>> run_tests(AnotherSillyExcTest)  # doctest: +ELLIPSIS
```

(continues on next page)

(continued from previous page)

```
test_it__... ok
test_it__... ok
test_it__... ok
test_it__... ok
test_it__... ok
test_it__... ok
...Ran 6 tests...
OK
>>> debug == [
...     'raising AssertionError',
...     'suppressing AssertionError',
...
...     'raising KeyError',
...     'replacing KeyError with RuntimeError',
...     'suppressing RuntimeError',
...
...     'raising OSError',
...     'replacing OSError with ZeroDivisionError',
...     'replacing ZeroDivisionError with ValueError',
...     'suppressing ValueError',
...
...     'raising UnboundLocalError',
...     'suppressing UnboundLocalError',
...     'raising ZeroDivisionError',
...     'suppressing ZeroDivisionError',
...     'raising ValueError',
...     'suppressing ValueError',
...
...     'no error',
...
...     'no error',
...     'raising TypeError',
...     'replacing TypeError with IndexError',
...     'suppressing IndexError',
... ]
True
```

Normally – without `_enable_exc_suppress_=True` – exceptions *are* propagated even when `__exit__()` returns a *true* value:

```
>>> @expand
... class AnotherSillyExcTest2(unittest.TestCase):
...
...     @foreach([
...         param(test_error=AssertionError)
...             .context(SillySuppressingCM),
...         param(test_error=KeyError)
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=RuntimeError),
...         param(test_error=None)
...             .context(SillySuppressingCM),
...         param(test_error=None)
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=IndexError)
...             .context(ErrorCM, error=TypeError),
...         param(test_error=OSError)
```

(continues on next page)

(continued from previous page)

```
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=ValueError)
...             .context(ErrorCM, error=ZeroDivisionError),
...         param(test_error=UnboundLocalError)
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=ValueError)
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=ZeroDivisionError)
...             .context(SillySuppressingCM),
...     ])
...     def test_it(self, test_error):
...         if test_error is None:
...             debug.append('no error')
...         else:
...             debug.append('raising {}'.format(test_error.__name__))
...             raise test_error('ha!')
...
>>> debug = []
>>> run_tests(AnotherSillyExcTest2)  # doctest: +ELLIPSIS
test_it__... FAIL
test_it__... ERROR
test_it__... ERROR
test_it__... ERROR
test_it__... ok
test_it__... ERROR
...Ran 6 tests...
FAILED (failures=1, errors=4)
>>> debug == [
...     # Note that the following 'suppressing ...' messages are lies
...     # because no errors are effectively suppressed here.
...
...     'raising AssertionError',
...     'suppressing AssertionError',
...
...     'raising KeyError',
...     'replacing KeyError with RuntimeError',
...     'suppressing RuntimeError',
...
...     'raising OSError',
...     'replacing OSError with ZeroDivisionError',
...     'replacing ZeroDivisionError with ValueError',
...     'suppressing ValueError',
...
...     'raising UnboundLocalError',
...     'suppressing UnboundLocalError',
...     'replacing UnboundLocalError with ZeroDivisionError',
...     'suppressing ZeroDivisionError',
...     'replacing ZeroDivisionError with ValueError',
...     'suppressing ValueError',
...
...     'no error',
...
...     'no error',
...     'raising TypeError',
...     'replacing TypeError with IndexError',
...     'suppressing IndexError',
... ]
```

(continues on next page)

(continued from previous page)

```
True
```

Note that `_enable_exc_suppress_=True` changes nothing when context manager's `__exit__()` returns a *false* value:

```
>>> @expand
... class AnotherSillyExcTest3(unittest.TestCase):
...
...     @foreach([
...         param(test_error=AssertionError)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True),
...         param(test_error=KeyError)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True)
...             .context(ErrorCM, error=RuntimeError),
...         param(test_error=None)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True),
...         param(test_error=None)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True)
...             .context(ErrorCM, error=IndexError, _enable_exc_suppress_=True)
...             .context(ErrorCM, error=TypeError, _enable_exc_suppress_=True),
...         param(test_error=OSError)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True)
...             .context(ErrorCM, error=ValueError)
...             .context(ErrorCM, error=ZeroDivisionError),
...         param(test_error=UnboundLocalError)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True)
...             .context(ErrorCM, error=ValueError)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True)
...             .context(ErrorCM, error=ZeroDivisionError)
...             .context(NamedTemporaryFile, 'w+t', _enable_exc_suppress_=True),
...     ])
...     def test_it(self, test_error):
...         if test_error is None:
...             debug.append('no error')
...         else:
...             debug.append('raising {}'.format(test_error.__name__))
...             raise test_error('ha!')
...
>>> debug = []
>>> run_tests(AnotherSillyExcTest3)  # doctest: +ELLIPSIS
test_it__... FAIL
test_it__... ERROR
test_it__... ERROR
test_it__... ERROR
test_it__... ok
test_it__... ERROR
...Ran 6 tests...
FAILED (failures=1, errors=4)
>>> debug == [
...     'raising AssertionError',
...
...     'raising KeyError',
...     'replacing KeyError with RuntimeError',
...
...     'raising OSError',
...     'replacing OSError with ZeroDivisionError',
```

(continues on next page)

(continued from previous page)

```
...     'replacing ZeroDivisionError with ValueError',
...
...     'raising UnboundLocalError',
...     'replacing UnboundLocalError with ZeroDivisionError',
...     'replacing ZeroDivisionError with ValueError',
...
...     'no error',
...
...     'no error',
...     'raising TypeError',
...     'replacing TypeError with IndexError',
... ]
True
```

## 2.9 Context order

As you can see in earlier examples, a test method call can be surrouned by multiple context managers – they form a hierarchy, i.e., are nested one in another.

```
>>> @expand
... class DummyTest(unittest.TestCase):
...     @foreach([
...         param(42).context(ErrDebugCM, tag='outer')
...                  .context(ErrDebugCM, tag='mid-outer')
...                  .context(ErrDebugCM, tag='mid-inner')
...                  .context(ErrDebugCM, tag='inner'),
...     ])
...     def test(self, n):
...         debug.append('test  # context_targets={!r}'
...                      .format(current.context_targets))
...
>>> debug = []  # see earlier definition of ErrDebugCM()...
>>> run_tests(DummyTest)  # doctest: +ELLIPSIS
test... ok
...Ran 1 test...
OK
>>> debug == [
...     "init:outer",
...     "enter:outer",
...     "init:mid-outer",
...     "enter:mid-outer",
...     "init:mid-inner",
...     "enter:mid-inner",
...     "init:inner",
...     "enter:inner",
...     "test  # context_targets=['outer', 'mid-outer', 'mid-inner', 'inner']",
...     "exit:inner",
...     "exit:mid-inner",
...     "exit:mid-outer",
...     "exit:outer",
... ]
True
```

As you can see, the rule seems to be simple: the *higher* (or more to the left) a context is declared, the more *outside* it is. The *lower* (or more to the right) – the more *inside*. (Let's name this rule: "higher=outer, lower=inner".)

Unfortunatelly, due to a design mistake made in early stages of development of *unittest_expander*, this rule is not kept when we stack up two or more *foreach()* decorators (to *obtain the Cartesian product* of given parameter collections):

**Warning:** Here we describe a **deprecated** behavior... But, please, read on!

```
>>> @expand
... class DummyTest(unittest.TestCase):
...     @foreach([
...         param(1).context(ErrDebugCM, tag='inner'),  # <- we'd prefer 'outer' here.
↪..
...     ])
...     @foreach([
...         param(2).context(ErrDebugCM, tag='mid-outer')
...                 .context(ErrDebugCM, tag='mid-inner'),
...     ])
...     @foreach([
...         param(3).context(ErrDebugCM, tag='outer'),  # <- we'd prefer 'inner' here.
↪..
...     ])
...     def test(self, *args):
...         debug.append('test  # context_targets={!r}'
...                      .format(current.context_targets))
...
>>> debug = []
>>> run_tests(DummyTest)  # doctest: +ELLIPSIS
test... ok
...Ran 1 test...
OK
>>> debug == [
...     "init:outer",
...     "enter:outer",
...     "init:mid-outer",
...     "enter:mid-outer",
...     "init:mid-inner",
...     "enter:mid-inner",
...     "init:inner",
...     "enter:inner",
...     "test  # context_targets=['outer', 'mid-outer', 'mid-inner', 'inner']",
...     "exit:inner",
...     "exit:mid-inner",
...     "exit:mid-outer",
...     "exit:outer",
... ]
True
```

What a mess!

We can, however, make the context ordering be consistently compliant with the aforementioned "higher=outer, lower=inner" rule by setting the *expand.legacy_context_ordering* global swich to `False`:

```
>>> expand.legacy_context_ordering = False
>>> @expand
... class DummyTest(unittest.TestCase):
...     @foreach([
...         param(1).context(ErrDebugCM, tag='outer'),
```

(continues on next page)

(continued from previous page)

```
...     ])
...     @foreach([
...         param(2).context(ErrDebugCM, tag='mid-outer')
...                 .context(ErrDebugCM, tag='mid-inner'),
...     ])
...     @foreach([
...         param(3).context(ErrDebugCM, tag='inner'),
...     ])
...     def test(self, *args):
...         debug.append('test  # context_targets={!r}'
...                      .format(current.context_targets))
...
>>> debug = []
>>> run_tests(DummyTest)  # doctest: +ELLIPSIS
test... ok
...Ran 1 test...
OK
>>> debug == [
...     "init:outer",
...     "enter:outer",
...     "init:mid-outer",
...     "enter:mid-outer",
...     "init:mid-inner",
...     "enter:mid-inner",
...     "init:inner",
...     "enter:inner",
...     "test  # context_targets=['outer', 'mid-outer', 'mid-inner', 'inner']",
...     "exit:inner",
...     "exit:mid-inner",
...     "exit:mid-outer",
...     "exit:outer",
... ]
True
```

Now the behavior is consistent and always compliant with the "higher=outer, lower=inner" rule.

And, also, is forward compatible!

**Warning:** In the current version of the library, the legacy (messy) behavior described earlier is enabled when the global option `expand.legacy_context_ordering` is set to `True`. Note that – for backward compatibility reasons – it *is* by default (however, in such a case, a deprecation warning is emitted when it is detected that two or more `foreach()` decorators that bring contexts are stacked up...). As stated above, you can swich to the new behavior (consistently providing the "higher=outer, lower=inner" context ordering) by setting `expand.legacy_context_ordering` to `False` (then the deprecation warning will not be emitted).

Note that the new behavior will become the only one in the *0.6.0* version of *unittest_expander*; then, the only legal value of `expand.legacy_context_ordering` will be `False`.

**What to do now?** To ensure your code is forward compatible, switch to the new behavior by setting `expand.legacy_context_ordering` to `False` and appropriately adjusting your test code (if necessary).

## 2.10 Access the current parametrized test's metadata via `current`

XXX TODO

## 2.11 Deprecated feature: accepting `label` and `context_targets` as keyword arguments

**Warning:** Here we describe a **deprecated** feature: automatic passing `label` and/or `context_targets` to test methods if the *`expand()`* decorator's machinery detected that the method is able to accept `label` and/or `context_targets` as keyword argument(s).

In the current version of the library, this feature is enabled when the global option *`expand.legacy_signature_introspection`* is set to `True`. Note that – for backward compatibility reasons – it *is* by default (however, a deprecation warning is emitted when it is detected that your code really makes use of the feature). The feature can be disabled by setting *`expand.legacy_signature_introspection`* to `False` (then the deprecation warning will not be emitted).

Note that the feature will be **removed** from *unittest_expander* in the version *0.6.0*; then, the only legal value of *`expand.legacy_signature_introspection`* will be `False`.

**What to do now?** To ensure your code is forward compatible, *make use* of *`current.label`* and *`current.context_targets`* whenever you need that information in your test method definitions (instead of having your test methods accepting `label` and/or `context_targets` as keyword arguments), and switch *`expand.legacy_signature_introspection`* to `False` (and, obviously, make sure your tests work correctly with that).

If the *`expand.legacy_signature_introspection`* global option is `True` and a test method is able to accept the `label` keyword argument, the appropriate label (either auto-generated from parameter values or explicitly specified with *`param.label()`*) will be passed to the method as that argument:

```
>>> expand.legacy_signature_introspection = True
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         param(sys.maxsize, expected=False).label('sys.maxsize'),
...         param(-sys.maxsize, expected=False).label('-sys.maxsize'),
...     )
...     def test(self, n, expected, label):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert label in ('sys.maxsize', '-sys.maxsize')
...         sys.stdout.write(' [DEBUG: {!r}] '.format(label))
...         sys.stdout.flush()
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test__<-sys.maxsize> ... [DEBUG: '-sys.maxsize'] ok
test__<sys.maxsize> ... [DEBUG: 'sys.maxsize'] ok
...Ran 2 tests...
OK
```

Similarly, if the aforementioned option is `True` and a test method is able to accept the `context_targets` keyword argument then a list of context manager as-targets* (i.e., objects returned by context managers' `__enter__()`) will be passed to the method as that argument:

```
>>> @expand
... class TestSaveLoad(unittest.TestCase):
```

(continues on next page)

(continued from previous page)

```
...
...     params_with_contexts = [
...         (
...             param(save='', load='', expected_tag='FOO')
...               .context(NamedTemporaryFile, 'w+t')
...               .context(debug_cm, tag='FOO')
...         ),
...         (
...             param(save='abc', load='abc', expected_tag='BAR')
...               .context(NamedTemporaryFile, 'w+t')
...               .context(debug_cm, tag='BAR')
...         ),
...     ]
...
...     @foreach(params_with_contexts)
...     def test_save_load(self, save, load, expected_tag, context_targets):
...         file, tag = context_targets
...         assert tag == expected_tag
...         file.write(save)
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load)
...         debug.append('test')
...
>>> # Note: the following change of the value of the attribute
>>> # `expand.legacy_signature_introspection` does not affect
>>> # further uses of the test class defined above because the
>>> # @expand decorator has already been applied to that class.
>>> expand.legacy_signature_introspection = False
>>> debug = []  # see earlier definition of debug_cm()...
>>> run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test_save_load__<expected_tag='BAR',load='abc',save='abc'> ... ok
test_save_load__<expected_tag='FOO',load='',save=''> ... ok
...Ran 2 tests...
OK
>>> debug == [
...     'enter:BAR', 'test', 'exit:BAR',
...     'enter:FOO', 'test', 'exit:FOO',
... ]
True
```

## 2.12 `Substitute` objects

One could ask: "What does the *`expand()`* decorator do with the original methods decorated with *`foreach()`*?"

```
>>> @expand
... class DummyTest(unittest.TestCase):
...
...     @foreach(1, 2)
...     def test_it(self, x):
...         pass
...
...     attr = [42]
...     test_it.attr = [43, 44]
```

They cannot be left where they are because, without parametrization, they are not valid tests (but rather kind of test templates). For this reason, they are always replaced (by the *expand()*'s machinery) with *Substitute* instances:

```
>>> test_it = DummyTest.test_it
>>> test_it                             # doctest: +ELLIPSIS
<...Substitute object at 0x...>
>>> test_it.actual_object               # doctest: +ELLIPSIS
<...test_it...>
>>> test_it.attr
[43, 44]
>>> test_it.attr is test_it.actual_object.attr
True
>>> (set(dir(test_it.actual_object)) - {'__call__'}
...  ).issubset(dir(test_it))
True
```

As you see, such a *Substitute* instance is kind of a non-callable proxy to the original method (preventing it from being included by test loaders, but still keeping it available for introspection, etc.).

## 2.13 Custom method name formatting

If you don't like how parametrized test method names are generated – you can customize that globally by:

- setting *expand.global_name_pattern* to a `format()`-able pattern, making use of zero or more of the following replacement fields:
  - `{base_name}` – the name of the original test method,
  - `{base_obj}` – the original test method (technically: function) object,
  - `{label}` – the test label (automatically generated or explicitly specified with *param.label()*),
  - `{count}` – the consecutive number (within a single application of *expand()*) of the generated parametrized test method;

  (in future versions of *unittest_expander* more replacement fields may be made available)

and/or

- setting *expand.global_name_formatter* to an instance of a custom subclass of the `string.Formatter` class from the Python standard library (or to any object whose `format()` method acts similarily to `string.Formatter.format()`).

For example:

```
>>> expand.global_name_pattern = '{base_name}__parametrized_{count:04}'
>>>
>>> params_with_contexts = paramseq(
...     param(save='', load=''),
...     param(save='abc', load='abc'),
... ).context(NamedTemporaryFile, 'w+t')
>>>
>>> @expand
... class TestSaveLoad(unittest.TestCase):
...     @foreach(params_with_contexts)
...     @foreach(param(suffix=' '), param(suffix='XX'))
...     def test_save_load(self, save, load, suffix):
...         file = current.context_targets[0]
```

(continues on next page)

(continued from previous page)

```
...         file.write(save + suffix)
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load + suffix)
...
>>> run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test_save_load__parametrized_0001 ... ok
test_save_load__parametrized_0002 ... ok
test_save_load__parametrized_0003 ... ok
test_save_load__parametrized_0004 ... ok
...Ran 4 tests...
OK
```

. . . or, let's say:

```
>>> import string
>>> class ExtremelySillyFormatter(string.Formatter):
...     def format(self, format_string, *args, **kwargs):
...         count = kwargs['count']
...         label = kwargs['label']
...         if 'abc' in label:
...             result = 'test__{}__"!!! {} !!!"'.format(count, label)
...         else:
...             result = super(ExtremelySillyFormatter,
...                            self).format(format_string, *args, **kwargs)
...         if count % 3 == 1:
...             result = result.replace('_', '-')
...         return result
...
>>> expand.global_name_formatter = ExtremelySillyFormatter()
>>>
>>> @expand
... class TestSaveLoad(unittest.TestCase):
...     @foreach(params_with_contexts)
...     @foreach(param(suffix=' '), param(suffix='XX'))
...     def test_save_load(self, save, load, suffix):
...         file = current.context_targets[0]
...         file.write(save + suffix)
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load + suffix)
...
>>> run_tests(TestSaveLoad)  # doctest: +ELLIPSIS
test--4--"!!! suffix='XX', load='abc',save='abc' !!!" ... ok
test-save-load--parametrized-0001 ... ok
test__2__"!!! suffix=' ', load='abc',save='abc' !!!" ... ok
test_save_load__parametrized_0003 ... ok
...Ran 4 tests...
OK
```

Set those attributes to `None` to restore the default behavior:

```
>>> expand.global_name_pattern = None
>>> expand.global_name_formatter = None
```

## 2.14 Name clashes avoided automatically

*expand()* does its best to avoid name conflicts: when it detects that a newly generated name could clash with an existing one (whether the latter was generated recently – as an effect of the ongoing application of *expand()* – or might have already existed), it adds a suffix to the newly generated name to avoid the clash.

As shown in the following examples, the machinery of *expand()* is really careful when it comes to avoiding name conflicts:

```
>>> def setting_attrs(attr_dict):
...     def deco(cls):
...         for k, v in attr_dict.items():
...             setattr(cls, k, v)
...         return cls
...     return deco
...
>>> @expand
... @setting_attrs({
...     'test_even__<4>': 'something',
...     'test_even__<4>__2': None,
... })
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         0,
...         4,
...         0,    # <- repeated parameter value
...         0,    # <- repeated parameter value
...         -16,
...         0,    # <- repeated parameter value
...     )
...     def test_even(self, n):
...         self.assertTrue(is_even(n))
...
>>> Function = type(lambda: None)
>>> {name: type(obj)
...  for name, obj in vars(Test_is_even).items()
...  if not name.startswith('_')
... } == {
...     'test_even': Substitute,
...     'test_even__<-16>': Function,
...     'test_even__<0>': Function,
...     'test_even__<0>__2': Function,
...     'test_even__<0>__3': Function,
...     'test_even__<0>__4': Function,
...     'test_even__<4>': str,
...     'test_even__<4>__2': type(None),
...     'test_even__<4>__3': Function,
... }
True
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_even__<-16> ... ok
test_even__<0> ... ok
test_even__<0>__2 ... ok
test_even__<0>__3 ... ok
test_even__<0>__4 ... ok
test_even__<4>__3 ... ok
```

(continues on next page)

(continued from previous page)

```
...Ran 6 tests...
OK
>>> @expand
... @setting_attrs({
...     'test_even__<0>__6': False,
...     'test_even__<0>__7': object(),
... })
... class Test_is_even_2(Test_is_even):
...
...     @foreach(
...         0,
...         4,
...         0,    # <- repeated parameter value
...         0,    # <- repeated parameter value
...         -16,
...         0,    # <- repeated parameter value
...     )
...     def test_even(self, n):
...         self.assertTrue(is_even(n))
...
>>> {name: type(obj)
...  for name, obj in vars(Test_is_even_2).items()
...  if not name.startswith('_')
... } == {
...     'test_even': Substitute,
...     'test_even__<-16>__2': Function,
...     'test_even__<0>__10': Function,
...     'test_even__<0>__5': Function,
...     'test_even__<0>__6': bool,
...     'test_even__<0>__7': object,
...     'test_even__<0>__8': Function,
...     'test_even__<0>__9': Function,
...     'test_even__<4>__4': Function,
... }
True
>>> run_tests(Test_is_even_2)  # doctest: +ELLIPSIS
test_even__<-16> ... ok
test_even__<-16>__2 ... ok
test_even__<0> ... ok
test_even__<0>__10 ... ok
test_even__<0>__2 ... ok
test_even__<0>__3 ... ok
test_even__<0>__4 ... ok
test_even__<0>__5 ... ok
test_even__<0>__8 ... ok
test_even__<0>__9 ... ok
test_even__<4>__3 ... ok
test_even__<4>__4 ... ok
...Ran 12 tests...
OK
```

## 2.15 Questions and answers about various details...

### 2.15.1 "Can I omit `expand()` and then apply it to subclasses?"

Yes, you can. Please consider the following example:

```
>>> debug = []
>>> parameters = paramseq(
...     7, 8, 9,
... ).context(debug_cm, tag='M')  # see earlier definition of debug_cm()...
>>>
>>> class MyTestMixIn(object):
...
...     @foreach(parameters)
...     def test(self, x):
...         debug.append((x, self.n))
...
>>> @expand
... class TestActual(MyTestMixIn, unittest.TestCase):
...     n = 42
...
>>> @expand
... class TestYetAnother(MyTestMixIn, unittest.TestCase):
...     n = 12345
...
>>> run_tests(TestActual, TestYetAnother)  # doctest: +ELLIPSIS
test__<7> (...TestActual...) ... ok
test__<8> (...TestActual...) ... ok
test__<9> (...TestActual...) ... ok
test__<7> (...TestYetAnother...) ... ok
test__<8> (...TestYetAnother...) ... ok
test__<9> (...TestYetAnother...) ... ok
...Ran 6 tests...
OK
>>> inspect.isfunction(vars(MyTestMixIn)['test'])      # (not touched by @expand)
True
>>> type(vars(TestActual)['test']) is Substitute       # (replaced by @expand)
True
>>> type(vars(TestYetAnother)['test']) is Substitute   # (replaced by @expand)
True
>>> debug == [
...     'enter:M', (7, 42), 'exit:M',
...     'enter:M', (8, 42), 'exit:M',
...     'enter:M', (9, 42), 'exit:M',
...     'enter:M', (7, 12345), 'exit:M',
...     'enter:M', (8, 12345), 'exit:M',
...     'enter:M', (9, 12345), 'exit:M',
... ]
True
```

Note that, most probably, you should name such mix-in or "test template" base classes in a way that prevents the test loader you use from including them; for the same reason, typically, it is better to avoid making them subclasses of `unittest.TestCase`.

### 2.15.2 "Can I `expand()` a subclass of an already `expand()`-ed class?"

Yes, you can (in some past versions of *unittest_expander* it was broken, but now it works perfectly):

```
>>> debug = []
>>> parameters = paramseq(
...     1, 2, 3,
... ).context(debug_cm)  # see earlier definition of debug_cm()...
>>>
>>> @expand
... class Test(unittest.TestCase):
...
...     @foreach(parameters)
...     def test(self, n):
...         debug.append(n)
...
>>> @expand
... class TestSubclass(Test):
...
...     @foreach(parameters)
...     def test_another(self, n):
...         debug.append(n)
...
>>> run_tests(TestSubclass)  # doctest: +ELLIPSIS
test__<1> (...TestSubclass...) ... ok
test__<2> (...TestSubclass...) ... ok
test__<3> (...TestSubclass...) ... ok
test_another__<1> (...TestSubclass...) ... ok
test_another__<2> (...TestSubclass...) ... ok
test_another__<3> (...TestSubclass...) ... ok
...Ran 6 tests...
OK
>>> type(TestSubclass.test) is type(Test.test) is Substitute
True
>>> type(TestSubclass.test_another) is Substitute
True
```

### 2.15.3 "Do my test classes need to inherit from `unittest.TestCase`?"

No, it doesn't matter from the point of view of the *unittest_expander* machinery.

```
>>> debug = []
>>> parameters = paramseq(
...     1, 2, 3,
... ).context(debug_cm)  # see earlier definition of debug_cm()...
>>>
>>> @expand
... class Test(object):  # not a unittest.TestCase subclass
...
...     @foreach(parameters)
...     def test(self, n):
...         debug.append(n)
...
>>> # confirming that unittest_expander machinery acted properly:
>>> instance = Test()
>>> type(instance.test) is Substitute
```

(continues on next page)

(continued from previous page)

```
True
>>> t1 = getattr(instance, 'test__<1>')
>>> t2 = getattr(instance, 'test__<2>')
>>> t3 = getattr(instance, 'test__<3>')
>>> t1()
>>> t2()
>>> t3()
>>> debug == [
...     'enter', 1, 'exit',
...     'enter', 2, 'exit',
...     'enter', 3, 'exit',
... ]
True
```

### 2.15.4 "What happens if I apply `expand()` when there's no `foreach()`?"

Just nothing – the test works as if *`expand()`* was not applied at all:

```
>>> @expand
... class TestIt(unittest.TestCase):
...
...     def test(self):
...         sys.stdout.write(' [DEBUG: OK] ')
...         sys.stdout.flush()
...
>>> run_tests(TestIt)  # doctest: +ELLIPSIS
test ... [DEBUG: OK] ok
...Ran 1 test...
OK
```

### 2.15.5 "To what objects can `foreach()` be applied?"

The *`foreach()`* decorator is designed to be applied *only* to regular test methods (i.e., instance methods, *not* static or class methods) – that is, technicaly, to *functions* being attributes of test (or test base/mix-in) classes.

If you apply *`foreach()`* to anything but a function object, a `TypeError` is raised:

```
>>> @foreach(1, 2, 3)
... class What:
...     '''I am not a function'''                    # doctest: +ELLIPSIS
...
Traceback (most recent call last):
  ...
TypeError: ...is not a function...
```

```
>>> @foreach(1, 2, 3)
... class WhatTest(unittest.TestCase):
...     '''I am not a function'''                    # doctest: +ELLIPSIS
...
Traceback (most recent call last):
  ...
TypeError: ...is not a function...
```

```
>>> @expand
... class ErroneousTest(unittest.TestCase):
...     @foreach(parameters)
...     @classmethod
...     def test_erroneous(cls, n):
...         '''I am not a function'''               # doctest: +ELLIPSIS
...
Traceback (most recent call last):
  ...
TypeError: ...is not a function...
```

CHAPTER 3

# Module Contents

The public interface of the *unittest_expander* module consists of the following functions, classes, objects and constants.

(See: *Narrative Documentation* – for a much richer description of most of them, including a lot of usage examples. . . )

## 3.1 The `expand()` class decorator

**`@expand`**

Apply this decorator to a *test class* to generate actual *parametrized test methods*, i.e., to "expand" parameter collections which have been bound (by applying *foreach()*) to original *test methods*.

—

The public interface of *expand()* includes also the attributes described below.

Two of them make it possible to *customize how names of parametrized test methods are generated*:

`expand.`**`global_name_pattern`**

**Type** `str` or `None`

**Value** `None` (use the default pattern)

`expand.`**`global_name_formatter`**

**Type** `string.Formatter`-like object or `None`

**Value** `None` (use the default formatter)

Other two allow – respectively – to *switch the context ordering style*, and to decide whether *a certain deprecated, signature-introspection-based feature* shall be enabled:

`expand.`**`legacy_context_ordering`**

**Type** `bool`

**Value** `True`

expand.**legacy_signature_introspection**

**Type** bool

**Value** True

**Warning:** For each of the last two attributes, True (the default value) dictates a deprecated (legacy) behavior, whereas only False is compatible with future versions of *unittest_expander*.

## 3.2 The foreach() method decorator

@**foreach**(*param_collection*)

*or*

@**foreach**(**param_collection_items*, ***param_collection_labeled_items*)

Call this function, specifying parameter collections to be bound to a *test method*, and then apply the resultant decorator to that method (only then it will be possible – by applying *expand()* to the *test class* owning the method – to generate actual *parametrized test methods*).

To learn more about what needs to be passed to *foreach()*, see the description (below) of the *paramseq*'s constructor (note that the call signatures of *foreach()* and that constructor are the same).

## 3.3 The paramseq class

**class paramseq**(*param_collection*)

*param_collection* must be a parameter collection – that is, one of:

- a *paramseq* instance,
- a *sequence* **not being** a tuple/str/unicode/bytes/bytearray (in other words, such an object for whom isinstance(obj, collections.abc.Sequence) and not isinstance(obj, (tuple, str, bytes, bytearray)) returns True in Python 3) – for example, a list,
- a *mapping* (i.e., such an object that isinstance(obj, collections.abc.Mapping) returns True in Python 3) – for example, a dict,
- a *set* (i.e., such an object that isinstance(obj, collections.abc.Set) returns True in Python 3) – for example, a set or frozenset,
- a *callable* (i.e., such an object that callable(obj) returns True) which is supposed to:
  - accept one positional argument (the *test class*) or no arguments at all,
  - return an *iterable* object (i.e., an object that could be used as a for loop's subject, able to yield consecutive items)

  – for example, a generator function.

Each *item* of a parameter collection is supposed to be:

- a *param* instance,
- a tuple (to be converted automatically to a *param* which will contain parameter values being the items of that tuple),
- any other object (to be converted automatically to a *param* which will contain only one parameter value: that object).

*or*

**class paramseq**(**param_collection_items*, ***param_collection_labeled_items*)

The total number of given arguments (positional and/or keyword ones) must be greater than 1. Each argument will be treated as a parameter collection's *item* (see above); for each keyword argument (if any), its name will be used to `label()` the *item* it refers to.

—

A `paramseq` instance is the canonical form of a parameter collection.

Its public interface includes the following methods:

**__add__**(*param_collection*)

Returns a new `paramseq` instance – being a result of concatenation of the `paramseq` instance we operate on and the given *param_collection* (see the above description of the `paramseq` constructor's argument *param_collection*...).

**__radd__**(*param_collection*)

Returns a new `paramseq` instance – being a result of concatenation of the given *param_collection* (see the above description of the `paramseq` constructor's argument *param_collection*...) and the `paramseq` instance we operate on.

**context**(*context_manager_factory*, **its_args*, ***its_kwargs*, *_enable_exc_suppress_=False*)

Returns a new `paramseq` instance containing clones of the items of the instance we operate on – each cloned with a `param.context()` call (see below...) to which all given arguments are passed.

## 3.4 The `param` class

**class param**(**args*, ***kwargs*)

*args* and *kwargs* specify actual (positional and keyword) arguments to be passed to test method call(s).

—

A `param` instance is the canonical form of a parameter collection's *item*. It represents *a single combination of test parameter values*.

Its public interface includes the following methods:

**context**(*context_manager_factory*, **its_args*, ***its_kwargs*, *_enable_exc_suppress_=False*)

Returns a new `param` instance being a clone of the the instance we operate on, with the specified context manager factory (and its arguments) attached.

By default, the possibility to suppress exceptions by returning a *true* value from context manager's `__exit__()` is *disabled* (exceptions are propagated even if `__exit__()` returns `True`); to enable this possibility you need to set the *_enable_exc_suppress_* keyword argument to `True`.

**label**(*text*)

Returns a new `param` instance being a clone of the instance we operate on, with the specified textual label attached.

## 3.5 The `current` special object

**current**

A special singleton object which, when used during execution of a parametrized test method, provides access (in a thread-local manner) to the following properties of the (currently executed) test:

`current.`**label**

**Type** str

**Value** the *test's label* (which was automatically generated or explicitly specified with *param.label()*...)

current.**context_targets**

**Type** Sequence

**Value** the *test contexts' as-targets* (i.e., objects returned by __enter__() of each of the context managers which were specified with *param.context()*...)

current.**all_args**

**Type** Sequence

**Value** all *positional arguments* obtained by the currently executed parametrized test method (in particular, including all positional arguments which were passed to the *param* constructor...)

current.**all_kwargs**

**Type** Mapping

**Value** all *keyword arguments* obtained by the currently executed parametrized test method (in particular, including all keyword arguments which were passed to the *param* constructor...)

current.**count**

**Type** int

**Value** the consecutive number (within a single application of *expand()*) of the generated parametrized test method

current.**base_name**

**Type** str

**Value** the name of the original (non-parametrized) test method

current.**base_obj**

**Type** function

**Value** the original (non-parametrized) test method itself

## 3.6 Non-essential constants and classes

**__version__**
The version of *unittest_expander* as a str being a **PEP 440**-compliant identifier.

**class Substitute**(*actual_object*)
A kind of attribute-access-proxying wrapper, *automatically applied* by the machinery of *expand()* to each test method previously decorated with *foreach()*.

The sole constructor argument (*actual_object*) is the object (typically, a test method) to be proxied.

Apart from exposing in a transparent way nearly all attributes of the proxied object, the public interface of a *Substitute* includes the following instance attribute:

**actual_object**
The proxied object itself (unwrapped).

**Note:** A `Substitute` instance is *never* callable – even though (typically) the proxied object is.

CHAPTER 4

# Changes

## 4.1 Unreleased (to be updated. . . )

- TBD. . .

## 4.2 0.4.4 (2023-03-21)

- Documentation: an important completion to the *Narrative Documentation* part (regarding the deprecation introduced in the previous version of *unittest_expander*) as well as several minor improvements and fixes.

## 4.3 0.4.3 (2023-03-21)

- **Deprecation notice:** using an instance of the Python 3 built-in type **bytes** or **bytearray** (or of any subclass thereof) as a *parameter collection* – passed as the *sole* argument to **foreach()** or **paramseq()**, or added (using +) to an existing **paramseq** object – is now deprecated (causing emission of a **DeprecationWarning**) and will become **illegal** in *unittest_expander 0.5.0*.
- A minor fix regarding CI and package metadata. . .
- Documentation and code comments: minor updates/cleanups.

## 4.4 0.4.2 (2023-03-18)

- A minor interface usability fix: from now on, the **expand()** decorator's attributes **global_name_pattern** and **global_name_formatter** are both initially set to **None** (previously, by default, they were not initialized at all, so trying to get any of them without first setting it caused an **AttributeError**).
- Documentation: several updates, improvements and minor fixes.

## 4.5 0.4.1 (2023-03-17)

- Added to the **unittest_expander** module the **__version__** constant.
- Improvements and additions related to tests, CI, generation of documentation, etc.; in particular: added a script that checks whether **unittest_expander.__version__** is equal to **version** in package metadata, and added invocation of that script to the *Install and Test* GitHub workflow.
- Documentation: improvements and minor fixes.

## 4.6 0.4.0 (2023-03-16)

- From now on, the following versions of Python *are officially supported:* **3.11**, **3.10**, **3.9**, **3.8**, **3.7**, **3.6** and **2.7** (still). This means, in particular, that the versions *3.5*, *3.4*, *3.3*, *3.2* and *2.6* are *no longer supported*.
- Now, if two (or more) parameter collections are combined to make the Cartesian product of them (as an effect of decorating a test with two or more **foreach(...)** invocations), and a conflict is detected between any *keyword arguments* passed earlier to the **param()** constructor to create the **param** instances that are being combined, the **expand()** decorator raises a **ValueError** (in older versions of *unittest_expander* no exception was raised; instead, a keyword argument being a component of one **param** was silently overwritten by the corresponding keyword argument being a component of the other **param**; that could lead to silent bugs in your tests...).
- When it comes to **param.context()** (and **paramseq.context()**), now the standard Python context manager's mechanism of suppressing exceptions (by making **__exit__()** return a *true* value) is, by default, consistently *disabled* (i.e. exceptions are *not* suppressed, as it could easily become a cause of silent test bugs; the previous behavior was inconsistent: exceptions could be suppressed in the case of applying **foreach()** decorators to test *methods*, but not in the case of applying them to test *classes*).

  If needed, the possibility of suppressing exceptions by **__exit__()** returning a *true* value can be explicitly *enabled* on a case-by-case basis by passing `_enable_exc_suppress_=True` to **param.context()** (or **paramseq.context()**).
- **Deprecation notice:** decorating test *classes* with **foreach()** – to generate new parametrized test *classes* – is now deprecated (causing emission of a **DeprecationWarning**); in future versions of *unittest_expander* it will first become **unsupported** (in *0.5.0*), and then, in some version, will (most probably) get a **new meaning**. The current shape of the feature is deemed broken by design (in particular, because of the lack of composability; that becomes apparent when class inheritance comes into play...).
- **Deprecation notice:** a change related to the deprecation described above is that now the **expand()**'s keyword argument `into` is also deprecated (its use causes emission of a **DeprecationWarning**) and will become **illegal** in *unittest_expander 0.5.0*.
- **Deprecation notice:** using a tuple (i.e., an instance of the built-in type **tuple** or of any subclass of it, e.g., a *named tuple*) as a *parameter collection* – passed as the *sole* argument to **foreach()** or **paramseq()**, or added (using +) to an existing **paramseq** object – is now deprecated (causing emission of a **DeprecationWarning**) and will become **illegal** in *unittest_expander 0.5.0*. Instead of a tuple, use a collection of another type (e.g., a **list**).

  Note: this deprecation concerns tuples used as *parameter collections*, *not* as *items* of parameter collections (tuples being such items, acting as simple substitutes of **param** objects, are – and will always be – perfectly OK).
- Two compatibility fixes:

  (1) now test methods with *keyword-only* arguments and/or *type annotations* are supported (previously an error was raised by **expand()** if such a method was decorated with **foreach()**); the background is that under Python

3.x, from now on, **inspect.getfullargspec()** (instead of its legacy predecessor, **inspect.getargspec()**) is used to inspect test method signatures;

(2) now standard *abstract base classes* of collections are imported from `collections.abc`; an import from `collections` is left only as a Python-2.7-dedicated fallback.

- Two bugfixes related to the **expand()** decorator:

  (1) now class/type objects and **Substitute** objects are ignored when scanning a test class for attributes (methods) that have **foreach()**-created marks;

  (2) **foreach()**-created marks are no longer retained on parametrized test methods generated by the **expand()**'s machinery.

  Thanks to those fixes, it is now possible to apply **expand()** to subclasses of an **expand()**-ed class – provided that **foreach()** has been applied only to test *methods* (not to test *classes*, which is a deprecated feature anyway – see the relevant *deprecation notice* above).

- A few bugfixes related to applying **foreach()** to test classes (which is a deprecated feature – see the relevant *deprecation notice* above):

  (1) a **foreach()**-decorated test class which does *not* inherit from **unittest.TestCase** is no longer required to provide its own implementations of **setUp()** and **tearDown()** (previously, if they were missing, an **AttributeError** were raised by the **setUp()** and **tearDown()** implementations provided by **expand()**-generated subclasses);

  (2) now the `context_targets` attribute of a test class instance is set also if there are no contexts (to an empty list – making it possible for a test code to refer to `self.context_targets` without fear of an **AttributeError**);

  (3) now a *context*'s **__exit__()** is never called without the corresponding **__enter__()** call being successful.

- A few really minor behavioral changes/improvements (in particular, now a *callable* parameter collection is required to satisfy the built-in **callable(...)** predicate, instead of the previously checked **isinstance(..., collections.Callable)** condition; that should not matter in practice).
- A bunch of package-setup-and-metadata-related additions, updates, fixes, improvements and removals (in particular, `pyproject.toml` and `setup.cfg` files have been added, and the `setup.py` file has been removed).
- Added `.gitignore` and `.editorconfig` files.
- A bunch of changes related to tests, CI, documentation, etc.: updates, fixes, improvements and additions (including addition of the *Install and Test* GitHub workflow).

**Many thanks** to:

- KOLANICH (@KOLANICH),
- Hugo van Kemenade (@hugovk),
- John Vandenberg (@jayvdb)

– for their invaluable contribution to this release!

## 4.7 0.3.1 (2014-08-19)

- Several tests/documentation-related fixes and improvements.

## 4.8 0.3.0 (2014-08-17)

- Improved signatures of the **foreach()** decorator and the **paramseq()** constructor: they take either exactly one positional argument which must be a test parameter collection (as previously: a sequence/mapping/set or a **paramseq** instance, or a callable returning an iterable...), or *any number of positional and/or keyword arguments* being test parameters (singular parameter values, tuples of parameter values or **param** instances...). For example, `@foreach([1, 42])` can now also be spelled as `@foreach(1, 42)`.
- Several tests/documentation-related updates, fixes and improvements.

## 4.9 0.2.1 (2014-08-12)

- Important setup/configuration fixes (repairing 0.2.0 regressions):
  - a setup-breaking bug in *setup.py* has been fixed;
  - a bug in the configuration of Sphinx (the tool used to generate the documentation) has been fixed.
- Some setup-related cleanups.

## 4.10 0.2.0 (2014-08-11)

- Now **unittest_expander** is a one-file module, not a directory-based package.
- Some documentation improvements and updates.
- Some library setup improvements and refactorings.

## 4.11 0.1.2 (2014-08-01)

- The signatures of the **foreach()** decorator and the **paramseq()** constructor have been unified.
- Tests/documentation-related updates and improvements.

## 4.12 0.1.1 (2014-07-29)

- Minor tests/documentation-related improvements.

## 4.13 0.1.0 (2014-07-29)

- Initial release.

CHAPTER 5

# License, Credits and Other Information

## 5.1 Copyright and License



## 5.2 Credits

The authors of *unittest_expander* are: Jan Kaliszewski (zuo) and the Contributors whose GitHub pull requests have been merged into the code repository.

Also, the project owes a lot to those who report bugs and propose/discuss enhancements.

There were also some inspirations; see the following section to learn about some of them...

## 5.3 Related Links (Historical Note)

Before/when creating first versions of *unittest_expander* I (Jan) checked out several other projects and resources related to unit test parametrization (aka *parameterization*) in Python.

Some of them were mature and actively maintained projects, others were just minor drafts; some depended (contrary to *unittest_expander*) on external libraries or testing frameworks, others did not; some had appealing, programmer-friendly interfaces, others felt more like low-level building blocks. . .

Anyway, here are the links:

- https://nose.readthedocs.org/en/latest/writing_tests.html#test-generators
- https://github.com/wolever/nose-parameterized
- https://github.com/msabramo/python_unittest_parameterized_test_case
- https://github.com/txels/ddt
- https://code.google.com/p/parameterized-testcase/
- *https://bitbucket.org/lothiraldan/unittest-templates/* (link no longer alive)
- https://launchpad.net/testscenarios
- https://eli.thegreenplace.net/2011/08/02/python-unit-testing-parametrized-test-cases/
- https://gist.github.com/mfazekas/1710455
- plus – of course! – certain brilliant features of *pytest*:
  - https://pytest.org/latest/parametrize.html
  - https://pytest.org/latest/fixture.html
- . . . as well as some interesting *nose2* plugins:
  - https://nose2.readthedocs.org/en/latest/plugins/generators.html
  - https://nose2.readthedocs.org/en/latest/params.html#nose2.tools.params

See also:

- https://github.com/python/cpython/issues/52145
- https://github.com/python/cpython/issues/56809
- https://github.com/python/cpython/commit/56517e5cb91c896024934a520d365d6e275eb1ad

CHAPTER 6

# Indices and tables

- genindex
- search

# Python Module Index

## u

# Index