unittest_expander Documentation

Release 0.4.4

Jan Kaliszewski (zuo) and others

Mar 21, 2023

Contents

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 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()`*,
- two helper classes: *`param`* and *`paramseq`*.

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 prevents subsequent tests from being run), less clear result messages (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 attach 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_case_classes):
...     suite = unittest.TestSuite(
...         unittest.TestLoader().loadTestsFromTestCase(cls)
...         for cls in test_case_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_is_even(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_is_even__<-1,False> ... ok
test_is_even__<-14,True> ... ok
test_is_even__<0,True> ... ok
test_is_even__<17,False> ... ok
test_is_even__<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_is_even(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_is_even__<-1,expected=False> ... ok
test_is_even__<-14,expected=True> ... ok
test_is_even__<0,expected=True> ... ok
test_is_even__<17,expected=False> ... ok
test_is_even__<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_is_even(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_is_even__<-sys.maxsize> ... ok
test_is_even__<sys.maxsize> ... ok
...Ran 2 tests...
OK
```

If a test method accepts the *label* keyword argument, the appropriate label (either auto-generated from parameter values or explicitly specified with `param.label()`) will be passed in as that argument:

```
>>> @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_is_even(self, n, expected, label):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert label in ('sys.maxsize', '-sys.maxsize')
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_is_even__<-sys.maxsize> ... ok
test_is_even__<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({
...         'noninteger': (1.2345, False),
...         'horribleabuse': ('%s', False),
...     })
...     def test_is_even(self, n, expected, label):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert label in ('noninteger', 'horribleabuse')
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_is_even__<horribleabuse> ... ok
test_is_even__<noninteger> ... ok
...Ran 2 tests...
OK
```

...or just using keyword arguments:

```
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     @foreach(
...         noninteger=(1.2345, False),
...         horribleabuse=('%s', False),
...     )
...     def test_is_even(self, n, expected, label):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...         assert label in ('noninteger', 'horribleabuse')
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_is_even__<horribleabuse> ... ok
test_is_even__<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),
...     })
...
...     other = paramseq(
...         (-15, False),
...         param(15, expected=False),
...         # explicit labelling with keyword arguments:
...         noninteger=param(1.2345, expected=False),
...         horribleabuse=param('%s', expected=False),
...     )
...
```

(continues on next page)

(continued from previous page)

```
...     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_is_even(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
>>> run_tests(Test_is_even)  # doctest: +ELLIPSIS
test_is_even__<-1,expected=False> ... ok
test_is_even__<-14,expected=True> ... ok
test_is_even__<-15,False> ... ok
test_is_even__<-sys.maxsize> ... ok
test_is_even__<15,expected=False> ... ok
test_is_even__<17,expected=False> ... ok
test_is_even__<18->True> ... ok
test_is_even__<2,expected=True> ... ok
test_is_even__<<12399999999...>,False> ... ok
test_is_even__<expected=True,n=<12399999999...>> ... ok
test_is_even__<horribleabuse> ... ok
test_is_even__<just zero, because why not?> ... ok
test_is_even__<noninteger> ... ok
test_is_even__<sys.maxsize> ... ok
...Ran 14 tests...
OK
```

Note: Parameter collections – such as *sequences* (e.g., `list` instances), *mappings* (e.g., `dict` instances), *sets* (e.g., `set` or `frozenset` instances) 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 signatures of the *`foreach()`* decorator and the *`paramseq`*'s 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 singular parameter values or tuples of parameter values, or *`param`* instances.

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 text string* (`str` in Python 3, `str` or `unicode` in Python 2).

Warning: Also, a parameter collection should *not* be a tuple (i.e., an instance of the built-in type `tuple` or, obviously, of any subclass of it, e.g., a *named tuple*), as this is **deprecated** and will become **illegal** in the *0.5.0* version of *unittest_expander*.

> Note that 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).

> **Warning:** Also, a parameter collection should *not* be a Python 3 *binary string-like sequence* (`bytes` or `bytearray`), as this is **deprecated** and will become **illegal** in the *0.5.0* version of *unittest_expander*.

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_case_cls):
...     LO, HI = test_case_cls.LO, test_case_cls.HI
...     print('DEBUG: LO = {}; HI = {}'.format(LO, HI))
...     print('----')
...     yield param(randint(LO, HI) * 2,
...                 expected=True).label('random even')
...     yield param(randint(LO, HI) * 2 + 1,
...                 expected=False).label('random odd')
...
>>> @expand
... class Test_is_even(unittest.TestCase):
...
...     LO = -100
...     HI = 100
...
...     @foreach(randomized)
...     def test_is_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_is_even_negated_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_is_even__<random even> ... ok
test_is_even__<random odd> ... ok
test_is_even_negated_when_incremented__<random even> ... ok
test_is_even_negated_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()`) can 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, *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_is_even(self, n, expected):
...         actual = is_even(n)
...         self.assertTrue(isinstance(actual, bool))
...         self.assertEqual(actual, expected)
...
...     @foreach(input_values_and_results)
...     def test_is_even_negated_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_is_even__<-1,expected=False> ... ok
test_is_even__<-14,expected=True> ... ok
test_is_even__<0,expected=True> ... ok
test_is_even__<17,expected=False> ... ok
test_is_even__<2,expected=True> ... ok
test_is_even__<random even> ... ok
test_is_even__<random odd> ... ok
test_is_even_negated_when_incremented__<-1,expected=False> ... ok
test_is_even_negated_when_incremented__<-14,expected=True> ... ok
test_is_even_negated_when_incremented__<0,expected=True> ... ok
test_is_even_negated_when_incremented__<17,expected=False> ... ok
test_is_even_negated_when_incremented__<2,expected=True> ... ok
```

(continues on next page)

(continued from previous page)

```
test_is_even_negated_when_incremented__<random even> ... ok
test_is_even_negated_when_incremented__<random odd> ... ok
...Ran 14 tests...
OK
```

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

You can apply 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_is_even(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_is_even__<floating, -1,expected=False> ... ok
test_is_even__<floating, -14,expected=True> ... ok
test_is_even__<floating, 0,expected=True> ... ok
test_is_even__<floating, 17,expected=False> ... ok
test_is_even__<floating, 2,expected=True> ... ok
test_is_even__<floating, random even> ... ok
test_is_even__<floating, random odd> ... ok
test_is_even__<integer, -1,expected=False> ... ok
test_is_even__<integer, -14,expected=True> ... ok
test_is_even__<integer, 0,expected=True> ... ok
```

(continues on next page)

(continued from previous page)

```
test_is_even__<integer, 17,expected=False> ... ok
test_is_even__<integer, 2,expected=True> ... ok
test_is_even__<integer, random even> ... ok
test_is_even__<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).

```
>>> from tempfile import NamedTemporaryFile
>>>
>>> @expand
... class TestSaveLoad(unittest.TestCase):
...
...     data_with_contexts = [
...         param(save='', load='').context(NamedTemporaryFile, 'w+t'),
...         param(save='abc', load='abc').context(NamedTemporaryFile, 'w+t'),
...     ]
...
...     @foreach(data_with_contexts)
...     def test_save_load(self, save, load, context_targets):
...         file = context_targets[0]
...         file.write(save)
...         file.flush()
...         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(data_with_contexts)
```

(continues on next page)

(continued from previous page)

```
...     def test_save_load_with_spaces(self, save, load, context_targets):
...         file = context_targets[0]
...         file.write(' ' + save + ' ')
...         file.flush()
...         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
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, if a test method accepts the *context_targets* keyword argument then a list of context manager *as-targets* (i.e., objects returned by context managers' `__enter__()`) will be passed in as that argument.

It is a list 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')
...         ),
...     ]
...
```

(continues on next page)

(continued from previous page)

```
...     @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.flush()
...         file.seek(0)
...         load_actually = file.read()
...         self.assertEqual(load_actually, load)
...         debug.append('test')
...
>>> 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*):

```
>>> @contextlib.contextmanager
... def err_debug_cm(tag):
...     if tag.endswith('context-enter-error'):
...         debug.append('ERR-enter:' + tag)
...         raise RuntimeError('error in __enter__')
...     debug.append('enter:' + tag)
...     try:
...         yield tag
...         if tag.endswith('context-exit-error'):
...             raise RuntimeError('error in __exit__')
...     except:
...         debug.append('ERR-exit:' + tag)
...         raise
...     else:
...         debug.append('exit:' + tag)
...
>>> debug = []
>>> err_params = [
...     (
...         param().label('no_error')
...                .context(err_debug_cm, tag='outer')
...                .context(err_debug_cm, tag='inner')
...     ),
...     (
...         param().label('test_fail')
...                .context(err_debug_cm, tag='outer')
...                .context(err_debug_cm, tag='inner')
...     ),
```

(continues on next page)

(continued from previous page)

```
...     (
...         param().label('test_error')
...                 .context(err_debug_cm, tag='outer')
...                 .context(err_debug_cm, tag='inner')
...     ),
...     (
...         param().label('inner_context_enter_error')
...                 .context(err_debug_cm, tag='outer')
...                 .context(err_debug_cm, tag='inner-context-enter-error')
...     ),
...     (
...         param().label('inner_context_exit_error')
...                 .context(err_debug_cm, tag='outer')
...                 .context(err_debug_cm, tag='inner-context-exit-error')
...     ),
...     (
...         param().label('outer_context_enter_error')
...                 .context(err_debug_cm, tag='outer-context-enter-error')
...                 .context(err_debug_cm, tag='inner')
...     ),
...     (
...         param().label('outer_context_exit_error')
...                 .context(err_debug_cm, tag='outer-context-exit-error')
...                 .context(err_debug_cm, 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, label):
...         if label == 'test_fail':
...             debug.append('FAIL-test')
...             self.fail()
...         elif 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)
```

(continues on next page)

(continued from previous page)

```
>>> debug == [
...     # inner_context_enter_error
...     'setUp',
...     'enter:outer',
...     'ERR-enter:inner-context-enter-error',
...     'ERR-exit:outer',
...     'tearDown',
...
...     # inner_context_exit_error
...     'setUp',
...     'enter:outer',
...     'enter:inner-context-exit-error',
...     'test',
...     'ERR-exit:inner-context-exit-error',
...     'ERR-exit:outer',
...     'tearDown',
...
...     # no_error
...     'setUp',
...     'enter:outer',
...     'enter:inner',
...     'test',
...     'exit:inner',
...     'exit:outer',
...     'tearDown',
...
...     # outer_context_enter_error
...     'setUp',
...     'ERR-enter:outer-context-enter-error',
...     'tearDown',
...
...     # outer_context_exit_error
...     'setUp',
...     'enter:outer-context-exit-error',
...     'enter:inner',
...     'test',
...     'exit:inner',
...     'ERR-exit:outer-context-exit-error',
...     'tearDown',
...
...     # test_error
...     'setUp',
...     'enter:outer',
...     'enter:inner',
...     'ERROR-test',
...     'ERR-exit:inner',
...     'ERR-exit:outer',
...     'tearDown',
...
...     # test_fail
...     'setUp',
...     'enter:outer',
...     'enter:inner',
...     'FAIL-test',
...     'ERR-exit:inner',
...     'ERR-exit:outer',
...     'tearDown',
```

(continues on next page)

(continued from previous page)

```
... ]
True
```

Note that contexts attached to test *method* params (in contrast to those attached to test *class* params – see below: *Deprecated feature: foreach() as a class decorator*) are handled *directly* before (by running `__enter__()`) and after (by running `__exit__()`) a given parametrized test method call, that is, *after* `setUp()` and *before* `tearDown()` calls – so `setUp()` and `tearDown()` are unaffected by any errors related to those contexts.

On the other hand, an error in `setUp()` prevents a test from being called – then contexts are not touched at all:

```
>>> 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)
...
...     @foreach(params_with_contexts)
...     def test_save_load(self, save, load, context_targets):
...         file, silly_cm_target = context_targets
...         assert silly_cm_target == 42
...         file.write(save)
...         file.flush()
...         file.seek(0)
...         load_actually = file.read()
```

(continues on next page)

(continued from previous page)

```
...         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 Deprecated feature: `foreach()` as a class decorator

> **Warning:** Here we describe a **deprecated** feature.
>
> Decorating a *class* with `foreach()` will become **unsupported** in the *0.5.0* version of *unittest_expander*.
>
> Another **deprecation**, strictly related to the above, is that the `into` keyword argument to `expand()` will become **illegal** in the *0.5.0* version of *unittest_expander*.

`foreach()` can be used not only as a test *method* decorator but also as a test *class* decorator – to generate parametrized test *classes*.

That allows you to share each specified parameter/context/label across all test methods. Parameters (and labels, and context targets) are accessible as instance attributes (*not* as method arguments) from any test method, as well as from the `setUp()` and `tearDown()` methods.

```
>>> params_with_contexts = paramseq(                          # 2 param items
...     param(save='', load=''),
...     param(save='abc', load='abc'),
... ).context(NamedTemporaryFile, 'w+t')
```

(continues on next page)

(continued from previous page)

```
>>> useless_data = [                                          # 2 param items
...     param('foo', b=42),
...     param('foo', b=433)]
>>>
>>> @expand(into=globals())  # note the 'into' keyword-only argument
... @foreach(params_with_contexts)
... @foreach(useless_data)
... class TestSaveLoad(unittest.TestCase):
...
...     def setUp(self):
...         self.file = self.context_targets[0]
...         assert self.save == self.load
...         assert self.params == ('foo',)  # self.params <- *positional* ones
...         assert self.b in (42, 433)
...         assert 'foo' in self.label
...
...     @foreach(param(suffix=' '), param(suffix='XX'))        # 2 param items
...     def test_save_load(self, suffix):
...         self.file.write(self.save + suffix)
...         self.file.flush()
...         self.file.seek(0)
...         load_actually = self.file.read()
...         self.assertEqual(load_actually, self.load + suffix)
...
>>> for name in dir():  # doctest: +ELLIPSIS
...     if name.startswith('TestSaveLoad'):
...         name
...
'TestSaveLoad'
"TestSaveLoad__<'foo',b=42, load='',save='>"
"TestSaveLoad__<'foo',b=42, load='abc',save='abc'>"
"TestSaveLoad__<'foo',b=433, load='',save='>"
"TestSaveLoad__<'foo',b=433, load='abc',save='abc'>"
>>>
>>> test_classes = [globals()[name] for name in dir()
...                 if name.startswith('TestSaveLoad__')]
>>> # (note: 2 * 2 * 2 param items -> 8 parametrized tests)
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test_save_load__<suffix=' '> (..._<'foo',b=42, load='',save='>...) ... ok
test_save_load__<suffix='XX'> (..._<'foo',b=42, load='',save='>...) ... ok
test_save_load__<suffix=' '> (..._<'foo',b=42, load='abc',save='abc'>...) ... ok
test_save_load__<suffix='XX'> (..._<'foo',b=42, load='abc',save='abc'>...) ... ok
test_save_load__<suffix=' '> (..._<'foo',b=433, load='',save='>...) ... ok
test_save_load__<suffix='XX'> (..._<'foo',b=433, load='',save='>...) ... ok
test_save_load__<suffix=' '> (..._<'foo',b=433, load='abc',save='abc'>...) ... ok
test_save_load__<suffix='XX'> (..._<'foo',b=433, load='abc',save='abc'>...) ... ok
...Ran 8 tests...
OK
```

As you see, you can combine `foreach()` as *class* decorator(s) with `foreach()` as *method* decorator(s) – you will obtain tests parametrized with the Cartesian product of the involved parameter collections.

Important: when using `foreach()` as a *class* decorator you must remember to place `expand()` as the topmost (the outer) class decorator (above all `foreach()` decorators).

The *into* keyword argument for the `expand()` decorator specifies where the generated (parametrized) subclasses of the decorated test case class should be placed; the attribute value should be either a mapping (typically: the `globals()` dictionary) or a (non-read-only) Python module object, or a (possibly dotted) name of such a module.

Below: an example with the *into* argument being a module object:

```
>>> import types
>>> module = types.ModuleType('_my_test_module')
>>>
>>> @expand(into=module)
... @foreach(params_with_contexts)
... class TestSaveLoad(unittest.TestCase):
...
...     def setUp(self):
...         self.file = self.context_targets[0]
...
...     def test_save_load(self):
...         self.file.write(self.save)
...         self.file.flush()
...         self.file.seek(0)
...         load_actually = self.file.read()
...         self.assertEqual(load_actually, self.load)
...
>>> for name in dir(module):
...     if not name.startswith('__'):
...         name  # doctest: +ELLIPSIS
...
"TestSaveLoad__<load='',save=''>"
"TestSaveLoad__<load='abc',save='abc'>"
>>>
>>> TestSaveLoad__1 = getattr(module, "TestSaveLoad__<load='',save=''>")
>>> TestSaveLoad__2 = getattr(module, "TestSaveLoad__<load='abc',save='abc'>")
>>>
>>> run_tests(TestSaveLoad__1, TestSaveLoad__2)  # doctest: +ELLIPSIS
test_save_load (...TestSaveLoad__<load='',save=''>...) ... ok
test_save_load (...TestSaveLoad__<load='abc',save='abc'>...) ... ok
...Ran 2 tests...
OK
```

…and with *into* being an importable module name:

```
>>> module = types.ModuleType('_my_test_module')
>>> sys.modules['_my_test_module'] = module
>>>
>>> @expand(into='_my_test_module')
... @foreach(params_with_contexts)
... class TestSaveLoad(unittest.TestCase):
...
...     def setUp(self):
...         self.file = self.context_targets[0]
...
...     def test_save_load(self):
...         self.file.write(self.save)
...         self.file.flush()
...         self.file.seek(0)
...         load_actually = self.file.read()
...         self.assertEqual(load_actually, self.load)
...
>>> for name in dir(module):
...     if not name.startswith('__'):
...         name  # doctest: +ELLIPSIS
...
```

(continues on next page)

(continued from previous page)

```
"TestSaveLoad__<load='',save=''>"
"TestSaveLoad__<load='abc',save='abc'>"
>>>
>>> TestSaveLoad__1 = getattr(module, "TestSaveLoad__<load='',save=''>")
>>> TestSaveLoad__2 = getattr(module, "TestSaveLoad__<load='abc',save='abc'>")
>>>
>>> run_tests(TestSaveLoad__1, TestSaveLoad__2)  # doctest: +ELLIPSIS
test_save_load (...TestSaveLoad__<load='',save=''>...) ... ok
test_save_load (...TestSaveLoad__<load='abc',save='abc'>...) ... ok
...Ran 2 tests...
OK
```

...and with *into* not specified – which has, generally, the same effect as setting it to the `globals()` dictionary (however, this implicit variant may not work with those Python implementations that do not support stack frame introspection; *note:* CPython and PyPy do support it perfectly `:-)`):

```
>>> @expand
... @foreach(params_with_contexts)
... class TestSaveLoadIt(unittest.TestCase):
...
...     def setUp(self):
...         self.file = self.context_targets[0]
...
...     def test_save_load(self):
...         self.file.write(self.save)
...         self.file.flush()
...         self.file.seek(0)
...         load_actually = self.file.read()
...         self.assertEqual(load_actually, self.load)
...
>>> for name in dir():
...     if name.startswith('TestSaveLoadIt'):
...         name
...
'TestSaveLoadIt'
"TestSaveLoadIt__<load='',save=''>"
"TestSaveLoadIt__<load='abc',save='abc'>"
>>>
>>> TestSaveLoadIt__1 = globals()["TestSaveLoadIt__<load='',save=''>"]
>>> TestSaveLoadIt__2 = globals()["TestSaveLoadIt__<load='abc',save='abc'>"]
>>>
>>> run_tests(TestSaveLoadIt__1, TestSaveLoadIt__2)  # doctest: +ELLIPSIS
test_save_load (...TestSaveLoadIt__<load='',save=''>...) ... ok
test_save_load (...TestSaveLoadIt__<load='abc',save='abc'>...) ... ok
...Ran 2 tests...
OK
```

Contexts are, obviously, properly handled – also when errors occur (with some legitimate subtle reservations – see: *Contexts cannot suppress exceptions unless you enable that explicitly*):

```
>>> debug = []              # see earlier definition of err_debug_cm()...
>>> err_params.extend([     # see earlier initialization of err_params...
...     (
...         param().label('setUp_error')
...                 .context(err_debug_cm, tag='outer')
...                 .context(err_debug_cm, tag='inner')
```

(continues on next page)

(continued from previous page)

```
...     ),
...     (
...         param().label('tearDown_error')
...                .context(err_debug_cm, tag='outer')
...                .context(err_debug_cm, tag='inner')
...     ),
... ])
>>> into_dict = {}  # this time we'll pass another mapping (not globals())
>>>
>>> @expand(into=into_dict)
... @foreach(err_params)
... class SillyTest(unittest.TestCase):
...
...     def setUp(self):
...         if self.label == 'setUp_error':
...             debug.append('ERR-setUp')
...             raise RuntimeError
...         debug.append('setUp')
...
...     def tearDown(self):
...         if self.label == 'tearDown_error':
...             debug.append('ERR-tearDown')
...             raise RuntimeError
...         debug.append('tearDown')
...
...     def test(self):
...         if self.label == 'test_fail':
...             debug.append('FAIL-test')
...             self.fail()
...         elif self.label == 'test_error':
...             debug.append('ERROR-test')
...             raise RuntimeError
...         else:
...             debug.append('test')
...
>>> for name in sorted(into_dict):
...     name
...
'SillyTest__<inner_context_enter_error>'
'SillyTest__<inner_context_exit_error>'
'SillyTest__<no_error>'
'SillyTest__<outer_context_enter_error>'
'SillyTest__<outer_context_exit_error>'
'SillyTest__<setUp_error>'
'SillyTest__<tearDown_error>'
'SillyTest__<test_error>'
'SillyTest__<test_fail>'
>>>
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test (...SillyTest__<inner_context_enter_error>...) ... ERROR
test (...SillyTest__<inner_context_exit_error>...) ... ERROR
test (...SillyTest__<no_error>...) ... ok
test (...SillyTest__<outer_context_enter_error>...) ... ERROR
test (...SillyTest__<outer_context_exit_error>...) ... ERROR
test (...SillyTest__<setUp_error>...) ... ERROR
test (...SillyTest__<tearDown_error>...) ... ERROR
```

(continues on next page)

(continued from previous page)

```
test (...SillyTest__<test_error>...) ... ERROR
test (...SillyTest__<test_fail>...) ... FAIL
...Ran 9 tests...
FAILED (failures=1, errors=7)
>>> debug == [
...     # inner_context_enter_error
...     'enter:outer',
...     'ERR-enter:inner-context-enter-error',
...     'ERR-exit:outer',
...
...     # inner_context_exit_error
...     'enter:outer',
...     'enter:inner-context-exit-error',
...     'setUp',
...     'test',
...     'tearDown',
...     'ERR-exit:inner-context-exit-error',
...     'ERR-exit:outer',
...
...     # no_error
...     'enter:outer',
...     'enter:inner',
...     'setUp',
...     'test',
...     'tearDown',
...     'exit:inner',
...     'exit:outer',
...
...     # outer_context_enter_error
...     'ERR-enter:outer-context-enter-error',
...
...     # outer_context_exit_error
...     'enter:outer-context-exit-error',
...     'enter:inner',
...     'setUp',
...     'test',
...     'tearDown',
...     'exit:inner',
...     'ERR-exit:outer-context-exit-error',
...
...     # setUp_error
...     'enter:outer',
...     'enter:inner',
...     'ERR-setUp',
...     'ERR-exit:inner',
...     'ERR-exit:outer',
...
...     # tearDown_error
...     'enter:outer',
...     'enter:inner',
...     'setUp',
...     'test',
...     'ERR-tearDown',
...     'ERR-exit:inner',
...     'ERR-exit:outer',
...
...     # test_error
```

(continues on next page)

(continued from previous page)

```
...     'enter:outer',
...     'enter:inner',
...     'setUp',
...     'ERROR-test',  # note:
...     'tearDown',    # *not* ERR-tearDown
...     'exit:inner',  # *not* ERR-exit:inner
...     'exit:outer',  # *not* ERR-exit:outer
...
...     # test_fail
...     'enter:outer',
...     'enter:inner',
...     'setUp',
...     'FAIL-test',   # note:
...     'tearDown',    # *not* ERR-tearDown
...     'exit:inner',  # *not* ERR-exit:inner
...     'exit:outer',  # *not* ERR-exit:outer
... ]
True
```

Note that contexts attached to test *class* params (in contrast to those attached to test *method* params – see: *Context-manager-based fixtures: param.context()*) are automatically handled within `setUp()` and (if applicable) `tearDown()` – so `setUp()` and `tearDown()` *are* affected by errors related to those contexts. On the other hand, context finalization is *not* affected by any exceptions from actual test methods (i.e., context managers' `__exit__()` methods are called with `None, None, None` arguments anyway – unless `setUp()`/`tearDown()` or an enclosed context manager's `__enter__()`/`__exit__()` raises an exception).

2.8.1 Additional note about extending `setUp()` and `tearDown()`

> **Warning:** Here we refer to applying the *`foreach()`* decorator to a *class* which is a **deprecated** feature (see the warning at the beginning of the *Deprecated feature: foreach() as a class decorator* section).

As you can see in the above examples, you can, without any problem, implement your own `setUp()` and/or `tearDown()` methods in test classes that are decorated with *`foreach()`* and *`expand()`*; the *unittest_expander* machinery, which provides its own version of these methods, will incorporate your implementations automatically – by obtaining them with `super()` and calling (*within* the scope of any contexts that have been attached to your parameters with *`param.context()`* or *`paramseq.context()`*).

However, if you need to create a subclass of one of the test classes generated by *`expand()`* applied to a class decorated with *`foreach()`* – you need to obey the following rules:

- you shall not apply *`foreach()`* to that subclass or any class that inherits from it (though you can still apply *`foreach()`* to methods of the subclass);
- when extending `setUp()` and/or `tearDown()` methods:
  - in `setUp()`, calling `setUp()` of the superclass should be the first action;
  - in `tearDown()`, calling `tearDown()` of the superclass should be the last action – and you shall ensure (by using a `finally` clause) that this action is *always* executed.

For example:

```
>>> # the SillyTest__<no_error> class from the previous code snippet
>>> base = into_dict['SillyTest__<no_error>']
```

(continues on next page)

(continued from previous page)

```
>>>
>>> class SillyTestSubclass(base):
...
...     def setUp(self):
...         debug.append('*** before everything ***')
...         # <- at this point no contexts are active (and there are
...         # no self.params, self.label, self.context_targets, etc.)
...         super(SillyTestSubclass, self).setUp()
...         # *HERE* is the place for your extension's implementation
...         debug.append('*** SillyTestSubclass.setUp ***')
...         assert hasattr(self, 'params')
...         assert hasattr(self, 'label')
...         assert hasattr(self, 'context_targets')
...
...     def tearDown(self):
...         try:
...             # *HERE* is the place for your extension's implementation
...             debug.append('*** SillyTestSubclass.tearDown ***')
...         finally:
...             super(SillyTestSubclass, self).tearDown()
...             # <- at this point no contexts are active
...         debug.append('*** after everything ***')
...
>>> debug = []
>>> run_tests(SillyTestSubclass)  # doctest: +ELLIPSIS
test (...SillyTestSubclass...) ... ok
...Ran 1 test...
OK
>>> debug == [
...     '*** before everything ***',
...     'enter:outer',
...     'enter:inner',
...     'setUp',
...     '*** SillyTestSubclass.setUp ***',
...     'test',
...     '*** SillyTestSubclass.tearDown ***',
...     'tearDown',
...     'exit:inner',
...     'exit:outer',
...     '*** after everything ***',
... ]
True
```

2.9 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 suppressing 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
```

(continues on next page)

(continued from previous page)

```
test_it__... ok
...Ran 2 tests...
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!')
...
>>> into_dict = {}
>>> @expand(into=into_dict)
... @foreach([
...     param(setup_error=OSError)
...         .context(SillySuppressingCM, _enable_exc_suppress_=True),
...     param(setup_error=OSError)
...         .context(SillySuppressingCM, _enable_exc_suppress_=True)
...         .context(ErrorCM, error=TypeError),
...     param(setup_error=None),
... ])
... class AnotherSillyExcTest(unittest.TestCase):
...
...     def setUp(self):
...         if self.setup_error is not None:
...             debug.append('raising {}'.format(self.setup_error.__name__))
...             raise self.setup_error('ooops!')
...
...     @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),
...     ])
...     def test_it(self, test_error):
...         debug.append('raising {}'.format(test_error.__name__))
...         raise test_error('ha!')
...
>>> debug = []
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test_it__... ok
test_it__... ok
```

(continues on next page)

(continued from previous page)

```
test_it__... ok
test_it__... ok
test_it__... ok
test_it__... ok
...Ran 6 tests...
OK
>>> debug == [
...     'raising OSError',
...     'suppressing OSError',
...     'raising AssertionError',
...     'suppressing AssertionError',
...
...     'raising OSError',
...     'suppressing OSError',
...     'raising KeyError',
...     'replacing KeyError with RuntimeError',
...     'suppressing RuntimeError',
...
...     'raising OSError',
...     'replacing OSError with TypeError',
...     'suppressing TypeError',
...     'raising AssertionError',
...     'suppressing AssertionError',
...
...     'raising OSError',
...     'replacing OSError with TypeError',
...     'suppressing TypeError',
...     'raising KeyError',
...     'replacing KeyError with RuntimeError',
...     'suppressing RuntimeError',
...
...     'raising AssertionError',
...     'suppressing AssertionError',
...
...     'raising KeyError',
...     'replacing KeyError with RuntimeError',
...     'suppressing RuntimeError',
... ]
True
```

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

```
>>> into_dict = {}
>>> @expand(into=into_dict)
... @foreach([
...     param(setup_error=OSError)
...         .context(SillySuppressingCM),
...     param(setup_error=OSError)
...         .context(SillySuppressingCM)
...         .context(ErrorCM, error=TypeError),
...     param(setup_error=None),
... ])
... class AnotherSillyExcTest2(unittest.TestCase):
...
...     def setUp(self):
```

(continues on next page)

(continued from previous page)

```
...         if self.setup_error is not None:
...             raise self.setup_error('ooops!')
...
...     @foreach([
...         param(test_error=AssertionError)
...             .context(SillySuppressingCM),
...         param(test_error=KeyError)
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=RuntimeError),
...     ])
...     def test_it(self, test_error):
...         raise test_error('ha!')
...
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test_it__... ERROR
test_it__... ERROR
test_it__... ERROR
test_it__... ERROR
test_it__... FAIL
test_it__... ERROR
...Ran 6 tests...
FAILED (failures=1, errors=5)
```

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

```
>>> into_dict = {}
>>> @expand(into=into_dict)
... @foreach([
...     param(setup_error=OSError)
...         .context(SillySuppressingCM),
...     param(setup_error=OSError)
...         .context(SillySuppressingCM)
...         .context(ErrorCM, error=TypeError,
...                  _enable_exc_suppress_=True),
...     param(setup_error=None),
... ])
... class AnotherSillyExcTest3(unittest.TestCase):
...
...     def setUp(self):
...         if self.setup_error is not None:
...             raise self.setup_error('ooops!')
...
...     @foreach([
...         param(test_error=AssertionError)
...             .context(SillySuppressingCM),
...         param(test_error=KeyError)
...             .context(SillySuppressingCM)
...             .context(ErrorCM, error=RuntimeError,
...                      _enable_exc_suppress_=True),
...     ])
...     def test_it(self, test_error):
...         raise test_error('ha!')
...
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
```

(continues on next page)

(continued from previous page)

```
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test_it__... ERROR
test_it__... ERROR
test_it__... ERROR
test_it__... ERROR
test_it__... FAIL
test_it__... ERROR
...Ran 6 tests...
FAILED (failures=1, errors=5)
```

2.10 `Substitute` objects

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

```
>>> @expand
... @foreach(useless_data)
... 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:

```
>>> DummyTest                          # doctest: +ELLIPSIS
<...Substitute object at 0x...>
>>> DummyTest.actual_object            # doctest: +ELLIPSIS
<class '...DummyTest'>
>>> DummyTest.attr
[42]
>>> DummyTest.attr is DummyTest.actual_object.attr
True
>>> (set(dir(DummyTest.actual_object)) - {'__call__'}
...  ).issubset(dir(DummyTest))
True
```

```
>>> 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 class or method (preventing it from being included by test loaders, but still keeping it available for introspection, etc.).

2.11 Custom method/class name formatting

If you don't like how parametrized test method/class 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 or test class,
  - `{base_obj}` – the original test method (technically: function) or test class,
  - `{label}` – the test label (automatically generated or explicitly specified with `param.label()`),
  - `{count}` – consecutive number (within a single application of `@expand()`) of the generated parametrized test method or test class;

  (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}'
>>>
>>> into_dict = {}
>>>
>>> @expand(into=into_dict)
... @foreach(params_with_contexts)
... @foreach(useless_data)
... class TestSaveLoad(unittest.TestCase):
...
...     def setUp(self):
...         self.file = self.context_targets[0]
...
...     @foreach(param(suffix=' '), param(suffix='XX'))
...     def test_save_load(self, suffix):
...         self.file.write(self.save + suffix)
...         self.file.flush()
...         self.file.seek(0)
...         load_actually = self.file.read()
...         self.assertEqual(load_actually, self.load + suffix)
...
>>> for name in sorted(into_dict):  # doctest: +ELLIPSIS
...     name
...
'TestSaveLoad__parametrized_1'
'TestSaveLoad__parametrized_2'
'TestSaveLoad__parametrized_3'
'TestSaveLoad__parametrized_4'
>>>
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
```

(continues on next page)

(continued from previous page)

```
test_save_load__parametrized_1 (...TestSaveLoad__parametrized_1...) ... ok
test_save_load__parametrized_2 (...TestSaveLoad__parametrized_1...) ... ok
test_save_load__parametrized_1 (...TestSaveLoad__parametrized_2...) ... ok
test_save_load__parametrized_2 (...TestSaveLoad__parametrized_2...) ... ok
test_save_load__parametrized_1 (...TestSaveLoad__parametrized_3...) ... ok
test_save_load__parametrized_2 (...TestSaveLoad__parametrized_3...) ... ok
test_save_load__parametrized_1 (...TestSaveLoad__parametrized_4...) ... ok
test_save_load__parametrized_2 (...TestSaveLoad__parametrized_4...) ... ok
...Ran 8 tests...
OK
```

…or, let's say:

```
>>> import string
>>> class SillyFormatter(string.Formatter):
...     def format(self, format_string, *args, **kwargs):
...         label = kwargs['label']
...         if '42' in label:
...             return '!{}!'.format(label)
...         else:
...             result = super(SillyFormatter,
...                            self).format(format_string, *args, **kwargs)
...             if isinstance(kwargs['base_obj'], type):
...                 result = result.replace('_', '^')
...             return result
...
>>> expand.global_name_formatter = SillyFormatter()
>>>
>>> into_dict = {}
>>>
>>> @expand(into=into_dict)
... @foreach(params_with_contexts)
... @foreach(*useless_data)
... class TestSaveLoad(unittest.TestCase):
...
...     def setUp(self):
...         self.file = self.context_targets[0]
...
...     @foreach([param(suffix=' '), param(suffix='XX')])
...     def test_save_load(self, suffix):
...         self.file.write(self.save + suffix)
...         self.file.flush()
...         self.file.seek(0)
...         load_actually = self.file.read()
...         self.assertEqual(load_actually, self.load + suffix)
...
>>> for name in sorted(into_dict):  # doctest: +ELLIPSIS
...     name
...
"!'foo',b=42, load='',save=''!"
"!'foo',b=42, load='abc',save='abc'!"
'TestSaveLoad^^parametrized^3'
'TestSaveLoad^^parametrized^4'
>>>
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test_save_load__parametrized_1 (...!'foo',b=42, load='',save=''!...) ... ok
```

(continues on next page)

(continued from previous page)

```
test_save_load__parametrized_2 (...!'foo',b=42, load='',save='')!...) ... ok
test_save_load__parametrized_1 (...!'foo',b=42, load='abc',save='abc'!...) ... ok
test_save_load__parametrized_2 (...!'foo',b=42, load='abc',save='abc'!...) ... ok
test_save_load__parametrized_1 (...TestSaveLoad^^parametrized^3...) ... ok
test_save_load__parametrized_2 (...TestSaveLoad^^parametrized^3...) ... ok
test_save_load__parametrized_1 (...TestSaveLoad^^parametrized^4...) ... ok
test_save_load__parametrized_2 (...TestSaveLoad^^parametrized^4...) ... ok
...Ran 8 tests...
OK
```

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

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

2.12 Name clashes avoided automatically

`expand()` tries to avoid name clashes: when it detects that a newly generated name clashes with an existing one, it adds a suffix to the new name. E.g.:

```
>>> def setting_attrs(attr_dict):
...     def deco(cls):
...         for k, v in attr_dict.items():
...             setattr(cls, k, v)
...         return cls
...     return deco
...
>>> into_dict = {
...     "Test_is_even__<'foo',b=42>": ('spam', 'spam', 'spam'),
... }
>>> extra_attrs = {
...     'test_even__<4>': 'something',
...     'test_even__<4>__2': None,
... }
>>>
>>> @expand(into=into_dict)
... @foreach(useless_data)
... @setting_attrs(extra_attrs)
... 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))
...
>>> for name, obj in sorted(into_dict.items()):  # doctest: +ELLIPSIS
...     if obj != ('spam', 'spam', 'spam'):
...         name
```

(continues on next page)

(continued from previous page)

```
...
"Test_is_even__<'foo',b=42>__2"
"Test_is_even__<'foo',b=433>"
>>>
>>> test_classes = [into_dict[name] for name, obj in sorted(into_dict.items())
...                 if obj != ('spam', 'spam', 'spam')]
...
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test_even__<-16> (...Test_is_even__<'foo',b=42>__2...) ... ok
test_even__<0> (...Test_is_even__<'foo',b=42>__2...) ... ok
test_even__<0>__2 (...Test_is_even__<'foo',b=42>__2...) ... ok
test_even__<0>__3 (...Test_is_even__<'foo',b=42>__2...) ... ok
test_even__<0>__4 (...Test_is_even__<'foo',b=42>__2...) ... ok
test_even__<4>__3 (...Test_is_even__<'foo',b=42>__2...) ... ok
test_even__<-16> (...Test_is_even__<'foo',b=433>...) ... ok
test_even__<0> (...Test_is_even__<'foo',b=433>...) ... ok
test_even__<0>__2 (...Test_is_even__<'foo',b=433>...) ... ok
test_even__<0>__3 (...Test_is_even__<'foo',b=433>...) ... ok
test_even__<0>__4 (...Test_is_even__<'foo',b=433>...) ... ok
test_even__<4>__3 (...Test_is_even__<'foo',b=433>...) ... ok
...Ran 12 tests...
OK
```

2.13 Questions and answers about various details...

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

Yes, you can. Please consider the following example:

```
>>> debug = []
>>> into_dict = {}
>>>
>>> # see earlier definition of debug_cm()...
>>> class_params = paramseq(1, 2, 3).context(debug_cm, tag='C')
>>> method_params = paramseq(7, 8, 9).context(debug_cm, tag='M')
>>>
>>> @foreach(class_params)
... class MyTestMixIn(object):
...
...     @foreach(method_params)
...     def test(self, y):
...         [x] = self.params
...         debug.append((x, y, self.n))
...
>>> @expand(into=into_dict)
... class TestActual(MyTestMixIn, unittest.TestCase):
...     n = 42
...
>>> @expand(into=into_dict)
... class TestYetAnother(MyTestMixIn, unittest.TestCase):
...     n = 12345
...
>>> for name in sorted(into_dict):
...     name
```

(continues on next page)

(continued from previous page)

```
...
'TestActual__<1>'
'TestActual__<2>'
'TestActual__<3>'
'TestYetAnother__<1>'
'TestYetAnother__<2>'
'TestYetAnother__<3>'
>>>
>>> test_classes = [into_dict[name] for name in sorted(into_dict)]
>>> run_tests(*test_classes)  # doctest: +ELLIPSIS
test__<7> (...TestActual__<1>...) ... ok
test__<8> (...TestActual__<1>...) ... ok
test__<9> (...TestActual__<1>...) ... ok
test__<7> (...TestActual__<2>...) ... ok
test__<8> (...TestActual__<2>...) ... ok
test__<9> (...TestActual__<2>...) ... ok
test__<7> (...TestActual__<3>...) ... ok
test__<8> (...TestActual__<3>...) ... ok
test__<9> (...TestActual__<3>...) ... ok
test__<7> (...TestYetAnother__<1>...) ... ok
test__<8> (...TestYetAnother__<1>...) ... ok
test__<9> (...TestYetAnother__<1>...) ... ok
test__<7> (...TestYetAnother__<2>...) ... ok
test__<8> (...TestYetAnother__<2>...) ... ok
test__<9> (...TestYetAnother__<2>...) ... ok
test__<7> (...TestYetAnother__<3>...) ... ok
test__<8> (...TestYetAnother__<3>...) ... ok
test__<9> (...TestYetAnother__<3>...) ... ok
...Ran 18 tests...
OK
>>> (type(MyTestMixIn) is type and
...  inspect.isfunction(vars(MyTestMixIn)['test']))                # (not touched by␣
↪@expand)
True
>>> type(TestActual) is type(TestYetAnother) is Substitute           # (replaced by␣
↪@expand)
True
>>> type(vars(TestActual.actual_object)['test']) is Substitute       # (replaced by␣
↪@expand)
True
>>> type(vars(TestYetAnother.actual_object)['test']) is Substitute   # (replaced by␣
↪@expand)
True
>>> debug == [
...     'enter:C', 'enter:M', (1, 7, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (1, 8, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (1, 9, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (2, 7, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (2, 8, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (2, 9, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (3, 7, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (3, 8, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (3, 9, 42), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (1, 7, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (1, 8, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (1, 9, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (2, 7, 12345), 'exit:M', 'exit:C',
```

(continues on next page)

(continued from previous page)

```
...     'enter:C', 'enter:M', (2, 8, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (2, 9, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (3, 7, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (3, 8, 12345), 'exit:M', 'exit:C',
...     'enter:C', 'enter:M', (3, 9, 12345), 'exit:M', 'exit:C',
... ]
True
```

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

A similar yet simpler example (*without* using `foreach()` as a *test class decorator*, which – *as stated earlier* – is a deprecated feature):

```
>>> debug = []
>>> class MyTestMixIn(object):
...
...     @foreach(method_params)  # (see method_params defined earlier...)
...     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
```

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

As long as you do *not* apply `foreach()` to test *classes* (but only to test *methods*) – *yes, you can* (in some of the past versions of *unittest_expander* it was broken, but now it works perfectly):

```
>>> debug = []
>>> into_dict = {}
>>> 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
```

But things complicate when you apply `foreach()` to test *classes*. For such cases the answer is: *do not try this at home*. *As it was said earlier*, the parts of *unittest_expander* related to applying `foreach()` to classes are **deprecated** anyway.

2.13.3 "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 = []
>>> into_dict = {}
>>> parameters = paramseq(
...     1, 2, 3,
... ).context(debug_cm)  # see earlier definition of debug_cm()...
>>>
>>> @expand
... class Test(object):  # not a unittest.TestCase subclass
...
```

(continues on next page)

(continued from previous page)

```
...     @foreach(parameters)
...     def test(self, n):
...         debug.append(n)
...
>>> # confirming that unittest_expander machinery acted properly:
>>> instance = Test()
>>> type(instance.test) is Substitute
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
```

However, note that if you decorate your test class (and not only its methods) with `foreach()`, the test running tools you use are expected to call `setUp()` and `tearDown()` methods appropriately – as *unittest*'s test running machinery does (though your test class does not need to implement these methods by itself).

Warning: Here we refer to applying the `foreach()` decorator to a *class* which is a **deprecated** feature (see the warning at the beginning of the *Deprecated feature: foreach() as a class decorator* section).

```
>>> debug = []
>>> into_dict = {}
>>>
>>> @expand(into=into_dict)
... @foreach(parameters)
... class Test(object):  # not a unittest.TestCase subclass
...
...     def test(self):
...         assert len(self.params) == 1
...         n = self.params[0]
...         debug.append(n)
...
>>> # confirming that unittest_expander machinery acted properly:
>>> type(Test) is Substitute
True
>>> orig_cls = Test.actual_object
>>> type(orig_cls) is type
True
>>> orig_cls.__bases__ == (object,)
True
>>> orig_cls.__name__ == 'Test'
True
>>> not hasattr(orig_cls, 'setUp') and not hasattr(orig_cls, 'tearDown')
True
>>> cls1 = into_dict['Test__<1>']
>>> cls2 = into_dict['Test__<2>']
```

(continues on next page)

(continued from previous page)

```
>>> cls3 = into_dict['Test__<3>']
>>> issubclass(cls1, orig_cls)
True
>>> issubclass(cls2, orig_cls)
True
>>> issubclass(cls3, orig_cls)
True
>>> hasattr(cls1, 'setUp') and hasattr(cls1, 'tearDown')
True
>>> hasattr(cls2, 'setUp') and hasattr(cls2, 'tearDown')
True
>>> hasattr(cls3, 'setUp') and hasattr(cls3, 'tearDown')
True
>>> instance1 = cls1()
>>> instance2 = cls2()
>>> instance3 = cls3()
>>> for inst in [instance1, instance2, instance3]:
...     # doing what any reasonable test runner should do
...     inst.setUp()
...     try: inst.test()
...     finally: inst.tearDown()
...
>>> debug == [
...     'enter', 1, 'exit',
...     'enter', 2, 'exit',
...     'enter', 3, 'exit',
... ]
True
```

2.13.4 "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
```

```
>>> into_dict = {}
>>> @expand(into=into_dict)
... class TestIt2(unittest.TestCase):
...
...     def test(self):
...         sys.stdout.write(' [DEBUG: OK] ')
...         sys.stdout.flush()
...
>>> run_tests(TestIt2)  # doctest: +ELLIPSIS
test ... [DEBUG: OK] ok
```

(continues on next page)

(continued from previous page)

```
...Ran 1 test...
OK
>>> into_dict
{}
```

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

The `foreach()` decorator is designed to be applied *only*:

- to regular test methods (being instance methods, *not* static or class methods), that is, to functions being attributes of test (or test mix-in) classes;
- to test (or test mix-in) classes themselves

(however, note that – *as noted earlier* – the latter is a deprecated feature).

You should *not* apply the decorator to anything else (especially, not to static or class methods). If you do, the effect is undefined: an exception or some other faulty/unexpected behavior may be observed (immediately or, for example, when `expand()` is applied, or when tests are run...).

CHAPTER 3

Module Contents

The module `unittest_expander`'s public interface consists of the following functions, classes 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

or

@expand(*, *into=globals()*)

Deprecated since version 0.4.0: The *into* argument will become *illegal* in the version *0.5.0*.

This decorator is intended to be applied to *test classes*: doing that causes that test parameters – previously attached to related test methods (and/or classes) by decorating them with `foreach()` – are "expanded", that is, actual parametrized versions of those methods (and/or classes) are generated.

The public interface provided by `expand()` includes also the following attributes (making it possible to *customize how names of parametrized test methods and classes are generated*):

expand.**global_name_pattern**

expand.**global_name_formatter**

3.2 The `foreach()` method/class decorator

@foreach(*param_collection*)

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

- a `paramseq` instance,

- a *sequence* **not being** a *text string* (in other words, such an object for whom `isinstance(obj, collections.abc.Sequence) and not isinstance(obj, str)` 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, and to 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.

Any valid parameter collection will be, under the hood, automatically coerced to a *`paramseq`*.

Deprecated since version 0.4.0: A parameter collection given as a tuple (i.e., an instance of the built-in type `tuple` or of a subclass of it, e.g., a *named tuple*) will become *illegal* in the version *0.5.0* (note that this deprecation concerns tuples used as *parameter collections* themselves, *not* as *items* of parameter collections; the latter are – and will be – perfectly OK). As a parameter collection, instead of a tuple, use another type (e.g., a `list`).

Deprecated since version 0.4.3: A parameter collection given as an instance of the Python 3 built-in type `bytes` or `bytearray` (or of a subclass thereof) will become *illegal* in the version *0.5.0*.

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

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

or

@foreach(**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.

———

This decorator is intended to be applied to test *methods* and/or test *classes* – to attach to those methods (or classes) the test parameters from the specified parameter collection (only then it is possible to generate, by using *`expand()`*, actual parametrized methods and/or classes...).

Deprecated since version 0.4.0: Support for decorating test *classes* with *`foreach()`* will be *removed* in the version *0.5.0*.

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 *text string* (in other words, such an object for whom `isinstance(obj, collections.abc.Sequence) and not isinstance(obj, str)` 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, and to 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.

Deprecated since version 0.4.0: A parameter collection given as a tuple (i.e., an instance of the built-in type `tuple` or of a subclass of it, e.g., a *named tuple*) will become *illegal* in the version *0.5.0* (note that this deprecation concerns tuples used as *parameter collections* themselves, *not* as *items* of parameter collections; the latter are – and will be – perfectly OK). As a parameter collection, instead of a tuple, use another type (e.g., a `list`).

Deprecated since version 0.4.3: A parameter collection given as an instance of the Python 3 built-in type `bytes` or `bytearray` (or of a subclass thereof) will become *illegal* in the version *0.5.0*.

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

- a `param` instance,
- a `tuple` (converted automatically to a `param` which contains parameter values being the items of that tuple),
- any other object (converted automatically to a `param` which contains 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 – whose items are `param` instances.

The public interface provided by this class includes the following instance methods:

__add__(*param_collection*)

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

Deprecated since version 0.4.0: *param_collection* being a tuple will become *illegal* in the version *0.5.0*.

Deprecated since version 0.4.3: *param_collection* being a Python 3 `bytes` or `bytearray` will become *illegal* in the version *0.5.0*.

__radd__(*param_collection*)

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

Deprecated since version 0.4.0: *param_collection* being a tuple will become *illegal* in the version *0.5.0*.

Deprecated since version 0.4.3: *param_collection* being a Python 3 `bytes` or `bytearray` will become *illegal* in the version *0.5.0*.

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

Returns a new `paramseq` instance contaning clones of the items of the current instance – 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*.

The public interface provided by this class includes the following instance methods:

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

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

3.5 Non-essential constants and classes

3.5.1 The `__version__` constant

__version__

The version of `unittest_expander` as a **PEP 440**-compliant identifier (being a `str`).

3.5.2 The `Substitute` class

class Substitute(*actual_object*)

actual_object is the object *to be proxied* (typically, it is a test method or test class, previously decorated with `foreach()`).

Apart from exposing in a transparent way nearly all attributes of the proxied object, the public interface provided by this class includes the following instance attribute:

actual_object

The proxied object itself (unwrapped).

`Substitute` instances are *not* callable.

CHAPTER 4

Changes

4.1 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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!

4.6 0.3.1 (2014-08-19)

- Several tests/documentation-related fixes and improvements.

4.7 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.

4.8 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.

4.9 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.

4.10 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.

4.11 0.1.1 (2014-07-29)

- Minor tests/documentation-related improvements.

4.12 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

Symbols

A

C

E

F

G

L

P

S

U