# Python unittest subtest



**Summary**: in this tutorial, you'll learn how to define parameterized tests using unittest (https://www.pythontutorial.net/python-unit-testing/python-unittest/) 's `subTest()` context manager.

## Introduction to the unittest subtest context manager

First, create a new module (https://www.pythontutorial.net/python-oop/what-is-a-python-module/) called `pricing.py` and define a `calculate()` function as follows:

```
def calculate(price, tax=0, discount=0):
    return round((price - discount) * (1+tax), 2)
```

The `calculate()` function calculates the net price from the price, tax, and discount.

Second, create the `test_pricing.py` test module to test the `calculate()` function:

```
import unittest

from pricing import calculate


class TestPricing(unittest.TestCase):
    def test_calculate(self):
        pass
```

To test the `calculate()` function, you need to come up with multiple test cases, for example:

- Has price with no tax and no discount
- Has price with tax but no discount
- Has price with no tax and discount
- Has price with tax and discount

To cover these cases, you need to have various test methods. Or you can define a single test method and supply test data from a list of cases. For example:

```
import unittest

from pricing import calculate


class TestPricing(unittest.TestCase):
    def test_calculate(self):
        items = (
            {'case': 'No tax, no discount', 'price': 10, 'tax': 0, 'discount': 0, 'net_price': 10},
            {'case': 'Has tax, no discount', 'price': 10, 'tax': 0.1, 'discount': 0, 'net_price': 10},
            {'case': 'No tax, has discount', 'price': 10, 'tax': 0, 'discount': 1, 'net_price': 10},
            {'case': 'Has tax, has discount', 'price': 10, 'tax': 0.1, 'discount': 1, 'net_price': 9.9},
        )

        for item in items:
            with self.subTest(item['case']):
                net_price = calculate(
                    item['price'],
                    item['tax'],
                    item['discount']
                )
```

```
                self.assertEqual(
                    net_price,
                    item['net_price']
                )
```

Run the test:

```
python -m unittest test_pricing -v
```

Output:

```
test_calculate (test_pricing.TestPricing) ... FAIL

======================================================================
FAIL: test_calculate (test_pricing.TestPricing)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python-unit-testing\test_pricing.py", line 26, in test_calculate
    self.assertEqual(
AssertionError: 11.0 != 10


----------------------------------------------------------------------
Ran 1 test in 0.002s
```

```
FAILED (failures=1)
```

The problem with this approach is that the test stops after the first failure. To resolve this, the unittest provides you with the `subTest()` context manager. For example:

```
import unittest

from pricing import calculate


class TestPricing(unittest.TestCase):
    def test_calculate(self):
        items = (
            {'case': 'No tax, no discount', 'price': 10, 'tax': 0, 'discount': 0, 'net_price': 10},
            {'case': 'Has tax, no discount', 'price': 10, 'tax': 0.1, 'discount': 0, 'net_price': 10},
            {'case': 'No tax, has discount', 'price': 10, 'tax': 0, 'discount': 1, 'net_price': 10},
            {'case': 'Has tax, has discount', 'price': 10, 'tax': 0.1, 'discount': 1, 'net_price': 9.9},
        )

        for item in items:
            with self.subTest(item['case']):
                net_price = calculate(
                    item['price'],
```

```
                    item['tax'],
                    item['discount']
                )
                self.assertEqual(
                    net_price,
                    item['net_price']
                )
```

Run the test:

```
python -m unittest test_pricing -v
```

Output:

```
test_calculate (test_pricing.TestPricing) ...
======================================================================
FAIL: test_calculate (test_pricing.TestPricing) [Has tax, no discount]
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python-unit-testing\test_pricing.py", line 26, in test_calculate
    self.assertEqual(
AssertionError: 11.0 != 10
```

```
======================================================================
FAIL: test_calculate (test_pricing.TestPricing) [No tax, has discount]
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python-unit-testing\test_pricing.py", line 26, in test_calculate
    self.assertEqual(
AssertionError: 9 != 10


----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=2)
```

By using the `subTest()` context manager, the test didn't stop after the first failure. Also, it shows a very detailed message after each failure so that you can examine the case.

## The subTest() context manager syntax

The following shows the `subTest()` context manager syntax:

```
def subTest(self, msg=_subtest_msg_sentinel, **params):
```

The `subTest()` returns a context manager. The optional `message` parameter identifies the closed block of the subtest returned by the context manager.

If a failure occurs, it'll mark the test case as failed. However, it resumes execution at the end of the enclosed block, allowing further test code to be executed.

## Summary

- Use the unittest `subTest()` context manager to parameterize tests.