

# UnitTest介绍

---

- unittest是python自带的自动化测试框架
- unittest主要包含的内容
  - TestCase(测试用例)
  - TestSuite(测试套件,把多个TestCase集成到一个测试TestSuite)
  - TestRunner(执行测试用例)
  - TestLoader(自动从代码中加载多个测试用例TestCase)
  - Fixture(unittest特性)

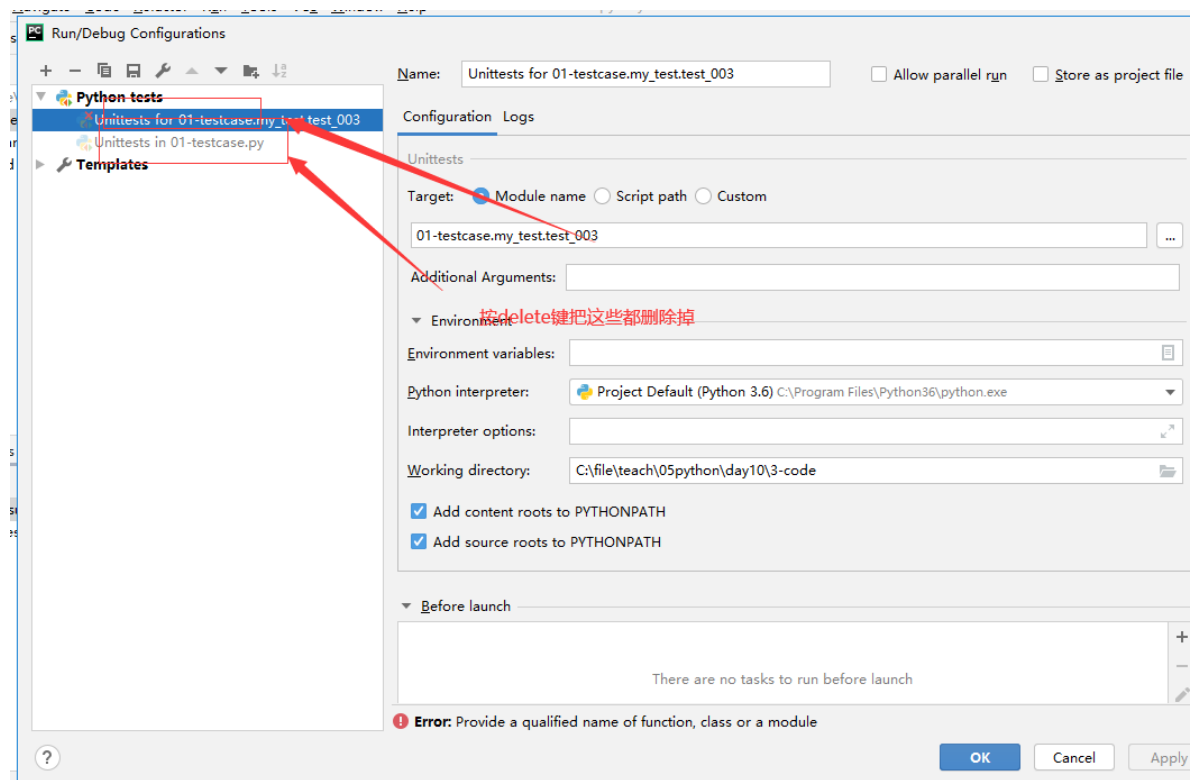
## TestCase

---

- 第一步:导入unittest模块
- 第二步:实现一个类,这个类必须继承自unittest.TestCase类
- 第三步:类中每个方法代表一个测试用例,方法名必须以test开头

```
1  import unittest
2
3  def my_sum(a, b):
4      return a + b
5
6  class my_test(unittest.TestCase):
7      def test_001(self):
8          print(my_sum(5, 6))
9
10     def test_002(self):
11         print(my_sum(0, 3))
12
13
```





## TestSuite

- 把多个测试用例整合成一个测试套件
- 使用方法
  - import导入unittest
  - import导入其他的包含测试用例的py文件
    - py文件的命名规则与变量名相同
  - 实例化unittest.TestSuite类的对象
  - 调用对象的addTest方法
    - addTest(py文件名.类名("方法名"))

```
1 import unittest
2 import testcase_01
3
4 suite = unittest.TestSuite()
5 suite.addTest(testcase_01.my_test("test_001"))
6 suite.addTest(testcase_01.my_test("test_002"))
7 # 只是把测试用例添加到了测试套件中,并不是执行测试用例
8
```

- 用unittest.makeSuite一次导入一个类中的所有测试方法

```
1 import unittest
2 import testcase_01
3
4 suite = unittest.TestSuite()
5 # suite.addTest(testcase_01.my_test("test_001"))
6 # suite.addTest(testcase_01.my_test("test_002"))
7 # 只是把测试用例添加到了测试套件中,并不是执行测试用例
8 suite.addTest(unittest.makeSuite(testcase_01.my_test))
9
```

## TextTestRunner

- 作用,执行在suite中的测试用例
- 使用方法
  - 先实例化TextTestRunner的对象
  - 调用对象的run方法
    - 只要把suite做为参数,放入到run方法里面

```
1 import unittest
2 import testcase_01
3
4 suite = unittest.TestSuite()
5 # suite.addTest(testcase_01.my_test("test_001"))
6 # suite.addTest(testcase_01.my_test("test_002"))
7 # 只是把测试用例添加到了测试套件中,并不是执行测试用例
8 suite.addTest(unittest.makeSuite(testcase_01.my_test))
9
10 runner = unittest.TextTestRunner() # 实例化TextTestRunner的对象
11 runner.run(suite) # 调用对象的run方法
12
```

## TestLoader

- 可以从指定目录查找指定py文件中的所有测试用例,自动加载到TestSuite中

```
1 import unittest
2 # 用TestLoader对象的discover方法来自动查找py,自动加载py文件中的方法
3 # 第一个参数是从哪里找py文件, "."从当前目录开始查找py文件
4 # 第二个参数是指定py文件的文件名,可以用通配符
5 suite = unittest.TestLoader().discover(".", "my*.py")
6 runner = unittest.TextTestRunner()
7 runner.run(suite)
```

## TestSuite和TestLoader的使用区别

- 当只是要执行py文件中多个测试用例中的几个,而不是全部执行那么适合用TestSuite的addTest加载指定的测试用例
- 当要执行所有的py文件中的所有的测试用例,那么适合使用TestLoader

## 小结

- 所有的TestCase最终都是用TextTestRunner来执行的
- TextTestRunner执行的是TestSuite
- 一个TestSuite中可以有多多个TestCase

## Fixture

- 可以在测试用例执行之前自动调用指定的函数,在测试用例执行之后自动调用指定的函数
- 控制级别
  - 方法级
    - 每个方法执行前和执行后都自动调用函数
  - 类级
    - 不管类中有多少方法,一个类执行前后都自动调用函数
  - 模块级
    - 不管一个模块(一个模块就是一个py文件)中有多少类,模块执行前后自动调用函数

## 方法级

- 在TestCase,也就是测试用例所在的class中定义方法
- def setUp(self) 当测试用例执行前,自动被调用
- def tearDown(self) 当测试用例执行后,自动被调用
- 如果一个TestCase中有多个测试用例,那么setUp和tearDown就会被自动调用多次

mytest.py内容修改如下:

```
1  import unittest
2
3  def my_sum(a, b):
4      return a + b
5
6  class my_test(unittest.TestCase):
7      def setUp(self):
8          print("setup被自动调用了")
9      def tearDown(self):
10         print("teardown被自动调用了")
11
12         def test_001(self):
13             print(my_sum(5, 6))
14
15         def test_002(self):
16             print(my_sum(0, 3))
```

## 类级

- 不管类中有多少方法,一个类开始的时候自动调用函数,结束之后自动调用函数
- 类级的fixture一定要是类方法
- `@classmethod def setUpClass(cls)` 类开始时自动调用的方法
- `@classmethod def tearDownClass(cls)` 类结束的时候自动调用的方法

mytest.py修改如下:

```
1  import unittest
2
3  def my_sum(a, b):
4      return a + b
5
6  class my_test(unittest.TestCase):
7      @classmethod
8      def setUpClass(cls):
9          print("setUpClass自动调用了")
10     @classmethod
11     def tearDownClass(cls):
12         print("tearDownClass自动调用了")
13     def setUp(self):
14         print("setUp被自动调用了")
15     def tearDown(self):
16         print("tearDown被自动调用了")
17
18     def test_001(self):
19         print(my_sum(5, 6))
20
21     def test_002(self):
22         print(my_sum(0, 3))
23
24
25
```

## 模块级

- 不管py文件中有多少个类,以及类中有多少方法,只自动执行一次
- `def setUpModule()` 在py文件开始的时候自动调用
- `def tearDownModule()` 在py文件结束的时候自动调用

修改后的mytest.py内容如下

```
1  import unittest
2
3  def setUpModule():
4      print("setUpModule自动调用了")
5
6  def tearDownModule():
7      print("tearDownModule自动调用了")
8
9  def my_sum(a, b):
10     return a + b
11
12  class my_test1(unittest.TestCase):
```

```

13     @classmethod
14     def setUpClass(cls):
15         print("setUpClass自动调用了")
16     @classmethod
17     def tearDownClass(cls):
18         print("tearDownClass自动调用了")
19     def setUp(self):
20         print("setUp被自动调用了")
21     def tearDown(self):
22         print("tearDown被自动调用了")
23
24     def test_001(self):
25         print(my_sum(5, 6))
26
27     def test_002(self):
28         print(my_sum(0, 3))
29
30 class my_test2(unittest.TestCase):
31     @classmethod
32     def setUpClass(cls):
33         print("setUpClass自动调用了")
34     @classmethod
35     def tearDownClass(cls):
36         print("tearDownClass自动调用了")
37     def setUp(self):
38         print("setUp被自动调用了")
39     def tearDown(self):
40         print("tearDown被自动调用了")
41
42     def test_001(self):
43         print(my_sum(5, 6))
44
45     def test_002(self):
46         print(my_sum(0, 3))
47
48
49

```

## fixture小结

- 一定要在继承于unittest.TestCase这个类的子类中使用
- setUp,tearDown, 每个方法执行开始和完毕后自动调用
- setUpClass, tearDownClass, 每个类开始时候和结束时候自动调用
- setUpModule, tearDownModule,每个py文件开始和结束的时候自动调用

## 断言

- 让程序来判断测试用例执行结果是否符合预期

## unittest断言

- assertEquals(参数1, 参数2)
  - 如果参数1,参数2的值相等,断言成功,否则断言失败

- 两个参数,有一个存放实际结果,有一个存放预期结果

修改后的mytest.py内容

```
1 import unittest
2
3 def setUpModule():
4     print("setUpModule自动调用了")
5
6 def tearDownModule():
7     print("tearDownModule自动调用了")
8
9 def my_sum(a, b):
10     return a - b
11
12 class my_test1(unittest.TestCase):
13     @classmethod
14     def setUpClass(cls):
15         print("setUpclass自动调用了")
16     @classmethod
17     def tearDownClass(cls):
18         print("tearDownclass自动调用了")
19     def setUp(self):
20         print("setUp被自动调用了")
21     def tearDown(self):
22         print("tearDown被自动调用了")
23
24     def test_001(self):
25         num1 = my_sum(5, 6) # 定义变量num1得到my_sum函数的返回值
26         self.assertEqual(num1, 11) # num1里存放的是实际结果,11是预期结果
27         # 实际结果与预期结果相符,代表测试用例测试通过
28         # 不相符代表测试用例测试失败
29
30     def test_002(self):
31         num1 = my_sum(0, 3)
32         self.assertEqual(num1, 3)
33
34
```

Traceback (most recent call last):

```
File "C:\file\teach\05python\day10\3-code\mytest.py", line 32, in test_002
    self.assertEqual(num1, 3)
```

AssertionError: -3 != 3

- assertIn(参数1, 参数2)
  - 如果参数1在参数2里面,断言通过,否则断言失败

修改后的mytest.py内容如下:

```
1 import unittest
2 import random
3
4 def setUpModule():
5     print("setUpModule自动调用了")
```



```

6
7 def tearDownModule():
8     print("tearDownModule自动调用了")
9
10 def my_sum(a, b):
11     return a + b
12
13 def my_rand(): # 返回从1到5之间的一个随机数
14     return random.randint(10, 50)
15
16 class my_test1(unittest.TestCase):
17     def test_001(self):
18         num1 = my_sum(5, 6) # 定义变量num1得到my_sum函数的返回值
19         self.assertEqual(num1, 11) # num1里存放的是实际结果,11是预期结果
20         # 实际结果与预期结果相符,代表测试用例测试通过
21         # 不相符代表测试用例测试失败
22
23     def test_002(self):
24         num1 = my_sum(0, 3)
25         self.assertEqual(num1, 3)
26
27     def test_003(self):
28         num1 = my_rand()
29         self.assertIn(num1, [1, 2, 3, 4, 5])
30
31

```

FAIL: test\_003 (mytest.my\_test1)

-----  
Traceback (most recent call last):

File "C:\file\teach\05python\day10\3-code\mytest.py", line 29, in test\_003  
 self.assertIn(num1, [1, 2, 3, 4, 5])  
AssertionError: 29 not found in [1, 2, 3, 4, 5]

## 参数化

### 测试用例中使用参数化的场景

- 多个测试用例代码相同,只是测试数据不同,预期结果不同,可以把多个测试用例通过参数化技术合并为一个测试用例

```

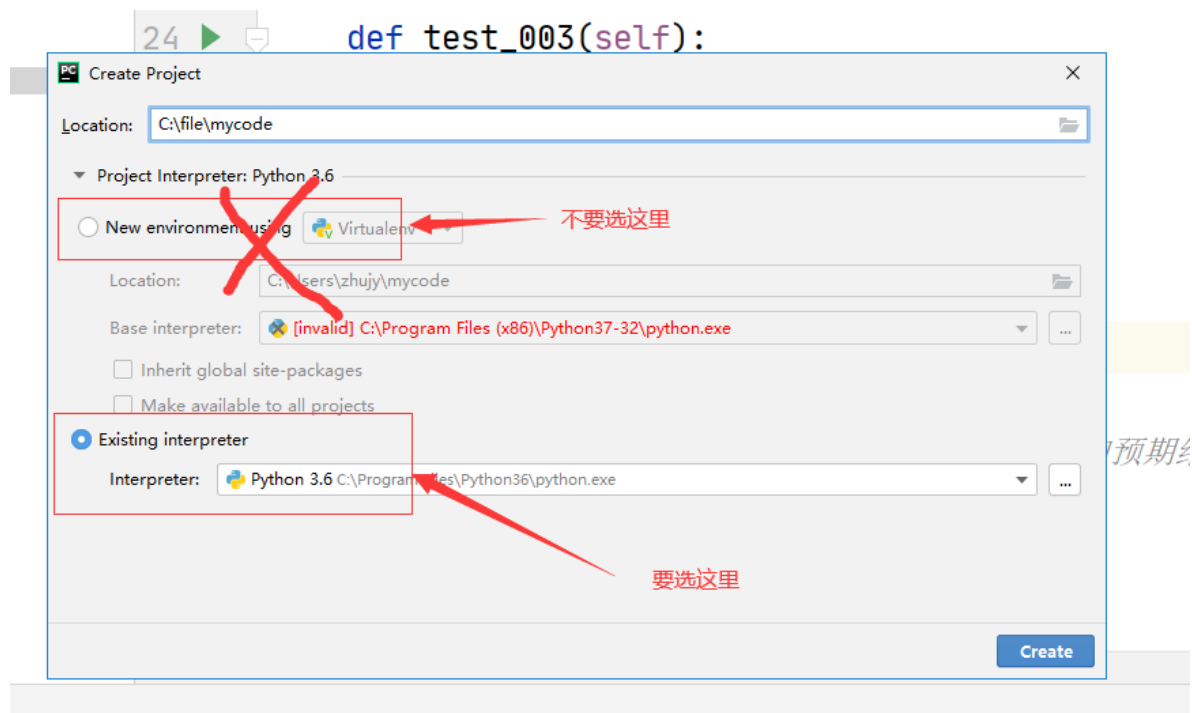
1 import unittest
2 import random
3
4 def setUpModule():
5     print("setUpModule自动调用了")
6
7 def tearDownModule():
8     print("tearDownModule自动调用了")
9
10 def my_sum(a, b):
11     return a + b

```

```
12
13 class my_test1(unittest.TestCase):
14     def test_001(self):
15         num1 = my_sum(5, 6) # 定义变量num1得到my_sum函数的返回值
16         self.assertEqual(num1, 11) # num1里存放的是实际结果,11是预期结果
17         # 实际结果与预期结果相符,代表测试用例测试通过
18         # 不相符代表测试用例测试失败
19
20     def test_002(self):
21         num1 = my_sum(0, 3)
22         self.assertEqual(num1, 3)
23
24     def test_003(self):
25         num1 = my_sum(-3, 7)
26         self.assertEqual(num1, 4)
27
28     def test_004(self):
29         num1 = my_sum(-4, -20)
30         self.assertEqual(num1, -24)
31
32     # 以上的测试用例,基本是一样的,测试用例的数据和预期结果不同
33
34
35
```

## 手工安装py包的过程

- 把parameterized目录和parameterized-0.7.4.dist-info拷贝到python安装目录的Lib/site-packages下
- 在pycharm中新建项目的时候,注意一个选项
  - 第一步:先新建了一个文件夹c:\file\mycode



## 参数化

- 第一步:导入from parameterized import parameterized
- 第二步在方法上面用@parameterized.expand()修饰方法
  - expand()里面是一个列表
  - 列表里面放多个元组, 每个元组中的成员就代表调用方法使用的实参
  - 列表中有几个元组,方法就会自动被调用几次

```
1 import unittest
2 from parameterized import parameterized
3
4 def my_sum(a, b):
5     return a + b
6
7 class my_test1(unittest.TestCase):
8     # a是调用my_sum的第一个参数
9     # b是调用my_sum的第二个参数
10    # c是预期结果
11    @parameterized.expand([(1, 2, 3), (5, 6, 110), (-1, 3, 2)])
12    def test_001(self, a, b, c):
13        num1 = my_sum(a, b) # 定义变量num1得到my_sum函数的返回值
14        self.assertEqual(num1, c) # num1里存放的是实际结果,11是预期结果
15        # 实际结果与预期结果相符,代表测试用例测试通过
16        # 不相符代表测试用例测试失败
17
```

```

@parameterized.expand([(1, 2, 3), (5, 6, 110), (-1, 3, 2)])
def test_001(self, a, b, c):
    num1 = my_sum(a, b) # 第一次调用a的值为1,b的值为2,c的值为3 定义变量num1得到my_sum函数的返回值
    self.assertEqual(num1, c) # num1里存放的是实际结果,11是预期结果
    # 实际结果与预期结果相符,代表测试用例测试通过
    # 不相符代表测试用例测试失败
    # 第二次调用的时候a的值是5, b的值是6, c的值是110
    # 第三笔调用的时候a的值为-1,b为3,c为2

```

由于有3个元组,所以test\_001被调用了三次

执行结果

Ran 3 tests in 0.002s

FAILED (failures=1)

参数化场景二

```

1 import unittest
2 from parameterized import parameterized
3
4 def my_sum(a, b):
5     return a + b
6
7 list1 = [(1, 2, 3), (5, 6, 110), (-1, 3, 2)]
8
9 class my_test1(unittest.TestCase):
10     # a是调用my_sum的第一个参数
11     # b是调用my_sum的第二个参数
12     # c是预期结果
13     @parameterized.expand(list1)
14     def test_001(self, a, b, c):
15         num1 = my_sum(a, b) # 定义变量num1得到my_sum函数的返回值
16         self.assertEqual(num1, c) # num1里存放的是实际结果,11是预期结果
17         # 实际结果与预期结果相符,代表测试用例测试通过
18         # 不相符代表测试用例测试失败
19
20
21
22
23

```

参数化场景三:

```

1 import unittest
2 from parameterized import parameterized
3
4 def my_sum(a, b):

```

```

5     return a + b
6
7     def get_data(): # 定义了一个函数,返回一个列表
8         return [(1, 2, 3), (5, 6, 110), (-1, 3, 2)]
9
10    class my_test1(unittest.TestCase):
11        # a是调用my_sum的第一个参数
12        # b是调用my_sum的第二个参数
13        # c是预期结果
14        @parameterized.expand(get_data())
15        def test_001(self, a, b, c):
16            num1 = my_sum(a, b) # 定义变量num1得到my_sum函数的返回值
17            self.assertEqual(num1, c) # num1里存放的是实际结果,11是预期结果
18            # 实际结果与预期结果相符,代表测试用例测试通过
19            # 不相符代表测试用例测试失败
20
21

```

## 跳过(了解即可)

- 可以通过@unittest.skip跳过指定的方法或者函数
- 语法

```

1 @unittest.skip
2 def 方法名():

```

```

1 import unittest
2 from parameterized import parameterized
3
4 def my_sum(a, b):
5     return a + b
6
7 def get_data(): # 定义了一个函数,返回一个列表
8     return [(1, 2, 3), (5, 6, 11), (-1, 3, 2)]
9
10 class my_test1(unittest.TestCase):
11     # a是调用my_sum的第一个参数
12     # b是调用my_sum的第二个参数
13     # c是预期结果
14     @parameterized.expand(get_data())
15     def test_001(self, a, b, c):
16         num1 = my_sum(a, b) # 定义变量num1得到my_sum函数的返回值
17         self.assertEqual(num1, c) # num1里存放的是实际结果,11是预期结果
18         # 实际结果与预期结果相符,代表测试用例测试通过
19         # 不相符代表测试用例测试失败
20     @unittest.skip
21     def test_002(self):
22         print("test002")
23
24

```

# 通过TextTestRunner生成测试报告

- 在实例化TextTestRunner对象的时候,需要写参数

```
1 stream=file, verbosity=2
2 file代表用open打开的一个文件
3 verbosity=2,固定
```

- 第一步:用open,w方式打开测试报告文件
- 第二步:实例化TextTestRunner对象
- 第三步调用对象的run方法执行测试套件
- 第四步:关闭open打开的文件

```
1 import unittest
2 # 用TestLoader对象的discover方法来自动查找py,自动加载py文件中的方法
3 # 第一个参数是从哪里找py文件, "."从当前目录开始查找py文件
4 # 第二个参数是指定py文件的文件名,可以用通配符
5 suite = unittest.TestLoader().discover(".", "my*.py")
6 # runner = unittest.TextTestRunner()
7 file = open("test01.txt", "w", encoding="utf8")
8 runner = unittest.TextTestRunner(stream=file, verbosity=2)
9 runner.run(suite)
10 file.close()
```

## HTML测试报告

- 把文件HTMLTestRunner.py拷贝到项目目录下
- 在代码中导入模块from HTMLTestRunner import HTMLTestRunner
- 调用HTMLTestRunner(stream=file, title="我的第一个html测试报告")
  - 第一个参数是用open打开的文件, 打开的文件扩展名一定是.html
  - open打开文件的时候,用wb,不用指定字符集
- 调用runner对象的run方法执行测试套件
- 关闭open打开的文件

```
1 import unittest
2 from HTMLTestRunner import HTMLTestRunner
3 # 用TestLoader对象的discover方法来自动查找py,自动加载py文件中的方法
4 # 第一个参数是从哪里找py文件, "."从当前目录开始查找py文件
5 # 第二个参数是指定py文件的文件名,可以用通配符
6 suite = unittest.TestLoader().discover(".", "my*.py")
7 # runner = unittest.TextTestRunner()
8 file = open("test01.html", "wb") # 用wb代表用二进制写方式打开文件
9 # runner = unittest.TextTestRunner(stream=file, verbosity=2)
10 runner = HTMLTestRunner(stream=file, title="我的第一个html测试报告")
11 runner.run(suite)
12 file.close()
```