Python XVII:
Тестирование

Тестирование – это процесс проверки программного обеспечения на соответствие заданным требованиям и ожидаемому поведению. Оно является важной частью разработки программного обеспечения, которая помогает выявить ошибки и недочеты в коде до его запуска в продакшн-среде.

В Python есть несколько популярных библиотек для тестирования, таких как unittest, pytest и nose.

unittest - это встроенная библиотека в Python, которая предоставляет средства для написания и запуска модульных тестов. Она обеспечивает наследование, чтобы уменьшить дублирование кода в тестах, и предоставляет удобный интерфейс для проверки результатов тестов.

pytest - это библиотека, которая упрощает написание тестов, предоставляя больше гибкости и меньше шаблонного кода, чем unittest. Она автоматически находит и запускает все тесты в проекте, а также поддерживает параметризацию тестов.

nose - это еще одна библиотека для тестирования, которая также облегчает написание тестов. Она имеет расширенный функционал, включая автоматическое обнаружение тестов, поддержку атрибутов тестирования и тестирование с помощью данных из базы данных.

Пример простого тестового кейса в **unittest**:

```
import unittest

class MyTestCase(unittest.TestCase):
      def test_addition(self):
      self.assertEqual(1+1, 2)

if __name__ == '__main__':
      unittest.main()
```

В этом примере мы создаем класс MyTestCase, который наследуется от unittest.TestCase. Мы определяем метод test_addition, который проверяет, что результат сложения 1 и 1 равен 2, с помощью метода assertEqual. Затем мы запускаем наши тесты с помощью unittest.main().

Рассмотрим **unittest** подробней:

unittest предоставляет класс **TestCase**, который является базовым классом для написания тестовых классов. В тестовом классе каждый метод с именем,

начинающимся с **test**, будет запущен как тест. Методы могут использовать различные утверждения для проверки ожидаемых результатов.

Вот пример тестового класса, который тестирует функцию add:

```
import unittest

def add(a, b):
    return a + b

class TestAdd(unittest.TestCase):
    def test_add_positive_numbers(self):
    result = add(2, 3)
    self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
    result = add(-2, -3)
    self.assertEqual(result, -5)

    def test_add_zero(self):
    result = add(0, 0)
    self.assertEqual(result, 0)

if __name__ == '__main__':
    unittest.main()
```

В этом примере определена функция add, которая складывает два числа. Затем определен класс **TestAdd**, который наследуется от unittest.TestCase. В классе определены три метода, каждый из которых тестирует функцию add с разными аргументами. Методы используют утверждения self.assertEqual для проверки, что результат выполнения функции add соответствует ожидаемому результату.

Для запуска тестов можно использовать команду python -m unittest <имя_модуля>, где <имя_модуля> - это имя файла, содержащего тестовый класс. В результате выполнения команды будут выведены результаты тестов.

unittest также предоставляет множество других методов для проверки результатов тестов. Например:

1. assertNotEqual(a, b) - проверяет, что значения a и b не равны.
2. assertTrue(expr) - проверяет, что выражение expr истинно.
3. assertFalse(expr) - проверяет, что выражение expr ложно.
4. assertIs(a, b) - проверяет, что a и b являются одним и тем же объектом.
5. assertIsNot(a, b) - проверяет, что a и b являются разными объектами.

   и многие другие.

unittest позволяет создавать фикстуры (fixtures) для тестов. Фикстуры - это предварительно подготовленное состояние, которое необходимо для проведения тестов. Фикстуры могут включать, например, инициализацию базы данных или создание объектов, которые будут использоваться в тестах.

unittest предоставляет два метода для создания фикстур: setUp() и tearDown(). Метод setUp() вызывается перед каждым тестом и используется для инициализации состояния, необходимого для проведения теста. Метод tearDown() вызывается после каждого теста и используется для очистки состояния, созданного в setUp().

Кроме того, **unittest** позволяет создавать параметризованные тесты, то есть тесты, которые запускаются несколько раз с разными параметрами. Для этого необходимо использовать декоратор @parameterized.

Например, рассмотрим следующий пример кода:

```
import unittest

class MyTestCase(unittest.TestCase):
      def setUp(self):
      self.my_list = [1, 2, 3]

      def tearDown(self):
      self.my_list = None

      def test_add_to_list(self):
      self.my_list.append(4)
      self.assertEqual(self.my_list, [1, 2, 3, 4])

      def test_remove_from_list(self):
      self.my_list.remove(2)
      self.assertEqual(self.my_list, [1, 3])

if __name__ == '__main__':
      unittest.main()
```

В этом примере мы создаем класс MyTestCase, который наследуется от класса unittest.TestCase. Мы определяем два метода setUp() и tearDown(), которые инициализируют и очищают переменную my_list.

Мы также определяем два тестовых метода test_add_to_list и test_remove_from_list, которые проверяют добавление и удаление элементов из списка my_list. В этих методах мы используем метод assertEqual(), чтобы проверить, что список my_list изменяется правильно.

Метод if __name__ == '__main__': unittest.main() используется для запуска всех тестов в классе MyTestCase.

Параметризованные тесты позволяют запустить один и тот же тестовый метод несколько раз с разными параметрами. Для этого в unittest есть декоратор @parameterized из модуля parameterized.

Для создания параметризованного теста нужно создать кортеж с параметрами, передать его в тестовый метод в качестве аргументов и указать ожидаемый результат.

Вот пример параметризованного теста:

```
import unittest
from parameterized import parameterized

class TestMath(unittest.TestCase):

     @parameterized.expand([("addition", 2, 3, 5), ("subtraction",
4, 2, 2), ("multiplication", 5, 6, 30), ("division", 8, 4, 2),])
     def test_math_operations(self, name, x, y, result):
     if name == "addition":
          self.assertEqual(x + y, result)
     elif name == "subtraction":
          self.assertEqual(x - y, result)
     elif name == "multiplication":
          self.assertEqual(x * y, result)
     elif name == "division":
          self.assertEqual(x / y, result)
```

В этом примере мы создали класс TestMath, в котором есть параметризованный тест test_math_operations. Метод test_math_operations принимает четыре аргумента: name - название операции, x и y - операнды, result - ожидаемый результат.

Декоратор @parameterized.expand принимает список кортежей, каждый из которых содержит параметры для теста. Мы указали, что мы хотим запустить тест метод test_math_operations четыре раза - для сложения, вычитания, умножения и деления.
В методе test_math_operations мы проверяем, что результат операции совпадает с ожидаемым результатом для каждой из операций.

При запуске тестов, unittest создаст четыре тестовых метода с уникальными именами и запустит их. Если какой-то из тестов завершится неудачно, unittest выведет сообщение об ошибке.

Важно помнить, что тесты должны быть написаны так, чтобы они проверяли не только правильную работу кода в идеальных условиях, но и обрабатывали различные ошибки и краевые случаи. Это поможет убедиться, что ваше программное обеспечение работает корректно в любых условиях.