Python Tutorial: Unittest

`unittest` ist ein eingebautes Modul in Python, das einen umfangreichen Satz von Werkzeugen für das Schreiben und Ausführen von Tests bereitstellt. Es ist stark von `JUnit` aus der Javawelt inspiriert, welches ursprünglich von `Smalltalk` abstammt.

Einführung in Unittest

Das `unittest`-Modul unterstützt die Entwicklung von Testfällen, die Organisation dieser Testfälle in Test-Suiten und die Ausführung der Tests mit einem Testrunner. Es bietet auch die Möglichkeit, vor und nach jedem Testfall Setup- und Teardown-Routinen auszuführen.

Grundlagen von Unittest

Einen Testfall schreiben

Testfälle in `unittest` werden durch das Erstellen einer Klasse definiert, die von `unittest.TestCase` erbt. Innerhalb dieser Klasse können Sie Methoden definieren, die mit `test` beginnen, um verschiedene Aspekte Ihres Codes zu testen.

```
import unittest

class MeineTests(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(2 + 2, 4)

    def test_subtraktion(self):
        self.assertEqual(4 - 2, 2)
```

Tests ausführen

Um die Tests auszuführen, können Sie das Modul als Skript ausführen oder die `unittest.main()`-Funktion in Ihrem Code verwenden, um einen Testrunner zu starten.

Ausführung von der Kommandozeile

```
python -m unittest deine_test_datei.py
```

Ausführung im Code

Wenn Sie die Tests innerhalb eines Python-Skripts ausführen möchten, fügen Sie am Ende des Skripts folgendes hinzu:

```

```

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

Setup und Teardown

`unittest` bietet Mechanismen, um vor und nach jedem Testfall oder vor und nach der gesamten Testklasse bestimmte Vorbereitungen bzw. Aufräumarbeiten durchzuführen.

```
class MeineTests(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # Wird einmal vor allen Testfällen ausgeführt
        pass

    def setUp(self):
        # Wird vor jedem Testfall ausgeführt
        pass

    def tearDown(self):
        # Wird nach jedem Testfall ausgeführt
        pass

    @classmethod
    def tearDownClass(cls):
        # Wird einmal nach allen Testfällen ausgeführt
        pass
```

Wichtige Assertions

`unittest` bietet eine Vielzahl von Methoden, um Erwartungen über die Testergebnisse auszudrücken, wie z.B. `assertEqual`, `assertTrue`, `assertFalse`, `assertRaises` und viele mehr.

Vorteile von Unittest

- **Strukturierte Testentwicklung**: Unterstützt die systematische Entwicklung und Organisation von Testfällen.
- **Reichhaltige Assertion-Bibliothek**: Bietet umfangreiche Möglichkeiten, um das Verhalten des Codes zu überprüfen.
- **Unabhängigkeit von Testfällen**: Ermöglicht die unabhängige Ausführung jedes Testfalls, was die Fehlersuche erleichtert.

Zusammenfassung

Das `unittest`-Modul ist ein leistungsfähiges Werkzeug für das Testen in Python, das es ermöglicht, robuste Testfälle und Test-Suiten zu entwickeln. Es ist ideal für größere Projekte und Situationen, in denen detaillierte Testberichte und die Überprüfung komplexer Szenarien erforderlich sind.

Das `unittest`-Framework in Python bietet eine Vielzahl von Methoden, um Testfälle effektiv zu schreiben und die Ergebnisse zu überprüfen. Diese Methoden, oft als Assertion-Methoden bezeichnet, ermöglichen es Ihnen, die Funktionalität Ihres Codes zu testen, indem Sie die tatsächlichen Ergebnisse mit den erwarteten Ergebnissen vergleichen. Hier stelle ich einige der wichtigsten `unittest`-Methoden vor, die für das Schreiben von Tests verwendet werden.

Unittest Methoden in Python

Wichtige Assertion-Methoden

`assertEqual(a, b)`

Überprüft, ob `a` und `b` gleich sind. Wenn nicht, schlägt der Test fehl.

```
self.assertEqual(1 + 1, 2)
```

`assertNotEqual(a, b)`

Überprüft, ob `a` und `b` nicht gleich sind. Wenn doch, schlägt der Test fehl.

```
self.assertNotEqual(2 * 2, 5)
```

`assertTrue(x)`

Überprüft, ob `x` True ist. Wenn nicht, schlägt der Test fehl.

```
self.assertTrue(2 < 3)
```

`assertFalse(x)`

Überprüft, ob `x` False ist. Wenn nicht, schlägt der Test fehl.

```
self.assertFalse(2 > 3)
```

`assertRaises(Exception, func, *args, **kwargs)`

Überprüft, ob die Ausführung von `func` mit den angegebenen Argumenten und Schlüsselwortargumenten eine Ausnahme vom Typ `Exception` auslöst.

```

```

```
with self.assertRaises(ValueError):
    int("abc")
```

`assertIn(a, b)`

Überprüft, ob `a` in `b` enthalten ist. Wenn nicht, schlägt der Test fehl.

```
self.assertIn(1, [1, 2, 3])
```

`assertNotIn(a, b)`

Überprüft, ob `a` nicht in `b` enthalten ist. Wenn doch, schlägt der Test fehl.

```
self.assertNotIn(4, [1, 2, 3])
```

`assertIsInstance(a, b)`

Überprüft, ob `a` eine Instanz von `b` ist. Wenn nicht, schlägt der Test fehl.

```
self.assertIsInstance("hello", str)
```

`assertIsNone(x)`

Überprüft, ob `x` None ist. Wenn nicht, schlägt der Test fehl.

```
self.assertIsNone(None)
```

`assertIsNotNone(x)`

Überprüft, ob `x` nicht None ist. Wenn doch, schlägt der Test fehl.

```
self.assertIsNotNone(42)
```

Setup und Teardown Methoden

`setUp()`

Wird vor jedem Testfall ausgeführt. Ideal für die Initialisierung, die für jeden Test benötigt wird.

`tearDown()`

Wird nach jedem Testfall ausgeführt. Nützlich für Aufräumarbeiten nach Tests.

`setUpClass()`

Wird einmal vor dem Start der Testfälle in einer Testklasse ausgeführt. Nützlich für kostspielige Operationen, die für alle Tests nur einmal benötigt werden.

`tearDownClass()`

Wird einmal nach dem Ende aller Testfälle einer Testklasse ausgeführt. Geeignet für Aufräumarbeiten, die einmal nach allen Tests durchgeführt werden müssen.

Zusammenfassung

Die `unittest`-Methoden bieten eine umfangreiche Bibliothek von Werkzeugen für das Schreiben von Tests in Python. Durch die Verwendung dieser Methoden können Sie die Korrektheit Ihres Codes effektiv überprüfen und sicherstellen, dass Änderungen am Code die bestehende Funktionalität nicht beeinträchtigen.