Pytest: Installation, Einführung und Beispiele

Pytest ist ein populäres, flexibles und leistungsfähiges Testframework für Python. Es wird für seine einfache Syntax geschätzt, die das Schreiben von Tests erleichtert, sowie für seine Fähigkeit, komplexe Test-Szenarien zu unterstützen. Pytest kann für einfache Unit-Tests bis hin zu komplexen Funktions- und Integrationstests verwendet werden.

Installation

Pytest kann leicht über `pip`, den Paketmanager von Python, installiert werden. Öffnen Sie eine Kommandozeile oder ein Terminal und führen Sie den folgenden Befehl aus:

```
pip install pytest
```

Nach der Installation können Sie `pytest` in der Kommandozeile oder Ihrem Terminal ausführen, um zu überprüfen, ob es korrekt installiert wurde.

Einführung in Pytest

Grundlegende Testfälle schreiben

Mit Pytest können Sie Testfälle schreiben, indem Sie Funktionen definieren, deren Namen mit `test` beginnen. Pytest erkennt diese automatisch als Testfälle.

Beispiel: Einen einfachen Test schreiben

```python
# test_math.py

def test_addition():
    assert 1 + 1 == 2

def test_subtraktion():
    assert 2 - 1 == 1
```

Um die Tests auszuführen, navigieren Sie im Terminal zum Verzeichnis, das Ihre Testdatei enthält, und führen Sie `pytest` aus:

```
pytest
```

Pytest wird automatisch alle Dateien finden, die mit `test_` beginnen oder auf `_test.py` enden, die Testfunktionen ausführen und Ihnen die Ergebnisse anzeigen.

Assertions

Pytest ermöglicht es Ihnen, Python's eingebaute `assert`-Anweisung für Assertions zu verwenden. Dies macht den Testcode sauber und einfach zu lesen.

Beispiel: Mehrere Assertions

```
def test_multiplikation():
    assert 2 * 3 == 6
    assert 3 * 3 == 9

def test_division():
    assert 8 / 2 == 4
    assert 9 / 3 == 3
```

Setup und Teardown

Pytest bietet verschiedene Möglichkeiten, Setup- und Teardown-Code auszuführen, zum Beispiel mit Fixtures.

Beispiel: Verwendung einer Fixture

```
import pytest

@pytest.fixture
def input_value():
    return 4

def test_doppelte(input_value):
    assert input_value * 2 == 8
```

Vorteile von Pytest

- **Einfachheit**: Pytest erfordert minimalen Boilerplate-Code für Tests.
- **Leistungsfähige Fixtures**: Bieten flexible Mechanismen für Setup und Teardown.
- **Reichhaltiges Plugin-Ökosystem**: Viele Plugins sind verfügbar, um die Funktionalität von Pytest zu erweitern.
- **Erweiterte Assertions**: Detaillierte Ausgabe bei Assertion-Fehlern.

Beispiele für fortgeschrittene Nutzung

Pytest unterstützt fortgeschrittene Test-Szenarien, einschließlich Parametrisierung von Tests, parallele Ausführung und Testabdeckung.

Parametrisierte Tests

```
@pytest.mark.parametrize("test_input,expected", [(3, 9), (5, 25), (10,
100)])
```

```
def test_quadrate(test_input, expected):
    assert test_input ** 2 == expected
```

Tests parallel ausführen

Um Tests parallel auszuführen, können Sie das Plugin `pytest-xdist` installieren und Pytest mit dem `-n` Argument ausführen:

```
pip install pytest-xdist
pytest -n NUM
```

Ersetzen Sie `NUM` durch die Anzahl der gewünschten parallelen Arbeiter.

Zusammenfassung

Pytest ist ein leistungsfähiges und flexibles Testframework für Python, das für seine einfache Syntax und seine Fähigkeit, komplexe Tests zu unterstützen, geschätzt wird. Es ist gut geeignet für Projekte jeder Größe, von kleinen bis zu großen, komplexen Anwendungen.

Mocking ist eine wichtige Technik beim Unit-Testing, insbesondere wenn es darum geht, Teile eines Systems zu isolieren, die in einem Test nicht direkt überprüft werden sollen. In Tests kann es notwendig sein, externe Abhängigkeiten oder komplexe Systeme zu simulieren, um das Verhalten von Code unter kontrollierten Bedingungen zu testen. Pytest unterstützt das Mocking durch das `unittest.mock`-Modul, das in der Python-Standardbibliothek enthalten ist, sowie durch verschiedene Plugins und Erweiterungen, wie `pytest-mock`, das die Verwendung von Mocks in Pytest vereinfacht.

Pytest: Mocking

Verwendung von `unittest.mock`

Das `unittest.mock`-Modul bietet eine flexible Mocking- und Patching-Funktionalität. Es ermöglicht das Ersetzen von Teilen des Systems, die getestet werden, durch Mock-Objekte und das Festlegen von Rückgabewerten oder das Überprüfen, ob bestimmte Methoden aufgerufen wurden.

Beispiel: Ein einfaches Mocking

```
from unittest.mock import MagicMock
import pytest

def externe_funktion():
    # Diese Funktion würde in der realen Anwendung externe Ressourcen
aufrufen
    pass

def zu_testende_funktion():
```

```
    # Diese Funktion ruft die externe Funktion auf
    externe_funktion()

def test_zu_testende_funktion():
    with pytest.mock.patch('pfad.zur.modul.externe_funktion') as mock_funk:
        mock_funk.return_value = "Mocked Value"
        zu_testende_funktion()
        mock_funk.assert_called_once()
```

Pytest-mock Plugin

Das `pytest-mock` Plugin vereinfacht das Mocking in Pytest, indem es eine `mocker`-Fixture bereitstellt, die das Erstellen und Verwenden von Mocks erleichtert.

Installation

Um `pytest-mock` zu verwenden, installieren Sie es zunächst über `pip`:

```
pip install pytest-mock
```

Beispiel: Verwendung von `pytest-mock`

```
def zu_testende_funktion(argument):
    # Diese Funktion ruft die externe Funktion mit einem Argument auf
    return externe_funktion(argument)

def test_zu_testende_funktion(mocker):
    mock = mocker.patch('pfad.zur.modul.externe_funktion')
    mock.return_value = "Mocked Value"

    ergebnis = zu_testende_funktion("test")

    mock.assert_called_once_with("test")
    assert ergebnis == "Mocked Value"
```

Vorteile von Mocking in Tests

- **Isolation**: Erlaubt das Testen von Code in Isolation von seinen Abhängigkeiten.
- **Kontrolle**: Ermöglicht die Kontrolle über das Verhalten von externen Abhängigkeiten innerhalb von Tests.
- **Sicherheit**: Verhindert, dass Tests mit realen Systemen oder Daten interagieren, was insbesondere bei der Arbeit mit externen Diensten wichtig ist.

Best Practices

- **Klarheit**: Verwenden Sie Mocks nur, wenn notwendig, um die Lesbarkeit und Wartbarkeit der Tests zu gewährleisten.
- **Überprüfung**: Stellen Sie sicher, dass Sie das Verhalten der Mocks überprüfen, um die Korrektheit des getesteten Codes zu validieren.
- **Aufräumen**: Achten Sie darauf, Mocks ordnungsgemäß zurückzusetzen oder zu entfernen, um Seiteneffekte zwischen Tests zu vermeiden.

Zusammenfassung

Mocking ist eine kritische Technik beim Unit-Testing, die es ermöglicht, Code effektiv in Isolation zu testen. Mit dem `unittest.mock`-Modul und Erweiterungen wie `pytest-mock` bietet Python leistungsfähige Werkzeuge, um externe Abhängigkeiten in Tests zu simulieren und zu kontrollieren.

Real World Beispiel

Lassen Sie uns ein realistischeres Beispiel für Mocking betrachten, indem wir eine einfache Anwendung simulieren, die eine externe API aufruft, um Wetterdaten zu erhalten. In diesem Beispiel wollen wir die Funktion testen, die den API-Aufruf durchführt, ohne dabei tatsächlich eine Anfrage an die externe API zu senden. Stattdessen verwenden wir Mocking, um die Antwort der API zu simulieren.

Die Anwendung

Stellen Sie sich vor, wir haben eine Funktion `hole_wetterdaten`, die einen API-Aufruf an einen Wetterdienst macht und die Wettervorhersage für eine bestimmte Stadt zurückgibt.

```
import requests

def hole_wetterdaten(stadt):
    url = f"http://api.wetterdienst.de/vorhersage/{stadt}"
    antwort = requests.get(url)
    if antwort.status_code == 200:
        daten = antwort.json()
        return daten['vorhersage']
    else:
        raise ValueError("Fehler beim Abrufen der Wetterdaten")
```

Der Test

Wir wollen nun sicherstellen, dass unsere Funktion `hole_wetterdaten` korrekt funktioniert, indem wir einen Test mit Pytest und dem `pytest-mock` Plugin schreiben. Dabei mocken wir die `requests.get` Funktion, um eine kontrollierte Antwort zu simulieren.

Zuerst installieren Sie `pytest` und `pytest-mock`, falls noch nicht geschehen:

```
pip install pytest pytest-mock
```

Jetzt schreiben wir unseren Test:

```
# Importieren Sie pytest und die zu testende Funktion
import pytest
from mein_modul import hole_wetterdaten

def test_hole_wetterdaten(mocker):
    # Mock für requests.get vorbereiten
    mock_get = mocker.patch('requests.get')

    # Eine simulierte Antwort vorbereiten
    mock_antwort = mocker.Mock()
    mock_antwort.status_code = 200
    mock_antwort.json.return_value = {'vorhersage': 'Sonnig mit Chancen auf
Regenbögen'}

    # Konfigurieren des Mocks, um die simulierte Antwort zurückzugeben
    mock_get.return_value = mock_antwort

    # Funktion aufrufen und das Ergebnis überprüfen. Wurde korrekt mit Json
gearbeitet?
    #
    ergebnis = hole_wetterdaten('Berlin')
    assert ergebnis == 'Sonnig mit Chancen auf Regenbögen'

    # Bestätigen, dass der Mock wie erwartet aufgerufen wurde

mock_get.assert_called_once_with('http://api.wetterdienst.de/vorhersage/Ber
lin')
```

In diesem Test haben wir die `requests.get` Methode mit `mocker.patch` gemockt, um zu verhindern, dass während des Tests ein echter HTTP-Aufruf gemacht wird. Stattdessen haben wir eine Mock-Antwort konfiguriert, die unsere Funktion verwenden soll. Schließlich haben wir überprüft, ob das Ergebnis unserer Funktion dem erwarteten Wert entspricht und ob der Mock korrekt aufgerufen wurde.

Wie reagiert deine Funktion auf unerwartete oder fehlerhafte Antworten? Im Folgenden Beispiel wird getestet, ob bei HTTP-Responsestatus 404 ein ValueError in `hole_wetterdaten` ausgelöst wird.

```
def test_hole_wetterdaten_fehler(mocker):
    mock_get = mocker.patch('requests.get')
    mock_antwort = mocker.Mock()
    mock_antwort.status_code = 404
    mock_antwort.json.return_value = {}
    mock_get.return_value = mock_antwort

    with pytest.raises(ValueError):
        hole_wetterdaten('Unbekannt')
```