استفاده از Mock و Assert در تست‌ها

در تست‌نویسی، گاهی اوقات نیاز است که وابستگی‌های خارجی مانند پایگاه داده، APIها، یا سیستم‌های دیگر را شبیه‌سازی کنیم تا از پیچیدگی‌ها و وابستگی‌های غیرضروری اجتناب کنیم. این کار را می‌توان با استفاده از **Mocking** انجام داد. در اینجا به‌طور کامل در مورد **Mocking** و نحوه استفاده از آن در تست‌ها توضیح می‌دهیم.

---

1. معرفی مفهوم Mocking در تست‌نویسی

Mocking فرآیند شبیه‌سازی اشیاء و وابستگی‌ها در طول تست است. به‌طور خاص، زمانی که کد شما به یک سیستم خارجی مانند پایگاه داده، فایل، یا API متصل است، ممکن است نخواهید به آن سیستم متصل شوید و در عوض بخواهید پاسخ‌هایی جعلی یا شبیه‌سازی‌شده ایجاد کنید.

در پایتون، ما می‌توانیم از ماژول `unittest.mock` برای ایجاد **mock objects** استفاده کنیم. این اشیاء mock می‌توانند رفتار خاصی را شبیه‌سازی کنند که برای آزمایش کد شما مفید است.

---

2. استفاده از `unittest.mock` برای ایجاد اشیاء Mock و تست وابستگی‌ها

ماژول `unittest.mock` ابزارهایی را برای شبیه‌سازی رفتارهای مختلف اشیاء فراهم می‌آورد. مهم‌ترین کلاس‌ها و توابع در این ماژول عبارتند از:

- **`Mock`**: برای شبیه‌سازی هر نوع شیء.
- **`patch`**: برای شبیه‌سازی یک شیء در مدت زمان خاص در هنگام انجام تست.

نمونه استفاده از `Mock`

```python
from unittest.mock import Mock

# ایجاد یک mock object
mock_object = Mock()

# شبیه‌سازی رفتارهای مختلف
mock_object.some_method.return_value = 'Hello, World!'

# استفاده از mock
print(mock_object.some_method())  # 'Hello, World!'
```

در این مثال، ما یک شیء `Mock` ایجاد کرده‌ایم و برای متد `some_method` یک مقدار برگشتی تعیین کرده‌ایم.

نمونه استفاده از `patch`

`patch` برای شبیه‌سازی وابستگی‌ها در هنگام اجرای تست‌ها استفاده می‌شود. این ابزار به‌طور معمول برای تغییر موقتی ویژگی‌ها و متدهای یک ماژول یا کلاس کاربرد دارد.

```python
from unittest.mock import patch

# شبیه‌سازی یک تابع خارجی
def external_api():
    return "Real API Response"

# تست با استفاده از patch
with patch('__main__.external_api', return_value='Mocked Response'):
    print(external_api())  # 'Mocked Response'
```

در اینجا، با استفاده از `patch` تابع `external_api` را موقتا تغییر دادیم تا پاسخ جعلی ارائه دهد.

---

3. استفاده از `assert` برای تایید رفتار صحیح کد

در تست‌های واحد، از متدهای مختلف `assert` برای بررسی صحت عملکرد کد استفاده می‌کنیم. این متدها برای مقایسه نتایج واقعی با نتایج مورد انتظار استفاده می‌شوند.

مثال استفاده از `assert`

```python
import unittest

class TestMathOperations(unittest.TestCase):

    def test_addition(self):
        result = 1 + 1
        self.assertEqual(result, 2)  # بررسی اینکه نتیجه برابر با 2 است

    def test_is_true(self):
        self.assertTrue(1 < 2)  # بررسی اینکه شرط درست است

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

در اینجا از `assertEqual` برای مقایسه دو مقدار و از `assertTrue` برای بررسی اینکه یک عبارت درست است استفاده کردیم.

---

4. نحوه تست کدهای با وابستگی‌های خارجی مانند پایگاه داده یا APIها با استفاده از Mock

برای تست کدهایی که به سیستم‌های خارجی مانند پایگاه داده یا APIها متصل هستند، می‌توانیم از `Mock` برای شبیه‌سازی پاسخ‌ها استفاده کنیم و از ایجاد وابستگی‌های خارجی اجتناب کنیم.

نمونه تست کدهایی با وابستگی به پایگاه داده

فرض کنید که کد شما یک اتصال به پایگاه داده برقرار می‌کند و داده‌ای را از آن دریافت می‌کند. می‌توانید از `Mock` برای شبیه‌سازی اتصال به پایگاه داده استفاده کنید.

```python
from unittest.mock import Mock
import unittest
```

```
# تابعی که به پایگاه داده متصل می‌شود
def get_data_from_db():
    # در عمل این تابع از پایگاه داده می‌خواند
    return "Real Data"

class TestDatabaseOperations(unittest.TestCase):

    @patch('__main__.get_data_from_db', return_value="Mocked Data")
    def test_get_data(self, mock_db):
        result = get_data_from_db()
        self.assertEqual(result, "Mocked Data")  # بررسی اینکه داده‌های شبیه‌سازی‌شده برگشت داده شدند

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

در این مثال، ما از `patch` برای شبیه‌سازی تابع `get_data_from_db` استفاده کردیم تا داده‌های واقعی از پایگاه داده نخوانده شوند.

---

5. بررسی خطاها و استثناها در تست‌ها با استفاده از `()assertRaises`

گاهی اوقات در کد ما باید رفتار خاصی را هنگام وقوع استثناها آزمایش کنیم. از متد `()assertRaises` برای بررسی اینکه یک استثنا در شرایط خاص رخ می‌دهد، استفاده می‌کنیم.

نمونه استفاده از `()assertRaises`

```
import unittest

def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y

class TestMathOperations(unittest.TestCase):

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)  # بررسی اینکه هنگام تقسیم بر صفر خطای ValueError ایجاد می‌شود

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

در اینجا، `()assertRaises` بررسی می‌کند که هنگام تقسیم بر صفر، یک خطای `ValueError` رخ می‌دهد.

نتیجه‌گیری

- **Mocking** ابزاری مهم برای شبیه‌سازی وابستگی‌های خارجی و جلوگیری از نیاز به ارتباط با سیستم‌های واقعی است.
- با استفاده از ماژول `unittest.mock` می‌توانیم به‌طور موقت رفتارهای اشیاء و توابع را تغییر دهیم و تست‌های واحد را به‌طور مؤثری اجرا کنیم.
- استفاده از متدهای **assert** مانند `assertTrue`، `()assertEqual()` و `()assertRaises` به ما کمک می‌کند تا رفتار درست کد را در شرایط مختلف بررسی کنیم.

- با استفاده از **Mock** و **assert** می‌توانیم به‌طور مؤثر تست‌هایی برای کدهایی که به سیستم‌های خارجی وابسته هستند بنویسیم.