Тестування

Тестування коду - це процес перевірки програмного коду на правильність його виконання та відповідність вимогам і очікуванням. Головна мета тестування полягає у виявленні помилок, недоліків або неправильного функціонування програми. Це дозволяє розробникам вчасно виявляти та виправляти проблеми перед випуском програмного продукту.

Важливі аспекти тестування коду включають в себе:

1. **Функціональні тести:** Перевірка, чи виконує програма очікувану функціональність та задачі.
2. **Інтеграційні тести:** Перевірка взаємодії різних компонентів програми для впевненості, що вони працюють коректно разом.
3. **Одиничні тести (unit tests):** Тести, які перевіряють окремі компоненти або функції програми для впевненості в їхній правильній роботі.
4. **Навантажувальні тести:** Визначення, як програма веде себе при великих обсягах даних або при великій кількості одночасних користувачів.
5. **Автоматизовані тести:** Спеціально створені скрипти чи програми, які автоматизують процес тестування для швидшого та більш ефективного виявлення помилок.
6. **Ручні тести:** Виконання тестів вручну для виявлення недоліків, які можуть бути важкі або неможливі автоматизувати.
7. **Тестування на реальних даних:** Використання реальних або схожих на реальні дані для більш точного визначення функціональності програми.

Тестування коду є невід'ємною частиною розробки програмного забезпечення та допомагає забезпечити якість продукту перед його впровадженням у реальні умови.

Тестування - велика тема в розробці програмного забезпечення. До того як продукт потрапить до кінцевого користувача, ймовірно, він пройшов кілька тестів, таких як інтеграційні, системні та приймальні тести. Ідея такого відносно важкого

тестування полягає в тому, щоб забезпечити, що поведінка програми працює так, як очікується з точки зору кінцевого користувача. Цей підхід до тестування відомий як "розробка з урахуванням поведінки" (BDD).

Останнім часом серед розробників значно зросла зацікавленість у "розробці з урахуванням тестування" (TDD). Глибоко вдаватися в цю тему може бути складним завданням для цієї статті, але загальна ідея полягає в тому, що традиційний процес розробки та тестування розвернутий в зворотньому напрямку - спочатку ви пишете свої модульні тести, а потім впроваджуєте зміни в код до тих пір, поки тести не пройдуть.

У цій статті ми зосередимося на модульних тестах і, зокрема, на тому, як їх робити за допомогою популярного тестувального фреймворку для Python, який називається Pytest.

Що таке модульні тести?

Модульні тести - це форма автоматизованих тестів, що просто означає, що план тестування виконується сценарієм, а не вручну людиною. Вони служать першим рівнем тестування програмного забезпечення і, як правило, пишуться у вигляді функцій, які перевіряють поведінку різних функціональностей в програмному продукті.

Ідея цих тестів полягає в тому, щоб дозволити розробникам виокремити найменшу логічно обґрунтовану одиницю коду та перевірити, чи вона працює так, як очікується. Іншими словами, модульні тести перевіряють, чи працює окремий компонент програмного продукту так, як розробники задумали.

Ідеально, ці тести повинні бути досить малими - чим вони менші, тим краще. Одна з причин створення менших тестів полягає в тому, що тест буде більш ефективним, оскільки тестування менших одиниць дозволяє виконувати тестовий код набагато швидше. Ще однією причиною тестування менших компонентів є те, що це дає вам більше уявлення про те, як поводиться код на найдрібнішому рівні при об'єднанні.

Навіщо нам потрібні модульні тести?

Загальне виправдання того, чому важливо проводити модульні тести, полягає в тому, що розробники повинні забезпечити, що код, який вони пишуть, відповідає вимогам якості, перш ніж дозволити йому потрапити до середовища продукції. Однак існують і інші чинники, які призводять до необхідності модульних тестів. Розглянемо докладніше деякі з цих причин.

1. Збереження ресурсів

Проведення модульних тестів допомагає розробникам виявити помилки в коді під час етапу конструювання програмного забезпечення, запобігаючи їх подальшому розповсюдженню у життєвому циклі розробки. Це зберігає ресурси, оскільки розробникам не потрібно витрачати час і зусилля на виправлення помилок пізніше у розробці. Це також означає, що кінцевим користувачам менше ймовірно доведеться мати справу з нестабільним кодом.

1. Додаткова документація

Ще однією важливою виправданою для проведення модульних тестів є те, що вони служать як додатковий шар живої документації для вашого програмного продукту. Розробники можуть просто посилатися на модульні тести для отримання комплексного розуміння загальної системи, оскільки вони деталізують, як повинні вести себе більш важливі компоненти.

1. Підвищення впевненості

Дуже просто допустити дрібні помилки у своєму коді під час написання функціональності. Однак більшість розробників погодяться з тим, що набагато краще виявити точки вразливості в кодові перед тим, як він потрапить до середовища продукції. Модульні тести надають розробникам таку можливість.

Можна сміливо сказати, що "код, покритий модульними тестами, може вважатися надійнішим за код, який ними не охоплений". Майбутні поломки в коді можна виявляти набагато швидше, ніж в коді без тестового покриття, що економить час і гроші. Розробники також користуються додатковою документацією, щоб швидше зрозуміти кодову базу, і є додаткова впевненість у тому, що якщо вони допустять помилку у своєму коді, її виявить модульний тест, а не кінцевий користувач.

Фреймворки для тестування Python

Python надзвичайно зросла в популярності протягом останніх років. Разом із зростанням популярності Python збільшилася кількість фреймворків для тестування, що призвело до великої кількості інструментів, доступних для тестування Python-коду. Глибоко занурюватися в кожен інструмент в рамках цієї статті виходить за її межі, але ми зупинимось на деяких найпоширеніших фреймворках для тестування Python.

1. unittest

Unittest - це вбудований фреймворк Python для модульного тестування. Він натхненний фреймворком для модульного тестування під назвою JUnit мови програмування Java. Оскільки він постачається разом з мовою Python, не потрібно встановлювати додаткових модулів, і більшість розробників використовують його для вивчення тестування.

1. Pytest

Pytest, ймовірно, найпоширеніший фреймворк для тестування Python - це означає, що в нього є велика спільнота, яка допоможе вам, якщо ви застрягнете. Це відкритий фреймворк, який дозволяє розробникам писати прості, компактні набори тестів, підтримуючи модульне тестування, функціональне тестування та тестування API.

1. doctest

Фреймворк doctest поєднує дві основні складові програмної інженерії: документацію і тестування. Ця функціональність забезпечує те, що всі програми детально документовані та протестовані, щоб гарантувати їх правильну роботу. Doctest постачається разом із стандартною бібліотекою Python і є досить простим у вивченні.

1. nose2

Nose2, спадкоємиця фреймворку nose, суттєво є unittest з плагінами. Люди часто називають nose2 "розширеними модульними тестами" або "модульними тестами з плагіном" через його тісні зв'язки з вбудованим фреймворком модульного тестування Python. Оскільки це практично розширення фреймворку unittest, nose2 дуже просто прийняти для тих, хто знайомий з unittest.

1. Testify

Testify - це фреймворк Python для модульного, інтеграційного та системного тестування, який відомий як фреймворк, розроблений для заміни unittest і nose. У фреймворку є велика кількість обширних плагінів і досить гладка крива навчання, якщо ви вже знайомі з unittest.

1. Hypothesis

Hypothesis дозволяє розробникам створювати модульні тести, які простіше писати і дуже потужні при виконанні. Оскільки цей фреймворк створений для підтримки проектів у сфері науки про дані, він допомагає знаходити варіанти вхідних даних,

які не є очевидними під час створення тестів, генеруючи приклади вхідних даних, що відповідають певним властивостям, які ви визначаєте.

Для нашого навчального посібника ми використовуватимемо pytest. Дивіться на наступний розділ, щоб дізнатися, чому ви можливо захочете вибрати Pytest перед іншими, які ми вказали.

Чому використовувати Pytest?

За великою підтримкою спільноти pytest криється кілька факторів, що роблять його одним із найкращих інструментів для створення автоматизованого набору тестів в Python. Філософія та можливості pytest створені так, щоб тестування програмного забезпечення ставало набагато кращим досвідом для розробників. Один із способів, яким створювачі pytest досягли цієї мети, - значуще зменшення кількості коду, необхідного для виконання загальних завдань, і можливість виконувати розширені завдання за допомогою розгорнутих команд і плагінів.

Додаткові причини використання pytest включають наступне:

1. **Простота вивчення:** Pytest дуже простий для вивчення. Якщо ви розумієте, як працює ключове слово `assert` в Python, ви вже добре розпочинаєте вивчення цього фреймворку. Тести з використанням pytest - це просто функції Python з префіксом "test_" або суфіксом "_test" у назві функції, хоча ви також можете використовувати класи для групування кількох тестів. Загалом, навчальна крива pytest набагато менша, ніж, наприклад, у unittest, оскільки вам не потрібно вивчати нові конструкції.
2. **Фільтрація тестів:** Під час зростання вашого набору тестів може бути ситуація, коли ви не хочете виконувати всі тести кожного разу. Pytest надає три способи ізоляції тестів:
    - Фільтрація за іменем: запускати лише ті тести, імена яких відповідають заданому шаблону.
    - Орієнтування на каталог: тести виконуються лише в поточному каталозі або в підкаталогах за замовчуванням.
    - Категоризація тестів: можливість визначити категорії тестів, які повинен включати або виключати pytest.
3. **Параметризація:** Pytest включає вбудований декоратор `parametrize`, який дозволяє параметризувати аргументи для функції тесту. Це означає, що, якщо

функції, які ви тестуєте, обробляють дані або виконують загальне перетворення, вам не потрібно писати декілька схожих тестів.

4. **Менше Байлерплейту:** У відміну від unittest, який вимагає створення класів, похідних від модуля `TestCase`, та визначення в них методів тестів, pytest вимагає лише визначення функції з префіксом "test_" та використання умов `assert` всередині них. Це допомагає скоротити кількість бойлерплейту, необхідного для написання тестових кейсів.

```
"""
An example test case with unittest.
See: https://docs.python.org/3/library/unittest.html
"""
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)
```

Assert

`assert` - це ключове слово в багатьох мовах програмування, включаючи Python. Воно використовується для перевірки певних умов в коді. Якщо умова, передбачена `assert`, виявиться `False`, програма генерує виключення (assertion error) і припиняє своє виконання.

Наприклад, в Python:

```
x = 5
assert x == 5  # Це умова, яка повинна бути правдивою. Нічого не станеться.
```

Але, якщо умова не виконується:

```
y = 3
assert y == 5  # Умова не виконана, буде викинуто AssertionError.
```

Це корисний інструмент для виявлення недоліків у програмному коді, особливо під час тестування. Його часто використовують у юніт-тестуванні для перевірки, чи повертає функція очікувані результати.

Наприклад, в юніт-тестах може бути використано так:

```
def add(x, y):
    return x + y

def test_add():
    assert add(3, 5) == 8
    assert add(1, 2) == 3
    assert add(0, 0) == 0
    assert add(-1, 1) == 0
```

У цьому прикладі, якщо будь-яка з умов не виконується, програма викине виключення, що дозволить вам швидко виявити, де виникають проблеми у вашому коді.

unittest

`unittest` - це вбудований модуль в Python для написання тестів. Він надає набір інструментів для створення тестів та автоматизації їх виконання.

Основні поняття `unittest` включають:

1. **TestCase:** Це базовий клас для створення тестових наборів. Він містить набір методів для перевірки умов та порівняння значень.
2. **Test Fixture:** Це підготовка середовища для виконання тесту. Це може включати встановлення початкових значень, створення необхідних об'єктів та

інше.

3. **Test Runner:** Це інструмент, який виконує тести та збирає результати. В Python вбудований тестовий ранер, який можна викликати з командного рядка.
4. **Assertions:** Це перевірки, які вказують, що очікується від тесту. Наприклад, `assertEqual`, `assertTrue`, `assertFalse`, тощо.
5. **Test Suite:** Це група тестів, яка дозволяє вам виконати кілька тестів разом.

Приклад використання `unittest`:

```
import unittest

def add(x, y):
    return x + y

class TestAddition(unittest.TestCase):

    def test_addition(self):
        self.assertEqual(add(3, 5), 8)
        self.assertEqual(add(1, 2), 3)
        self.assertEqual(add(0, 0), 0)
        self.assertEqual(add(-1, 1), 0)

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

У цьому прикладі створено клас `TestAddition`, який успадковує `unittest.TestCase`. Метод `test_addition` містить набір перевірок, які викликають функцію `add` та порівнюють результат з очікуваним значенням.

Запуск тестів може бути викликаний з командного рядка, наприклад, за допомогою `python your_test_file.py`.

`unittest` дозволяє створювати комплексні тести та групи тестів для перевірки різних частин програми. Він є потужним інструментом для забезпечення якості програмного забезпечення.

pytest

`pytest` - це популярний фреймворк для автоматизованого тестування в Python. Він надає простий та зручний спосіб написання та виконання тестів, забезпечуючи багато корисних можливостей.

Основні переваги `pytest` включають:

1. **Простий синтаксис:** `pytest` дозволяє писати тести в дуже простому та природньому синтаксисі, що полегшує розуміння та підтримку тестів.
2. **Автоматичне виявлення тестів:** `pytest` автоматично знаходить тести в вашому коді, навіть якщо ви не використовуєте стандартний синтаксис `unittest`.
3. **Могутність параметризації тестів:** Ви можете легко параметризувати тести, щоб виконати їх з різними наборами даних.
4. **Підтримка фікстур:** Фікстури в `pytest` дозволяють підготувати середовище для виконання тестів та поділити це середовище між різними тестами.
5. **Розширені можливості ассертів:** `pytest` надає багато розширених можливостей для перевірки очікуваних результатів.

Приклад використання `pytest`:

```
# Файл тесту: test_example.py

def add(x, y):
    return x + y

def test_addition():
    assert add(3, 5) == 8
    assert add(1, 2) == 3
    assert add(0, 0) == 0
    assert add(-1, 1) == 0
```

Для запуску тестів, вам просто потрібно викликати команду `pytest` в командному рядку, надавши ім'я файлу з тестами.

`pytest` надає додаткові можливості для організації тестів, включаючи фікстури, маркери, параметризацію тестів та інше. Це робить його потужним інструментом для автоматизованого тестування в Python.