Юніт-тестування: Теорія та Концепції

1.1. Що таке Тестування Коду?

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:

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

1.2. Підходи до Розробки: BDD та TDD

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" (Behavior-Driven Development - BDD)**.

Останнім часом серед розробників значно зросла зацікавленість у **"розробці з урахуванням тестування" (Test-Driven Development - 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 цьому уроці ми зосередимося на **модульних тестах** і, зокрема, на тому, як їх робити за допомогою популярного тестувального фреймворку для Python, який називається **Pytest**.

1.3. 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.

1.4. 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.
2. **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.
3. **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.

1.5. 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.
   Основні поняття unittest включають:
   - **TestCase:** Це базовий клас для створення тестових наборів. Він містить набір методів для перевірки умов та порівняння значень.

- **Test Fixture:** Це підготовка середовища для виконання тесту. Це може включати встановлення початкових значень, створення необхідних об'єктів та інше.
- **Test Runner:** Це інструмент, який виконує тести та збирає результати. В Python вбудований тестовий ранер, який можна викликати з командного рядка.
- **Assertions:** Це перевірки, які вказують, що очікується від тесту. Наприклад, assertEqual, assertTrue, assertFalse, тощо.
- **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.

2. 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.
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.

3. 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.
4. 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.
5. 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.

6. 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**.

1.6. 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:

- **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.
- **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.
- **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.

- **Менше Бойлерплейту:** На 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.

1.7. Ключове слово 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 виняток (AssertionError) і припиняє своє виконання.

Приклад:

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

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

виняток, що дозволить вам швидко виявити, де виникають проблеми у вашому коді.

Частина 2: Практичні Приклади Pytest

Функції, які ми будемо тестувати

Зазвичай ці функції знаходяться в окремому файлі (наприклад, `app.py` або `utils.py`)

Для простоти прикладу, вони знаходяться тут.

```
def add(a, b):
    """Додає два числа."""
    return a + b

def subtract(a, b):
    """Віднімає два числа."""
    return a - b

def multiply(a, b):
    """Множить два числа."""
    return a * b

def divide(a, b):
    """Ділить два числа. Обробляє ділення на нуль."""
    if b == 0:
        raise ValueError("Ділення на нуль неможливе.")
    return a / b

def is_palindrome(text):
    """Перевіряє, чи є рядок паліндромом (читається однаково вперед і
назад)."""
    processed_text = "".join(filter(str.isalnum, text)).lower()
    return processed_text == processed_text[::-1]
```

--- Тестові функції для Pytest ---

Зазвичай ці функції знаходяться у файлах, що починаються з `test_` або

закінчуються на `_test.py`

(наприклад, `test_math_operations.py`, `test_strings.py`).

Приклад 1: Базові тести з assert

Тестування функції `add`

Щоб запустити ці тести, збережіть цей код у файл (наприклад, `test_my_app.py`)

і виконайте в терміналі: `pytest test_my_app.py`

або просто `pytest` (якщо файл знаходиться в поточному каталозі або підкаталозі,

і Pytest може його автоматично виявити).

```
def test_add_positive_numbers():
    """Тестує додавання додатних чисел."""
    assert add(2, 3) == 5

def test_add_negative_numbers():
    """Тестує додавання від'ємних чисел."""
    assert add(-1, -5) == -6

def test_add_zero():
    """Тестує додавання з нулем."""
    assert add(0, 7) == 7
    assert add(-10, 0) == -10

def test_add_float_numbers():
    """Тестує додавання чисел з плаваючою комою."""
    assert add(2.5, 3.5) == 6.0
    assert add(0.1, 0.2) == pytest.approx(0.3) # Використовуємо
pytest.approx для порівняння float
```

Приклад 2: Тестування винятків (Exceptions)

Тестування функції `divide` на ділення на нуль.

```
import pytest # Імпортуємо pytest для використання його функціоналу

def test_divide_by_zero():
    """Тестує, чи функція `divide` викликає ValueError при діленні на нуль."""
    with pytest.raises(ValueError): # Очікуємо, що буде викликано ValueError
        divide(10, 0)

def test_divide_positive_numbers():
    """Тестує ділення додатних чисел."""
    assert divide(10, 2) == 5.0

def test_divide_negative_numbers():
    """Тестує ділення від'ємних чисел."""
    assert divide(-10, 2) == -5.0
    assert divide(10, -2) == -5.0
```

Приклад 3: Тестування рядків (Паліндром)

Тестування функції `is_palindrome`.

```
def test_is_palindrome_simple():
    """Тестує прості паліндроми."""
    assert is_palindrome("madam") == True
    assert is_palindrome("racecar") == True

def test_is_palindrome_with_spaces_and_punctuation():
    """Тестує паліндроми з пробілами та пунктуацією."""
```

```
    assert is_palindrome("A man, a plan, a canal: Panama") == True
    assert is_palindrome("No lemon, no melon") == True

def test_is_palindrome_not_palindrome():
    """Тестує не-паліндроми."""
    assert is_palindrome("hello") == False
    assert is_palindrome("world") == False

def test_is_palindrome_empty_string():
    """Тестує порожній рядок (вважається паліндромом)."""
    assert is_palindrome("") == True

def test_is_palindrome_single_char():
    """Тестує рядок з одним символом."""
    assert is_palindrome("a") == True
```

Приклад 4: Параметризація тестів

Використання `@pytest.mark.parametrize` для тестування функції `multiply` з різними вхідними даними.

```
@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 6),      # Додатні числа
    (-1, 5, -5),    # Від'ємне і додатне
    (-4, -2, 8),    # Два від'ємні
    (0, 10, 0),     # Множення на нуль
    (2.5, 2, 5.0)   # Числа з плаваючою комою
])
def test_multiply_parametrized(a, b, expected):
    """Тестує функцію `multiply` з різними наборами вхідних даних."""
    assert multiply(a, b) == expected

# Приклад 5: Фікстури (Fixtures)
# Використання фікстур для підготовки даних або ресурсів, які потрібні кільком тестам.
# Фікстури забезпечують повторюваність та чистоту тестів.
```

```python
# Приклад фікстури, яка створює тимчасовий файл
@pytest.fixture
def temp_file(tmp_path):
    """
    Фікстура, яка створює тимчасовий файл для тестів.
    `tmp_path` - це вбудована фікстура Pytest для тимчасових директорій.
    """
    file_path = tmp_path / "test_data.txt"
    file_path.write_text("Це тестові дані.")
    print(f"\n[Фікстура] Створено тимчасовий файл: {file_path}")
    yield file_path # Код до yield виконується перед тестом, після yield - після тесту
    # Тут можна додати логіку очищення, якщо вона не обробляється автоматично (як для tmp_path)
    print(f"[Фікстура] Видалено тимчасовий файл: {file_path}")

def test_read_temp_file(temp_file):
    """Тестує читання з тимчасового файлу, створеного фікстурою."""
    print(f"[Тест] Читаю з файлу: {temp_file}")
    content = temp_file.read_text()
    assert "тестові дані" in content
    assert len(content) > 0

def test_write_to_temp_file(temp_file):
    """Тестує запис у тимчасовий файл, створений фікстурою."""
    print(f"[Тест] Записую у файл: {temp_file}")
    temp_file.write_text("Нові дані.")
    content = temp_file.read_text()
    assert "Нові дані" in content
```

Щоб запустити всі ці приклади, збережіть їх у файл `test_examples.py`

і виконайте `pytest` у терміналі в тому ж каталозі.