Конспект лекції (9)

▼ Загальні відомості про unittest

`unittest` в Python є вбудованим фреймворком тестування, який дозволяє створювати і виконувати тести для перевірки правильності вашого коду. В основі `unittest` лежить концепція одиниць тестування, які дозволяють перевірити правильність роботи окремих частин вашого програмного коду.

Основні етапи використання `unittest` виглядають наступним чином:

Імпорт бібліотеки:

```
import unittest
```

Створення класу тестування:

Ваш клас тестування повинен успадковувати клас `unittest.TestCase`. У цьому класі ви визначаєте різні тести за допомогою методів, які починаються зі слова "test".

```
class MyTest(unittest.TestCase):
    def test_example(self):
        # Ваш код для перевірки правильності
```

Визначення тестів:

Для визначення тестів використовуйте методи, які починаються зі слова "test". У цих методах ви можливо викликаєте ваш код і використовуєте різні методи асерції для перевірки очікуваних результатів.

```
def test_example(self):
    result = some_function()
    self.assertEqual(result, expected_result)
```

Запуск тестів:

Використовуйте вбудований метод `unittest.main()` для автоматичного виконання всіх тестів, коли скрипт запускається.

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

▼ Імпорт файлів для тесту

Зазвичай функції не пишуться в тесті, а імпортуються туди.

Найпростіше зробити імпорт файлу, що лежить в тій самій папці що й тест.

```
import unittest
from main_module import add   # у папці де створено файл тесту
                              # є файл main_module.py і в ньому
```

Проте якщо файл не лежить в поточному каталозі то python потребує “пояснення” де взяти файл.

Зробити це можна декількома методами.

1. **Динамічний імпорт модуля**

```
from importlib.machinery import SourceFileLoader

module_name = 'math'
math_module = SourceFileLoader(module_name, '/folder/math.py').load_module()

result = math_module.add(2, 3)
```

Цей код використовує клас `SourceFileLoader` з модуля `importlib.machinery` для динамічного імпорту модуля `math` з файлу `/folder/math.py`. Давайте розглянемо його крок за кроком:

1. `from importlib.machinery import SourceFileLoader`: Цей рядок імпортує клас `SourceFileLoader` з модуля `importlib.machinery`. `SourceFileLoader` є інструментом для завантаження модуля з файлу на диску.
2. `module_name = 'math'`: Задаємо ім'я модуля, яке ми хочемо завантажити. У цьому випадку це ім'я `math`.
3. `math_module = SourceFileLoader(module_name, '/folder/math.py').load_module()`: Створюємо об'єкт `SourceFileLoader`, який завантажує модуль з вказаного файлу. Метод `load_module()` викликається для завантаження самого модуля.

- `module_name` - ім'я модуля, яке буде використовуватися після завантаження.
- `/folder/math.py` - шлях до файлу, з якого ми хочемо завантажити модуль.

4. `result = math_module.add(2, 3)`: Зараз, коли модуль завантажений, ми можемо використовувати його функції чи класи. У цьому випадку викликається функція `add` з завантаженого модуля `math`.

2. **Додавання папки з модулем в шлях пошуку**

```
import sys
import pathlib
sys.path.insert(0, str(pathlib.Path(__file__).parent.parent))
from folder.main_module import add
```

Цей код використовує модулі `sys` та `pathlib`, щоб змінити шлях пошуку модулів у Python та імпортувати функцію `add` з модуля `main_module` в папці `folder`.

Розглянемо кожен рядок коду окремо:

1. `import sys`: Цей оператор імпортує модуль `sys`, який надає доступ до функціоналу, пов'язаного з виконанням Python.
2. `import pathlib`: Цей оператор імпортує модуль `pathlib`, який надає об'єктно-орієнтоване інтерфейс для роботи з шляхами файлів та каталогів.
3. `sys.path.insert(0, str(pathlib.Path(__file__).parent.parent))`: Цей рядок додає два рівні вище шлях до поточного файлу до шляху пошуку модулів Python.
   - `pathlib.Path(__file__)` отримує шлях до поточного файлу.
   - `parent.parent` двічі визначає два рівні вище каталогу від поточного файлу.
   - `str(...)` конвертує об'єкт шляху в рядок.
   - `sys.path.insert(0, ...)` вставляє цей шлях на початок списку шляхів пошуку модулів.
4. `from folder.main_module import add`: Після зміни шляху пошуку модулів у `sys.path`, цей рядок імпортує функцію `add` з модуля `main_module` в

папці `folder`.

Загальною метою цього коду є те, щоб забезпечити можливість імпортувати функцію `add` з модуля `main_module`, навіть якщо поточний файл розташований в іншій частині проекту. Це дозволяє вам структурувати свій проект і використовувати функції або класи з інших частин проекту, що знаходяться на різних рівнях вкладеності.

Перевірка тверджень

В `unittest` фреймворку для перевірки тверджень використовуються методи-асерти (assert methods), які належать до класу `unittest.TestCase`. Ці методи дозволяють вам встановлювати різноманітні твердження та перевіряти їхню правильність під час виконання тестів.

`self.assertEqual()`

`self.assertEqual()` - це метод, який використовується у тестах фреймворку `unittest` для перевірки того, чи два значення є рівними. Цей метод входить до складу класу `unittest.TestCase` і використовується для порівняння фактичного результату з очікуваним.

Ось простий приклад використання `self.assertEqual()`:

```
import unittest

def add(a, b):
    return a + b

class TestAddFunction(unittest.TestCase):
    def test_add_positive_numbers(self):
        result = add(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        result = add(-2, 3)
        self.assertEqual(result, 1)

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

`self.assertTrue()`

`self.assertTrue()` - це метод у фреймворку `unittest`, який використовується для перевірки того, чи вираз `expr` є істинним (True). Якщо вираз істинний, тест вважається успішним, в іншому випадку він

вважається неуспішним і генерується відповідне повідомлення про помилку.

Приклад використання `self.assertTrue()`:

```
import unittest

def is_even(num):
    return num % 2 == 0

class TestIsEvenFunction(unittest.TestCase):
    def test_even_number(self):
        result = is_even(4)
        self.assertTrue(result)

    def test_odd_number(self):
        result = is_even(7)
        self.assertTrue(not result)

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

▼ інші твердження

В фреймворку `unittest` є ще дуже багато засобів перевірки тверджень.

self.assertNotEqual(a, b): Перевіряє, чи `a` не дорівнює `b`.

- `self.assertNotEqual(result, unexpected_result)`

self.assertFalse(expr): Перевіряє, чи вираз `expr` є хибним.

- `self.assertFalse(is_invalid)`

self.assertIsNone(a): Перевіряє, чи `a` є `None`.

- `self.assertIsNone(result)`

self.assertIsNotNone(a): Перевіряє, чи `a` не є `None`.

- `self.assertIsNotNone(result)`

self.assertIn(a, b): Перевіряє, чи `a` входить в `b`.

- `self.assertIn(item, container)`

self.assertNotIn(a, b): Перевіряє, чи `a` не входить в `b`.

- `self.assertNotIn(item, container)`

self.assertAlmostEqual(a, b): Перевіряє, чи числа `a` та `b` приблизно рівні.

- `self.assertAlmostEqual(result, expected_result, places=2)`

▼ Вивід даних

У `unittest` фреймворку можна виводити повідомлення користувача за допомогою методу `print()`
або будь-якого іншого механізму виводу. Однак результати тестів (повідомлення про успіх чи помилку) виводяться в консоль за замовчуванням.
Крім того assert methods мають параметр `msg` в якому можна вказати помилку, яка буде виведена коли тест провалиться, наприклад:

```
self.assertEqual(result,
                 expected_result,
                 msg=f"Тест не пройшов. Результат {result} не дорівнює очікуваному."
                )
```

▼ Тестування виключень

У `unittest` фреймворку є спеціальні методи для тестування винятків. Ви можете використовувати ці методи для перевірки, чи виникає виняток під час виклику певної функції чи методу. Основні методи для тестування винятків:

1. `self.assertRaises(exception, callable, *args, **kwargs)`: Перевіряє, чи виклик `callable(*args, **kwargs)` викликає виняток типу `exception`.

   ```
   import unittest

   def divide(x, y):
       return x / y

   class TestDivision(unittest.TestCase):
       def test_divide_by_zero(self):
           with self.assertRaises(ZeroDivisionError):
               result = divide(5, 0)

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

   У цьому прикладі тестування перевіряє, чи виклик функції `divide(5, 0)` викликає `ZeroDivisionError`.

2. `self.assertWarns(warn, callable, *args, **kwargs)`: Перевіряє, чи виклик `callable(*args, **kwargs)` викликає попередження (warning) типу `warn`.

```
import unittest
import warnings

def issue_warning():
    warnings.warn("This is a warning", UserWarning)

class TestWarning(unittest.TestCase):
    def test_warning(self):
        with self.assertWarns(UserWarning):
            issue_warning()

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

У цьому прикладі тест перевіряє, чи виклик функції `issue_warning()` викликає попередження `UserWarning`.

3. `self.assertLogs(logger=None, level=None)`: Перевіряє, чи логгер реєструє повідомлення на певному рівні.

```
import unittest
import logging

class TestLogging(unittest.TestCase):
    def test_log_warning(self):
        logger = logging.getLogger(__name__)
        with self.assertLogs(logger=logger, level='WARNING'):
            logger.warning("This is a warning message")

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

У цьому прикладі тест перевіряє, чи логгер реєструє попередження (WARNING) під час виклику `logger.warning()`.

Ці методи дозволяють вам ефективно тестувати винятки та інші види повідомлень в рамках тестів вашого програмного коду.

▼ Зміна рівня деталізації виводу

Параметр `verbosity` при виклику `unittest.main(verbosity=x)` визначає рівень деталізації виведених повідомлень під час виконання тестів. Різні значення

`verbosity` вказують на різний рівень деталізації виводу.

Значення `verbosity` може бути в межах від 0 до 2:

- `verbosity=0`: Не виводить додаткових повідомлень. Виводяться тільки крапки або F/S/E/R в залежності від результатів тестів.
- `verbosity=1`: Виводить інформацію про кожен тест, якщо він пройшов чи не пройшов
- `verbosity=2`: Виводить детальну інформацію про кожен тест, включаючи імена тестів та час виконання.

Параметр `verbosity` є корисним при відладці або аналізі результатів тестів, оскільки він дозволяє збільшити обсяг інформації, яку ви бачите на консолі під час виконання тестів.

Якщо ви не вказуєте `verbosity` (або вказуєте `verbosity=None`), тоді використовується значення за замовчуванням, яке часто є `1` або іншим значенням, що визначається залежно від реалізації конкретного тестувального середовища або фреймворку.

▼ Підміна реального коду – unittest.mock

Однією з причин використання фіктивних об’єктів Python є ситуація, коли об’єкти для тестування можуть бути відсутні, є лише вимоги до їх реалізації.

Ще може бути ситуація, коли тимчасова зміна в поведінці зовнішніх служб може призвести до періодичних збоїв у вашому наборі тестів.

Приклад використання

Давайте розглянемо приклад використання `unittest.mock` для тестування коду, який взаємодіє з API-ендпоінтом (наприклад, HTTP-запитом). Для цього ми можемо використати бібліотеку `requests` для виконання запитів, а `unittest.mock` - для створення макетів об'єктів, які взаємодіють з цими запитами.

Давайте розглянемо клас, який використовує `requests` для виконання запиту до API-ендпоінту і повертає результат:

```
import requests

class APIClient:
    def __init__(self, base_url):
```

```
        self.base_url = base_url

    def get_data(self):
        url = f"{self.base_url}/data"
        response = requests.get(url)
        return response.json() if response.status_code == 200 else None
```

Тепер ми можемо написати тести для цього класу, використовуючи макети для емуляції відповідей від API-ендпоінта. Для цього використаємо `unittest.mock.patch`:

```
import unittest
from unittest.mock import patch, Mock
from my_module import APIClient

class TestAPIClient(unittest.TestCase):
    @patch('requests.get')
    def test_get_data_success(self, mock_get):
        # Створюємо макет відповіді API-ендпоінта
        mock_response = Mock()
        mock_response.status_code = 200
        mock_response.json.return_value = {'data': 'example_data'}

        # Встановлюємо макет для функції get() з бібліотеки requests
        mock_get.return_value = mock_response

        # Тестуємо метод get_data() з класу APIClient
        api_client = APIClient(base_url='https://api.example.com')
        result = api_client.get_data()

        # Перевіряємо, чи метод get() був викликаний з очікуваним URL
        mock_get.assert_called_once_with('https://api.example.com/data')

        # Перевіряємо результат
        self.assertEqual(result, {'data': 'example_data'})

    @patch('requests.get')
    def test_get_data_failure(self, mock_get):
        # Створюємо макет відповіді API-ендпоінта для
        # симуляції невдачі (status_code != 200)
        mock_response = Mock()
        mock_response.status_code = 404

        # Встановлюємо макет для функції get() з бібліотеки requests
        mock_get.return_value = mock_response

        # Тестуємо метод get_data() з класу APIClient при невдачі
        api_client = APIClient(base_url='https://api.example.com')
        result = api_client.get_data()

        # Перевіряємо, чи метод get() був викликаний з очікуваним URL
        mock_get.assert_called_once_with('https://api.example.com/data')
```

```
        # Перевіряємо, що результат - None при невдачі
        self.assertIsNone(result)

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

У цьому прикладі ми використали `patch` для створення макета функції `requests.get`, яку використовує наш клас `APIClient`. Таким чином, ми можемо емулювати відповіді від API-ендпоінта і тестувати різні сценарії викликів, включаючи успішний та невдалий
запити.

Покроковий розбір коду

Зазначений приклад використовує бібліотеку `unittest.mock` для тестування класу `APIClient`, який взаємодіє з API-ендпоінтом за допомогою бібліотеки `requests`. Давайте розглянемо його крок за кроком:

```
import unittest
from unittest.mock import patch, Mock
from my_module import APIClient
```

Імпорт бібліотек: Починаємо з імпорту необхідних бібліотек. `unittest.mock` містить інструменти для створення макетів об'єктів.

Клас `APIClient`:

```
class APIClient:
    def __init__(self, base_url):
        self.base_url = base_url

    def get_data(self):
        url = f"{self.base_url}/data"
        response = requests.get(url)
        return response.json() if response.status_code == 200 else None
```

Цей клас `APIClient` має метод `get_data`, який використовує бібліотеку `requests` для виконання GET-запиту до API-ендпоінту та повертає JSON-дані в разі успіху або `None` у випадку невдачі.

Тести:

```
class TestAPIClient(unittest.TestCase):
```

Визначаємо клас тестів, який успадковується від `unittest.TestCase`.

```
    @patch('requests.get')
    def test_get_data_success(self, mock_get):
```

Визначаємо тестовий метод для успішного виклику `get_data`. Декоратор `@patch('requests.get')` вказує, що ми хочемо макетувати (замінити) функцію `requests.get`. В параметрі `mock_get` буде створений макет.

```
        # Створюємо макет відповіді API-ендпоінта
        mock_response = Mock()
        mock_response.status_code = 200
        mock_response.json.return_value = {'data': 'example_data'}
```

Створюємо макет відповіді API-ендпоінта для симуляції успішного запиту.

```
        # Встановлюємо макет для функції get() з бібліотеки requests
        mock_get.return_value = mock_response
```

Встановлюємо макет для функції `requests.get`. Тепер будь-який виклик `requests.get` під час виконання тесту буде повертати наш макетований об'єкт відповіді.

```
        # Тестуємо метод get_data() з класу APIClient
        api_client = APIClient(base_url='https://api.example.com')
        result = api_client.get_data()
```

Створюємо екземпляр `APIClient` та викликаємо його метод `get_data`.

```
        # Перевіряємо, чи метод get() був викликаний з очікуваним URL
        mock_get.assert_called_once_with('https://api.example.com/data')
```

Викликаємо метод `assert_called_once_with` на макеті, щоб перевірити, чи функція `requests.get` була викликана з очікуваним URL.

```
        # Перевіряємо результат
        self.assertEqual(result, {'data': 'example_data'})
```

Перевіряємо, чи результат відповідає очікуваним даним.

Аналогічно реалізований тест для випадку, коли відповідь має статус-код не рівний 200:

```
    @patch('requests.get')
    def test_get_data_failure(self, mock_get):
        # Створюємо макет відповіді API-ендпоінта
        # для симуляції невдачі (status_code != 200)
        mock_response = Mock()
        mock_response.status_code = 404

        # Встановлюємо макет для функції get() з бібліотеки requests
        mock_get.return_value = mock_response

        # Тестуємо метод get_data() з класу APIClient при невдачі
        api_client = APIClient(base_url='https://api.example.com')
        result = api_client.get_data()

        # Перевіряємо, чи метод get() був викликаний з очікуваним URL
        mock_get.assert_called_once_with('https://api.example.com/data')

        # Перевіряємо, що результат - None при невдачі
        self.assertIsNone(result)
```

Запуск тестів

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

Вказуємо Python, що потрібно виконати тести, якщо цей файл запускається як основний скрипт.

У цьому прикладі ми використали `unittest.mock` для створення макетів функцій бібліотеки `requests`, щоб забезпечити контрольоване середовище для тестування взаємодії з API-ендпоінтом у класі `APIClient`. Такий підхід дозволяє нам ефективно тестувати різні сценарії, включаючи успішні та невдалі випадки.