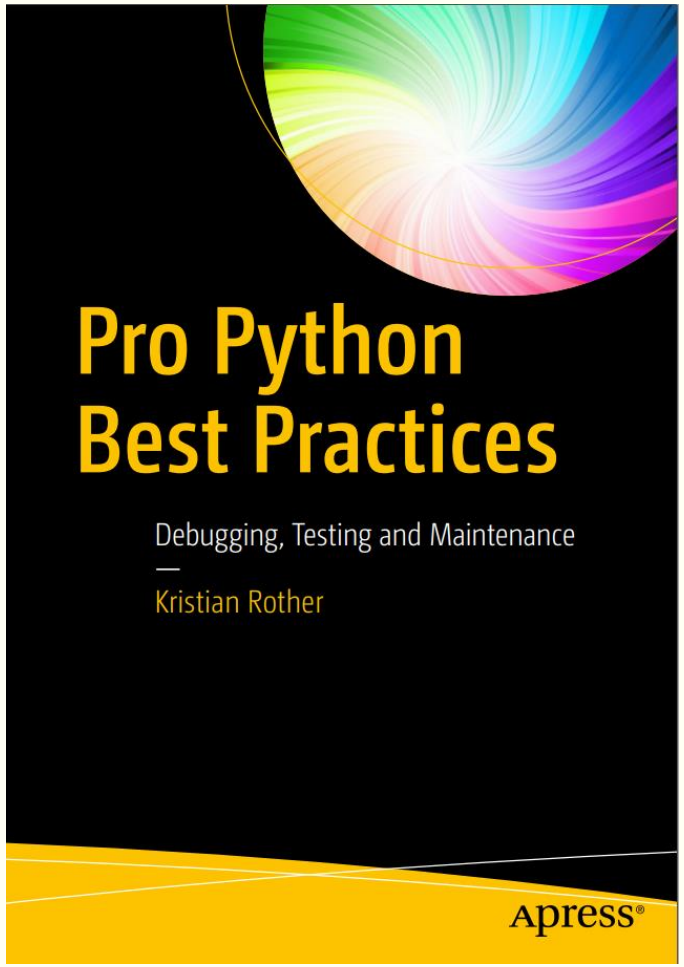
ОСНОВИ МОДУЛЬНОГО ТЕСТУВАННЯ КОДУ МОВОЮ PYTHON

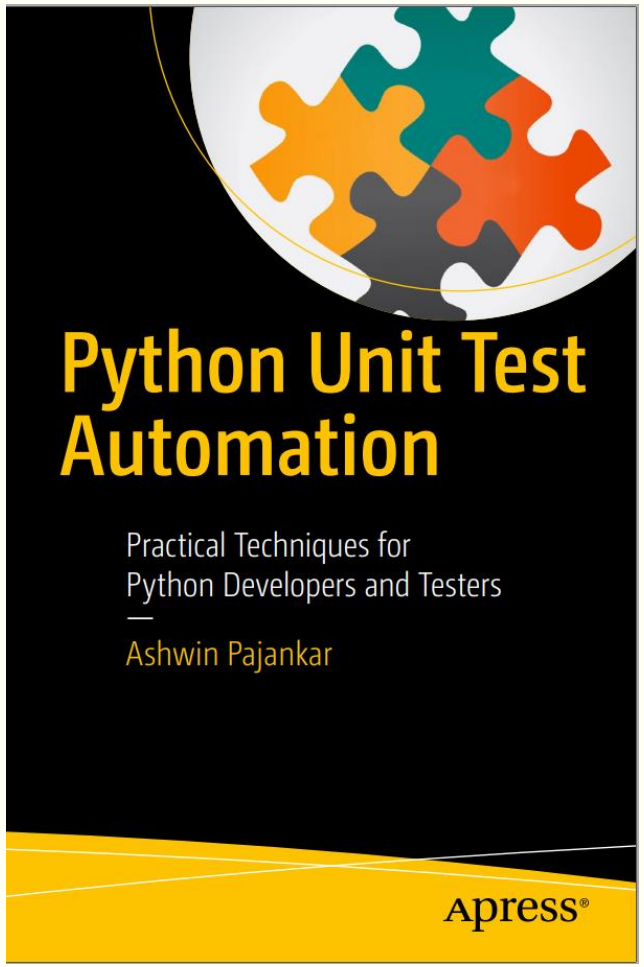
Тема 08

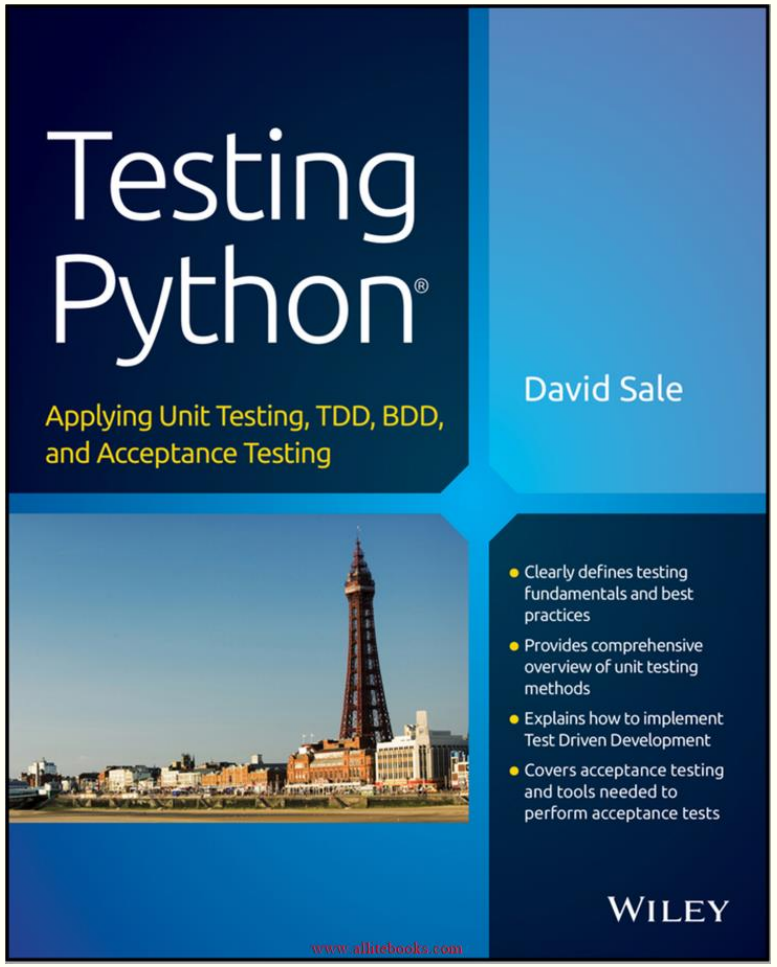
План лекції

- Основні поняття в галузі тестування програмного забезпечення
- Організація даних для тестування
- Написання тестових наборів

Рекомендована література





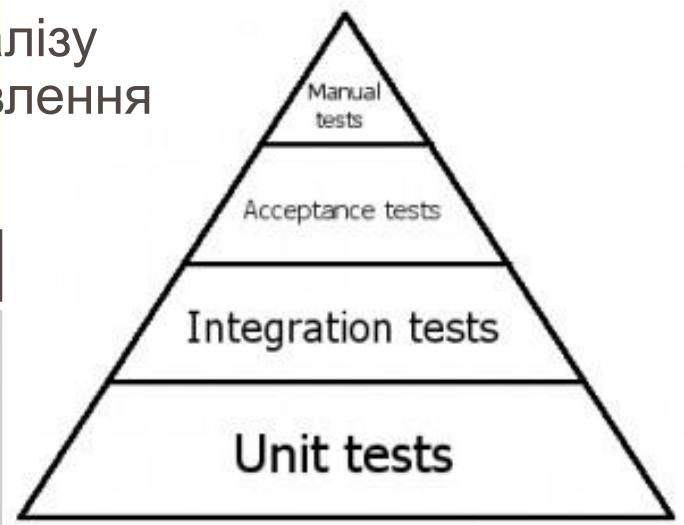


ОСНОВНІ ПОНЯТТЯ В ГАЛУЗІ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Питання 8.1

Мануальне та автоматизоване тестування

- ***Тестування програмного забезпечення*** – це процес аналізу програмного засобу та супутньої документації з метою виявлення дефектів та підвищення якості продукту.



	Мануальне тестування	Автоматизоване тестування
Переваги	✓ Будь-хто може тестувати ✓ Найпростіший спосіб покращити якість ✓ Доречне для людей без формального досвіду в тестуванні ✓ В основному концентрується на робочому процесі клієнта	✓ Швидші цикли тестування ✓ Може знайти більше дефектів за коротший період часу ✓ Зберігає кошти компанії після певної кількості тестових прогонів ✓ Чудовий спосіб для відповідності процесам при гнучкій розробці ✓ Вища якість продукту в порівнянні з мануальним тестуванням
Недоліки	− Може не ідентифікувати всі тестові випадки − Може не ідентифікувати всі дефекти − Загалом нижча якість результуючого продукту	− Скрипти для тестування зазвичай пишуть відповідні спеціалісти − Вимагає мануальних тестових випадків при первинній автоматизації − Вимагає спеціальну тестову платформу − Більше витрат при менш, ніж 3 циклах тестування.

Види автоматизованих тестів

Тип тестів	Опис
Модульні тести (Unit Test)	Тестують малі, ізольовані блоки коду.
Інтеграційні тести (Integration Test)	Тестують співпрацю кількох більших програмних компонентів.
Приймальні тести (Acceptance Test)	Тестують функціональність з точки зору користувача.
Регресійні тести (Regression Test)	Тести, що повторно запускаються для перевірки працездатності раніше протестованого коду.
Тести продуктивності (Performance Test)	Тестують швидкодію, використання пам'яті та інші метрики продуктивності.
Тести завантаженості (Load Test)	Тестують продуктивність під високим навантаженням, особливо для веб-серверів.
Стрес-тести	Тестують функціональність під форс-мажорними ситуаціями (відмова «заліза», атаки тощо)

Модульні тести

- Тести для одної функції, класу або модуля.
  - Для єдиної частини коду підтверджують виконання нею базових вимог.
  - Модульні тести зазвичай покривають багато межових випадків: порожній ввід, занадто довгий ввід, некоректний ввід даних тощо.
- Рекомендації до написання модульних тестів представлені Tim Ottinger та Jeff Langr.
  - **F**ast—швидке виконання, максимум кілька секунд.
  - **I**solated—тестується тільки один шматок коду за раз.
  - **R**epeatable—тест може повторно запускатись з тим же результатом.
  - **S**elf-verifying—набір тестів (Test Suite) не потребує додаткової інформації, щоб оцінити тест.
  - **T**imely—спочатку пишуться тести, а потім – код (Test-Driven Development, TDD)

```python
# test_port1.py

import unittest
from portfolio1 import Portfolio

class PortfolioTest(unittest.TestCase):
    def test_buy_one_stock(self):
        p = Portfolio()
        p.buy("IBM", 100, 176.48)
        assert p.cost() == 17648.0
```

- У Python модульний тест – це метод всередині тестового класу, який представляє собою тестовий набір (Test Suite).

```
$ python -m unittest test_port1
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Використання Docstring

- Docstring – рядковий літерал, який є першою інструкцією в оголошенні модуля, функції, класу чи метода.
  - Використовується для документування конкретної частини коду.
  - Підчас розбору коду компілятор пропускає коментарі, проте docstring залишається.
  - docstring отримує спеціальний атрибут __doc__ для об'єкту.
  - Основна превага – доступність під час виконання програми.

Детальніше:

- https://www.python.org/dev/peps/pep-0256
- https://www.python.org/dev/peps/pep-0257
- https://www.python.org/dev/peps/pep-0258

```
1  """
2  Це test_module01.
3  Це приклад багаторядкового docstring-а.
4  """
5  class TestClass01:
6      """Ви в TestClass01."""
7
8      def test_case01(self):
9          """Ви в test_case01()."""
10
11 def test_function01():
12     """Ви в test_function01()."""
```

```
In [13]: import test_module01

In [14]: help(test_module01)
Help on module test_module01:

NAME
    test_module01

DESCRIPTION
    Це test_module01.
    Це приклад багаторядкового docstring-а.

CLASSES
    builtins.object
        TestClass01

    class TestClass01(builtins.object)
     |  Ви в TestClass01.
     |
     |  Methods defined here:
     |
     |  test_case01(self)
     |      Ви в test_case01().
     |
     |  ----------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    test_function01()
        Ви в test_function01().

FILE
    c:\users\spuasson\desktop\test_module01.py
```

Структура модульного тесту. Паттерн AAA

- Назва тесту: ***def test_buy_one_stock(self):***
  - Має бути описовою. Хороша назва може замінити docstring!
- Рекомендований шаблон написання модульного тесту – «Arrange-Act-Assert»
  - ***Arrange*** – блок коду, який налаштовує умови виконання для тесту.
  - ***Act*** – уривок коду, який запускає на виконання тестований блок коду. Рекомендується зберігати результат у змінну result та робити код виклику однорядковим.
  - ***Assert*** – перевірка отриманого результату на відповідність очікуванням

```
def test_buy_one_stock(self):
    p = Portfolio()
    p.buy("IBM", 100, 176.48)
    assert p.cost() == 17648.0
```

Arrange
Act
Assert

  - Для мови Python існує статичний аналізатор (лінтер) тестів - Flake8.
- Поширена практика написання модульних тестів – використовувати лише одну assert-перевірку на тест та використовувати об’єкти-заглушки (mock objects) для заміни складних програмних компонентів простішими структурами.

Інтеграційні тести

- Тестують взаємодію між програмними модулями.
  - Зазвичай, між великими компонентами: базою даних, веб-сервером, стороннім додатком тощо.
  - Також програмне забезпечення може тестуватись з різними версіями бібліотек та Python.
  - На цьому етапі не використовуються об’єкти-заглушки чи інші замісники коду.
  - Інтеграційні тести намагаються відтворити середовище, в якому працюватиме програмне забезпечення, яке розробляється.
- Для мови Python доступні кілька інструментів інтеграційного тестування:
  - Tox – підтримує запуск набору тестів на різних версіях платформи Python.

Приймальні (acceptance) тести

- Перевіряють, чи певна функціональність працює так, як було заявлено.
  - Типовий приймальний тест запускає всю програму та перевіряє результати роботи кількох програмних характеристик.
  - Імітується поведінка та дії реального користувача, проте абсолютно всі ситуації не розглядаються – для цього було модульне тестування.
  - Такі тести перевіряють, чи програма обробляє вхідні дані та забезпечує бажаний результат за припущення, що всі окремі компоненти працюють коректно.
- Реалізація приймальних тестів сильно залежить від користувацького інтерфейсу програми.
  - При розробці бібліотеки коду тести будуть подібні до модульних, проте тестуватимуть інтерфейс для взаємодії з користувачем.
  - Для графічного інтерфейсу тестування набагато складніше. Для веб-інтерфейсів існують спеціальні інструменти приймального тестування - Selenium, Cucumber, Codecept.js тощо.
- Приймальне тестування не усуває потребу в мануальному тестуванні – комунікація програмістів з користувачами важлива для визначення релевантної функціональності додатку.

Регресійні тести

- Передбачають повторний запуск раніше пройдених тестів, щоб перевірити, чи внесені зміни не порушили роботу протестованих модулів.
  - Можуть включати модульні, інтеграційні та приймальні тести.
  - Швидкий набір тестів може запускатись навіть кожні кілька хвилин.
- Регресійне тестування запускається у кількох стандартних ситуаціях:
  - Після додавання нової функціональності.
  - Після виправлення дефекту.
  - Після рефакторингу (реорганізації) коду.
  - Перед комітингом коду в репозиторій.
  - Після отримання (check out) коду з репозиторію.

Тести продуктивності (Performance Tests)

- Попередні тести перевіряли функціональність; тести продуктивності перевіряють ефективність роботи програми: швидкість, респонсивність, ресурсні витрати тощо.
- Python пропонує кілька інструментів для ручного тестування продуктивності.
  - Наприклад, магічний метод %timeit в IPython:

```
In [1]: from generate_maze import create_maze
In [2]: create_maze(10, 10)
In [3]: %timeit create_maze(10, 10)
1000 loops, best of 3: 589 s per loop
```

  - Використання timeit краще, ніж зовнішньої програми для замірів часу.

```
In [4]: %timeit range(100)
The slowest run took 9.16 times longer than the fastest.
This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 392 ns per loop
```

  - Імпортуючи модуль timeit, можна виміряти час виконання будь-якої Python-команди всередині програми:

```
def test_fast_maze_generation():
    """Maze generation is fast"""
    seconds = timeit.timeit("create_maze(10, 10)", number=1000)
    assert seconds <= 1.0
```

Оптимізація продуктивності коду

```
def create_maze(xsize, ysize):
    """Returns a xsize*ysize maze as a string"""
    dots = generate_dot_positions(xsize, ysize)
    maze = create_grid_string(dots, xsize, ysize)
    return maze
```

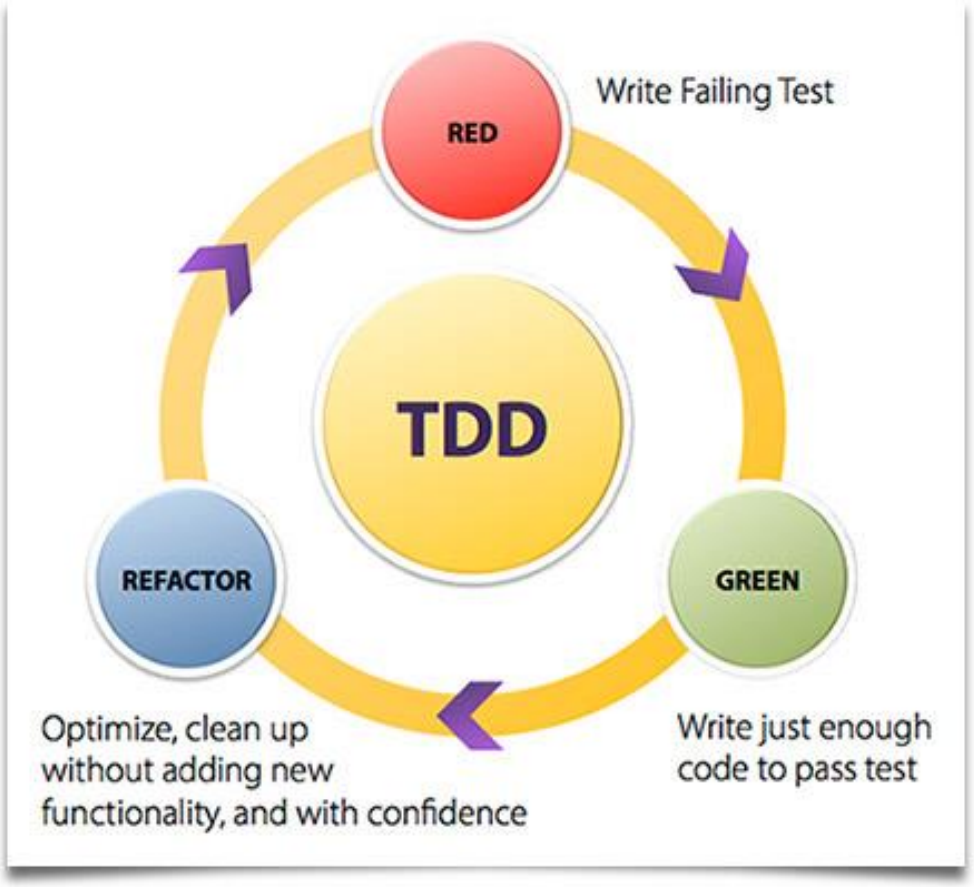
- Модуль cProfile дозволяє детальніше перевірити продуктивність коду.

```
cProfile.run("create_maze(200, 200)")
         395494 function calls in 19.689 seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filename:lineno(function)
     1   0.003   0.003  19.689  19.689 <string>:1(<module>)
     1   0.000   0.000   0.009   0.009 generate_maze.py:22(get_all_dot_pos)
     1   0.009   0.009   0.009   0.009 generate_maze.py:24(<listcomp>)
 39204   0.078   0.000   0.078   0.000 generate_maze.py:27(get_neighbors)
     1   0.262   0.262  19.670  19.670 generate_maze.py:35(generate_dot_pos)
 39204   0.126   0.000   0.126   0.000 generate_maze.py:42(<listcomp>)
     1   0.000   0.000  19.686  19.686 generate_maze.py:49(create_maze)
     1   0.017   0.017   0.017   0.017 generate_maze.py:9(create_grid_string)
 ...
 39204  18.914   0.000  18.914   0.000 {method 'remove' of 'list' objects
```

- Стовпчик ncalls вказує, скільки разів викликалась функція.
- tottime – загальний час, проведений всередині функції, далі – час роботи одного виклику (percall).
- cumtime – кумулятивний час, проведений всередині функції та всіх викликаних з неї функцій, далі – час у конкретній функції (percall).

Розробка через тестування (Test-Driven Development, TDD)

- Базується на концепції test-first з екстремального програмування.
- Процедура, яка управляє циклами розробки, називається red-green-refactor:
  - 1. Напишіть тест.
  - 2. Запустіть усі тести.
  - 3. Напишіть код реалізації.
  - 4. Запустіть усі тести.
  - 5. Виконайте рефакторинг.
  - 6. Запустіть усі тести.



Обмеження автоматизованого тестування

- Вимагає структуру коду, яку можливо тестувати.
  - Розбиття коду на функції та класи спрощує тестування.
- Тестування погано працює на динамічно еволюціонуючих проектах.
  - Наприклад, для дослідницьких проектів, які оперують даними з файлів та баз даних і генерують графіки та діаграми, писати автоматизовані тести даремно.
  - Більш доречними будуть альтернативи: прототипування, огляд коду (code review) тощо.
  - Повсякчасні зміни в коді програми можуть свідчити про її погане структурування.
- Тестування не доводить коректності коду.
  - Неможливо протестувати все, навіть з розглядом найбільш пограничних випадків.
  - Покриття коду 100% не означає повного його тестування.
  - Навіть у програмі з невеликою кількістю галужень число шляхів виконання перевищує кількість ситуацій, які можна відтестувати на практиці.
  - Дослідження показують, що це нормально, якщо програма містить невідомі дефекти.

Програми, які складно тестувати

- Програми з випадковими числами.
  - Доречний підхід – контролювати зерно генератора псевдовипадкових чисел (random.seed()), що дозволяє спрогнозувати результати генерації.
- Графічні інтерфейси користувачів.
  - Важко написати автоматизований тест щодо змістовності візуалізації: білий текст на світло-сірому фоні може бути коректним з боку тесту, проте даремним для використання.
  - Рекомендований підхід – архітектурно відокремити візуальні компоненти додатку від його логіки, після чого тестувати логіку за допомогою інтерфейсу командного рядка.
- Програми зі складним чи великим виводом.
  - Наприклад, результатом роботи є генерація великих текстових чи мультимедійних файлів.
  - Стратегія 1: використовувати готові бібліотеки з підтримкою різних форматів збереження даних.
  - Стратегія 2: писати тести для невеликого датасету-зразка.
- Багатопоточний код.
  - Баг може з’являтись та зникати залежно від впливу іншого потоку. Такі баги називають гейсенбагами.
  - Рекомендація – добре розібрати питання багатопоточності та роботу з певною бібліотекою в цьому контексті - asyncio, gevent, Twisted тощо.

Альтернативи автоматизованому тестуванню

- ***Прототипування*** – розробка первинної, концептуальної версії програми.
  - Наступна версія програми буде більш ретельно спроектована та усувати недоліки прототипу.
  - Саме тут автоматизоване тестування може знадобитись.
- ***Огляд (review) коду*** іншою особою.
  - Оглядач може задати багато наївних запитань, які вкажуть на моменти, що раніше не розглядались.
  - Досвідчений оглядач вкаже не лише на дефекти, але й заплутані інструкції чи слабкості програмної архітектури.
  - Можлива організація – порядковий перегляд коду з поясненнями розробника.
- ***Чеклісти*** – серії кроків, запитань чи нагадувань, які послідовно перевіряються. Елементи чеклісту можуть бути наступні:
  - Типові баги для усунення: «Чи всі змінні було ініціалізовано?»
  - Етапи релізу програмного забезпечення: «Створити zip-архів. Завантажити на сторінку проекту»
  - Явні мануальні тести: «Чи можливо завантажити та розпакувати zip-файл?»

ДЯКУЮ ЗА УВАГУ!

Наступне питання: організація даних для тестування