Звіт
До лабораторної роботи №1
З предмету
“Інструментальні засоби розробки програмного забезпечення”

Виконав:
Студент групи ІПС-22
Рибаков Денис Олександрович

Київ 2025

Тема: Практичне застосування системи контролю версій Git та юніт-тестування у процесі розробки програмного забезпечення. Формування повного робочого циклу з GitHub — від створення репозиторію до Pull Request.

Мета: Отримати практичні навички використання сучасних інструментів розробки ПЗ.

Посилання на репозиторій: https://github.com/RybakovDenys/dev-tools-lab-1

Теоретичний вступ

Керування проєктом: Git та GitHub

Система контролю версій (VCS) є інструментом, що дозволяє відстежувати зміни у файлах проєкту впродовж часу. **Git** — це найпопулярніша на сьогодні розподілена VCS. Вона дозволяє розробникам зберігати "знімки" стану проєкту, які називаються комітами (commits).

GitHub — це веб-сервіс, який надає хостинг для Git-репозиторіїв. Він розширює можливості Git, додаючи інструменти для візуалізації, обговорення коду та управління проєктами. Ключовим елементом співпраці на GitHub є **Pull Request** (PR). Це запит на включення (злиття) змін із однієї гілки до іншої. PR є центральним місцем для проведення огляду коду (code review), автоматичного запуску тестів та обговорення запропонованих змін перед тим, як вони потраплять до основної гілки.

Юніт-тестування

Юніт-тестування — це метод тестування програмного забезпечення, при якому окремі компоненти або модулі (тобто "юніти") програми тестуються ізольовано. Мета полягає в тому, щоб переконатися, що кожна окрема частина коду працює коректно.

У об'єктно-орієнтованому програмуванні "юнітом" найчастіше є окремий метод класу. Тести пишуться для перевірки:

> **Очікуваної поведінки**: Чи повертає метод правильний результат за типових вхідних даних.

Граничних випадків (Edge Cases): Як поводиться код на межах допустимих значень (наприклад, порожня множина, максимальний індекс, краї поля).

Виняткових ситуацій: Чи коректно обробляються помилкові дані (наприклад, передача неправильного типу даних).

Використання фреймворків, таких як unittest у Python, дозволяє автоматизувати цей процес, створюючи набір тестів, які можна запускати після будь-яких змін у коді. Це гарантує, що нові зміни не зламали існуючу функціональність.

«Гра 'Життя'» Конвея

Об'єктом для тестування у цій роботі є програмна реалізація «Гри 'Життя'» (Conway's Game of Life). Це відомий клітинний автомат, що симулює еволюцію популяції клітин на двовимірній сітці (полі).

Основна логіка: Стан кожної клітини (жива чи мертва) у наступному поколінні (кроці) визначається станом її 8 сусідів за трьома простими правилами:

Виживання: Жива клітина з 2 або 3 живими сусідами виживає.

Смерть: Жива клітина помирає від "самотності" (< 2 сусідів) або "перенаселення" (> 3 сусідів).

Народження: Мертва клітина з рівно 3 живими сусідами стає живою.

Особливості реалізації:

- Клас **GameOfLife** зберігає стан поля, використовуючи множину set координат (x, y) лише живих клітин.

- Метод **step()** обчислює наступне покоління на основі правил.

- Клас підтримує два режими:

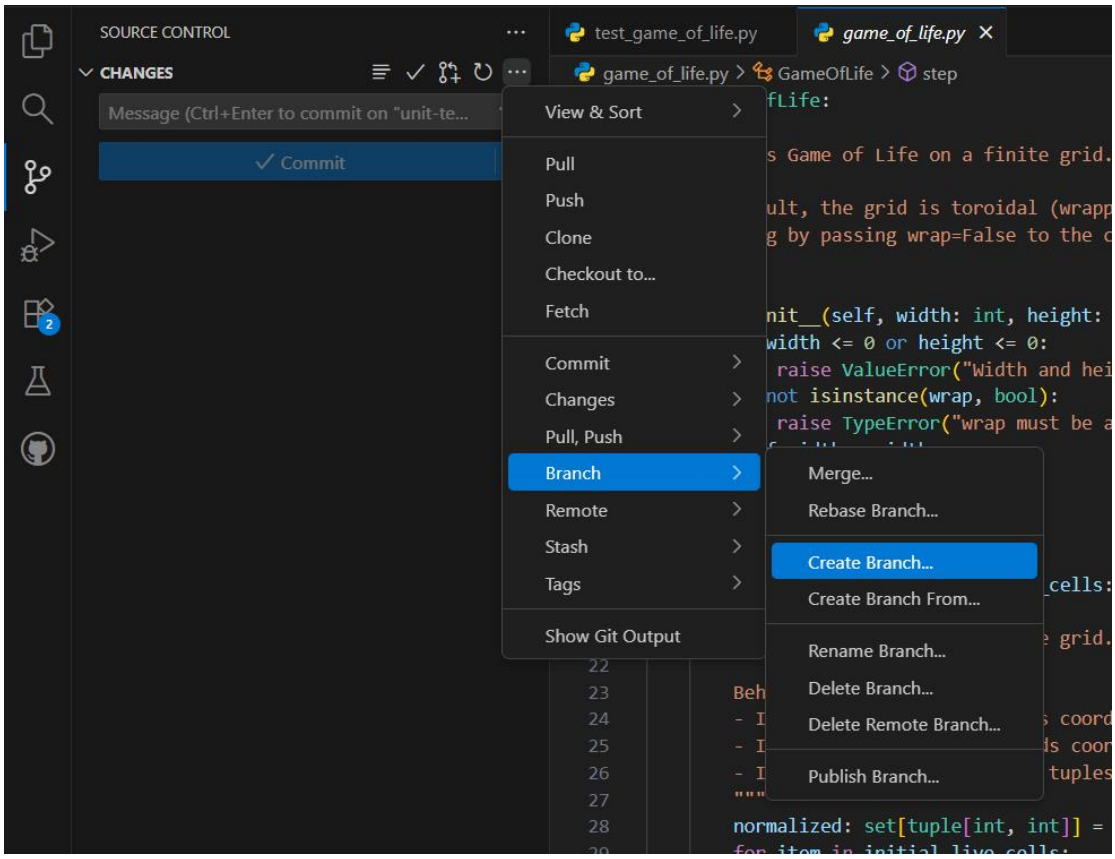
Тороїдальний (wrap=True): Поле "зациклене", тобто крайні праві клітини є сусідами крайніх лівих, а верхні — сусідами нижніх.

Обмежений (wrap=False): Поле має жорсткі кордони, за якими клітин не існує.

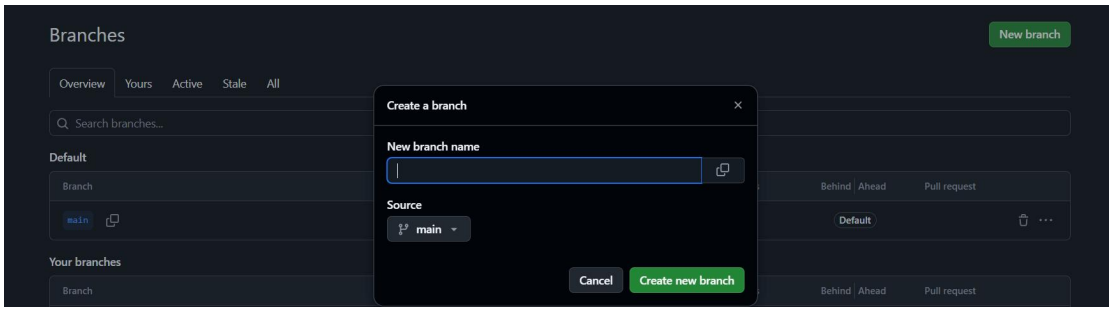
Основні етапи роботи

1. Створення гілки

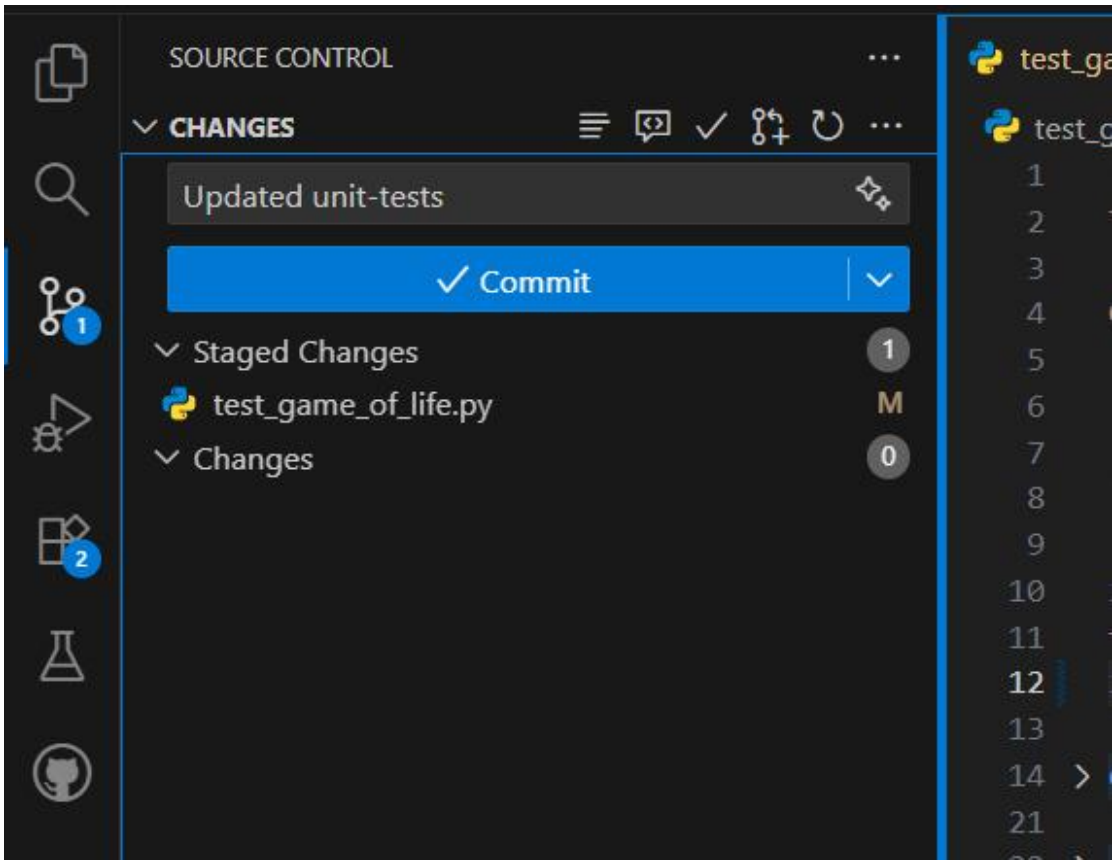
Через VSCode



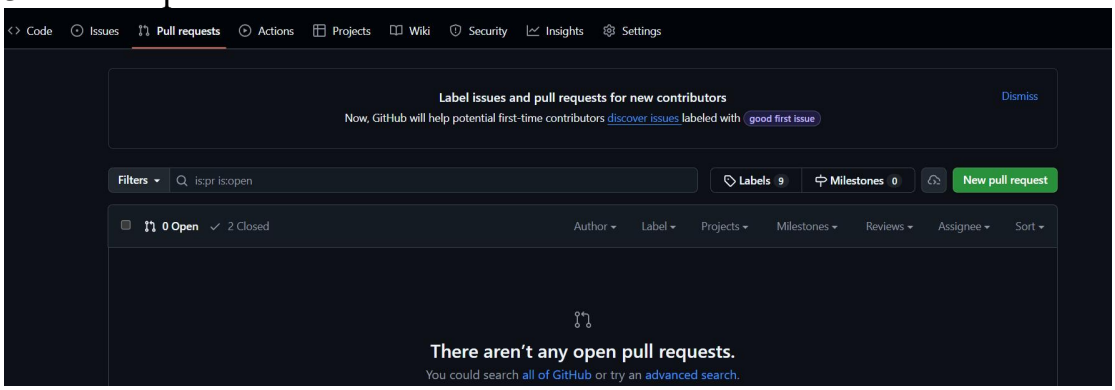
Через GitHub

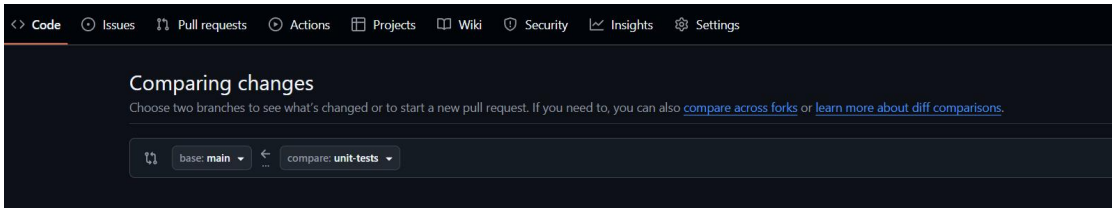


2. Коміти



3. Pull Request





Опис написання юніт-тестів

Для виконання лабораторної роботи я обрав реалізацію «Гри 'Життя'» Конвея мовою Python. Для написання тестів я використав вбудований фреймворк unittest.

Процес тестування був структурований для поступового покриття всієї функціональності класу GameOfLife, від базової логіки до складних граничних випадків.

Тестування валідації та початкового стану

Насамперед я створив тести для перевірки коректності роботи конструктора (__init__) та методу set_state.

test_constructor_validation: Цей тест переконується, що клас GameOfLife коректно генерує винятки ValueError при спробі створення поля з нульовими або від'ємними розмірами (наприклад, GameOfLife(0, 10)), а також TypeError, якщо параметр wrap не є булевим.

test_set_state_rejects_invalid_types: Перевіряє, що метод set_state відхиляє некоректні дані (наприклад, множини, що містять не-кортежі, кортежі неправильної довжини або з не-числовими значеннями), генеруючи ValueError.

test_get_state_is_defensive_copy: Важливий тест, який підтверджує, що get_state() повертає копію внутрішнього стану (множини live_cells), а не посилання на нього. Це гарантує, що зовнішній код не може випадково змінити стан гри в обхід її методів.

Тестування основних правил гри

Далі я реалізував набір тестів, що перевіряють кожне з чотирьох основних правил «Життя» ізольовано:

test_underpopulation: Перевіряє, що жива клітина з одним сусідом (або нулем) помирає.

test_survival_2_neighbors / test_survival_3_neighbors: Перевіряє, що жива клітина з двома або трьома сусідами виживає на наступному кроці.

test_overpopulation: Перевіряє, що жива клітина з чотирма (або більше) сусідами помирає.

test_reproduction: Перевіряє, що мертва клітина, оточена рівно трьома живими сусідами, "народжується".

test_stasis_dead_cell: Додатково перевіряє, що мертва клітина (наприклад, з 2 або 4 сусідами) залишається мертвою.

Тестування відомих патернів

Після перевірки базових правил я перейшов до тестування відомих патернів, щоб переконатися, що їхня сукупна поведінка відповідає очікуванням.

Стабільні фігури ("Still Lifes"): test_still_life_block, test_still_life_beehive, test_still_life_boat. Ці тести встановлюють початковий стан фігури, викликають game.step() і переконуються, що кінцевий стан get_state() точно дорівнює початковому.

Осцилятори ("Oscillators"): test_oscillator_blinker_period_2, test_oscillator_toad_period_2. Ці тести перевіряють, що патерн повертається до свого початкового стану після N кроків (у даному випадку N=2).

Космічні кораблі ("Spaceships"): test_spaceship_glider_move. Цей тест перевіряє, що після 4 кроків глайдер не просто виживає, а й зміщується по діагоналі на (1, 1) від своєї початкової позиції (для чого була використана допоміжна функція translate).

Тестування граничних випадків (Toroidal vs Bounded)

Найважливішою частиною тестування була перевірка логіки "зациклення" поля (wrap=True) та обмеженого поля (wrap=False).

Тести wrap=True (Тороїдальне поле):

test_torus_reproduction_top_left_corner: Спеціально перевіряє народження клітини (0, 0) за рахунок сусідів, що "зациклюються" з протилежних країв поля (наприклад, (9, 9), (0, 9), (9, 0)).

test_torus_overpopulation_right_edge: Перевіряє смерть клітини на правому краю (9, 5) через сусідів, що включають клітини на лівому краю (0, 4), (0, 5).

test_torus_blinker_on_edge: Перевіряє, що "Блінкер", розміщений на межі поля (координати 9, 0, 1), коректно осцилює, використовуючи зациклення.

test_set_state_wraps_out_of_bounds: Переконується, що set_state при wrap=True автоматично нормалізує координати за межами поля (наприклад, (10, 10) стає (0, 0) на полі 10x10).

Тести wrap=False (Обмежене поле):

test_full_board_bounded_differs: Цей тест демонструє ключову відмінність логіки. На відміну від test_full_wrapping_board_dies (де повне тороїдальне поле вимирає, оскільки кожна клітина має 8 сусідів), у обмеженому 3x3 полі клітини в кутах мають лише 3 сусідів і виживають.

test_set_state_ignores_out_of_bounds: Перевіряє, що set_state при wrap=False просто ігнорує будь-які координати за межами поля.

Завдяки цьому набору тестів зміг впевнитись у коректній роботі кожного аспекту реалізації «Гри 'Життя'».

Висновок

Під час цієї роботи я на практиці освоїв ключові інструменти розробки. Я навчився використовувати Git, зокрема створювати гілки (unit-tests) для ізольованої роботи та робити коміти. Також я здобув навички написання юніт-тестів, приділяючи увагу не лише базовій логіці, але й важливим граничним випадкам (наприклад, логіці "зациклення" поля). Завершальним етапом стало створення Pull Request на GitHub, що продемонструвало повний цикл командної роботи. Я зрозумів, як поєднання Git та тестування дозволяє підтримувати високу якість коду та впевнено вносити зміни.