UNICHECK
by Turnitin

Ім'я користувача:
приховано налаштуваннями конфіденційності

ID перевірки:
1015655549

Дата перевірки:
20.06.2023 11:37:17 EEST

Тип перевірки:
Doc vs Library

Дата звіту:
20.06.2023 11:38:49 EEST

ID користувача:
100011372

Назва документа: **Цьось Ілля КН 320**

Кількість сторінок: 26 **Кількість слів:** 4437 **Кількість символів:** 34238 **Розмір файлу:** 250.33 KB **ID файлу:** 1015300767

1.8%
Схожість

Найбільша схожість: 0.29% з джерелом з Бібліотеки (ID файлу: 1015186994)

Пошук збігів з Інтернетом не проводився

1.8% Джерела з Бібліотеки 23 ..... Сторінка 28

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

1 ЗБІР ТА ОПИС ТЕОРЕТИЧНИХ ВІДОМОСТЕЙ З РОБОТИ З АВТОМАТИЗАЦІЄЮ ВИКОНАННЯ ЮНІТ-ТЕСТІВ

1.1 Інструменти для автоматизації виконання юніт-тестів

Автоматизація виконання юніт-тестів є важливим етапом розробки програмного забезпечення. Юніт-тести дозволяють перевірити правильність роботи окремих фрагментів коду та виявити помилки на ранніх етапах розробки, що дозволяє забезпечити високу якість програмного забезпечення.

Автоматизація виконання юніт-тестів полягає в тому, що ви встановлюєте спеціальні інструменти, які дозволяють запускати тести автоматично. Це дозволяє зекономити час та зусилля розробника, а також забезпечує виконання тестів на кожному етапі розробки.

Для автоматизації виконання юніт-тестів можна використовувати різні інструменти, залежно від мови програмування, у якій ви працюєте. Наприклад, для PHP є PHPUnit, для JavaScript - Jest, для Python - Pytest та інші. Встановлення та налаштування цих інструментів дозволяє автоматично запускати тести на кожному етапі розробки, наприклад, під час виконання коміту в гіт-репозиторій або на основі розкладу виконання завдань.

Також можна використовувати сервіси для автоматизації виконання юніт-тестів, такі як Travis CI, CircleCI, GitLab CI/CD та інші. Вони дозволяють автоматично запускати тести на кожному етапі розробки, включаючи підтримку різних платформ та інструментів.

1.2 Вигода автоматизації виконання юніт-тестів:

Автоматизація виконання юніт-тестів дозволяє забезпечити високу якість програмного забезпечення та покращити продуктивність розробки, зменшивши кількість помилок, які можуть з'явитися на пізніших етапах розробки. Залежно від розміру проекту, може бути важко вручну виконувати юніт-тести під час розробки.

Автоматизація виконання тестів зменшує час, потрібний для виконання тестів та забезпечує більшу точність виконання тестів, оскільки не залежить від розробника та його власного стилю роботи.

Одним із підходів до автоматизації виконання тестів є використання засобів ContinuousIntegration (CI). CI дозволяє автоматично виконувати тести кожного разу, коли комітиться новий код до репозиторію. Якщо тест не проходить, CI-сервер надсилає повідомлення про помилку на електронну пошту, відповідні платформи чатів (наприклад, Slack) або ж самому розробнику на GitHub.

GitHubActions - це новий інструмент, що дозволяє автоматизувати різні процеси в GitHub. Зокрема, використовуючи GitHubActions, можна автоматизувати виконання юніт-тестів та побудувати звіти з покриттям коду тестами.

GitHubActions дозволяє виконувати автоматичне виконання юніт-тестів з використанням засобів, які підтримуються вашою мовою програмування. Використання GitHubActions для автоматизації тестів дозволяє зменшити зусилля, необхідні для їх виконання, та забезпечити більшу точність виконання тестів. Крім того, засіб автоматично створює звіти з покриттям коду тестами, що дозволяє розробникам визначати, які частини коду вимагають додаткових тестів.

GitHubAction - це безкоштовний сервіс автоматизації розробки програмного забезпечення, який надається GitHub. Він дозволяє розробникам автоматизувати свої рутинні задачі, такі як збірка проекту, запуск тестів, розгортання та інші, без необхідності встановлювати та налаштовувати власний сервер автоматизації.

GitHubAction дозволяє вам налаштовувати сценарії автоматизації, які виконуються при певних подіях, таких як коміти до гілки репозиторію, пул-реквести, створення відгалуження та інші. Кожен сценарій складається з одного або декількох кроків, які виконуються послідовно.

GitHubAction має безліч готових шаблонів, які допомагають вам почати автоматизувати свої задачі з мінімальними зусиллями. Ви можете вибрати шаблон, який найкраще підходить вашому проекту, та налаштувати його відповідно до своїх потреб.

GitHubAction інтегрується з іншими сервісами, такими як Docker, AmazonWebServices, GoogleCloud та інші, що дозволяє вам використовувати їх у своїх сценаріях автоматизації.

Загалом, GitHubAction є потужним інструментом для автоматизації вашого процесу розробки та забезпечення високої якості вашого програмного забезпечення. Він дозволяє вам економити час та зусилля, які ви можете витратити на більш складні задачі.

Одним з унікальних аспектів автоматизації виконання юніт-тестів є можливість інтеграції з системами ContinuousDeployment (CD). CD - це практика автоматичного розгортання програмного забезпечення після успішного проходження тестів і перевірки на різних етапах розробки.

Коли тести проходять успішно, можна використовувати автоматизацію, щоб автоматично розгорнути нову версію програмного забезпечення на продакшн серверах або інших середовищах. Це дозволяє швидко впроваджувати нові функції та виправляти помилки, забезпечуючи неперервну доставку програмного забезпечення.

Інтеграція між автоматизацією тестування та розгортанням дозволяє забезпечити автоматичну перевірку на відповідність функціональним вимогам і якості перед розгортанням. Якщо тести не проходять успішно, процес розгортання призупиняється, що допомагає уникнути введення нестабільного або некоректного програмного забезпечення.

Цей підхід дозволяє автоматизувати не лише виконання тестів, але й розгортання програмного забезпечення, створюючи повний цикл неперервної інтеграції та розгортання (CI/CD). CI/CD є потужним інструментом для підтримки якості програмного забезпечення та прискорення процесу розробки, дозволяючи розробникам швидко та надійно внести зміни у великі проекти.

Одним зі способів поліпшити ефективність автоматизації виконання юніт-тестів є використання паралельного виконання тестів. При великому обсязі тестів, виконання їх послідовно може зайняти значну кількість часу. Однак,

використовуючи можливості паралельного виконання, можна значно скоротити час, необхідний для виконання всього тестового набору.

Для цього можна розділити тестовий набір на окремі групи або категорії тестів і запускати їх одночасно на різних пристроях або в різних потоках виконання. Такий підхід дозволяє використовувати паралельні ресурси для ефективного виконання тестів і прискорення процесу.

Багато інструментів автоматизації тестування, таких як Pytest для Python або Jest для JavaScript, надають можливість паралельного виконання тестів. Вони можуть автоматично розподіляти тестові задачі між різними потоками або процесами, забезпечуючи ефективне використання обчислювальних ресурсів.

Паралельне виконання тестів дозволяє зекономити значний час під час виконання тестового набору, особливо в ситуаціях, коли мається велика кількість тестів або вимагаються тривалі операції, такі як взаємодія з базою даних або зовнішніми сервісами.

Крім того, використання паралельного виконання тестів може допомогти виявити потенційні проблеми залежностей між тестами. Якщо тести паралельно виконуються на різних потоках або пристроях, то вони повинні бути незалежними один від одного і не впливати один на одного. Це допомагає виявити можливі конфлікти або проблеми з певними ресурсами, які можуть виникнути при одночасному виконанні тестів.

Загалом, паралельне виконання тестів є потужним інструментом для покращення ефективності автоматизованого тестування і забезпечення швидкого зворотного зв'язку з розробниками щодо стану програмного забезпечення.

Одним зі способів поліпшення автоматизації виконання юніт-тестів є використання контейнеризації. Контейнери дозволяють упаковувати програмне забезпечення та всі його залежності в ізольовані та переносні середовища.

Використання контейнерів, таких як Docker, дозволяє створювати стандартизовані тестові середовища, які можна легко реплікувати та відтворювати на різних платформах. За допомогою контейнерів можна запускати тести

відокремлено від основної інфраструктури, що спрощує конфігурацію тестових середовищ та уникнення конфліктів залежностей.

Крім того, використання контейнерів забезпечує консистентність середовища виконання тестів між різними розробниками та машинами. Це означає, що всі тести запускаються в однакових умовах, що дозволяє виявляти потенційні проблеми та несумісності, які можуть виникати на різних конфігураціях.

Крім того, контейнери можна легко інтегрувати з іншими інструментами автоматизації, такими як засоби ContinuousIntegration (CI). Наприклад, можна створювати Docker-образи з тестовим середовищем та використовувати їх у скриптах CI для автоматичного запуску тестів під час кожного коміту або зміни в кодовій базі.

Ще одним підходом є використання хмарних платформ для автоматизації виконання юніт-тестів. Наприклад, хмарні сервіси, такі як AmazonWebServices (AWS) або GoogleCloudPlatform (GCP), надають можливість запускати тести на віртуальних машинах або контейнерах в розподіленому середовищі. Це дозволяє масштабувати виконання тестів залежно від потреб проекту та забезпечує велику швидкість виконання завдяки паралельному виконанню тестів на багатьох вузлах.

Таким чином, використання контейнерів та хмарних платформ може сприяти покращенню швидкості та ефективності виконання юніт-тестів, а також забезпечити більшу гнучкість та стабільність у процесі автоматизованого тестування програмного забезпечення.

Одним із популярних підходів до автоматизації виконання юніт-тестів є використання техніки, відомої як "тестування на основі властивостей" (property-basedtesting).

У традиційному юніт-тестуванні ми описуємо конкретні вхідні дані та очікувані результати. Проте, у великих та складних системах може бути важко передбачити всі можливі комбінації вхідних даних. Тут на допомогу приходить тестування на основі властивостей.

У тестуванні на основі властивостей ми визначаємо загальні властивості або інваріанти, які повинні виконуватись для певного фрагмента коду. Наприклад, якщо у нас є функція сортування, загальна властивість може звучати так: "після сортування списку, елементи мають бути відсортовані за зростанням".

Тестування на основі властивостей автоматично генерує випадкові вхідні дані, які задовольняють визначеним властивостям, та перевіряє, чи виконуються ці властивості. Такий підхід дозволяє виявляти непередбачувані помилки та неочевидні проблеми у коді.

Для тестування на основі властивостей існують різні інструменти та бібліотеки. Наприклад, в мові програмування Haskell популярним інструментом є QuickCheck, у JavaScript - fast-check, а в Python - Hypothesis. Ці інструменти надають зручний спосіб опису властивостей та автоматично здійснюють генерацію випадкових вхідних даних для тестування.

Тестування на основі властивостей доповнює традиційні юніт-тести, дозволяючи перевірити код на більш широкому наборі вхідних даних та знайти проблеми, які можуть залишитись непоміченими при класичному підході. Воно сприяє покращенню якості програмного забезпечення та допомагає розробникам уникнути непередбачуваних проблем у майбутньому.

1.3 Тестування та його види

Тестування в програмуванні - це процес перевірки програмного забезпечення з метою виявлення помилок, недоліків та некоректностей у програмному коді. Його основна мета полягає в переконанні, що програма працює вірно і задовольняє вимоги, встановлені для неї.

Тестування може проводитися на різних етапах розробки програмного продукту, таких як модульне тестування, інтеграційне тестування, системне тестування та приймальне тестування. Кожен з цих етапів спрямований на перевірку різних аспектів програмного забезпечення, починаючи від окремих функцій і модулів і закінчуючи його поведінкою в цілому.

Під час тестування створюються тестові сценарії або тестові набори, які включають набір тестових випадків. Кожен тестовий випадок має певний очікуваний результат, і його виконання допомагає виявити проблеми у програмі. Після виконання тестових випадків результати аналізуються, і виявлені проблеми та помилки документуються для подальшої виправлення.

Тестування в програмуванні сприяє забезпеченню якості програмного продукту, зменшенню ризиків і покращенню користувацького досвіду. Воно допомагає розробникам і командам розробки виявляти проблеми на ранніх етапах розробки, що дозволяє їх виправити перед випуском продукту.

Деталі про тестування в програмуванні:

Типи тестування: Тестування в програмуванні включає різні типи тестів, такі як:

- Модульне тестування: Перевірка окремих функцій або модулів програми. Часто використовується автоматизовані тестові фреймворки.
- Інтеграційне тестування: Перевірка взаємодії між різними модулями або компонентами програми.
- Системне тестування: Тестування всієї системи або програмного продукту в цілому.
- Функціональне тестування: Перевірка, чи працюють функції програми відповідно до вимог.
- Навантажувальне тестування: Вимірювання та перевірка продуктивності програми при великому навантаженні.
- Автоматизоване тестування: Використання спеціальних інструментів та скриптів для автоматизації виконання тестів.

Тестові сценарії та тестові набори: Під час тестування розробники створюють тестові сценарії, які описують послідовність кроків для виконання тестів. Тестові сценарії можуть включати один або кілька тестових випадків, які перевіряють конкретні аспекти програми. Комбінація різних тестових випадків утворює тестовий набір, який може бути виконаний для цілісного тестування програми.

Інструменти тестування: У сфері тестування програмного забезпечення є багато спеціалізованих інструментів, які допомагають автоматизувати тестові процеси та спрощують роботу тестувальників. Деякі популярні інструменти включають JUnit (для тестування Java-програм), Selenium (для функціонального тестування веб-додатків), Appium (для тестування мобільних додатків), Postman (для тестування API) та багато інших.

Цикл тестування: Тестування в програмуванні часто виконується у циклі, що називається циклом життя тестування. Цей цикл включає планування тестування, створення тестових сценаріїв і наборів, виконання тестів, аналіз результатів та документування помилок. Інформація про помилки надсилається розробникам для виправлення, після чого тести повторюються для перевірки, чи виправлені проблеми.

Тестування ітераційного розробки: У контексті ітераційного розробки, такої як Scrum або Agile, тестування відбувається на кожній ітерації розробки. Це дозволяє забезпечити постійний контроль якості та швидке виявлення проблем.

Тестування в програмуванні є важливою складовою частиною розробки програмного забезпечення. Воно допомагає забезпечити якість продукту, зменшити ризики виникнення помилок та недоліків, підвищити задоволеність користувачів та сприяє покращенню загального досвіду використання програмного продукту.

Модульне тестування: Це перевірка окремих модулів або функцій програми для підтвердження їх коректності. Воно зазвичай виконується на ранніх етапах розробки, коли модулі ще не були інтегровані в більшу систему. Модульні тести часто автоматизуються за допомогою спеціалізованих фреймворків, таких як JUnit для Java або NUnit для .NET.

Інтеграційне тестування: Цей вид тестування перевіряє взаємодію між різними модулями або компонентами програми після їх інтеграції. Інтеграційні тести спрямовані на виявлення помилок, які можуть виникнути при комунікації між модулями, передачі даних або обміну повідомленнями.

Функціональне тестування: Це перевірка функціональності програми для забезпечення відповідності до вимог. Функціональні тести перевіряють, чи працюють окремі функції, операції та алгоритми програми так, як очікується. Вони перевіряють, чи повертають функції правильні результати, чи обробляють вони вхідні дані належним чином та чи виконуються очікувані дії.

Вимоги тестування: Це форма тестування, яка перевіряє, чи відповідає програма заданим вимогам. Вимоги тестування базуються на функціональних та нефункціональних вимогах до програми. Вони встановлюються перед початком розробки програми і слугують основою для тестування, щоб переконатися, що програма відповідає цим вимогам.

Навантажувальне тестування: Це перевірка продуктивності програми за умови великого навантаження. Його мета - визначити межі, при яких програма може працювати ефективно, і виявити проблеми з продуктивністю, такі як високий час відгуку або перевантаження ресурсів.

Системне тестування: Це тестування всієї системи або програмного продукту в цілому. Воно включає перевірку взаємодії між всіма компонентами програми, а також перевірку їх взаємодії з зовнішніми системами чи сервісами.

Ці види тестування використовуються для забезпечення високої якості програмного забезпечення. Комбінація різних видів тестування допомагає виявити різноманітні проблеми та помилки у програмах перед їх випуском.

2 НАЛАШТУВАННЯ СЕРЕДОВИЩА ВИКОНАННЯ РОБОТИ ТА ВСТАНОВЛЕННЯ ПЗ

2.1 Застосування автоматизації юніт тестів

GitHubActions - це сервіс автоматизації робочих процесів, який надається GitHub. Ви можете використовувати його для створення власних автоматичних дій, які будуть виконуватися на основі певних подій у вашому репозиторії. Такі дії можуть включати будь-які кроки, від установки залежностей та компіляції до розгортання програмного забезпечення.

Окрім автоматичного виконання юніт-тестів, GitHubActions може виконувати безліч інших завдань, таких як:

1. Автоматична збірка (ContinuousIntegration): Ви можете налаштувати GitHubActions, щоб автоматично збирати ваш проект при кожному коміті або пул-реквесті. Це допомагає перевіряти, чи працює ваш код, чи проходять тести і чи відповідає він вимогам проекту.
2. Автоматичне розгортання (ContinuousDeployment): Ви можете налаштувати GitHubActions для автоматичного розгортання вашого проекту після успішної збірки. Це дозволяє швидко виводити ваше програмне забезпечення в експлуатацію.
3. Тестування в різних середовищах: Ви можете налаштувати GitHubActions для виконання вашого тестування в різних середовищах, таких як різні версії операційних систем або різні версії мов програмування. Це допомагає переконатися, що ваш код працює на різних платформах.
4. Візуалізація результатів: GitHubActions надає можливість візуалізувати результати вашого тестування за допомогою графіків та гістограм. Це дозволяє вам швидко виявляти проблеми та аналізувати якість вашого коду.
5. Інтеграція з іншими сервісами: GitHubActions може бути інтегрований з іншими сервісами, такими як Slack, Email або інші, для надсилання повідомлень

про результати тестування або помилки. Це допомагає вам швидко отримувати сповіщення та реагувати на проблеми.

6. Кастомні сценарії: Ви можете налаштувати власні сценарії автоматизації, які відповідають вашим потребам. Ви можете визначити власні кроки, налаштувати залежності та параметри, які використовуються в діях.

7. Параметризовані дії: GitHubActions дозволяє використовувати параметри в діях, що полегшує конфігурування та перевикористання коду. Ви можете використовувати змінні, отримані з попередніх кроків, або передавати значення параметрів у ваші дії.

8. Сховище готових дій (Actions Marketplace): GitHub має сховище готових дій, де ви можете знайти велику кількість готових дій, які створили інші користувачі. Ви можете використовувати ці дії в своїх власних робочих процесах, щоб заощадити час і скоротити розробку.

9. Розширення функціональності за допомогою скриптів: Ви можете використовувати скрипти у вашому робочому процесі, щоб здійснювати різні дії та маніпулювати змінними та даними. Це дозволяє вам розширити функціональність GitHubActions та адаптувати його під ваші потреби.

10. Матриці та паралельне виконання: GitHubActions підтримує використання матриць, що дозволяє вам визначити набір параметрів, за якими робочий процес буде виконуватися для різних комбінацій значень. Крім того, ви можете налаштувати паралельне виконання, щоб прискорити час виконання робочих процесів.

11. Управління правами доступу: Ви можете налаштувати права доступу до робочих процесів GitHubActions для окремих користувачів або команд. Це дозволяє контролювати, хто може переглядати, редагувати або запускати робочі процеси в вашому репозиторії.

12. Локальний режим виконання (Self-hostedrunners): GitHubActions надає можливість використовувати власні ресурси для виконання робочих процесів. Ви можете налаштувати власний сервер або використовувати існуючі машини у вашій

інфраструктурі для виконання дій. Це особливо корисно для випадків, коли вам потрібен доступ до приватних ресурсів або особливих середовищ.

Загалом, GitHubActions - потужний інструмент для автоматизації робочих процесів у вашому репозиторії. Він надає широкі можливості для автоматичного виконання тестів, збірки, розгортання та інших завдань, що допомагають підтримувати якість вашого коду та прискорюють розробку.

Також GitHubAction - це сервіс, що дозволяє автоматизувати процеси від GitHub репозиторію. Один з найбільш корисних варіантів автоматизації - це автоматичне виконання юніт-тестів під час відправки змін в репозиторій.

2.2 Робота з GitHubAction

Для того, щоб автоматизувати виконання юніт-тестів за допомогою GitHubAction, спочатку потрібно створити файл у вашому репозиторії з назвою .github/workflows/test.yml. У цьому файлі можна вказати конфігурацію тестування, включаючи інструкції для запуску тестів. Наприклад, якщо використовувати мову програмування Python, тоді запуск юніт-тестів буде реалізовуватися як показано на блок-схемі 2.1

Блок-схема 2.1 - Блок-схема виконання юніт тестів у GitHub Action

У цьому прикладі ми вказали, що ми хочемо запустити юніт-тести під час кожного push в гілку репозиторію, використовуючи Python 3.x. Ми також вказали,

що ми хочемо встановити залежності з requirements.txt і запустити юніт-тести, використовуючи python -m unittestdiscover -s tests.

Після того, як ви створили файл конфігурації, ви можете перевірити його на помилки та зберегти його у вашому репозиторії. Кожного разу, коли ви відправляєте зміни до свого репозиторію, GitHubAction буде автоматично запускати ваші тести на серверах GitHub і повертати результати вам.

Таким чином, автоматизація виконання юніт-тестів за допомогою GitHubAction дозволяє вам відслідковувати помилки тестування на ран.

GitHubAction має безліч можливостей, які дозволяють автоматизувати не тільки виконання юніт-тестів, але й інші процеси, такі як автоматична збірка, розгортання та багато інших. Окрім цього, GitHubAction має декілька корисних функцій, таких як:

Візуалізація результатів тестування: GitHubAction дозволяє вам відслідковувати результати вашого тестування на графіках та гістограмах. Це допомагає швидко виявляти помилки в коді та вдосконалювати ваші тести.

Кастомні сценарії: GitHubAction дозволяє вам налаштувати сценарії автоматизації, які відповідають вашим потребам. Наприклад, ви можете налаштувати GitHubAction, щоб автоматично запускати тести при кожному коміті в гілку репозиторію або при пул-реквесті.

Налаштування середовища: GitHubAction дозволяє вам налаштувати середовище, у якому виконуються ваші тести. Наприклад, ви можете вибрати ОС, встановити необхідні залежності та налаштувати змінні середовища.

Інтеграція з іншими сервісами: GitHubAction дозволяє інтегрувати ваші тести з іншими сервісами, такими як Slack, Email та інші. Це допомагає швидко отримувати повідомлення про помилки та виявляти їх. Загалом, автоматизація виконання юніт-тестів за допомогою GitHubAction дозволяє вам швидко виявляти помилки в вашому коді та забезпечувати високу якість вашого продукту.

Конфігурація GitHubActions полягає в створенні файлу конфігурації з іменем workflow.yml, де задається список кроків, які потрібно виконати під час виконання автоматичної дії на основі певних подій у репозиторії. Файл

конфігурації повинен бути розміщений в директорії .github/workflows у кореневому каталозі репозиторію.

Конфігурація GitHubActions складається з наступних елементів: name - назва дії, яку ви виконуєте.

1. on - події, на основі яких дія буде виконана. Наприклад, можна налаштувати дію на запуск при пул-реквестах, коммітах або випусках.
2. jobs - список задач, які потрібно виконати в ході дії. Кожна задача повинна мати свою назву та список кроків, які необхідно виконати.
3. steps - список кроків, які потрібно виконати в рамках кожної задачі. Кожен крок може бути скриптом, командою або дією, яка виконується на основі певного контейнера.

Для прикладу, ось конфігурація GitHubActions, яка автоматично запускає тести при кожному комміті у репозиторії:

У цьому прикладі конфігурація складається з наступних елементів:

1. name: CI - назва дії.
2. on: - список подій, які спричиняють запуск дії.
3. jobs: - список задач, які мають бути виконані. побачити реалізацію запуску

2.3 Налаштування локального середовища розробки юніт-тестів

У даному коді використовується модуль unittest для написання тестів. Це стандартний модуль у Python, який надає засоби для автоматизованого тестування.

Щоб використовувати unittest, вам не потрібно додатково встановлювати жодну бібліотеку. Модуль unittest вже входить до стандартної бібліотеки Python. Клас TestCalculator унаслідується від unittest.TestCase, що робить його тестовим класом. У цьому класі визначені різні методи, які представляють окремі тести. Кожен метод тесту починається зі слова test, щоб unittest впізнав його як тестовий метод.

У методах тесту використовуються різні методи з модулю unittest, такі як assertEqual() для перевірки рівності, assertRaises() для перевірки підняття виключення, assertIsNone() для перевірки на None і т. д.

Щоб запустити тести, можна використовувати конструкцію if __name__ == '__main__': unittest.main()

Це перевіряє, чи файл був запущений безпосередньо, а не імпортований як модуль. Якщо файл був запущений безпосередньо, то викликається функція unittest.main(), яка автоматично знаходить і запускає всі тести у файлі.

Для запуску тестів можна виконати цей файл з командного рядка або за допомогою середовища розробки, якщо ви його використовуєте. Коли тести запускаються, можна буде побачити результати виконання кожного тесту і повідомлення про будь-які помилки чи невідповідності.

3 СТВОРЕННЯ GITHUB ACTION ДЛЯ ТЕСТУВАННЯ РОЗРОБЛЕНОГО КЛАСУ

3.1 Опис тестової програми та створення юніт-тестів

В коді роботи використовувалися наступні типи тестів:

- Позитивні тести: Це тести, які перевіряють очікувані випадки, коли програма повинна повернути коректні результати. Наприклад, тест test_add() перевіряє, чи правильно додаються два числа, очікуваний результат - 5.
- Негативні тести: Ці тести перевіряють, як програма поводиться в некоректних ситуаціях або з некоректними даними. Наприклад, тест test_zero_division() перевіряє, чи програма повертає None при діленні на нуль.
- Тести на винятки: Ці тести перевіряють, чи виникають очікувані виключення при виконанні певних операцій. Наприклад, тест test_missing_method() перевіряє, чи піднімається виключення AttributeError, коли викликається неіснуючий метод.
- Тести на коректність даних: Ці тести перевіряють, чи програма правильно обробляє різні типи вхідних даних. Наприклад, тест test_invalid_input() перевіряє, чи програма піднімає TypeError при спробі додати рядок до числа.

Весь процес роботи проводиться в GitHub в новоствореному репозиторію під назвою `my_module` в якій знаходиться директорія ‘workflows’ яка веде до yml файлу під назвою ‘tests.yml’

Також в цьому репозиторії знаходиться Python файл під назвою ‘test.py’. Цей файл містить в собі код тестів з які повинні запускатись.

Для того щоб почати роботу з тестами потрібно було провести імпорт бібліотек тестування та математичної функції. Це було реалізовано за допомогою:

```
importunittest
    importmath
```

unittest - це модуль, який надає функціональність для написання тестів у стилі юніт-тестування. Він містить класи тестування, такі як ‘TestCase’, а також методи перевірки, такі як ‘assertEqual’ та ‘assertIsNone’, які використовуються для перевірки результатів тестів.

math - це вбудований модуль Python, який містить різноманітні математичні функції. Наприклад, ви можете використовувати функцію ‘math.sqrt()’ для обчислення квадратного кореня числа або інші математичні функції, які можуть бути корисними для вашого коду.

Після цього слідувало створення класу ‘Calculator’, який містить математичні функції для додавання, віднімання, множення, ділення та обчислення квадратного кореня. Кожна функція реалізована як метод класу ‘Calculator’.

```
importunittest
importmath

classCalculator :
```

Визначення класу ‘TestCalculator’, який наслідується від класу ‘unittest.TestCase’ для створення тестів.

```
Class TestCalculator(unittest.TestCase)
    Def setUp(self) :
        Self.calculator = Calculator()
```

У методі ‘setUp()’ створюється екземпляр класу ‘Calculator’, який буде використовуватися в усіх тестах.

```
class TestCalculator(unittest.TestCase):
    def setUp(self)
        self.calculator = Calculator()
```

Визначення методів-тестів для перевірки роботи різних математичних функцій. Цей пункт включає визначення методів-тестів для перевірки роботи різних математичних функцій класу ‘Calculator’. У даному випадку, кожен метод-тест перевіряє коректність роботи окремої математичної операції.

Наприклад, для перевірки методу ‘square_root’, який обчислює квадратний корінь числа, визначений метод-тест ‘test_square_root’. У цьому методі виконується декілька кроків:

1. Створюється екземпляр класу ‘Calculator’ в методі ‘setUp’, який буде використовуватися в цьому тесті.
2. Викликається метод ‘square_root’ об'єкта ‘calculator’ з передачею значення 16 як аргументу.
3. Отримане значення результату зберігається у змінну result.
4. Використовується метод ‘self.assertEqual’, щоб порівняти значення result з очікуваним значенням 4.
5. Якщо значення result не дорівнює 4, тест викличе помилку і виведе повідомлення про невдалу перевірку.

Аналогічним чином визначаються інші методи-тести для інших математичних операцій, таких як ‘add’, ‘subtract, multiply’ і ‘divide’. Кожен метод-тест викликає відповідний метод класу ‘Calculator’ з відповідними вхідними параметрами, а потім перевіряє результат за допомогою відповідного методу перевірки, такого як ‘self.assertEqual’.

Цей підхід дозволяє систематично перевіряти коректність роботи кожної окремої математичної функції та специфічних випадків, щоб забезпечити

правильність їхньої реалізації.

Всі типи тестів якій використовуються в програмі мають наступний вигляд в блок-схемі 3.1

Блок-схема 3.1 – Типи юніт тестів у Python

6.В кожному методі-тесті викликається відповідний метод ‘Calculator’, а результат порівнюється з очікуваним значенням за допомогою методів перевірки, таких як ‘self.assertEqual’ або ‘self.assertIsNone, зображено на рисунках 3.1 та 3.2

Рисунок 3.1 – Тест з помилкою у методі square_root

Рисунок 3.2 – Тест з помилкою при діленні на нуль

Для кожного тесту вказано детальне повідомлення про помилку, яке буде виводитись, якщо перевірка не пройде. Для прикладу:

self.assertEqual(result, 5, "Помилка у методі add()")

В методі ‘test_missing_method’ перевіряється виклик неіснуючого методу ‘nonexistent_method’. Очікується, що при спробі виклику методу буде виникати помилка `AttributeError`. Завдання тесту полягає в тому, щоб переконатися, що відповідна помилка виникає при виклику неіснуючого методу як на рисунку 3.3

Рисунок 3.3 – Тест який перевіряє неіснуючий метод

9.В методі ‘test_invalid_input’ перевіряється передача неправильного типу аргументів у метод ‘add’. Очікується, що при спробі передачі рядкового значення "2" замість числового значення буде виникати помилка ‘TypeError’. Тест має

переконатися, що відповідна помилка виникає при передачі неправильного типу аргументів.

```
def test_invalid_input(self):
    with self.assertRaises(TypeError):
        self.calculator.add("2",3)
```

10.В методі 'test_zero_division' перевіряється ділення на нуль. Очікується, що при спробі поділити число на нуль результатом буде значення 'None'. Завдання тесту полягає в перевірці правильності повернення значення 'None' при діленні на нуль. Зображено це на рисунку 3.4.

Рисунок 3.4 – Тест який перевіряє ділення на нуль

11.Для запуску усіх тестів за допомогою unittest.main(), потрібно було додати наступний код в кінці файлу з тестами:

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

Цей код перевіряє, чи поточний файл є головним (основним) файлом, який запускається безпосередньо. Якщо це так, то виконується функція 'unittest.main()', яка автоматично виконує всі методи-тести, що містяться в файлі. Запуск

виконання юніт-тестів у вигляді блок-схеми 3.1

Блок-схема 3.2 – Запуск виконання тестів

3.2 Автоматичне виконання тестів через GitHubAction

При запуску файлу з тестами, наприклад, за допомогою команди ‘python tests.py’ у командному рядку або з інтегрованого середовища розробки, буде автоматично виконано всі методи-тести. Результат кожного тесту буде надруковано у консолі або відображено у вікні виконання, залежно від того, як запускати тести.

Щоб запустити тести, можна використати команду

python<ім'я_файлу.py>

в командному рядку. В даному випадку файл називається tests.py, а це означає що щоб запустити тести потрібно виконати команду

python tests.py.

Результати виконання тестів були успішно виведені у командному рядку. Успішно пройдені тести були позначені як ‘.’ . Якщо були неуспішні тести тоді вони будуть позначатись як ‘F’, тести з винятками - як ‘E’, пропущені тести - як s. Загальна статистика, така як кількість пройдених тестів, час виконання тощо, також була відображена (0.001 сек.).

Завершивши виконання тестів, можна побачити підсумкову інформацію про тести, таку як загальна кількість тестів, кількість пройдених, кількість неуспішних.

За допомогою GitHubAction можна автоматизувати весь цей процес. Для цього в репозиторії під назвою ‘my_module’ находиться yml файл під назвою ‘tests.yml’. Цей YML-код є файлом конфігурації для GitHubActions. Він виконує деякі дії при кожному push на гілку main або при створенні pullrequest на гілку main.

У даному випадку, назва цієї конфігурації є "Tests". У ній визначено одну роботу з назвою "build", яка запускається на операційній системі ubuntu-latest.

Опис кроків у роботі "build":

1. Крок "Checkoutrepository" використовує дію actions/checkout@v2 для клонування репозиторію.
2. Крок "SetupPython" використовує дію actions/setup-python@v2 для налаштування середовища Python з вказаною версією (3.x).
3. Крок "Runtests" виконує команду python tests.py, яка запускає файл tests.py з використанням Python.

Отже, загальна мета цього YML-коду полягає в тому, щоб при кожному push на гілку main або при створенні pullrequest на гілку main виконувалися тести, які знаходяться у файлі tests.py. Виглядає це наступним чином на рисунку 3.5

Рисунок 3.5 – Результат виконання тестів в Action

Схожість

Джерела з Бібліотеки 23

1	Студентська робота	ID файлу: 1015186994	Навчальний заклад: Lviv Polytechnic National University	10 Джерело	0.29%
2	Студентська робота	ID файлу: 1015283227	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.25%
3	Студентська робота	ID файлу: 1015227382	Навчальний заклад: Lviv Polytechnic National University		0.23%
4	Студентська робота	ID файлу: 1015037392	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.2%
5	Студентська робота	ID файлу: 1015226861	Навчальний заклад: Poltava National Technical Yuri Kond	2 Джерело	0.2%
6	Студентська робота	ID файлу: 1015058600	Навчальний заклад: Taras Shevchenko National University of Kyiv		0.18%
7	Студентська робота	ID файлу: 1015069347	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.18%
8	Студентська робота	ID файлу: 1015069747	Навчальний заклад: National Technical University of Ukraine “Kyi...		0.18%
9	Студентська робота	ID файлу: 1015249154	Навчальний заклад: Poltava National Technical Yuri Kondratyuk U...		0.18%
10	Студентська робота	ID файлу: 1015166602	Навчальний заклад: National Technical University of Ukra	2 Джерело	0.18%
11	Студентська робота	ID файлу: 1015170330	Навчальний заклад: Lviv Polytechnic National University		0.18%
12	Студентська робота	ID файлу: 1002697678	Навчальний заклад: Lviv Polytechnic National University		0.18%