Python series

Introduction to Unit Testing

With Code Examples

save for later

Introduction

Unit testing is a software testing technique where individual units or components of a software system are tested in isolation.

The primary goal of unit testing is to validate that each unit of the software performs as expected. It is an essential practice in software development for ensuring code quality, maintainability, and early bug detection.

Unit testing helps developers catch and fix issues early in the development cycle, reducing the cost and effort required for debugging and fixing bugs later on.

Swipe next ⟶

follow for more

Python's Built-in unittest Module

Python comes with a built-in module called unittest, which provides a framework for writing and running unit tests.

The unittest module is based on the xUnit architecture and offers a rich set of features for test automation, including test case organization, test fixtures, and test runners.

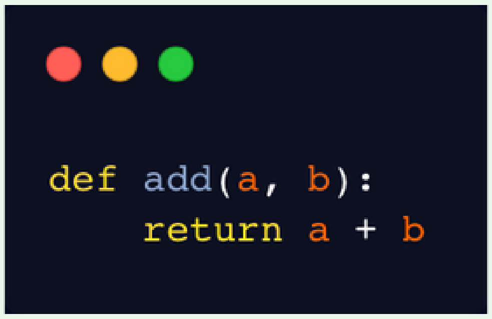
With the unittest module, you can write test cases by creating subclasses of unittest.TestCase and defining test methods that start with the prefix "test_".

Swipe next ⟶

save for later

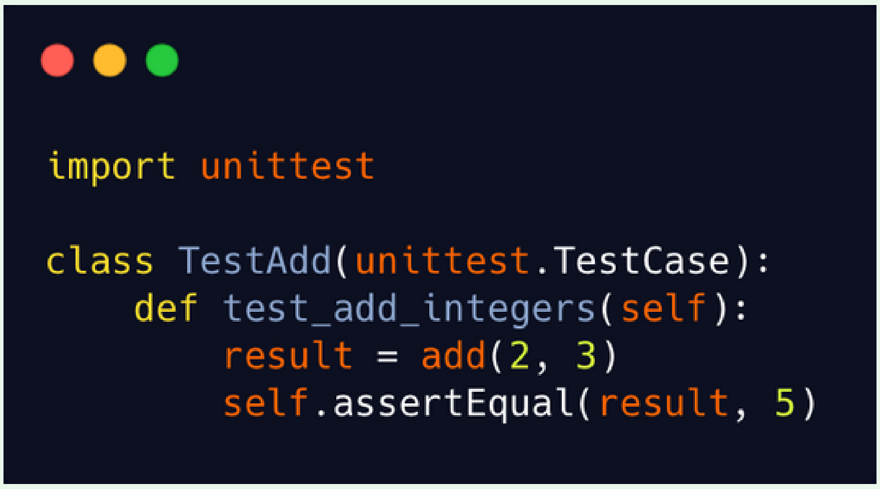
Testing a Simple Function

Let's start with a simple example of testing a function using the unittest module. Consider the following function that adds two numbers:



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

To test this function, we can create a test case like this:



```python
import unittest

class TestAdd(unittest.TestCase):
    def test_add_integers(self):
        result = add(2, 3)
        self.assertEqual(result, 5)
```

In this example, we create a subclass of unittest.TestCase called TestAdd and define a test method test_add_integers(). Inside the test method, we call the add() function with arguments 2 and 3, and assert that the result is equal to 5 using the assertEqual() method provided by unittest.TestCase.

Swipe next ⟶

follow for more

Testing a Class

Unit testing is not limited to testing functions; it can also be used to test classes and their methods. Consider the following Circle class:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

To test the area() method of the Circle class, we can write a test case like this:

```
import unittest

class TestCircle(unittest.TestCase):
    def test_area(self):
        circle = Circle(5)
        self.assertEqual(circle.area(), 78.5)
```

In this example, we create an instance of the Circle class with a radius of 5, and assert that the area() method returns the expected value of 78.5.

Swipe next ⟶

save for later

Test Case Organization

As the number of tests grows, it becomes important to organize them into logical groups or test suites.

The unittest module provides a way to group related tests into a test suite and run them together. This can be done using the unittest.TestSuite class and the addTest() or addTests() methods.

Grouping tests into suites can help in several ways:

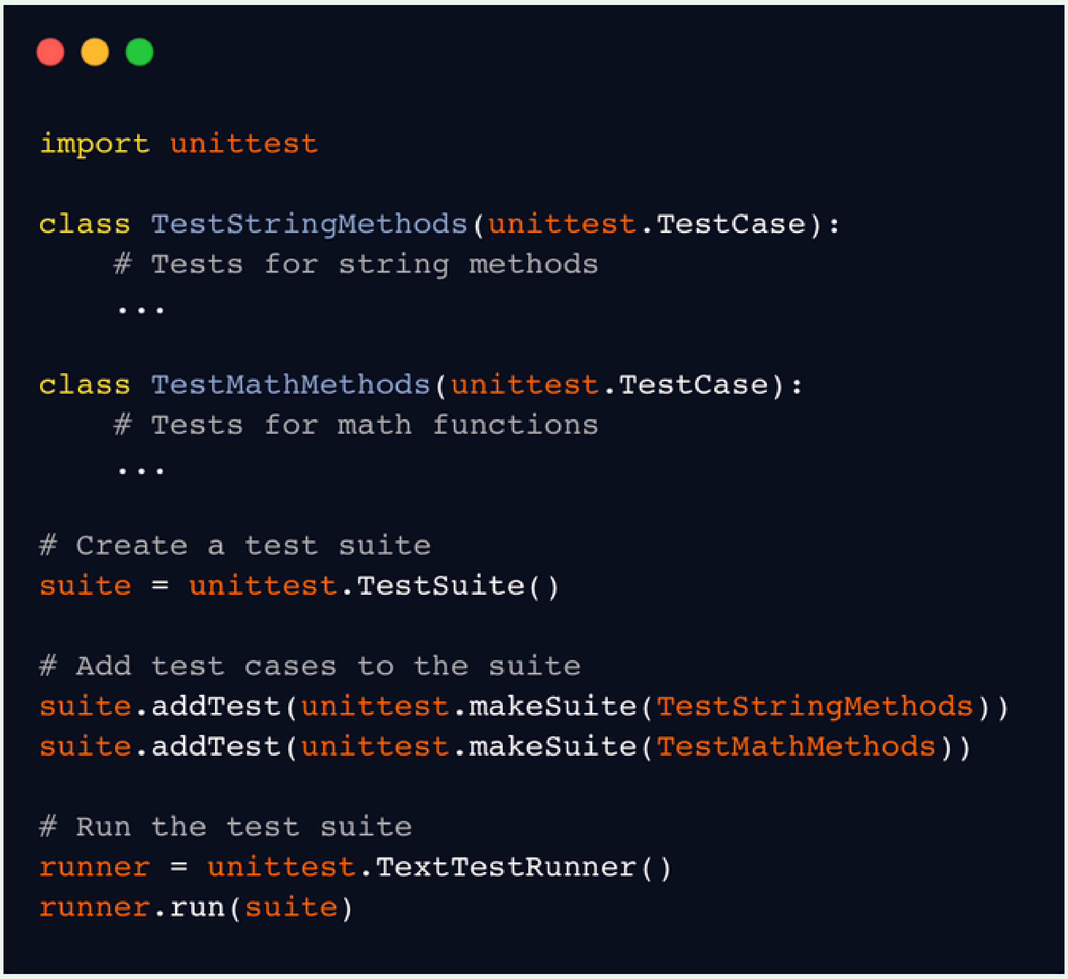
- Improved organization and readability of tests
- Selective execution of tests based on suites
- Parallel execution of tests across multiple suites
- Reporting and summary generation at the suite level

Swipe next ⟶

follow for more

Test Suite

Here's an example of how to create a test suite and run multiple test cases simultaneously using the unittest module:



```python
import unittest

class TestStringMethods(unittest.TestCase):
    # Tests for string methods
    ...

class TestMathMethods(unittest.TestCase):
    # Tests for math functions
    ...

# Create a test suite
suite = unittest.TestSuite()

# Add test cases to the suite
suite.addTest(unittest.makeSuite(TestStringMethods))
suite.addTest(unittest.makeSuite(TestMathMethods))

# Run the test suite
runner = unittest.TextTestRunner()
runner.run(suite)
```

In this example, we define two test case classes: TestStringMethods and TestMathMethods. We then create a unittest.TestSuite object and add the test cases from both classes using the addTest() method. Finally, we create a unittest.TextTestRunner object and run the test suite using the run() method.

Swipe next ⟶

save for later

Test Fixtures

Test fixtures are used to set up and tear down the test environment before and after running tests.

This is particularly useful when tests require specific conditions or resources to be set up, such as creating temporary files, setting up a database connection, or initializing objects.

The unittest module provides two special methods for managing test fixtures: **setUp()** and **tearDown()**. The **setUp()** method is called before each test method is executed, and the **tearDown()** method is called after each test method has completed.

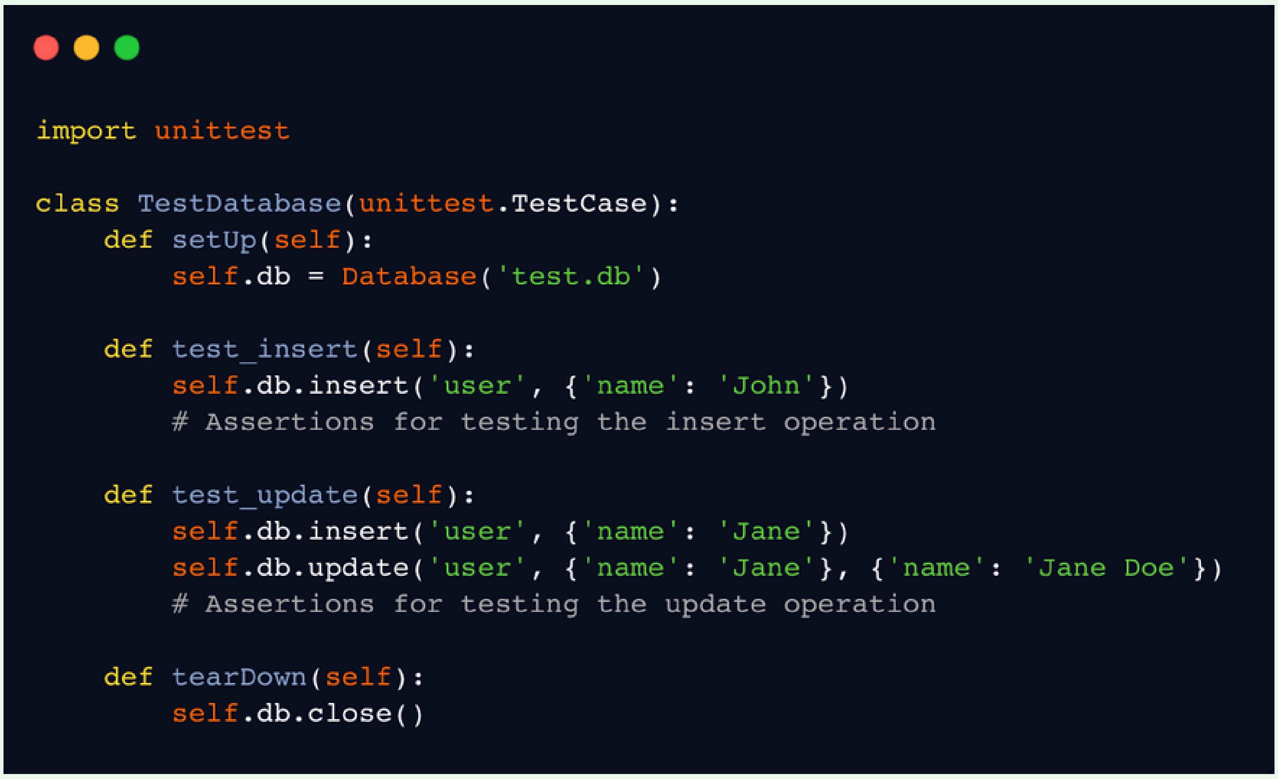
Using test fixtures helps ensure that each test runs in a consistent and isolated environment, preventing tests from interfering with each other or leaving behind residual data that could affect subsequent tests.

Swipe next ⟶

follow for more

Test Fixtures

Here's an example of how to use test fixtures in a practical scenario, such as setting up and tearing down a database connection:



```python
import unittest

class TestDatabase(unittest.TestCase):
    def setUp(self):
        self.db = Database('test.db')

    def test_insert(self):
        self.db.insert('user', {'name': 'John'})
        # Assertions for testing the insert operation

    def test_update(self):
        self.db.insert('user', {'name': 'Jane'})
        self.db.update('user', {'name': 'Jane'}, {'name': 'Jane Doe'})
        # Assertions for testing the update operation

    def tearDown(self):
        self.db.close()
```

In this example, we define two test case classes: TestStringMethods and TestMathMethods. We then create a unittest.TestSuite object and add the test cases from both classes using the addTest() method. Finally, we create a unittest.TextTestRunner object and run the test suite using the run() method.

Swipe next ⟶

save for later

Edge Cases and Boundary Conditions

Testing edge cases and boundary conditions is crucial for ensuring the robustness and reliability of your code. Edge cases refer to inputs or situations that are at the extreme limits of the expected range, while boundary conditions are the values that separate valid and invalid inputs.

Failing to test edge cases and boundary conditions can lead to unexpected behavior, crashes, or security vulnerabilities in your software. It's essential to identify and test these scenarios to ensure that your code handles them correctly and gracefully.

Examples of edge cases and boundary conditions include:

- Inputs at the minimum or maximum allowed values
- Empty or null inputs
- Inputs with special characters or edge cases (e.g., extremely large or small numbers, strings with Unicode characters)
- Boundary conditions for loops and conditional statements

Swipe next ⟶

follow for more

Edge Cases

Here's an example of how to test an edge case, such as handling division by zero:

```python
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

To test the divide() function and ensure it correctly handles the division by zero case, we can write a test like this:

```python
import unittest

class TestDivide(unittest.TestCase):
    def test_divide_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)
```

In this example, we use the assertRaises() context manager provided by unittest.TestCase to assert that the divide() function raises a ValueError when the second argument (b) is zero. This ensures that the edge case of division by zero is properly handled by the function.

Swipe next ⟶

save for later

Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development approach where tests are written before writing the actual code. In TDD, the development cycle follows these steps:

1. Write a failing test for the desired functionality.
2. Write the minimum amount of code necessary to make the test pass.
3. Refactor the code to improve its design and structure while ensuring all tests still pass.
4. Repeat steps 1-3 for the next piece of functionality.

By following the TDD approach, developers focus on writing tests first, which serves as a specification for the code they will write. This encourages a more modular and testable code design, and helps catch issues early in the development process.
TDD promotes a tight feedback loop between writing tests and writing code, resulting in a robust and well-tested code

Swipe next ⟶

Testing with Libraries

While Python's built-in unittest module is powerful and widely used, there are also third-party libraries available for unit testing that provide additional features and a more user-friendly syntax. Some popular testing libraries for Python include:

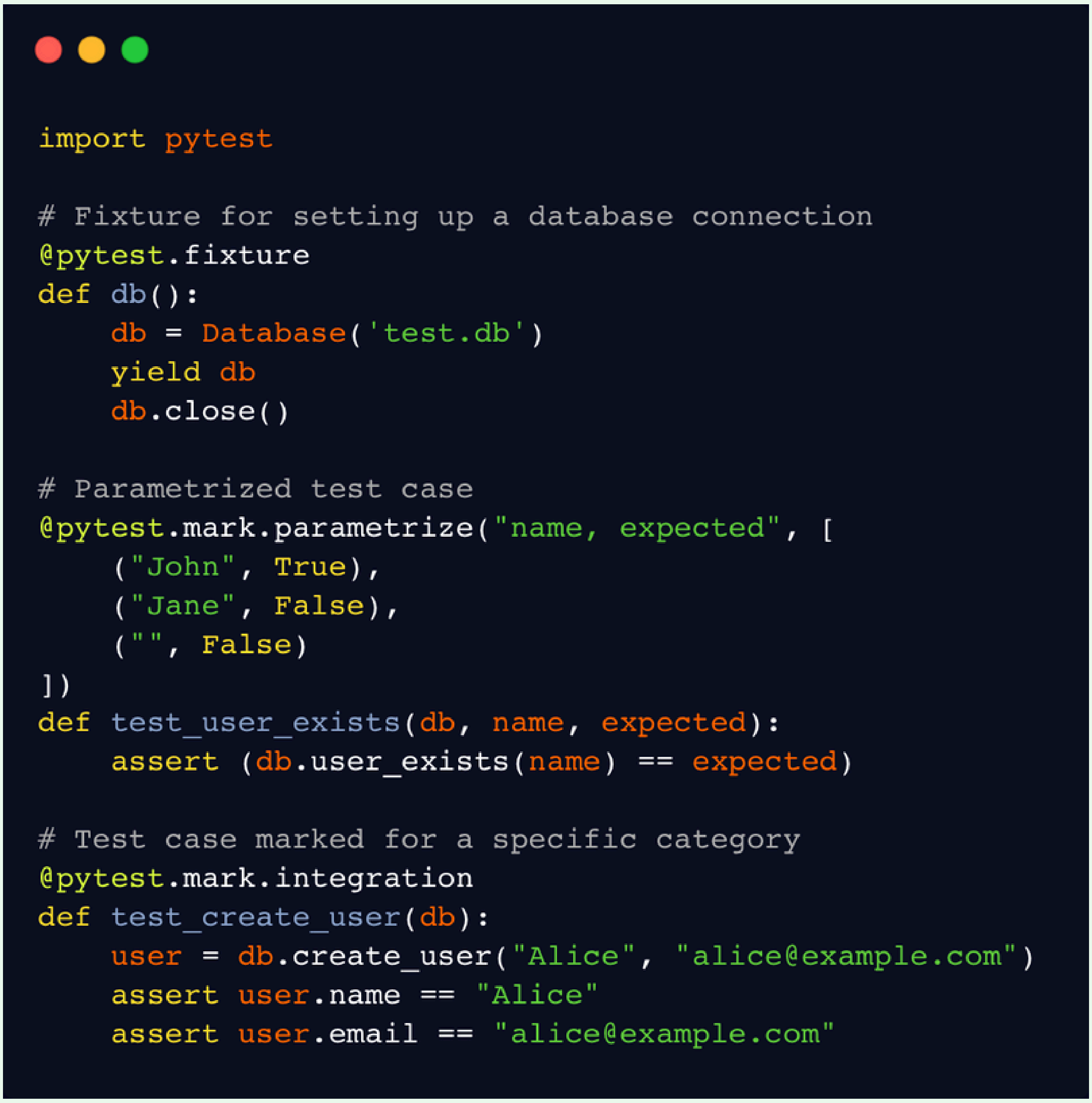
1. **pytest**: A flexible and user-friendly testing framework that offers a more expressive syntax and powerful features like fixtures, parameterization, and plugins.
2. **doctest**: A module that allows you to write tests as examples in your docstrings, making it easy to integrate testing with documentation.
3. **unittest.mock**: A library for creating mock objects, which can be useful for testing code that interacts with external dependencies or services.
4. **hypothesis**: A library for property-based testing, which generates random test cases based on user-defined specifications.

Swipe next ⟶

save for later

pytest: A Popular Testing Framework

Here's an example of how to write a test case using pytest, showcasing its features such as fixtures, parametrization, and markers:



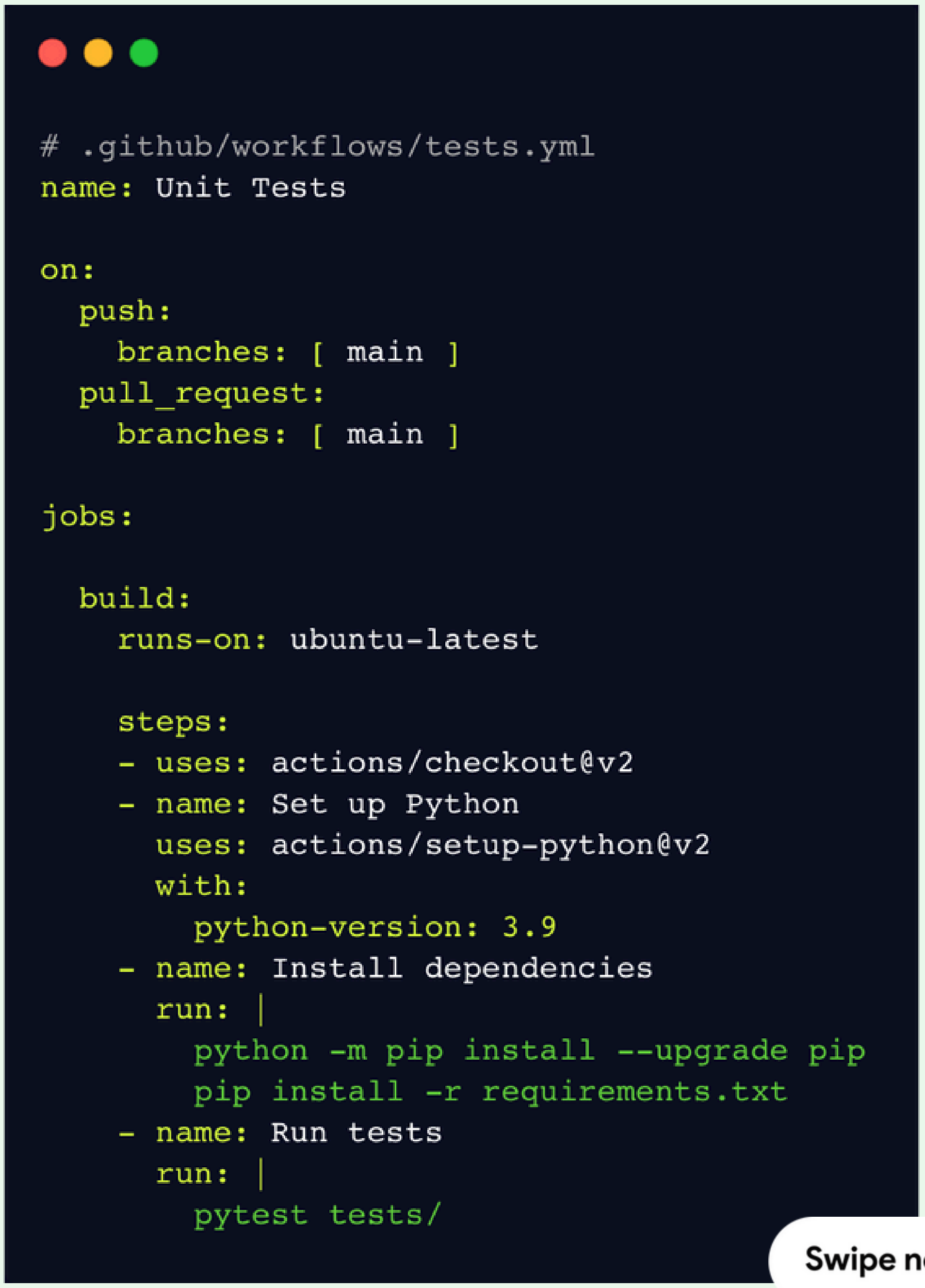
```python
import pytest

# Fixture for setting up a database connection
@pytest.fixture
def db():
    db = Database('test.db')
    yield db
    db.close()

# Parametrized test case
@pytest.mark.parametrize("name, expected", [
    ("John", True),
    ("Jane", False),
    ("", False)
])
def test_user_exists(db, name, expected):
    assert (db.user_exists(name) == expected)

# Test case marked for a specific category
@pytest.mark.integration
def test_create_user(db):
    user = db.create_user("Alice", "alice@example.com")
    assert user.name == "Alice"
    assert user.email == "alice@example.com"
```

Swipe next

follow for more

GitHub Actions for Testing

GitHub Actions is a popular Continuous Integration and Continuous Deployment (CI/CD) platform provided by GitHub. It allows you to automate various software development workflows, including testing, directly from your GitHub repository.



```yaml
# .github/workflows/tests.yml
name: Unit Tests

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:

  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.9
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
    - name: Run tests
      run: |
        pytest tests/
```

Swipe next →

save for later

GitHub Actions for Testing

- The workflow is triggered on pushes and pull requests to the **main** branch.
- The **actions/checkout** step checks out the repository code.
- The **actions/setup-python** step sets up the Python environment with the specified version.
- The **Install dependencies** step installs the project dependencies from **requirements.txt.**
- The **Run tests** step runs the pytest command to execute all tests in the tests/ directory.

This workflow ensures that the unit tests are automatically executed on every code change, providing continuous validation of the codebase.