
OOP (Object-Oriented Programming) Basics

What is OOP?
OOP is a programming paradigm that is based on the concept of "objects", which can contain both data (attributes) and methods (functions). This approach helps organize and structure code in a more efficient way.

Class & Object:

- **Class**: A blueprint for creating objects. A class defines properties (attributes) and methods (functions) that an object of that class can have.
- **Object**: An instance of a class. You can create multiple objects from a class, and each will have its own data and behavior.

Example:

```
class Dog:
    def __init__(self, name):
        self.name = name  # Attribute

    def bark(self):  # Method
        print(f"{self.name} says woof!")

# Creating an object (instance of the class)
dog1 = Dog("Max")
dog1.bark()  # Calling the bark method
```

Explanation:

1. class Dog: Defines a class called Dog.
2. def __init__(self, name): This is the constructor method. It gets called when you create a new object of the class. The self parameter refers to the current instance of the object.
3. dog1 = Dog("Max"): Creates an object dog1 of the class Dog with the name "Max".
4. dog1.bark(): Calls the bark() method, which prints "Max says woof!".

Key OOP Concepts:

- **Encapsulation**: Bundling data and methods that operate on that data within a class.
- **Inheritance**: A class can inherit the properties and methods of another class.
- **Polymorphism**: The ability to use the same method name to do different things.
- **Abstraction**: Hiding the complex implementation details and showing only the necessary features of an object.

What is Pytest?

Pytest is a powerful testing framework for Python that makes it easy to write simple and scalable test cases. It's highly extensible and can handle everything from simple unit tests to complex functional testing of applications.

Features of Pytest:

- Simple to use: Minimal setup required.
- Automatic test discovery: Pytest automatically finds tests by searching for files that start with test_ and functions that start with test_.
- Rich assertions: It provides easy-to-read assertion failures with detailed error messages.
- Extensibility: Supports plugins and fixtures to make testing more flexible and modular.
- Parallel test execution: Pytest can run tests in parallel, speeding up test suites.
- Supports multiple types of testing: Unit, functional, integration, etc.

Basic Structure of Pytest Tests

1. Test Function Names:

Pytest identifies test functions by their name. By default, Pytest will discover any function in your codebase that begins with test_.

2. Assertions:

Assertions are used to check if the output of a function or block of code matches the expected result.

- Basic assertion:

```
def test_addition():
    assert 1 + 1 == 2
```

- Failing test:

```
def test_addition_fail():
    assert 1 + 1 == 3
```

When a test fails, Pytest will provide detailed information on what went wrong, including the values that were compared.

Test Discovery and Running Tests

1. Automatic Test Discovery:

Once Pytest is installed, it will automatically discover any function starting with test_ within files starting with test_ in your directory.

- Example directory structure:

```
tests/
    test_math.py
    test_string_operations.py
```

- To run all tests:
  Navigate to your tests folder and run:

```
pytest
```

pytest will find all the tests and execute them.

2. Running Specific Tests:

You can also run specific tests by providing the test name or path.

- Run a specific test function:

```
pytest -s tests/test_math.py::test_addition
```

- Run all tests in a specific file:

```
pytest tests/test_math.py
```

- Run tests with a specific marker (useful for tagging tests for features or categories):

```
pytest -m "smoke"
```

3. Run Tests with Output to Terminal:

To print the detailed output while running the tests, use the -s flag:

pytest -s tests/test_math.py

Pytest Advanced Features

1. Parametrization:

You can run a test function with different sets of inputs by using pytest.mark.parametrize.

```
import pytest

@pytest.mark.parametrize("a, b, result", [
    (1, 1, 2),
    (2, 3, 5),
    (5, 6, 11)
])
def test_addition(a, b, result):
    assert a + b == result
```

This will run the test_addition function three times, with different parameter sets.

2. Marking Tests:

You can use markers to categorize tests. For example, you might want to mark certain tests as slow, or for a specific feature.

```
import pytest


@pytest.mark.slow
def test_long_computation():
    # A test that takes a long time
    assert 1 + 1 == 2
```

To run tests marked as slow, you can use:

pytest -m slow

Pytest Configuration

You can configure Pytest's behavior via command-line options or by creating a configuration file, such as pytest.ini, tox.ini, or setup.cfg.

Example pytest.ini:

[pytest]

Add the markers for categorization

markers =

  smoke: Quick smoke tests

  regression: Regression test suite

Now, when you run the tests, Pytest will recognize your custom markers.

Test Reporting and Logs

Pytest supports a variety of reporting tools, including creating reports in html

- Generate an HTML report:

  pytest --html=report.html

Example Directory Structure for Pytest Projects

```
project/
│
├── app/
│   ├── __init__.py
│   ├── math.py
│   └── string_operations.py
│
├── tests/
│   ├── __init__.py
│   ├── test_math.py      # Tests for math.py
│   └── test_string.py    # Tests for string_operations.py
│
└── pytest.ini
```

In this structure:

- The application code resides in the app/ directory.
- The test cases are in the tests/ directory.
- pytest.ini holds configuration options, such as markers.

What is Selenium WebDriver?

Selenium is a tool for automating web browsers. It allows you to programmatically control a web browser, interact with web pages, and perform tests or automate repetitive tasks.

Setting up Selenium in Python:

1. **Install Selenium** via pip:
2. pip install selenium
3. **Install a WebDriver**:
   To interact with browsers, you need the appropriate WebDriver. For example:
   - **Chrome**: Download ChromeDriver

- **Firefox**: Download GeckoDriver

Launch a Browser & Navigate to URLs:

```
from selenium import webdriver


# Open Chrome browser
driver = webdriver.Chrome()  # You can also use webdriver.Firefox()


# Navigate to a website
driver.get("https://www.google.com")


# Print the title of the page
print(driver.title)


# Close the browser window
driver.quit()
```

Explanation:

- webdriver.Chrome(): Creates a new Chrome browser session.
- driver.get("https://www.google.com"): Navigates to the Google homepage.
- driver.title: Fetches the title of the current page.
- driver.quit(): Closes the browser session.

Day 4: Locators and Web Element Interactions

Locators are used to find elements on a web page so that you can interact with them. Selenium supports several ways to locate elements:

- **id**: Finds an element by its unique id attribute.
- **name**: Finds an element by its name attribute.
- **xpath**: Finds an element using XPath, a language for selecting elements in XML documents.
- **css_selector**: Finds an element using CSS selectors.

Examples of Locators:

```
# Find element by id
username_element = driver.find_element("id", "username")
# Find element by name
password_element = driver.find_element("name", "password")
# Find element by XPath
submit_button = driver.find_element("xpath", "//button[@type='submit']")
# Find element by CSS selector
login_button = driver.find_element("css selector", ".login-btn")
```

Interactions with Web Elements:

Once you have located an element, you can perform actions such as sending keystrokes or clicking the element.

```
# Send keys to the username field
username_element.send_keys("your_username")

# Send keys to the password field
password_element.send_keys("your_password")

# Click the login button
login_button.click()
```

Explanation:

- send_keys(): Simulates typing into an input field.
- click(): Simulates a click on the button or link.

Full Example of a Web Interaction (Login Form Automation):

```
from selenium import webdriver
from selenium.webdriver.common.by import By

# Start Chrome
```

```
driver = webdriver.Chrome()

# Go to login page
driver.get("https://example.com/login")

# Find and fill out the username and password fields
driver.find_element(By.ID, "username").send_keys("my_username")
driver.find_element(By.ID, "password").send_keys("my_password")

# Submit the form by clicking the login button
driver.find_element(By.ID, "login-button").click()

# Check if login is successful by verifying the page title
assert "Dashboard" in driver.title

# Close the browser
driver.quit()
```

Explanation:

1. We first open a login page.
2. We interact with form fields by sending a username and password.
3. We submit the form by clicking the login button.
4. We verify that the login was successful by checking if the title of the resulting page contains "Dashboard".