# Python Unit Testing with PyUnit (unittest)

## 1. What is PyUnit (unittest)?

PyUnit (also known as unittest) is the built-in unit testing framework in Python, inspired by Java's JUnit. It supports test automation, sharing of setup and shutdown code, aggregation of tests into collections, and independence of the tests from the reporting framework.

## 2. Why Use unittest?

• To automate and validate small units (functions/classes) of source code.

• Helps in regression testing and code refactoring without breaking existing logic.

• Provides detailed test reports.

• Reduces bugs and improves code quality.

## 3. Recommended Folder Structure

project/  
│  
├── src/  
│ └── cart.py  
│ └── employee.py  
│  
├── tests/  
│ └── test\_cart.py  
│ └── test\_employee.py

## 4. Real-world Example: Add Item to Cart & Add New Employee

cart.py

class Cart:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def add\_item(self, item):  
 if item:  
 self.items.append(item)  
 return True  
 return False  
  
 def get\_items(self):  
 return self.items

employee.py

class EmployeeManager:  
 def \_\_init\_\_(self):  
 self.employees = {}  
  
 def add\_employee(self, emp\_id, name):  
 if emp\_id not in self.employees:  
 self.employees[emp\_id] = name  
 return True  
 return False  
  
 def get\_employee(self, emp\_id):  
 return self.employees.get(emp\_id, None)

## 5. Writing Test Cases

test\_cart.py

import unittest  
from src.cart import Cart  
  
class TestCart(unittest.TestCase):  
 def setUp(self):  
 self.cart = Cart()  
  
 def test\_add\_item\_success(self):  
 self.assertTrue(self.cart.add\_item("Laptop"))  
 self.assertIn("Laptop", self.cart.get\_items())  
  
 def test\_add\_item\_empty(self):  
 self.assertFalse(self.cart.add\_item(""))

test\_employee.py

import unittest  
from src.employee import EmployeeManager  
  
class TestEmployeeManager(unittest.TestCase):  
 def setUp(self):  
 self.manager = EmployeeManager()  
  
 def test\_add\_employee(self):  
 result = self.manager.add\_employee(101, "Alice")  
 self.assertTrue(result)  
 self.assertEqual(self.manager.get\_employee(101), "Alice")  
  
 def test\_add\_duplicate\_employee(self):  
 self.manager.add\_employee(101, "Alice")  
 result = self.manager.add\_employee(101, "Bob")  
 self.assertFalse(result)

## 6. Different Ways to Write Test Cases

• Using unittest.TestCase class (most common, built-in).

• Using pytest (third-party library, more concise syntax).

• Using doctest for inline documentation testing.

### Comparison

• unittest: Verbose, more setup, great for class-based testing.

• pytest: Lightweight, powerful fixtures, easier syntax, popular in modern dev.

• doctest: Good for beginners or validating examples in docstrings.

## 7. Performance Considerations

• Unit tests are fast and should test isolated functionality only.

• Avoid hitting real databases/APIs in unit tests; use mocks or in-memory objects.

• Use pytest-xdist for parallel execution if needed.