Intermediate Python Programming – Lesson 8

Facilitated by Kent State University
Topic: Testing and Writing Reliable Code
Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Understand the purpose of testing in software development
- Write simple unit tests using Python's built-in `unittest` module
- Use assertions and test runners to validate code behavior
- Follow testing best practices to improve code quality and maintainability

---

Lesson 8: Testing and Writing Reliable Code

I. Introduction to Testing (10 minutes)

Testing is a critical part of software development. It ensures that your code does what it's supposed to do and helps prevent regressions when making changes.

Types of testing:

- **Unit Testing**: Tests individual functions or components in isolation
- **Integration Testing**: Tests combined parts of the system together
- **System Testing**: Tests the complete application in a production-like environment

Benefits of Testing:

- Catch bugs early
- Improve code design
- Make refactoring safer
- Enable continuous integration and automated workflows

Multiple-Choice Question:

What is the primary purpose of unit testing?

A. To test user interfaces B. To ensure individual parts of the code work as expected C. To replace documentation D. To measure performance

Answer: B. To ensure individual parts of the code work as expected

Short Answer Question:

Why is automated testing important in modern development workflows?

Expected Answer: It ensures consistent, repeatable verification of code correctness and allows changes to be made with confidence.

II. Writing Unit Tests with `unittest` (20 minutes)

Python's built-in `unittest` framework provides a way to write and run unit tests.

Structure of a test case:

```
import unittest

class MathTests(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(2 + 2, 4)

    def test_division(self):
        self.assertAlmostEqual(10 / 4, 2.5)

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

Common assertions:

- `assertEqual(a, b)`
- `assertNotEqual(a, b)`
- `assertTrue(x)` / `assertFalse(x)`
- `assertRaises(ExceptionType)`

Exercise 1:

Write a function `is_even(n)` and test it using `unittest` to ensure it returns `True` for even numbers and `False` otherwise.

III. Using Assertions and Test Runners (10 minutes)

In addition to test cases, Python also supports simple assertion-based tests using the `assert` statement:

```
def square(x):
    return x * x

assert square(3) == 9
assert square(-4) == 16
```

Assertions are useful for quick checks, but for production-quality testing, structured frameworks like `unittest` or `pytest` are preferred.

Running Tests:

- Use `python test_file.py`
- Or run via IDEs and CI pipelines
- `pytest` can auto-discover tests with minimal boilerplate

Exercise 2:

Convert the assertion-based checks above into a `unittest.TestCase` class.

IV. Best Practices for Testing (10 minutes)

- Keep tests **isolated** and independent from each other
- Use descriptive test names
- Test **normal cases**, **edge cases**, and **error cases**
- Include tests in version control
- Run tests automatically on code changes (e.g., GitHub Actions)

Recommended Directory Structure:

```
project/
|– src/
|– tests/
    |– test_module1.py
    |– test_module2.py
```

Exercise 3:

Create a `calculator.py` module with functions for add, subtract, multiply, and divide. Write a `test_calculator.py` that tests each function.

V. Recap and Q&A (10 minutes)

- Unit testing verifies code correctness in isolation
- The `unittest` module provides structure and consistency
- Assertions validate assumptions
- Testing supports better design and reduces the cost of bugs

Final Exercise:

Use `unittest` to test a custom exception that is raised when a withdrawal exceeds an account balance in a simple banking class.
