Python Programming Bootcamp

Week 8: Testing, Best Practices & Project Design

Center for Artificial Intelligence & Emerging Technologies

November 8, 2025

Quick Recap: What We've Learned

Week 1:

- Python basics, data types, operators, conditionals, loops

Week 2:

- Data structures: Lists, strings, tuples, dictionaries, sets

Week 3:

- Functions, parameters, return values, modules and imports

Week 4:

- File I/O (text, CSV, JSON), Error handling (try-except)

Quick Recap (continued)

Week 5:

- OOP fundamentals: Classes, objects, methods, association, UML

Week 6:

- Advanced OOP: Encapsulation, inheritance, polymorphism, abstraction

Week 7:

- Advanced functions: Lambda, map/filter/reduce, generators, decorators

Week 8: Overview - Part 1

This Week's Focus
Professional development practices and preparing for real-world projects

Part 1: Testing & Debugging

- Unit testing with unittest and pytest
- Writing effective test cases
- Test-driven development (TDD)
- Debugging techniques and tools
- Code coverage concepts

Week 8: Overview - Part 2

Part 2: Best Practices & Project Design

- Code organization and structure
- Documentation and docstrings
- Virtual environments and pip
- Project planning and design patterns
- Git basics for version control

Capstone Project
You'll apply all these concepts in a complete Python application

Learning Objectives

By the end of this week, you will be able to:

- Write and run unit tests for your Python code
- Apply test-driven development principles
- Use debugging tools to find and fix errors
- Organize code into maintainable modules and packages
- Document code effectively with docstrings
- Manage project dependencies with virtual environments
- Plan and structure software projects
- Use basic Git commands for version control
- Apply common design patterns
- Build a complete Python application (Capstone Project)

Part 1: Testing & Debugging

Why Testing Matters

The Importance of Testing
Testing ensures your code works correctly and continues to work as you make changes

Benefits of Testing:

- **Catch bugs early:** Find errors before users do
- **Confidence in changes:** Refactor without fear
- **Documentation:** Tests show how code should be used
- **Better design:** Writing testable code leads to better structure
- **Save time:** Automated tests are faster than manual testing

Real-World Impact
Professional developers spend 30-50% of their time writing tests!

Types of Testing

Unit Testing:

- Test individual functions/methods
- Fast and isolated
- Most common type
- Focus of this week

Integration Testing:

- Test multiple components together
- Verify interactions work
- More complex setup

Functional Testing:

- Test entire features
- User's perspective
- End-to-end workflows

Regression Testing:

- Ensure fixes don't break existing features
- Run all tests regularly
- Automated with CI/CD

Introduction to unittest i

unittest Module
Python's built-in testing framework (no installation needed)

Basic Structure:

```python
import unittest

class TestMyFunction(unittest.TestCase):
    def test_something(self):
        # Arrange: Set up test data
        result = my_function(input_data)

        # Assert: Check the result
        self.assertEqual(result, expected_value)
```

Introduction to unittest ii

```python

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

Key Components:

- Import `unittest`
- Create test class inheriting from `unittest.TestCase`
- Test methods must start with `test_`
- Use assertion methods to verify results

Example: Testing a Calculator i

calculator.py:

```python
def add(a, b):
    """Add two numbers"""
    return a + b

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

def is_even(n):
```

Example: Testing a Calculator ii

```
    """Check if number is even"""
    return n % 2 == 0
```

Example: Testing a Calculator iii

test_calculator.py:

```python
import unittest
from calculator import add, divide, is_even

class TestCalculator(unittest.TestCase):

    def test_add_positive_numbers(self):
        """Test adding positive numbers"""
        result = add(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        """Test adding negative numbers"""
```

Example: Testing a Calculator iv

```python
        result = add(-1, -1)
        self.assertEqual(result, -2)

    def test_divide_normal(self):
        """Test normal division"""
        result = divide(10, 2)
        self.assertEqual(result, 5)

    def test_divide_by_zero(self):
        """Test division by zero raises error"""
        with self.assertRaises(ValueError):
            divide(10, 0)

```

Example: Testing a Calculator v

```
    def test_is_even_true(self):
        """Test even number returns True"""
        self.assertTrue(is_even(4))

    def test_is_even_false(self):
        """Test odd number returns False"""
        self.assertFalse(is_even(3))

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

Common Assertion Methods i

```python
# Equality assertions
self.assertEqual(a, b)        # a == b
self.assertNotEqual(a, b)     # a != b

# Boolean assertions
self.assertTrue(x)            # x is True
self.assertFalse(x)           # x is False

# Membership assertions
self.assertIn(item, list)     # item in list
self.assertNotIn(item, list)  # item not in list

# Exception assertions
```

Common Assertion Methods ii

```python
with self.assertRaises(ValueError):
    function_that_raises()

# Approximate equality (for floats)
self.assertAlmostEqual(a, b, places=2)

# Identity assertions
self.assertIs(a, b)           # a is b
self.assertIsNone(x)          # x is None
self.assertIsNotNone(x)       # x is not None

# Type assertions
self.assertIsInstance(obj, MyClass)
```

Common Assertion Methods iii

setUp and tearDown Methods i

Test Fixtures
setUp runs before each test, tearDown runs after each test

```python
import unittest

class TestBankAccount(unittest.TestCase):

    def setUp(self):
        """Run before each test method"""
        self.account = BankAccount("John", 1000)
        print("Setting up test...")

    def tearDown(self):
```

setUp and tearDown Methods ii

```python
        """Run after each test method"""
        self.account = None
        print("Cleaning up test...")

    def test_deposit(self):
        self.account.deposit(500)
        self.assertEqual(self.account.balance, 1500)

    def test_withdraw(self):
        self.account.withdraw(300)
        self.assertEqual(self.account.balance, 700)

    def test_withdraw_insufficient_funds(self):
```

setUp and tearDown Methods iii

```
        with self.assertRaises(ValueError):
            self.account.withdraw(2000)
```

Benefits:

- Avoid code duplication
- Each test starts with fresh state
- Automatic cleanup after tests

Running Tests i

Method 1: Direct execution

```
$ python test_calculator.py
...
--------------------------------------------------------------------

Ran 6 tests in 0.001s

OK
```

Method 2: Using unittest discovery

Running Tests ii

```
$ python -m unittest discover
# Finds all test_*.py files automatically
```

Method 3: Run specific test

```
$ python -m unittest test_calculator.TestCalculator.
    test_add_positive_numbers
```

Verbose output:

Running Tests iii

```
$ python -m unittest -v test_calculator.py
test_add_negative_numbers (test_calculator.TestCalculator) ... ok
test_add_positive_numbers (test_calculator.TestCalculator) ... ok
test_divide_by_zero (test_calculator.TestCalculator) ... ok
test_divide_normal (test_calculator.TestCalculator) ... ok
test_is_even_false (test_calculator.TestCalculator) ... ok
test_is_even_true (test_calculator.TestCalculator) ... ok
```

Introduction to pytest i

pytest
Modern, powerful testing framework with simpler syntax than unittest

Why pytest?

- Simpler syntax (no classes required)
- Better error messages
- Fixtures for setup/teardown
- Powerful plugins ecosystem
- Can run unittest tests too

Installation:

Introduction to pytest ii

```
$ pip install pytest
```

pytest Example i

test_calculator_pytest.py:

```python
import pytest
from calculator import add, divide, is_even

def test_add_positive_numbers():
    """Test adding positive numbers"""
    assert add(2, 3) == 5

def test_add_negative_numbers():
    """Test adding negative numbers"""
    assert add(-1, -1) == -2

```

pytest Example ii

```python
def test_divide_normal():
    """Test normal division"""
    assert divide(10, 2) == 5

def test_divide_by_zero():
    """Test division by zero raises error"""
    with pytest.raises(ValueError):
        divide(10, 0)

def test_is_even_true():
    """Test even number returns True"""
    assert is_even(4) is True

```

pytest Example iii

```python
def test_is_even_false():
    """Test odd number returns False"""
    assert is_even(3) is False
```

Key Differences from unittest:

- No class needed (just functions)
- Use Python's `assert` keyword
- Simpler and more readable

pytest Fixtures i

Fixtures
Reusable setup code for tests (similar to setUp but more powerful)

```python
import pytest

@pytest.fixture
def bank_account():
    """Create a bank account for testing"""
    account = BankAccount("John", 1000)
    return account

def test_deposit(bank_account):
    """Test gets the fixture automatically"""
```

pytest Fixtures ii

```python
    bank_account.deposit(500)
    assert bank_account.balance == 1500

def test_withdraw(bank_account):
    """Each test gets a fresh fixture"""
    bank_account.withdraw(300)
    assert bank_account.balance == 700

@pytest.fixture
def sample_data():
    """Fixture can return any data"""
    return [1, 2, 3, 4, 5]

```

pytest Fixtures iii

```python
def test_sum(sample_data):
    assert sum(sample_data) == 15

def test_length(sample_data):
    assert len(sample_data) == 5
```

Parametrized Tests with pytest i

Parametrization
Run the same test with different input values

```
import pytest

@pytest.mark.parametrize("a, b, expected", [
    (2, 3, 5),
    (0, 0, 0),
    (-1, 1, 0),
    (10, 20, 30),
    (-5, -5, -10)
])
def test_add(a, b, expected):
```

Parametrized Tests with pytest ii

```python
    """Test add function with multiple inputs"""
    assert add(a, b) == expected

@pytest.mark.parametrize("n, expected", [
    (2, True),
    (3, False),
    (0, True),
    (-4, True),
    (-3, False)
])
def test_is_even(n, expected):
    """Test is_even with multiple inputs"""
    assert is_even(n) == expected
```

Parametrized Tests with pytest iii

Benefits:

- Test multiple scenarios without code duplication
- Each parameter combination is reported separately
- Easy to add new test cases

Running pytest i

Basic commands:

```
$ pytest                    # Run all tests
$ pytest test_file.py       # Run specific file
$ pytest -v                 # Verbose output
$ pytest -v -s              # Verbose + print statements
$ pytest -k "add"           # Run tests matching pattern
$ pytest --maxfail=1        # Stop after first failure
```

Example output:

Running pytest ii

```
$ pytest -v test_calculator_pytest.py
========================= test session starts
   ==========================
test_calculator_pytest.py::test_add_positive_numbers PASSED
        [ 16%]
test_calculator_pytest.py::test_add_negative_numbers PASSED
        [ 33%]
test_calculator_pytest.py::test_divide_normal PASSED
               [ 50%]
test_calculator_pytest.py::test_divide_by_zero PASSED
              [ 66%]
```

Running pytest iii

```
test_calculator_pytest.py::test_is_even_true PASSED
                [ 83%]
test_calculator_pytest.py::test_is_even_false PASSED
               [100%]
========================== 6 passed in 0.02s
   ===========================
```

What is Test-Driven Development?

TDD Philosophy
Write tests **before** writing the actual code

The TDD Cycle (Red-Green-Refactor):

1. **Red:** Write a failing test
2. **Green:** Write minimal code to make it pass
3. **Refactor:** Improve the code while keeping tests passing
4. Repeat

Benefits:

- Forces you to think about requirements first
- Ensures 100% test coverage
- Code is automatically testable

TDD Example: Password Validator i

Step 1: Write the test first (Red)

```python
# test_password.py
import pytest
from password_validator import is_valid_password

def test_password_length():
    """Password must be at least 8 characters"""
    assert is_valid_password("short") is False
    assert is_valid_password("long_enough_pass") is True
```

Run test - it fails (function doesn't exist yet):

TDD Example: Password Validator ii

```
$ pytest test_password.py
ModuleNotFoundError: No module named 'password_validator'
```

TDD Example: Password Validator iii

Step 2: Write minimal code to pass (Green)

```
# password_validator.py
def is_valid_password(password):
    """Check if password is valid"""
    return len(password) >= 8
```

Run test - it passes:

```
$ pytest test_password.py
========================= 1 passed in 0.01s
    ===========================
```

TDD Example: Password Validator iv

Step 3: Add more requirements (Red again)

```python
def test_password_has_uppercase():
    """Password must contain uppercase letter"""
    assert is_valid_password("alllowercase123") is False
    assert is_valid_password("HasUpperCase123") is True

def test_password_has_digit():
    """Password must contain a digit"""
    assert is_valid_password("NoDigitsHere") is False
    assert is_valid_password("HasDigit1") is True
```

TDD Example: Password Validator v

Step 4: Implement new requirements (Green)

```python
def is_valid_password(password):
    """Check if password is valid"""
    # Check length
    if len(password) < 8:
        return False

    # Check for uppercase
    if not any(c.isupper() for c in password):
        return False

    # Check for digit
    if not any(c.isdigit() for c in password):
```

TDD Example: Password Validator vi

```
        return False

    return True
```

TDD Example: Password Validator vii

Step 5: Refactor for clarity

```python
def is_valid_password(password):
    """
    Check if password is valid.
    Requirements:
    - At least 8 characters
    - Contains uppercase letter
    - Contains digit
    """
    checks = [
        len(password) >= 8,
        any(c.isupper() for c in password),
        any(c.isdigit() for c in password)
```

TDD Example: Password Validator viii

```
    ]

    return all(checks)
```

All tests still pass!

```
$ pytest test_password.py -v
========================= 3 passed in 0.01s ===========================
```

Common Debugging Strategies - Part 1

1. **Print Debugging (Simple but effective)**

    - Add `print()` statements to see variable values
    - Track program flow
    - Quick and easy for simple bugs

2. **Using the Python Debugger (pdb)**

    - Step through code line by line
    - Inspect variables at any point
    - Set breakpoints
    - More powerful than print debugging

Common Debugging Strategies - Part 2

3. IDE Debuggers

- Visual debugging with breakpoints
- Watch variables in real-time
- Step over/into/out of functions
- Available in VS Code, PyCharm, etc.

Choose the Right Tool

Start with print debugging for simple issues, use pdb or IDE debuggers for complex problems!

Print Debugging i

```python
def calculate_average(numbers):
    """Calculate average of numbers"""
    print(f"Input: {numbers}")  # Debug: See input

    total = sum(numbers)
    print(f"Total: {total}")    # Debug: See sum

    count = len(numbers)
    print(f"Count: {count}")    # Debug: See count

    average = total / count
    print(f"Average: {average}") # Debug: See result

```

Print Debugging ii

```python
    return average

# Test with empty list (will cause error)
result = calculate_average([])
```

Output reveals the bug:

```
Input: []
Total: 0
Count: 0
ZeroDivisionError: division by zero
```

Print Debugging iii

Fixed version with validation:

```python
def calculate_average(numbers):
    """Calculate average of numbers"""
    if not numbers:
        raise ValueError("Cannot calculate average of empty list")

    total = sum(numbers)
    count = len(numbers)
    average = total / count

    return average
```

Using pdb (Python Debugger) i

```python
import pdb

def find_maximum(numbers):
    """Find maximum number in list"""
    pdb.set_trace()  # Debugger will stop here

    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num

    return max_num

```

Using pdb (Python Debugger) ii

```
result = find_maximum([3, 1, 4, 1, 5, 9, 2, 6])
```

Common pdb commands:

- `n` (next) - Execute next line
- `s` (step) - Step into function
- `c` (continue) - Continue until next breakpoint
- `p variable` - Print variable value
- `l` (list) - Show current code
- `h` (help) - Show help
- `q` (quit) - Exit debugger

Using pdb (Python Debugger) iii

Modern alternative: breakpoint() (Python 3.7+)

```python
def process_data(data):
    """Process some data"""
    result = []

    for item in data:
        breakpoint()  # Easier than import pdb; pdb.set_trace()
        processed = item * 2
        result.append(processed)

    return result

# When you run this, it will drop into debugger
```

Using pdb (Python Debugger) iv

```python
process_data([1, 2, 3, 4, 5])
```

Debugging Best Practices i

1. Read the error message carefully:

```
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    result = process_data(data)
  File "script.py", line 5, in process_data
    value = data['key']
KeyError: 'key'
```

- Error type: `KeyError`
- Location: line 5 in `process_data()`
- Problem: `'key'` doesn't exist in dictionary

Debugging Best Practices ii

2. Use descriptive error messages:

```
if age < 0:
    raise ValueError(f"Age cannot be negative, got {age}")

if not filename.endswith('.txt'):
    raise ValueError(f"Expected .txt file, got {filename}")
```

Debugging Best Practices iii

3. **Add logging for complex applications:**

```python
import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

def process_transaction(amount):
    logger.debug(f"Processing transaction: ${amount}")

    if amount < 0:
        logger.error(f"Invalid amount: {amount}")
        raise ValueError("Amount must be positive")

```

Debugging Best Practices iv

```
    logger.info(f"Transaction successful: ${amount}")
    return True
```

4. Write tests to catch bugs:

```
def test_process_transaction_negative():
    """Negative amounts should raise error"""
    with pytest.raises(ValueError):
        process_transaction(-100)
```

What is Code Coverage?

Code Coverage
Measure of how much of your code is executed by your tests

Types of Coverage:

- **Line coverage:** Percentage of lines executed
- **Branch coverage:** Percentage of if/else branches tested
- **Function coverage:** Percentage of functions called

Why it matters:

- Identifies untested code
- Helps find missing test cases
- Not a perfect metric (100% coverage doesn't mean bug-free)
- Good guideline: Aim for 80-90% coverage

Using coverage.py i

Installation:

```
$ pip install coverage
```

Running coverage with unittest:

```
$ coverage run -m unittest test_calculator.py
$ coverage report
```

Running coverage with pytest:

```
$ pip install pytest-cov
$ pytest --cov=calculator test_calculator.py
```

Using coverage.py ii

Example coverage report:

```
Name                Stmts   Miss  Cover
---------------------------------------
calculator.py          10      2    80%
test_calculator.py     25      0   100%
---------------------------------------
TOTAL                  35      2    94%
```

Detailed HTML report:

Using coverage.py iii

```
$ coverage html
# Opens htmlcov/index.html in browser
# Shows exactly which lines are not covered
```

Missing coverage example:

```python
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")  # Covered
    return a / b  # Not covered if we never test normal division!
```

Part 2: Best Practices & Project Design

Why Code Organization Matters

Organized Code is Maintainable Code
Good structure makes code easier to understand, test, and modify

Benefits of Good Organization:

- **Readability:** Others (and future you) can understand it
- **Maintainability:** Easy to fix bugs and add features
- **Testability:** Well-organized code is easier to test
- **Reusability:** Modular code can be reused
- **Collaboration:** Teams can work on different parts

Remember
Code is read far more often than it's written!

Project Structure Best Practices i

Typical Python Project Structure:

```
my_project/
|-- README.md           # Project description
|-- requirements.txt    # Dependencies
|-- setup.py           # Package configuration
|-- .gitignore         # Git ignore file
|-- src/               # Source code
|   |-- __init__.py
|   |-- main.py        # Entry point
|   |-- module1.py
|   |-- module2.py
|   |-- utils/         # Utility functions
```

Project Structure Best Practices ii

```
|       |-- __init__.py
|       |-- helpers.py
|-- tests/             # Test files
|   |-- __init__.py
|   |-- test_module1.py
|   |-- test_module2.py
|-- docs/              # Documentation
|-- data/              # Data files
|-- config/            # Configuration files
```

Organizing Code into Modules i

Bad: Everything in one file (main.py - 500 lines)

```python
# All code in one huge file
class User:
    pass

class Product:
    pass

class Order:
    pass

def validate_email(email):
```

Organizing Code into Modules ii

```python
    pass

def send_email(to, subject, body):
    pass

# ... 500 more lines ...
```

Organizing Code into Modules iii

Good: Organized into logical modules

```python
# models/user.py
class User:
    pass

# models/product.py
class Product:
    pass

# models/order.py
class Order:
    pass

```

Organizing Code into Modules iv

```
# utils/validation.py
def validate_email(email):
    pass

# utils/email.py
def send_email(to, subject, body):
    pass

# main.py
from models.user import User
from models.product import Product
from utils.email import send_email
```

Creating Python Packages i

Package structure with __init__.py:

```
ecommerce/
|-- __init__.py        # Makes it a package
|-- models/
|   |-- __init__.py
|   |-- user.py
|   |-- product.py
|-- services/
|   |-- __init__.py
|   |-- payment.py
|   |-- shipping.py
|-- utils/
```

Creating Python Packages ii

```
    |-- __init__.py
    |-- validators.py
```

Creating Python Packages iii

ecommerce/__init__.py:

```python
"""E-commerce package"""
__version__ = "1.0.0"

# Make common imports available at package level
from .models.user import User
from .models.product import Product
```

Usage:

Creating Python Packages iv

```python
# Can import directly from package
from ecommerce import User, Product

# Or from specific modules
from ecommerce.services.payment import process_payment
from ecommerce.utils.validators import validate_email
```

Code Organization Principles i

1. **Single Responsibility Principle:**

```python
# Bad: Class does too many things
class User:
    def save_to_database(self):
        pass

    def send_welcome_email(self):
        pass

    def generate_pdf_report(self):
        pass

```

Code Organization Principles ii

```python
# Good: Separate responsibilities
class User:
    """User model - only handles user data"""
    pass

class UserRepository:
    """Handles database operations"""
    def save(self, user):
        pass

class EmailService:
    """Handles email sending"""
    def send_welcome_email(self, user):
```

Code Organization Principles iii

```
        pass

class ReportGenerator:
    """Handles report generation"""
    def generate_user_report(self, user):
        pass
```

Code Organization Principles iv

2. Don’t Repeat Yourself (DRY):

```python
# Bad: Repeated code
def calculate_student_average(grades):
    total = sum(grades)
    return total / len(grades)

def calculate_class_average(all_grades):
    total = sum(all_grades)
    return total / len(all_grades)

# Good: Reuse common functionality
def calculate_average(numbers):
    """Generic average calculator"""
```

Code Organization Principles v

```
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)

def calculate_student_average(grades):
    return calculate_average(grades)

def calculate_class_average(all_grades):
    return calculate_average(all_grades)
```

Code Organization Principles vi

3. **Keep functions small and focused:**

```python
# Bad: Function does too much
def process_order(order_data):
    # Validate data
    if not order_data.get('customer'):
        raise ValueError("Missing customer")
    # Calculate total
    total = sum(item['price'] for item in order_data['items'])
    # Apply discount
    if total > 100:
        total *= 0.9
    # Save to database
    db.save(order_data)
```

Code Organization Principles vii

```
    # Send confirmation email
    send_email(order_data['customer']['email'])
    return total

# Good: Split into focused functions
def validate_order(order_data):
    if not order_data.get('customer'):
        raise ValueError("Missing customer")

def calculate_total(items):
    return sum(item['price'] for item in items)

def apply_discount(total):
```

Code Organization Principles viii

```
    return total * 0.9 if total > 100 else total

def process_order(order_data):
    validate_order(order_data)
    total = calculate_total(order_data['items'])
    total = apply_discount(total)
    db.save(order_data)
    send_email(order_data['customer']['email'])
    return total
```

Why Documentation Matters

Good Documentation
Explains what code does, how to use it, and why decisions were made

Types of Documentation:

- **Docstrings:** Built into the code (functions, classes, modules)
- **Comments:** Explain complex logic
- **README files:** Project overview and setup instructions
- **API documentation:** How to use your library/package
- **Type hints:** Show expected data types

Remember
Code tells you HOW, documentation tells you WHY!

Writing Good Docstrings i

Python docstring formats (we'll use Google style):

```python
def calculate_discount(price, discount_percent):
    """
    Calculate the discounted price.

    Args:
        price (float): Original price of the item
        discount_percent (float): Discount percentage (0-100)

    Returns:
        float: The discounted price

```

Writing Good Docstrings ii

```
    Raises:
        ValueError: If discount_percent is not between 0 and 100

    Example:
        >>> calculate_discount(100, 20)
        80.0
    """
    if not 0 <= discount_percent <= 100:
        raise ValueError("Discount must be between 0 and 100")

    discount_amount = price * (discount_percent / 100)
    return price - discount_amount
```

Writing Good Docstrings iii

Docstrings for Classes i

```python
class BankAccount:
    """
    A bank account with basic operations.

    This class manages a single bank account with deposit,
    withdrawal, and balance tracking capabilities.

    Attributes:
        account_holder (str): Name of the account holder
        balance (float): Current account balance
        account_number (str): Unique account identifier

    Example:
```

Docstrings for Classes ii

```python
        >>> account = BankAccount("John Doe", 1000)
        >>> account.deposit(500)
        >>> print(account.balance)
        1500.0
    """

    def __init__(self, account_holder, initial_balance=0):
        """
        Initialize a new bank account.

        Args:
            account_holder (str): Name of the account holder
```

Docstrings for Classes iii

```
            initial_balance (float, optional): Starting balance.
                Defaults to 0.
        """
        self.account_holder = account_holder
        self.balance = initial_balance
        self.account_number = self._generate_account_number()

    def deposit(self, amount):
        """
        Deposit money into the account.

        Args:
```

Docstrings for Classes iv

```
            amount (float): Amount to deposit (must be positive)

        Raises:
            ValueError: If amount is negative or zero
        """
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount
```

Type Hints for Better Documentation i

```python
from typing import List, Dict, Optional, Union

def process_students(
    students: List[Dict[str, Union[str, int]]],
    passing_grade: int = 60
) -> Dict[str, List[str]]:
    """
    Categorize students into passed and failed.

    Args:
        students: List of student dictionaries with ’name’ and ’grade’
        passing_grade: Minimum grade to pass (default: 60)
```

Type Hints for Better Documentation ii

```

    Returns:
        Dictionary with 'passed' and 'failed' lists of student names
    """
    passed = []
    failed = []

    for student in students:
        if student['grade'] >= passing_grade:
            passed.append(student['name'])
        else:
            failed.append(student['name'])
```

Type Hints for Better Documentation iii

```

    return {’passed’: passed, ’failed’: failed}

def find_student(
    students: List[Dict[str, str]],
    name: str
) -> Optional[Dict[str, str]]:
    """
    Find a student by name.

    Args:
        students: List of student dictionaries
        name: Name to search for
```

Type Hints for Better Documentation iv

```

    Returns:
        Student dictionary if found, None otherwise
    """
    for student in students:
        if student[’name’] == name:
            return student
    return None
```

Comments: When and How i

Good comments explain WHY, not WHAT:

```python
# Bad: Comment repeats what code does
# Increment i by 1
i += 1

# Good: Comment explains why
# Skip the header row when processing CSV
i += 1

# Bad: Obvious comment
# Create a list
numbers = []
```

Comments: When and How ii

```python

# Good: Explains reasoning
# Use list instead of set to preserve insertion order
numbers = []

# Bad: Redundant comment
def calculate_total(items):
    # Calculate the total price
    total = sum(item.price for item in items)
    return total

# Good: Explains business logic
def calculate_total(items):
```

Comments: When and How iii

```
    # Apply bulk discount if more than 10 items
    # Business rule: 5% off for orders > 10 items
    total = sum(item.price for item in items)
    if len(items) > 10:
        total *= 0.95
    return total
```

Why Virtual Environments? - Part 1

Virtual Environment
Isolated Python environment with its own packages and dependencies

Problems without virtual environments:

- Project A needs Django 3.2, Project B needs Django 4.0
- Installing packages globally can cause conflicts
- Hard to track which packages a project needs
- Difficult to share project with others

Why Virtual Environments? - Part 2

Benefits of virtual environments:

- Each project has its own dependencies
- No conflicts between projects
- Easy to recreate environment on another machine
- Clean separation of concerns

Best Practice
Always use virtual environments for Python projects!

Creating Virtual Environments i

Using venv (built-in):

```
# Create virtual environment
$ python -m venv myenv

# Activate on Linux/Mac
$ source myenv/bin/activate

# Activate on Windows
$ myenv\Scripts\activate

# Your prompt now shows (myenv)
(myenv) $
```

Creating Virtual Environments ii

```

# Deactivate when done
(myenv) $ deactivate
```

Project structure with venv:

```
my_project/
|-- venv/              # Virtual environment (don't commit to git!)
|-- src/
|-- tests/
|-- requirements.txt
|-- .gitignore         # Add venv/ to this
```

Creating Virtual Environments iii

Managing Packages with pip i

Installing packages:

```
# Install a package
(myenv) $ pip install requests

# Install specific version
(myenv) $ pip install requests==2.28.0

# Install multiple packages
(myenv) $ pip install requests pandas numpy

# Upgrade a package
(myenv) $ pip install --upgrade requests
```

Managing Packages with pip ii

```

# Uninstall a package
(myenv) $ pip uninstall requests
```

Managing Packages with pip iii

requirements.txt:

```
# Generate requirements.txt from current environment
(myenv) $ pip freeze > requirements.txt

# Install all packages from requirements.txt
(myenv) $ pip install -r requirements.txt
```

Example requirements.txt:

Managing Packages with pip iv

```
requests==2.28.0
pandas==1.5.0
numpy==1.23.0
pytest==7.2.0
```

Typical workflow:

```
# 1. Create project
$ mkdir my_project && cd my_project

# 2. Create virtual environment
$ python -m venv venv
```

Managing Packages with pip v

```

# 3. Activate it
$ source venv/bin/activate

# 4. Install packages
(venv) $ pip install requests pytest

# 5. Save dependencies
(venv) $ pip freeze > requirements.txt

# 6. Commit requirements.txt to git (not venv/)
```

Software Development Lifecycle - Part 1

SDLC Phases
Structured approach to building software projects

1. Requirements Gathering

- What problem are we solving?
- Who are the users?
- What features are needed?
- Document functional and non-functional requirements

2. Design

- How will the system work?
- What classes/modules are needed?
- Database design, API design

Software Development Lifecycle - Part 2

3. Implementation

- Write the code
- Follow best practices and coding standards
- Write tests alongside code
- Use version control (Git)

4. Testing & Deployment

- Test thoroughly (unit, integration, system tests)
- Fix bugs and issues
- Deploy to production
- Monitor system performance

Software Development Lifecycle - Part 3

5. Maintenance

- Add new features based on user feedback
- Fix bugs reported by users
- Improve performance and scalability
- Refactor and optimize code
- Update documentation

Iterative Process
SDLC is cyclical - you continuously improve and enhance your software!

Breaking Down Problems i

Example: Library Management System

Step 1: Identify main entities (nouns)

- Book
- Member
- Library
- Loan/Borrow

Step 2: Identify actions (verbs)

- Add book

Breaking Down Problems ii

- Remove book
- Register member
- Borrow book
- Return book
- Search books

Breaking Down Problems iii

Step 3: Define classes and methods

```python
class Book:
    """Represents a book in the library"""
    def __init__(self, isbn, title, author):
        self.isbn = isbn
        self.title = title
        self.author = author
        self.is_available = True

class Member:
    """Represents a library member"""
    def __init__(self, member_id, name):
        self.member_id = member_id
```

Breaking Down Problems iv

```python
        self.name = name
        self.borrowed_books = []

class Library:
    """Main library management system"""
    def __init__(self):
        self.books = {}
        self.members = {}

    def add_book(self, book):
        """Add a book to the library"""
        pass

```

Breaking Down Problems v

```python
    def register_member(self, member):
        """Register a new member"""
        pass

    def borrow_book(self, member_id, isbn):
        """Allow member to borrow a book"""
        pass

    def return_book(self, member_id, isbn):
        """Process book return"""
        pass

    def search_books(self, keyword):
```

Breaking Down Problems vi

```
        """Search for books by title or author"""
        pass
```

Breaking Down Problems vii

Step 4: Plan the workflow

```python
# Borrow book workflow
def borrow_book(self, member_id, isbn):
    """
    Workflow:
    1. Validate member exists
    2. Validate book exists
    3. Check if book is available
    4. Check member's borrow limit (max 3 books)
    5. Mark book as unavailable
    6. Add book to member's borrowed list
    7. Record borrow date
    """
```

Breaking Down Problems viii

```python
    # Implement validation
    if member_id not in self.members:
        raise ValueError("Member not found")

    if isbn not in self.books:
        raise ValueError("Book not found")

    book = self.books[isbn]
    if not book.is_available:
        raise ValueError("Book is not available")

    member = self.members[member_id]
    if len(member.borrowed_books) >= 3:
```

Breaking Down Problems ix

```python
            raise ValueError("Member has reached borrow limit")

        # Process the borrowing
        book.is_available = False
        member.borrowed_books.append(isbn)

        return f"Book '{book.title}' borrowed successfully"
```

What are Design Patterns?

Design Patterns
Reusable solutions to common programming problems

Benefits:

- Proven solutions to recurring problems
- Makes code more maintainable
- Common vocabulary for developers
- Promotes best practices

We'll cover three common patterns:

1. Singleton Pattern
2. Factory Pattern
3. Strategy Pattern

1. Singleton Pattern i

Purpose
Ensure a class has only one instance and provide global access to it

Use case: Database connection, configuration manager, logger

```python
class DatabaseConnection:
    """Singleton: Only one database connection exists"""
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.connection = "Connected to DB"
        return cls._instance
```

1. Singleton Pattern ii

```

# Usage
db1 = DatabaseConnection()
db2 = DatabaseConnection()

print(db1 is db2)  # True - same instance!
print(id(db1) == id(db2))  # True - same memory address
```

1. Singleton Pattern iii

Real-world example: Configuration Manager

```python
class Config:
    """Singleton configuration manager"""
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.settings = {}
        return cls._instance

    def set(self, key, value):
        self.settings[key] = value
```

1. Singleton Pattern iv

```

    def get(self, key):
        return self.settings.get(key)

# Anywhere in your application
config = Config()
config.set(’database_url’, ’localhost:5432’)

# In another file
config = Config()  # Gets the same instance
print(config.get(’database_url’))  # ’localhost:5432’
```

2. Factory Pattern i

Purpose
Create objects without specifying the exact class to create

Use case: Creating different types of objects based on input

```python
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

class Bird:
```

2. Factory Pattern ii

```
    def speak(self):
        return "Tweet!"

class AnimalFactory:
    """Factory to create animals"""
    @staticmethod
    def create_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        elif animal_type == "bird":
            return Bird()
```

2. Factory Pattern iii

```python
        else:
            raise ValueError(f"Unknown animal type: {animal_type}")

# Usage
factory = AnimalFactory()
animal = factory.create_animal("dog")
print(animal.speak())  # "Woof!"

animal = factory.create_animal("cat")
print(animal.speak())  # "Meow!"
```

2. Factory Pattern iv

Real-world example: Payment Processing

```python
class CreditCardPayment:
    def pay(self, amount):
        return f"Paid ${amount} with Credit Card"

class PayPalPayment:
    def pay(self, amount):
        return f"Paid ${amount} with PayPal"

class BankTransferPayment:
    def pay(self, amount):
        return f"Paid ${amount} with Bank Transfer"

```

2. Factory Pattern v

```
class PaymentFactory:
    """Factory for creating payment processors"""
    @staticmethod
    def create_payment(payment_type):
        if payment_type == "credit_card":
            return CreditCardPayment()
        elif payment_type == "paypal":
            return PayPalPayment()
        elif payment_type == "bank_transfer":
            return BankTransferPayment()
        else:
            raise ValueError(f"Unknown payment type: {payment_type}")
```

2. Factory Pattern vi

```python

# Usage in checkout
def process_checkout(amount, payment_method):
    factory = PaymentFactory()
    payment = factory.create_payment(payment_method)
    return payment.pay(amount)

print(process_checkout(100, "credit_card"))
print(process_checkout(50, "paypal"))
```

3. Strategy Pattern i

Purpose
Define a family of algorithms and make them interchangeable

Use case: Different ways to perform the same task

```python
from abc import ABC, abstractmethod

class SortStrategy(ABC):
    """Abstract strategy for sorting"""
    @abstractmethod
    def sort(self, data):
        pass

class BubbleSortStrategy(SortStrategy):
```

3. Strategy Pattern ii

```python
    def sort(self, data):
        # Simplified bubble sort
        n = len(data)
        for i in range(n):
            for j in range(0, n-i-1):
                if data[j] > data[j+1]:
                    data[j], data[j+1] = data[j+1], data[j]
        return data

class QuickSortStrategy(SortStrategy):
    def sort(self, data):
        if len(data) <= 1:
            return data
```

3. Strategy Pattern iii

```python
        pivot = data[len(data) // 2]
        left = [x for x in data if x < pivot]
        middle = [x for x in data if x == pivot]
        right = [x for x in data if x > pivot]
        return self.sort(left) + middle + self.sort(right)

class Sorter:
    """Context that uses a strategy"""
    def __init__(self, strategy: SortStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy: SortStrategy):
        self.strategy = strategy
```

3. Strategy Pattern iv

```python

    def sort_data(self, data):
        return self.strategy.sort(data.copy())
```

3. Strategy Pattern v

```python
# Usage
data = [64, 34, 25, 12, 22, 11, 90]

# Use bubble sort
sorter = Sorter(BubbleSortStrategy())
print(sorter.sort_data(data))

# Switch to quick sort
sorter.set_strategy(QuickSortStrategy())
print(sorter.sort_data(data))
```

Real-world example: Discount Calculation

3. Strategy Pattern vi

```python
class DiscountStrategy(ABC):
    @abstractmethod
    def calculate_discount(self, price):
        pass

class NoDiscount(DiscountStrategy):
    def calculate_discount(self, price):
        return price

class PercentageDiscount(DiscountStrategy):
    def __init__(self, percent):
        self.percent = percent
```

3. Strategy Pattern vii

```python

    def calculate_discount(self, price):
        return price * (1 - self.percent / 100)

class FixedDiscount(DiscountStrategy):
    def __init__(self, amount):
        self.amount = amount

    def calculate_discount(self, price):
        return max(0, price - self.amount)

class ShoppingCart:
    def __init__(self):
```

3. Strategy Pattern viii

```python
        self.items = []
        self.discount_strategy = NoDiscount()

    def set_discount(self, strategy):
        self.discount_strategy = strategy

    def calculate_total(self):
        total = sum(self.items)
        return self.discount_strategy.calculate_discount(total)
```

What is Git? - Part 1

Git
Version control system that tracks changes to your code over time

Why use Git?

- **Track history:** See all changes made to your code
- **Collaborate:** Work with others without conflicts
- **Backup:** Your code is safe on remote servers
- **Experimentation:** Try new features without breaking main code
- **Undo mistakes:** Revert to previous versions

What is Git? - Part 2

Core concepts:

- **Repository (repo):** Project folder tracked by Git
- **Commit:** Snapshot of your code at a point in time
- **Branch:** Separate line of development
- **Remote:** Server hosting your repository (GitHub, GitLab)

Essential Tool
Git is the industry standard for version control - every developer uses it!

Basic Git Workflow i

Initial setup (one time):

```
# Set your identity
$ git config --global user.name "Your Name"
$ git config --global user.email "your.email@example.com"
```

Creating a repository:

Basic Git Workflow ii

```
# Initialize Git in your project
$ cd my_project
$ git init

# Or clone existing repository
$ git clone https://github.com/username/repo.git
```

Basic Git Workflow iii

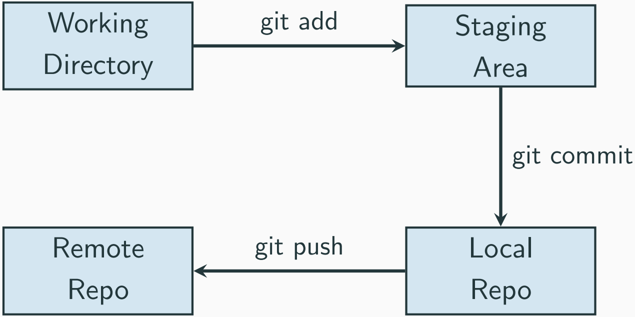
Basic Git workflow:

```
# 1. Check status
$ git status

# 2. Add files to staging area
$ git add file.py              # Add specific file
$ git add .                    # Add all files

# 3. Commit changes
$ git commit -m "Add login feature"

# 4. Push to remote repository
$ git push origin main
```

Basic Git Workflow iv

Basic Git Workflow v

Git workflow visualization:



Essential Git Commands i

Viewing history:

```
# View commit history
$ git log

# View compact history
$ git log --oneline

# View changes in a file
$ git diff file.py
```

Branching:

Essential Git Commands ii

```
# Create new branch
$ git branch feature-login

# Switch to branch
$ git checkout feature-login

# Create and switch in one command
$ git checkout -b feature-login

# List all branches
$ git branch

```

Essential Git Commands iii

```
# Merge branch into current branch
$ git merge feature-login

# Delete branch
$ git branch -d feature-login
```

Essential Git Commands iv

Working with remote:

```
# Add remote repository
$ git remote add origin https://github.com/user/repo.git

# View remotes
$ git remote -v

# Fetch changes from remote
$ git fetch origin

# Pull changes (fetch + merge)
$ git pull origin main

```

Essential Git Commands v

```
# Push changes
$ git push origin main
```

Undoing changes:

```
# Discard changes in working directory
$ git checkout -- file.py

# Unstage file
$ git reset HEAD file.py

# Undo last commit (keep changes)
$ git reset --soft HEAD~1
```

Essential Git Commands vi

```

# Undo last commit (discard changes)
$ git reset --hard HEAD~1
```

.gitignore File i

.gitignore
Tells Git which files to ignore (not track)

Common patterns for Python projects:

```
# .gitignore
# Virtual environment
venv/
env/
ENV/

# Python cache
__pycache__/
*.pyc
```

.gitignore File ii

```
*.pyo
*.pyd

# IDE files
.vscode/
.idea/
*.swp

# Test coverage
.coverage
htmlcov/

# Environment variables
```

.gitignore File iii

```
.env

# OS files
.DS_Store
Thumbs.db

# Build artifacts
dist/
build/
*.egg-info/
```

Git Best Practices i

1. **Write good commit messages:**

```
# Bad
$ git commit -m "fix"
$ git commit -m "changes"
$ git commit -m "update"

# Good
$ git commit -m "Fix login button not responding on mobile"
$ git commit -m "Add email validation to signup form"
$ git commit -m "Refactor database connection to use connection pool"
```

Git Best Practices ii

2. **Commit frequently:**
    - Commit logical units of work
    - Don't wait until end of day
    - Each commit should be functional
3. **Use branches for features:**

Git Best Practices iii

```
# Create branch for new feature
$ git checkout -b feature-user-profile

# Work on the feature...
$ git add .
$ git commit -m "Add user profile page"

# Merge when done
$ git checkout main
$ git merge feature-user-profile
```

Git Best Practices iv

4. Pull before you push:

```
# Always pull latest changes first
$ git pull origin main

# Then push your changes
$ git push origin main
```

5. Never commit sensitive data:

- Passwords, API keys, tokens
- Use environment variables instead

Git Best Practices v

- Add .env to .gitignore

```
# Bad
API_KEY = "abc123secret456"

# Good
import os
API_KEY = os.getenv(’API_KEY’)  # Read from .env file
```

Summary and Next Steps

Summary: Testing

What we learned about testing:

- **unittest:** Python's built-in testing framework
- **pytest:** Modern, simpler alternative
- **TDD:** Write tests before code (Red-Green-Refactor)
- **Debugging:** Print statements, pdb, IDE debuggers
- **Coverage:** Measure how much code is tested

Key Takeaway
Testing is not optional - it's essential for professional development!

Summary: Best Practices

What we learned about best practices:

- **Code Organization:** Modules, packages, separation of concerns
- **Documentation:** Docstrings, comments, type hints
- **Virtual Environments:** Isolate project dependencies
- **Project Planning:** Break down problems, design before coding
- **Design Patterns:** Singleton, Factory, Strategy
- **Git:** Version control for collaboration and safety

Key Takeaway
Professional code is well-organized, documented, and maintainable!

Skills Checklist

After this week, you should be able to:

- ✓ Write unit tests with unittest or pytest
- ✓ Apply test-driven development
- ✓ Debug code using various tools
- ✓ Organize code into maintainable modules
- ✓ Write comprehensive documentation
- ✓ Use virtual environments and manage dependencies
- ✓ Plan and structure projects effectively
- ✓ Apply common design patterns
- ✓ Use Git for version control

You're Ready!
You now have all the skills needed to build professional Python applications!

Preparing for the Capstone Project - Part 1

What's next: Capstone Project Workshop

The instructor will guide you through building a complete application that demonstrates:

- **OOP principles** (classes, inheritance, polymorphism)
- **File I/O** (reading/writing data)
- **Error handling** (try-except, validation)
- **Testing** (unit tests for critical functions)
- **Best practices** (organization, documentation)

Preparing for the Capstone Project - Part 2

Project ideas you might build:

- Library Management System
- Student Grade Tracker
- Personal Finance Manager
- Inventory Management System
- Task/Todo List Application
- Hotel Booking System
- E-commerce Product Catalog

Your Turn
After the demonstration, you'll build your own project applying all the concepts learned!

Recommended Resources

Testing:

- pytest documentation: `docs.pytest.org`
- Python Testing with pytest (book)

Best Practices:

- PEP 8 (Python Style Guide): `pep8.org`
- Clean Code by Robert Martin
- The Pragmatic Programmer

Design Patterns:

- Design Patterns in Python: `refactoring.guru/design-patterns/python`
- Head First Design Patterns

Final Thoughts

Congratulations!
You've completed the theoretical portion of Week 8!

Remember:

- **Testing** saves time in the long run
- **Documentation** helps your future self
- **Good organization** makes code maintainable
- **Design patterns** provide proven solutions
- **Version control** is essential for collaboration

Next: Capstone Project
Get ready to apply everything you've learned in a real-world project!

Thank You!

Thank you for your attention!

You're now equipped with professional Python development skills!

Access Course Materials:

Download Course Materials