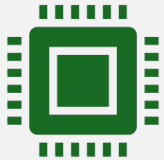
Advanced Python

Surendra Panpaliya

Week1 (Mon, Tue, Thurs)

Day 1: Python Recap + Environment & Tooling



Day 2: Functional Programming & Object-Oriented Design

Day 3: Advanced Python Concepts

Week2 (Mon, Tue, Wed, Thurs)

Day 4: Concurrency and Async Programming

Day 5: Web Services with FastAPI

Day 6: Azure Functions & Cloud Deployment

Day 7: Testing, Linting & Final Project

Day 1: Python Recap + Environment & Tooling

- Quick Python vs C# syntax mapping
- Variables, data types, control flow in Python
- Comprehensions (List, Dict, Set)
- Functions: *args, **kwargs, lambda
- Tooling: pip, venv, poetry, dependency locking

Day 1: Python Recap + Environment & Tooling

Hands-On Lab:

Set up Python project & create utility functions

Python vs C#: Static typing vs dynamic,

Main method vs scripts,

No semicolons/curly braces

Day 2: Functional Programming & Object-Oriented Design

Pythonic FP: map, filter, reduce, functools

Object-Oriented Programming:

C# vs Python class structure

SOLID principles and Mixins

Design Patterns: Singleton, Factory, Strategy

Day 2: Functional Programming & Object-Oriented Design

- Hands-On Lab:
- Create modular design for plugin architecture
- Authentication using OAuth / Azure AD
- *Python vs C#: Interfaces vs duck typing,*
- *Access modifiers*

Day 3: Advanced Python Concepts

- Decorators: Logging, validation, chaining
- Context Managers: with statement, __enter__, __exit__
- Generators and yield, pipelines
- Metaclasses: Framework-level magic

Day 3: Advanced Python Concepts

Hands-On Lab:

Logger with decorators and context managers

C# Attributes vs Python Decorators

Day 4: Concurrency and Async Programming

Concurrency: threading vs multiprocessing

Async IO: asyncio, event loop, aiohttp

Profiling Tools: cProfile,

line_profiler, memory_profiler

Day 4: Concurrency and Async Programming

Hands-On Lab:

Build async scraper with aiohttp

C# async/await vs Python async/await

Day 5: Web Services with FastAPI

FastAPI project structure vs ASP.NET Core WebAPI

Path/query parameters, request validation

Dependency Injection (DI)

Serving with Uvicorn + Gunicorn

Day 5: Web Services with FastAPI

Hands-On Lab:

Build CRUD APIs using FastAPI

ASP.NET Route attributes vs FastAPI decorators

Day 6: Azure Functions & Cloud Deployment

Azure Functions for Python:

Local setup and deployment

Comparing Dockerized App vs Azure Function

CI/CD in Azure Pipelines / GitHub Actions

Day 6: Azure Functions & Cloud Deployment

Hands-On Lab:

Convert API to Azure Function and deploy

C# Azure Functions vs Python Azure Functions

Day 7: Testing, Linting & Final Project

Code Quality & Linting:

mypy, ruff, black, flake8, pre-commit

Testing: pytest, mocking, assertions

Git Best Practices: PRs, reviews, branching strategy

Day 7: Testing, Linting & Final Project

- Final Capstone Project:
- Build Azure Function or FastAPI-based application
- Apply Clean Architecture & CQRS
- Present project
- *C# unit test framework vs pytest*

Software Installation Requirements

Download Link:Anaconda:

https://www.anaconda.com/products/distribution

Visual Studio Code :

https://code.visualstudio.com/Download

Software Installation Requirements

Azure CLI Version: 2.50.0+

https://docs.microsoft.com/cli/azure/install-azure-cli

Azure Functions Tools:

https://docs.microsoft.com/azure/azure-functions/functions-run-local#v2

Python Libraries (installed via pip or Conda)

- FastAPI (>= 0.95.0)
- aiohttp (>= 3.8.0)
- requests (>= 2.31.0)
- pandas (>= 2.0.0)
- pytest (>= 7.0.0)
- uvicorn (>= 0.22.0)

Python Code Quality Principle

1. Follow the PEP 8 Style Guide

Consistency in naming, spacing, indentation, etc.

Bad:

```
def add(x,y):return x+y
```

1. Follow the PEP 8 Style Guide

Consistency in naming, spacing, indentation, etc.

Good (PEP 8 Compliant):

```
def add(x, y):
    return x + y
```

2. Use Meaningful Variable and Function Names

Names should describe what the variable or function does.

Bad:

```
def f(a, b):
    return a * b
```

2. Use Meaningful Variable and Function Names

Names should describe what the variable or function does.

Good:

```
def calculate_area(length, width):
    return length * width
```

3. Single Responsibility Principle (SRP)

Each function or class should do one thing only.

Bad:

```
def process_order(order):
    validate_order(order)
    save_to_database(order)
    send_email(order)
```

3. Single Responsibility Principle (SRP)

Good:

```
def validate_order(order):
    # validation logic
    pass

def save_to_database(order):
    # database logic
    pass

def send_email(order):
    # email logic
    pass
```

4. Avoid Code Duplication

Use functions to avoid repeating code.

Bad:

```
print("Welcome, John")
print("Welcome, Mary")
print("Welcome, Alice")
```

4. Avoid Code Duplication

Good:

```
def greet(name):
    print(f"Welcome, {name}")


greet("John")
greet("Mary")
greet("Alice")
```

5. Use List Comprehensions (Where Appropriate)

More Pythonic and readable for simple operations.

Bad:

```
squares = []
for i in range(10):
    squares.append(i * i)
```

5. Use List Comprehensions (Where Appropriate)

More Pythonic and readable for simple operations.

Good:

```
squares = [i * i for i in range(10)]
```

6. Error Handling with try-except

Avoid crashing the program due to runtime errors.

Bad:

```
value = int(input("Enter a number: "))
result = 100 / value
```

6. Error Handling with try-except

Good:

```
try:
    value = int(input("Enter a number: "))
    result = 100 / value
except ValueError:
    print("Invalid input! Enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

7. Write Docstrings and Comments

📌 *Explain the purpose and usage of functions and modules.*

```
def calculate_bmi(weight, height):
    """
    Calculate BMI using weight in kg and height in meters.
    """
    return weight / (height ** 2)
```

8. Use Type Hints

Improve readability and help static analysis tools.

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

10. Write Unit Tests

Use unittest or pytest to ensure code works as expected.

```
import unittest

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

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

Python Linting

Surendra Panpaliya

GKTCS Innovations

https://www.gktcs.com

What is Linting in Python?

Process of analyzing your code

to **identify programming errors,**

style violations, bugs, and

bad practices—*before* you run it.

Why is Linting Important?

Enforces **PEP 8** (Python's style guide)

Detects **syntax errors and bugs early**

Improves **readability and maintainability**

Helps during **code reviews and collaboration**

Common Python Linting Tools

Tool	Purpose
pylint	Full-featured linting & scoring
flake8	Lightweight PEP8 + error checker
black	Auto code formatter
mypy	Checks type annotations
ruff	Super fast linter + formatter

Linting Summary

Benefit	Result
Catch errors early	Save time & debugging
Consistent code style	Easier to read & maintain
Better collaboration	Cleaner code reviews

Why Code Quality Tools?

These tools help:

Catch bugs early ✅

Ensure consistent formatting 🧹

Enforce good coding practices 🔍

Improve maintainability 🔧

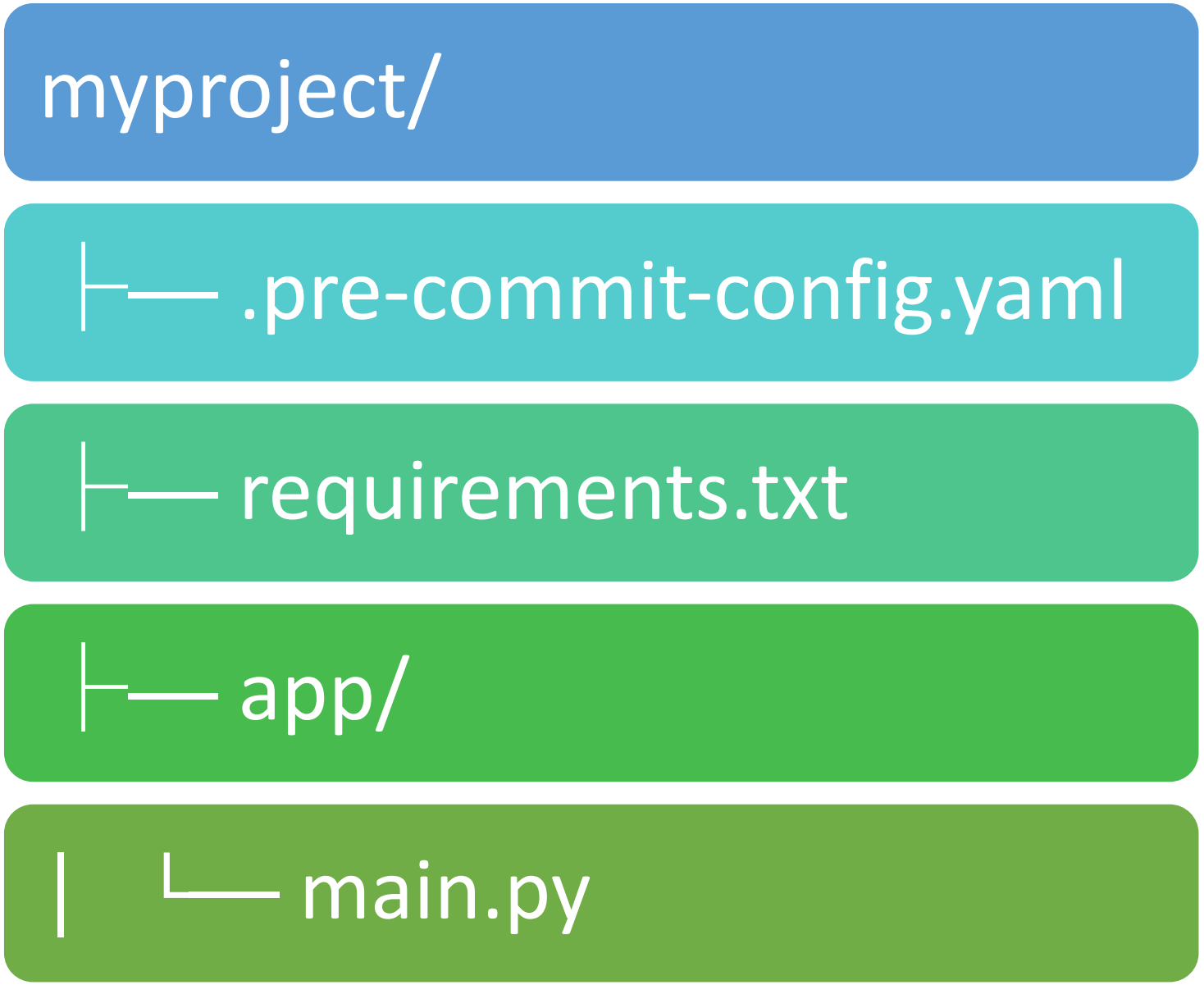
Summary Comparison

Tool	Purpose	Speed	Fix Support	Git Hook
mypy	Type checking	🟡	❌	✅
black	Code formatting	✅	✅	✅
flake8	Style + lint errors	🟡	❌	✅
ruff	Fast linting (all-in-1)	✅✅	✅	✅
pre-commit	Git integration	✅	✅	✅

Summary

Tool	Purpose	Type
black	Format code (PEP 8)	Formatter
flake8	Lint code for style + errors	Linter
mypy	Type check using hints	Type Checker
ruff	Fast linter + fixer	Linter/Fixer
pre-commit	Auto-run tools before commit	Git Hook Manager

Best Practice Project Setup

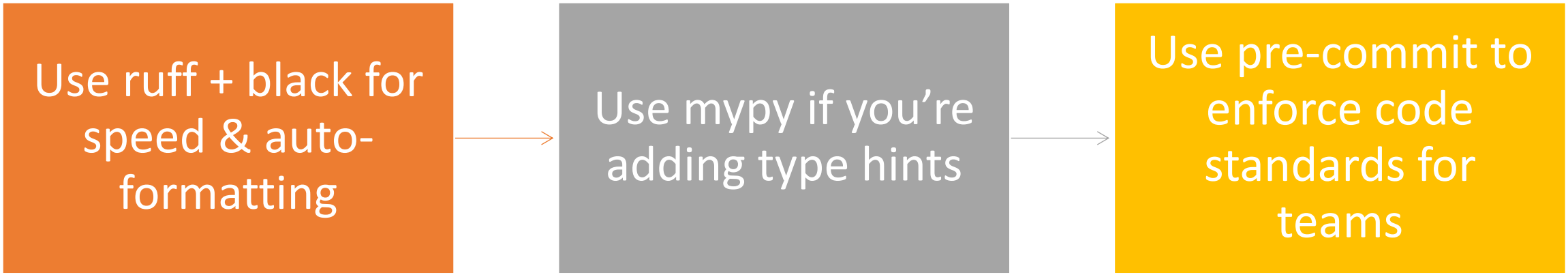


Best Practice Project Setup

requirements.txt

- black
- flake8
- mypy
- ruff
- pre-commit

Best Practice Project Setup



What
is pytest?

Powerful,

easy-to-use

popular testing framework

What is pytest?

used to write **unit tests**,

integration tests, and

even **functional tests**

with less code and more readability.

Why Use pytest?

- ✅ Simple syntax
- ✅ Fast execution
- ✅ Supports fixtures, mocks, parametrization
- ✅ Rich plugin ecosystem (pytest-cov, pytest-mock, etc.)

Why Use pytest?

Feature	Benefit
No boilerplate	No need for class-based tests or main() runner
Simple syntax	Use plain assert statements
Rich plugins	Has plugins for coverage, mocking, CI, etc.
Auto-discovery	Automatically finds test files/functions
Easy mocking support	Works well with unittest.mock or pytest-mock

Assertions

Syntax	Meaning
assert a == b	a equals b
assert x in y	x exists in y
assert func() raises Error	check error handling
assertAlmostEqual(a, b, tol)	for float comparisons

Tips

Always name test files: test_*.py

Use pytest-mock or unittest.mock for mocking

Use pytest.ini for consistent config

Combine with CI (GitHub Actions, Azure Pipelines)

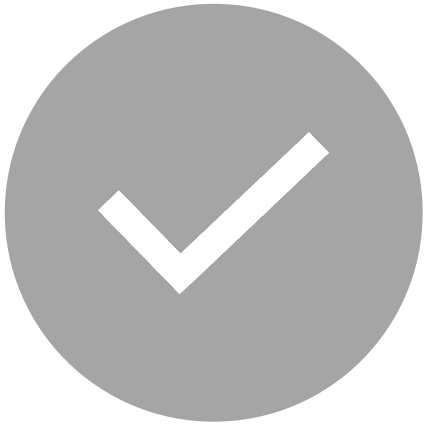
Summary

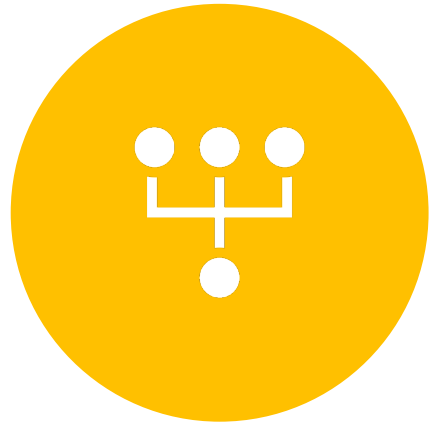
Topic	Key Point
pytest	Python testing framework – simple, readable, powerful
assert	Built-in syntax used for validation of test output
mocking	Replaces real objects (like APIs, DBs) with controlled fake behavior
parametrize	Lets you run the same test with multiple inputs easily

Summary Table

Concept	Tool	Description
Test runner	pytest	Executes tests
Assertions	assert	Validates output
Fixtures	@pytest.fixture	Reusable setup for tests
Parametrize	@pytest.mark.parametrize	Loop over test cases
Mocking	unittest.mock	Fake dependencies like API/DB
Coverage	pytest-cov	Test coverage %

Git Best Practices





✅ PULL REQUESTS (PRS)

🔍 CODE REVIEWS

🌿 BRANCHING STRATEGY

Why Git Best Practices Matter

Poor Git hygiene leads to:

- Merge conflicts 😣
- Lost history 🔍
- Broken builds 💥
- Team frustration 😠

Why Git Best Practices Matter

Good practices ensure:

- Stable codebase ✅
- Faster code reviews 🕵️
- Clean collaboration 🫧
- CI/CD compatibility 🚀

1. Branching Strategy

A **good branching strategy**

helps teams collaborate efficiently

without breaking the main codebase.

Git Flow Lite (Simplified Git Flow)

Branch	Purpose
main / master	Always production-ready 🟢
dev	Integration of all features 🔧
feature/*	Individual features 🧩
bugfix/*	Urgent bug fixes 🐞
hotfix/*	Direct patch to production 🔥
release/*	Pre-production final testing 🚦

Create a feature branch

git checkout -b feature/user-login

Common Strategies

Strategy	Description
Git Flow	Main, Develop, Feature, Release, Hotfix branches
GitHub Flow	Main + Feature branches with PRs
Trunk-Based	Commit to main/master with short-lived branches and CI/CD

Feature Branch Strategy (GitHub Flow)

Branch Naming Convention:

Branch Type	Naming Example
main	stable production branch
feature/	feature/login-page
bugfix/	bugfix/crash-on-submit
hotfix/	hotfix/v1.2.1-crash-fix
release/	release/v2.0

Example Workflow

```
git checkout -b feature/user-registration
# work on code...
git add .
git commit -m "Add user registration with validation"
git push origin feature/user-registration
```

Then you open a **Pull Request** (PR).

2. Pull Requests (PRs)

Purpose	Purpose of a PR:
Get	Get feedback before merging code
Ensure	Ensure quality through review, testing, and CI
Keep	Keep main branch clean and deployable

Why PRs?

- Code quality checks
- CI/CD validation
- Peer review & knowledge sharing
- Approval trail

What a good PR includes:

Element	Example
Title	feat: Add user login API
Description	Bullet points on what changed, why, any notes
Linked Issues	Fixes #123 (auto-close on merge)
Test Coverage	Include screenshots or pytest results
Small Size	Prefer < 500 lines of code

What a good PR includes

⚠️ **Avoid**

One huge PR with everything

Vague messages: fixed stuff, WIP, etc.

PR Best Practices

Practice	Why It Matters
✅ Small, focused PRs	Easier to review, faster feedback
✅ Clear titles & descriptions	Helps reviewers understand purpose
✅ Link to issues/tickets	Keeps context connected
✅ Add screenshots/tests	Speeds up UI/logic validation
✅ Self-review your PR	Avoid obvious mistakes

PR Description Template

Feature Summary

Implemented user registration page with email/password validation.

Changes Made

- New `RegisterUserForm`
- Email validation logic
- Basic styling and form submit handler

PR Description Template

- ### Test Plan
- - [x] Unit test for input validation
- - [x] Manual UI test on desktop & mobile
-
- ### Linked Issue
- Closes #123

3. Code Reviews

A CODE REVIEW ENSURES THAT

YOUR TEAM PRODUCES

CLEAN,

SECURE, AND

MAINTAINABLE CODE.

Review Checklist for Reviewers

Question	Why
Is the code readable and well-formatted?	Clarity
Are there any logic bugs or edge cases missed?	Functionality
Are there unit or integration tests?	Reliability
Are naming conventions followed?	Consistency
Are there any unnecessary code changes?	Cleanliness

Code Review Checklist

Area	Questions to Ask
Readability	Is the code easy to understand? Any comments needed?
Functionality	Does the code do what it says? Any edge cases missed?
Security	Are secrets exposed? Input validations done?
Testing	Are unit/integration tests added and passing?
Consistency	Does it follow formatting rules (black, flake8, etc.)?
Performance	Any unnecessary loops, queries, I/O?

Code Review Checklist

✅ Always respond to comments in PR review

✅ Approve only if you've read the code and tested locally (if applicable)

Reviewer Comments Example

💬 Suggestion:

- Variable `temp` can be renamed to `userInput` for clarity.

💬 Question:

- Is this API call retrying on failure?

🟢 Looks good overall! Just minor cleanup required before merge.

4. Merging the PR

Merge Method	When to Use
Squash and Merge	Clean history; for single-feature PRs
Rebase and Merge	Clean linear history; advanced teams only
Merge Commit	Keeps full history; useful for big projects

5. Bonus Best Practices

- ✅ Always pull latest changes:
- git pull origin main

5. Bonus Best Practices

- ✅ Rebase your branch (optional but cleaner history):
- git fetch
- git rebase origin/main

5. Bonus Best Practices

- ✅ Delete your feature branch after merge:
- git push origin --delete feature/user-registration

Summary Table

Practice	Goal
Feature branches	Isolate and parallelize work
Small PRs	Easy to review and test
Code review checklist	Maintain quality & consistency
Merge strategy	Keep Git history clean and useful

C# vs Python Unit Testing

Feature	**C# (xUnit/NUnit/MSTest)**	**Python (pytest)**
Language	Statically typed (C#)	Dynamically typed (Python)
Test Frameworks	MSTest, NUnit, xUnit	Pytest, Unittest
Assertions	Assert.Equal(), etc.	assert keyword
Fixtures (setup/teardown)	[SetUp], [TearDown], [Fact]	@pytest.fixture, setup/teardown

C# vs Python Unit Testing

Feature	C# (xUnit/NUnit/MSTest)	Python (pytest)
Parameterized Tests	[Theory][InlineData] (xUnit)	@pytest.mark.parametrize
Test Discovery	Via attributes	By naming convention (test_*.py)
Coverage Tools	Coverlet, dotCover	pytest-cov
Mocking Tools	Moq, NSubstitute	unittest.mock, pytest-mock

Summary Table

Feature	C# (xUnit / NUnit / MSTest)	Python (pytest)
Framework Setup	Attributes-based [Fact]	Naming conventions + decorators
Parametrized Tests	[Theory][InlineData]	@pytest.mark.parametrize
Fixtures	IClassFixture, [SetUp]	@pytest.fixture

Summary Table

Feature	C# (xUnit / NUnit / MSTest)	Python (pytest)
Assertions	Assert.Equal, etc.	Native assert keyword
Mocking	Moq, FakeItEasy	unittest.mock, pytest-mock
Coverage Tool	Coverlet, dotCover	pytest-cov

When to Use What?

Scenario	Use
Enterprise/ASP.NET backend	C# with xUnit/NUnit
ML, Data, FastAPI, scripting	Python with pytest
Rapid prototyping or scripting	Python
Integration with Azure Pipelines	Both support CI/CD and coverage

Capstone Project

Bank Application using **Azure Functions** or **FastAPI**,
applying **Clean Architecture +**
CQRS (Command Query Responsibility Segregation)

Capstone Title

“Modern Bank API with Clean Architecture & CQRS using FastAPI/Azure Function”

Stack Options:

- **API Framework**: FastAPI (preferred for web) or Azure Functions (serverless)
- **Architecture**: Clean Architecture + CQRS
- **Tools**: Python, Pydantic, SQLAlchemy/Databases, PostgreSQL

Stack Options:

- **Testing**: Pytest, Mock, Coverage
- **CI/CD**: GitHub Actions / Azure Pipelines
- **Linting & Formatting**: Black, Ruff, Mypy, Pre-commit

What is Clean Architecture?

- Clean Architecture separates your code into **well-defined layers**, improving:
- Modularity
- Testability
- Maintainability

Layers

Layer	Responsibility
Domain	Core business logic (Entities, Interfaces)
Use Cases	Application logic (Commands, Queries)
Interface/Adapters	REST API, Database, External services
Infrastructure	Database, Logging, External APIs

What is CQRS?

CQRS = Command Query Responsibility Segregation

Split the application into:

🔄 **Commands**: Write operations (Create, Update, Delete)

📥 **Queries**: Read operations (Get Account Info, Transaction History)

CQRS Benefits

BETTER SEPARATION OF CONCERNS

OPTIMIZED PERFORMANCE

EASIER TO SCALE AND SECURE

Project Structure

```
bank_app/
 ├── app/
|    ├── domain/
|    |    ├── entities.py         # Core business entities
|    |    └── interfaces.py       # Interface definitions
|    ├── use_cases/
|    |    ├── commands/           # Write logic
|    |    └── queries/            # Read logic
```

Project Structure

```
bank_app/
 ├── app/
|    ├── adapters/
|    |       ├── api/              # FastAPI or Azure Functions
|    |       └── repository/       # Database logic
|    ├– – infra/
|    |         └── db.py           # PostgreSQL connection
|    └── main.py
```

Project Structure

```
bank_app/
 ├── app/
 |   |   └── main.py
 ├── tests/
 ├── requirements.txt
 ├── README.md
 └── .pre-commit-config.yaml
```

Final Deliverables

Fully structured project (Clean Architecture + CQRS)

FastAPI or Azure Function API

Working PostgreSQL connection

Tests with Pytest

Final Deliverables

- Linting + Pre-commit hooks
- README with instructions
- Deployed version (optional: Render, Azure App Service, etc.)

Happy Learning !!
Thanks for Your Patience ☺

Surendra Panpaliya

GKTCS Innovations

https://www.gktcs.com