StructuredPy: Master Python through Objects and Functions

Ameyanagi

2025-01-10

Table of contents

Introduction **5**
- Who This Book Is For . . . 5
- What You'll Build . . . 5
- How to Use This Book . . . 6
  - Project-Based Learning . . . 6
  - Code Examples . . . 6
  - Exercises and Practice . . . 6
- Prerequisites . . . 6
- Getting Started . . . 7
- What Makes This Book Different . . . 7
- Navigation . . . 7
- Acknowledgments . . . 7
- Contributing . . . 7

Preface **8**
- Why This Book Exists . . . 8
- Our Approach . . . 8
- Book Structure . . . 8
- Key Principles . . . 9
- A Note on Style . . . 9
- Getting Help . . . 9

I Initial setting and basic workflow **11**

1 Initial Setting and Tutorial **12**
- 1.1 Prerequisites . . . 12
- 1.2 Setting up Git . . . 12
  - 1.2.1 Initial Setup . . . 12
  - 1.2.2 Configuring Git . . . 13
- 1.3 GitHub Setup (Optional) . . . 13
  - 1.3.1 Creating a GitHub Account . . . 13
  - 1.3.2 Creating a Personal Access Token . . . 14
- 1.4 uv: Modern Python Package Management . . . 14
  - 1.4.1 Installation . . . 14

1.4.2 Verifying Installation . . . 15
1.5 Summary . . . 15

2 Project Initialization Guide **16**
2.1 Understanding Project Structure . . . 16
2.2 Using Cookiecutter . . . 16
2.2.1 What is Cookiecutter? . . . 16
2.2.2 Project Template Selection . . . 17
2.2.3 Project Initialization Workflow . . . 17
2.2.4 Creating Your Project . . . 19
2.2.5 Project directory . . . 20
2.2.6 Initializing git repository . . . 21
2.2.7 Installing the python dependencies . . . 21
2.3 Summary . . . 23

3 General workflow of coding **24**
3.1 Open code in Editor . . . 24
3.2 Running the code . . . 24
3.3 Creating a python file and running it . . . 25
3.4 Test the code . . . 25
3.4.1 Running the test . . . 26
3.5 Documentation . . . 28
3.5.1 Writing Good Documentation . . . 28
3.5.2 Generating API Documentation . . . 29
3.6 Commit the code to Git . . . 30
3.7 Summary . . . 31

II Appendices **32**

4 Python Basics **33**
4.1 Variables and Data Types . . . 33
4.1.1 Basic Data Types . . . 33
4.1.2 Lists and Dictionaries . . . 33
4.2 Control Flow . . . 34
4.2.1 If Statements . . . 34
4.2.2 Loops . . . 35
4.3 Functions . . . 35
4.3.1 Basic Functions . . . 35
4.3.2 Return Values . . . 36
4.4 Exception Handling . . . 36
4.5 String Formatting . . . 37
4.6 File Operations . . . 37

4.7 Modules and Imports 37
4.8 Essential Built-in Functions 38
4.9 List Operations 38
4.10 Dictionary Operations 39
4.11 Common Modules Used in This Tutorial 39
4.12 Code Style Guidelines 40
4.13 What's Next 40

Introduction

Welcome to StructuredPy, a practical guide to building well-structured Python applications. This book takes a hands-on approach to learning Python through real-world projects, focusing on both functional and object-oriented programming approaches.

Who This Book Is For

This book is designed for:

- Python beginners who understand basic syntax and want to write better code
- Programmers transitioning to Python from other languages
- Anyone interested in building practical Python applications

What You'll Build

Throughout this book, you'll work on three main projects:

1. **Time Series Data Analysis**
    - Analyze sensor data using both functions and classes
    - Learn data processing techniques
    - Visualize time series data
2. **Device Communication**
    - Build a mock serial communication system
    - Handle device commands and responses
    - Manage device state and errors
3. **API Development**
    - Create a FastAPI server
    - Connect devices through web APIs
    - Build a complete application

Each project is presented twice: first using functions, then using classes. This approach helps you understand when to use each style and how to combine them effectively.

How to Use This Book

Project-Based Learning

Each chapter focuses on building something practical. We start with a problem, break it down into manageable pieces, and build a solution step by step.

Code Examples

All code examples are available in the `code` directory. For each project, you'll find:

```
code/
  timeseries/
  device/
  api/
```

Exercises and Practice

Each chapter includes: - Hands-on exercises - Code challenges - Suggestions for extending the projects

Prerequisites

To get the most out of this book, you should:

- Know basic Python syntax (variables, loops, conditionals)
- Have Python installed on your computer
- Be comfortable using a text editor or IDE
- Have a basic understanding of the command line

If you need to review Python basics, check the appendix for a quick reference.

Getting Started

1. Clone the book's repository:

```
git clone https://github.com/Ameyanagi/StructuredPy
```

2. Install the required packages:

```
pip install -r requirements.txt
```

3. Start with Chapter 1 and follow along with the code examples.

What Makes This Book Different

Instead of teaching Python concepts in isolation, this book:

- Shows both functional and object-oriented approaches
- Uses real-world examples
- Focuses on writing maintainable code
- Demonstrates practical project organization

Navigation

- Use the sidebar to navigate between chapters
- Code examples can be copied directly from the book
- Projects build on each other progressively

Let's begin by analyzing time series data in Chapter 1!

Acknowledgments

Thanks to all contributors and reviewers who helped improve this book.

Contributing

Found a mistake or want to improve the book? Visit the GitHub repository to:

- Report issues
- Suggest improvements
- Contribute content

Preface

Why This Book Exists

Python's flexibility is both a blessing and a curse. While it allows you to solve problems in many ways, it can be challenging to know which approach is best. This book aims to bridge the gap between writing code that just works and writing code that's clear, maintainable, and scalable.

Our Approach

We believe in learning by doing. Instead of abstract concepts, we focus on practical projects that you might encounter in real work. Each project is implemented twice:

1. First using functions - Simple, straightforward, and easy to understand
2. Then using classes - Organized, maintainable, and scalable

This dual approach helps you understand: - When to use each style - The trade-offs involved - How to combine approaches effectively

Book Structure

Each project follows this pattern:

1. **Problem Introduction**
    - What we're building
    - Why it matters
    - Expected outcomes
2. **Functional Solution**
    - Step-by-step implementation
    - Key concepts explained
    - Code organization

3. **Object-Oriented Solution**
    - Converting to classes
    - Benefits and trade-offs
    - Design decisions
4. **Practical Extensions**
    - Real-world considerations
    - Common challenges
    - Further improvements

Key Principles

Throughout the book, we emphasize:

- Writing clear, focused code
- Building maintainable solutions
- Making conscious design decisions
- Testing and error handling
- Real-world applications

A Note on Style

There's rarely one "right" way to solve a programming problem. We present approaches that are:

- Clear to understand
- Easy to maintain
- Practical to implement
- Well-tested in real applications

Getting Help

If you get stuck:

1. Check the code in the book's repository
2. Review the chapter's key concepts
3. Try breaking the problem into smaller pieces
4. Check the online resources in the appendix

Remember: Every experienced programmer was once a beginner. Take your time, practice with the examples, and don't hesitate to experiment with the code.

Let's start building better Python applications!

Part I

Initial setting and basic workflow

1 Initial Setting and Tutorial

This tutorial will guide you through setting up a modern Python development environment with best practices for:

- Version control with Git and GitHub
- Dependency management using uv (a fast Python package installer and resolver)
- Project structure using cookiecutter templates
- Testing with pytest and Test-Driven Development (TDD)
- Documentation with MkDocs
- Code quality tools and pre-commit hooks

By following this guide, you'll create a well-structured Python project that follows some of the best practices that people use for python. The setup includes automatic testing, documentation generation, and code quality checks - essential tools for professional Python development.

1.1 Prerequisites

Before starting this tutorial, you should have:

- Basic Python knowledge
- A GitHub account
- A text editor or IDE of your choice
- Terminal/command line familiarity

But if you don't have any of these, don't worry! You can refer to the Appendix for more information.

1.2 Setting up Git

1.2.1 Initial Setup

1.2.1.1 Installing Git

Before you can use Git, you need to install it on your system:

- **Windows**: Download and install from git-scm.com
- **macOS**: Install via Homebrew: `brew install git`
- **Linux**: Install via package manager:

```
# Ubuntu/Debian
sudo apt-get install git

# Fedora
sudo dnf install git
```

1.2.2 Configuring Git

After installation, configure your identity: This will be required when you are pushing the code to the GitHub.

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

Additional recommended configurations: Whenever you are pushing to the GitHub, you will be asked for the username and password. To avoid this, you can store the credentials securely. If you are not using GitHub, then you can skip this step.

```
# Store credentials securely
git config --global credential.helper store
```

1.3 GitHub Setup (Optional)

GitHub is a popular platform for hosting Git repositories and collaborating on code. You can use it to store your code, track changes, and work with others on projects. You can also use GitHub Actions for continuous integration and deployment (CI/CD) workflows.

You will need to create an account on GitHub and set up a personal access token to push code to GitHub.

1.3.1 Creating a GitHub Account

1. Go to github.com and sign up for an account
2. Verify your email address
3. Set up two-factor authentication (recommended)

1.3.2 Creating a Personal Access Token

Modern GitHub security requires using tokens instead of passwords:

1. Go to GitHub Settings → Developer settings → Personal access tokens → Tokens (classic)
2. Click "Generate new token (classic)"
3. Select scopes:
    - `repo` (Full control of private repositories)
    - `workflow` (if using GitHub Actions)
    - `read:org` (if working with organization repositories)
4. Set an expiration date
5. Generate and copy the token immediately (it won't be shown again)

Store your token securely - you'll need it when pushing to GitHub.

1.4 uv: Modern Python Package Management

uv is a new-generation Python package installer and resolver written in Rust. It offers several advantages over traditional tools like pip:

- Extremely fast package installation and dependency resolution
- Built-in virtual environment management
- Compatible with pip's interface and package formats
- Reliable dependency resolution
- Optimized for modern Python development

Here, we will only be focusing on the installation and minimal validation of the uv that is required for the project.

If you want to know more about uv, you can visit the official website here

1.4.1 Installation

1.4.1.1 Installation Methods

1. **Unix/macOS**

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

2. **Windows (PowerShell)**

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

> **Note**
>
> Note: After installation on Windows, you may need to add uv to your PATH. The installation location is `%USERPROFILE%\.local\bin`. You can add this to your PATH by:
>
> 1. Press Windows + X and select "System"
> 2. Click "Advanced system settings"
> 3. Click "Environment Variables"
> 4. Under "User variables", find and select "Path"
> 5. Click "Edit"
> 6. Click "New"
> 7. Add `%USERPROFILE%\.local\bin`
> 8. Click "OK" on all windows

1.4.2 Verifying Installation

You can verify the installation by opening a new terminal or PowerShell window and running:

```
uv --version
```

1.5 Summary

In this tutorial, you learned how to set up a modern Python development environment with Git, GitHub, and uv. These tools will help you manage your code, collaborate with others, and install Python packages efficiently.

In the next chapter, we'll create a new Python project using a cookiecutter template and set up a basic project structure.

2 Project Initialization Guide

We will be using boilerplate that includes modern dependency management with uv, comprehensive testing setup with pytest, documentation with MkDocs, code quality tools, CI/CD with GitHub Actions, Docker support, and dev container configuration for VSCode. This will give you a solid foundation for professional Python development without having to set up each component manually.

2.1 Understanding Project Structure

Before we dive into creating a new project, it's important to understand what makes up a well-structured Python project:

- Source code directory (your main package)
- Tests directory
- Documentation
- Build configuration files
- Development tools configuration
- CI/CD configuration
- License and README files

2.2 Using Cookiecutter

2.2.1 What is Cookiecutter?

Cookiecutter is a project template tool that we'll use through `uvx` to create our project structure. It uses a templating engine to replace project variables with your custom values, ensuring consistent project structure across all your repositories.

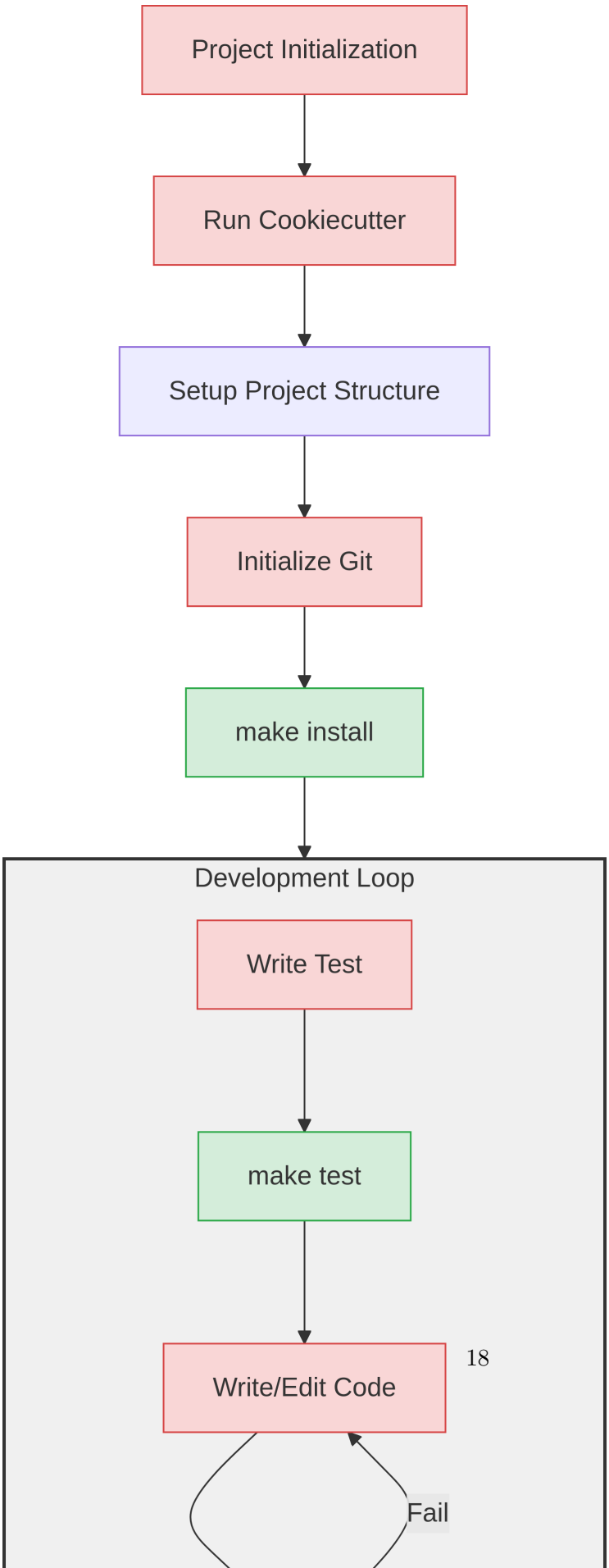
2.2.2 Project Template Selection

We'll use the `cookiecutter-uv` template created by Florian Maas, which is specifically designed for modern Python projects using `uv`. This well-maintained template includes:

- Modern dependency management with uv
- Comprehensive testing setup with pytest
- Documentation with MkDocs
- Code quality tools (ruff, mypy)
- CI/CD with GitHub Actions
- Docker support
- Dev container configuration for VSCode

2.2.3 Project Initialization Workflow

Here's a high-level overview of the project initialization workflow: The green boxes represent automated steps, while the red boxes represent manual steps.


Project Initialization
Run Cookiecutter
Setup Project Structure
Initialize Git
make install
Development Loop
Write Test
make test
Write/Edit Code
Fail

2.2.4 Creating Your Project

Run the template generator:

```
uvx cookiecutter https://github.com/fpgmaas/cookiecutter-uv.git
```

Once you run the command, you will be prompted to enter some information about your project. This information will be used to generate the project structure and files.

```
Installed 22 packages in 11ms
  [1/14] author (Florian Maas): Ameyanagi
  [2/14] email (fpgmaas@gmail.com): contact@ameyanagi.com
  [3/14] author_github_handle (fpgmaas): Ameyanagi
  [4/14] project_name (example-project): StructuredPy-code
  [5/14] project_slug (structuredpy_code):
  [6/14] project_description (This is a template repository for Python projects that use uv
  [7/14] Select include_github_actions
    1 - y
    2 - n
    Choose from [1/2] (1):
  [8/14] Select publish_to_pypi
    1 - y
    2 - n
    Choose from [1/2] (1):
  [9/14] Select deptry
    1 - y
    2 - n
    Choose from [1/2] (1):
  [10/14] Select mkdocs
    1 - y
    2 - n
    Choose from [1/2] (1):
  [11/14] Select codecov
    1 - y
    2 - n
    Choose from [1/2] (1):
  [12/14] Select dockerfile
    1 - y
    2 - n
    Choose from [1/2] (1):
  [13/14] Select devcontainer
    1 - y
```

```
    2 - n
    Choose from [1/2] (1):
  [14/14] Select open_source_license
    1 - MIT license
    2 - BSD license
    3 - ISC license
    4 - Apache Software License 2.0
    5 - GNU General Public License v3
    6 - Not open source
    Choose from [1/2/3/4/5/6] (1):
```

2.2.5 Project directory

You can check the initial directory tree by running the following command:

```
tree -a -L 4 StructuredPy-code # Please change the directory name to your project name or a c
```

The output will be like this:

```
StructuredPy-code
   codecov.yaml
   CONTRIBUTING.md
   .devcontainer
       devcontainer.json
       postCreateCommand.sh
   Dockerfile
   docs
       index.md
       modules.md
   .editorconfig
   .github
       actions
           setup-python-env
               action.yml
       workflows
           main.yml
           on-release-main.yml
           validate-codecov-config.yml
   .gitignore
   LICENSE
   Makefile
```

```
   mkdocs.yml
   .pre-commit-config.yaml
   pyproject.toml
   README.md
   structuredpy_code
       foo.py
       __init__.py
   tests
       test_foo.py
   tox.ini

9 directories, 23 files
```

2.2.6 Initializing git repository

```
git init
git add .
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/Ameyanagi/StructuredPy-code.git
git push -u origin main
```

2.2.7 Installing the python dependencies

```
make install
```

This will run the following command:

```
uv sync
uv run pre-commit install
```

The output will be like this:

```
 Creating virtual environment using uv
Using CPython 3.13.1 interpreter at: /usr/bin/python3
Creating virtual environment at: .venv
Resolved 63 packages in 620ms
   Built structuredpy-code @ file:///home/ameyanagi/StructuredPy-code
```

```
Prepared 59 packages in 1.12s
Installed 59 packages in 169ms
 + babel==2.16.0
 + cachetools==5.5.0
 + certifi==2024.12.14
 + cfgv==3.4.0
 + chardet==5.2.0
 + charset-normalizer==3.4.1
 + click==8.1.8
 + colorama==0.4.6
 + coverage==7.6.10
 + deptry==0.21.2
 + distlib==0.3.9
 + filelock==3.16.1
 + ghp-import==2.1.0
 + griffe==1.5.4
 + identify==2.6.5
 + idna==3.10
 + iniconfig==2.0.0
 + jinja2==3.1.5
 + markdown==3.7
 + markupsafe==3.0.2
 + mergedeep==1.3.4
 + mkdocs==1.6.1
 + mkdocs-autorefs==1.2.0
 + mkdocs-get-deps==0.2.0
 + mkdocs-material==9.5.49
 + mkdocs-material-extensions==1.3.1
 + mkdocstrings==0.27.0
 + mkdocstrings-python==1.13.0
 + mypy==1.14.1
 + mypy-extensions==1.0.0
 + nodeenv==1.9.1
 + packaging==24.2
 + paginate==0.5.7
 + pathspec==0.12.1
 + platformdirs==4.3.6
 + pluggy==1.5.0
 + pre-commit==4.0.1
 + pygments==2.19.1
 + pymdown-extensions==10.13
 + pyproject-api==1.8.0
```

```
 + pytest==8.3.4
 + pytest-cov==6.0.0
 + python-dateutil==2.9.0.post0
 + pyyaml==6.0.2
 + pyyaml-env-tag==0.1
 + regex==2024.11.6
 + requests==2.32.3
 + requirements-parser==0.11.0
 + ruff==0.8.6
 + six==1.17.0
 + structuredpy-code==0.0.1 (from file:///home/ameyanagi/StructuredPy-code)
 + tox==4.23.2
 + tox-uv==1.17.0
 + types-setuptools==75.6.0.20241223
 + typing-extensions==4.12.2
 + urllib3==2.3.0
 + uv==0.5.15
 + virtualenv==20.28.1
 + watchdog==6.0.0
pre-commit installed at .git/hooks/pre-commit
```

2.3 Summary

In this chapter, you learned how to initialize a new Python project using the `cookiecutter-uv` template. This template provides a modern Python project structure with uv, pytest, MkDocs, and other tools to help you get started quickly.

In the next chapter, we'll explore how to write and run tests for your Python project using the project structure we just created.

3 General workflow of coding

This guide will walk you through creating and maintaining a Python project, from writing code to testing, documentation, and version control. You'll learn how to write testable code following Test Driven Development (TDD) principles, create clear documentation, and manage your codebase with Git.

3.1 Open code in Editor

Open the root directory of the project in your preferred code editor (VSCode, Neovim, etc.).

3.2 Running the code

The source code is included in the `structuredpy_code` directory. By default the `foo.py` file is included in the directory. You can run the code by running the following command:

```
uv run structuredpy_code/foo.py
```

At this moment, there will be no output, because the `foo.py` file is empty.

```
def foo(bar: str) -> str:
    """Summary line.

    Extended description of function.

    Args:
        bar: Description of input argument.

    Returns:
        Description of return value
    """

    return bar
```

```

if __name__ == "__main__":  # pragma: no cover
    pass
```

We will be make a new file and work on that file, but configuration would be the same.

3.3 Creating a python file and running it

Let's create a new file called `hello.py` in the `structuredpy_code` directory. In the `hello.py` file, we will write a simple function that prints "hello world" to the console.

```
def main():
    print("hello world")


if __name__ == "__main__":
    main()
```

You can run the code by running the following command:

```
uv run structuredpy_code/hello.py
```

The output will be like this:

```
hello world
```

3.4 Test the code

Let's look at the test file in the `tests` directory. The test file is called `test_foo.py` and it tests the `foo` function in the `foo.py` file.

```
from structuredpy_code.foo import foo


def test_foo():
    assert foo("foo") == "foo"
```

3.4.1 Running the test

You can run the test by running the following command:

```
make test
```

This command will run the following command:

```
uv run python -m pytest --cov --cov-config=pyproject.toml --cov-report=xml
```

The output will be like this:

```
 Testing code: Running pytest
============================================================================ test session starts =====
platform linux -- Python 3.13.1, pytest-8.3.4, pluggy-1.5.0
rootdir: /home/ameyanagi/StructuredPy-code
configfile: pyproject.toml
testpaths: tests
plugins: cov-6.0.0
collected 1 item

tests/test_foo.py .

---------- coverage: platform linux, python 3.13.1-final-0 -----------
Coverage XML written to file coverage.xml


============================================================================ 1 passed in 0.03s ======
```

What this does is that it runs the test file cnd see if the code runs with out the error. In the `test_foo.py` file, we are testing the `foo` function in the `foo.py` file will return the same string that we passed to it. `assert` is a keyword that is used to check if the condition is true. If the condition is false, it will raise an `AssertionError`.

3.4.1.1 Basics of Test Driven Development (TDD)

TDD is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved so that the tests pass. The cycle is repeated until the software is complete.

Let's assume we want to add a new function called `add` in the `foo.py` file. We will first create a add `function stub` in the `test_foo.py` file.

```python
def add(a, b):
    pass
```

Then in the `test_foo.py` file, we will write a test for the `add` function. Here, we will write 4 tests for the `add` function to check if the function works as expected.

```python
from structuredpy_code.foo import add

def test_add():
    assert add(1, 2) == 3

def test_add_negative():
    assert add(-1, -2) == -3

def test_add_zero():
    assert add(0, 0) == 0

def test_add_float():
    assert add(1.5, 2.5) == 4.0
```

Then we will run the test by running the following command:

```
make test
```

We now see that all the tests have failed, because we have not implemented the `add` function yet. The main idea of TDD is to write the test first, then implement the function to make the test pass.

Let's implement the `add` function in the `foo.py` file to make the test pass.

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

Then run the test again by running the following command:

```
make test
```

3.5 Documentation

3.5.1 Writing Good Documentation

3.5.1.1 Module-Level Documentation

Every Python module should start with a docstring explaining its purpose:

```
"""
Data processing module for time series analysis.

This module provides functions for loading, processing, and analyzing
time series data with a focus on financial applications.

Classes:
    TimeSeriesData: Container for time series with metadata
    DataProcessor: Handles data cleaning and transformation

Functions:
    load_data: Load time series from various file formats
    process_data: Clean and preprocess time series data
"""
```

3.5.1.2 Writing Effective Docstrings

Docstrings serve two key purposes: 1. In-code documentation for developers 2. Source for automated documentation tools

Documentation generators like MkDocs and Sphinx can parse docstrings to create: - API reference documentation - Function/class indexes - Example galleries - Module hierarchies

For best results with these tools: 1. Follow Google Style Guide conventions 2. Include all parameters and return types 3. Add examples for complex functions 4. Document exceptions and edge cases

Example of a well-documented function:

```
def process_data(data: np.ndarray, window_size: int = 5) -> np.ndarray:
    """Process time series data using rolling window calculations.

    Args:
        data: Input time series array of shape (n_samples,)
        window_size: Size of rolling window for calculations
```

```
    Returns:
        Processed data array of shape (n_samples - window_size + 1,)

    Raises:
        ValueError: If window_size is larger than data length
        TypeError: If data is not a numpy array

    Examples:
        >>> data = np.array([1, 2, 3, 4, 5])
        >>> process_data(data, window_size=3)
        array([2., 3., 4.])
    """
```

3.5.2 Generating API Documentation

MkDocs with Material theme provides excellent API documentation:

```
make docs
```

This will run the following command:

```
uv run mkdocs serve
```

The output will be like this:

```
make docs
INFO    -  Building documentation...
INFO    -  Cleaning site directory
INFO    -  Documentation built in 0.25 seconds
INFO    -  [02:42:02] Watching paths for changes: 'docs', 'mkdocs.yml'
INFO    -  [02:42:02] Serving on http://127.0.0.1:8000/StructuredPy-code/
```

In this case, the documentation will be available at `http://127.0.0.1:8000/StructuredPy-code/` in your web browser. You can quickly see that the documentation of the `foo` function is available in `module` section of the documentation, but the `add` function is not available in the documentation. This is because the `add` function does not have a docstring.

The `docstring` is a string that is used to document the code. It is written in the first line of the function, class, or module. There are several formats for writing docstrings, but we will be using the Google style for writing docstrings.

Let's add a docstring to the `add` function in the `foo.py` file.

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

    Args:
        a: The first number.
        b: The second number.

    Returns:
        The sum of the two numbers.
    """
    return a + b
```

Then run the test again by running the following command:

```
make docs
```

You can check from the docs that the `add` function is now available in the documentation. We have to note that the type definition is not available in the documentation, because we have not added the type definition to the function. Let's add the type definition to the `add` function in the `foo.py` file.

```python
def add(a: number, b: int) -> int:
    """Add two numbers.

    Args:
        a: The first number.
        b: The second number.

    Returns:
        The sum of the two numbers.
    """
    return a + b
```

Now we can see that the type definition is now available in the documentation. The type definition is not necessary for the code to run, but it is a good practice to add the type definition to the function in terms of readability and maintainability of the code.

3.6 Commit the code to Git

You can commit the code to Git by running the following command:

```
git add .
git commit -m "add add function"
```

Then if you want to push the code to GitHub, you can run the following command:

```
git push
```

3.7 Summary

In this chapter, we have learned how to create a new file, write a simple function, test the function, write documentation, and commit the code to Git. This is the basic workflow of coding that you will follow throughout the project.

Part II

Appendices

4 Python Basics

This appendix covers the fundamental Python concepts you'll need for this tutorial.

4.1 Variables and Data Types

4.1.1 Basic Data Types

```
# Numbers
temperature = 25.5       # Float
count = 10               # Integer

# Strings
message = "Hello"        # String
device_id = 'DEV001'     # String (single or double quotes)

# Booleans
is_connected = True      # Boolean
is_ready = False         # Boolean

# None (null value)
result = None            # None type
```

4.1.2 Lists and Dictionaries

```
# Lists (ordered, changeable)
temperatures = [25.5, 26.0, 25.8]
devices = ['DEV001', 'DEV002', 'DEV003']

# Accessing list elements
first_temp = temperatures[0]     # 25.5
last_device = devices[-1]        # 'DEV003'
```

```
8
9  # Changing list elements
10 temperatures[0] = 26.5
11
12 # Adding to lists
13 temperatures.append(26.2)
14
15 # Dictionaries (key-value pairs)
16 device_info = {
17     'id': 'DEV001',
18     'temperature': 25.5,
19     'connected': True
20 }
21
22 # Accessing dictionary values
23 device_id = device_info['id']           # 'DEV001'
24 temp = device_info.get('temperature')   # 25.5
```

4.2 Control Flow

4.2.1 If Statements

```
1  temperature = 25.5
2
3  if temperature > 30:
4      print("Temperature too high")
5  elif temperature < 20:
6      print("Temperature too low")
7  else:
8      print("Temperature normal")
9
10 # Using and/or
11 if temperature > 20 and temperature < 30:
12     print("Temperature in range")
13
14 # Checking for None
15 if result is not None:
16     print("Have result")
```

4.2.2 Loops

```
# For loops with lists
for temp in temperatures:
    print(f"Temperature: {temp}")

# For loops with range
for i in range(5):
    print(f"Reading {i}")

# While loops
count = 0
while count < 5:
    print(f"Count: {count}")
    count += 1

# Breaking loops
while True:
    temperature = get_temperature()
    if temperature > 30:
        break
```

4.3 Functions

4.3.1 Basic Functions

```
# Simple function
def get_temperature():
    return 25.5

# Function with parameters
def check_temperature(temp, min_temp, max_temp):
    return min_temp <= temp <= max_temp

# Function with default values
def read_sensor(retries=3):
    for i in range(retries):
        # Try reading sensor
        pass
```

4.3.2 Return Values

```
# Single return value
def calculate_average(values):
    if not values:
        return 0
    return sum(values) / len(values)

# Multiple return values
def get_stats(values):
    minimum = min(values)
    maximum = max(values)
    average = sum(values) / len(values)
    return minimum, maximum, average
```

4.4 Exception Handling

```
# Basic try/except
try:
    temperature = get_temperature()
except Exception as e:
    print(f"Error: {e}")

# Multiple exception types
try:
    with open('data.txt') as f:
        data = f.read()
except FileNotFoundError:
    print("File not found")
except PermissionError:
    print("Permission denied")
except Exception as e:
    print(f"Unknown error: {e}")

# Using finally
try:
    device.connect()
    device.read_data()
except Exception as e:
```

```
    print(f"Error: {e}")
finally:
    device.disconnect()  # Always runs
```

4.5 String Formatting

```
# f-strings (recommended)
temp = 25.5
message = f"Temperature: {temp}°C"

# Format method
message = "Temperature: {:.1f}°C".format(temp)

# String concatenation
message = "Temperature: " + str(temp) + "°C"
```

4.6 File Operations

```
# Reading files
with open('data.txt', 'r') as file:
    content = file.read()        # Read entire file
    lines = file.readlines()     # Read lines into list

# Writing files
with open('log.txt', 'w') as file:
    file.write("Temperature log\n")
    file.write(f"Reading: {temp}\n")

# Appending to files
with open('log.txt', 'a') as file:
    file.write(f"New reading: {temp}\n")
```

4.7 Modules and Imports

```python
# Importing entire modules
import time
time.sleep(1)

# Importing specific items
from datetime import datetime
current_time = datetime.now()

# Importing with alias
import pandas as pd
df = pd.DataFrame()
```

4.8 Essential Built-in Functions

```python
# len() - Length of sequences
length = len([1, 2, 3])     # 3

# range() - Number sequences
for i in range(5):          # 0 to 4

# print() - Output
print(f"Value: {value}")    # Formatted output

# type() - Get type of object
data_type = type(value)     # Get type

# str(), int(), float() - Type conversion
text = str(123)             # Convert to string
number = int("123")         # Convert to integer
decimal = float("12.3")     # Convert to float
```

4.9 List Operations

```python
# Common list methods
values = [1, 2, 3]
values.append(4)            # Add to end
```

```
values.insert(0, 0)          # Insert at position
values.remove(2)             # Remove value
values.pop()                # Remove and return last item

# List slicing
first_two = values[0:2]      # Get first two items
reversed_list = values[::-1]  # Reverse list

# List comprehension
squares = [x**2 for x in range(5)]
```

4.10 Dictionary Operations

```
# Common dictionary methods
info = {'id': 'DEV001'}
info['temp'] = 25.5          # Add/update key-value
value = info.get('id')       # Safe get with default
info.update({'connected': True})  # Update multiple

# Checking keys
if 'id' in info:
    print("Has ID")

# Dictionary comprehension
squares = {x: x**2 for x in range(3)}
```

4.11 Common Modules Used in This Tutorial

```
# Time and dates
import time
time.sleep(1)                # Wait 1 second

from datetime import datetime
now = datetime.now()         # Current time

# JSON handling
import json
```

```
data = json.loads('{"temp": 25.5}')  # Parse JSON
text = json.dumps(data)    # Convert to JSON

# System operations
import os
path = os.path.join('folder', 'file.txt')
```

4.12 Code Style Guidelines

1. Use clear, descriptive names
2. Use snake_case for functions and variables
3. Add spaces around operators
4. Use 4 spaces for indentation
5. Add comments for complex logic
6. Keep functions focused and small
7. Handle errors appropriately

Example of well-styled code:

```
def calculate_average_temperature(readings):
    """Calculate average temperature from a list of readings."""
    if not readings:
        return None

    try:
        total = sum(readings)
        return total / len(readings)
    except Exception as e:
        print(f"Error calculating average: {e}")
        return None
```

4.13 What's Next

This appendix covered the Python basics needed for our tutorial. As you work through the main chapters, refer back here if you need to review any concepts.