Bootcamp 134 | Python

Course 13 | Clean Code

Amir Hossein Chegouniyan

Head of the Technical Team at Dariche Tejarat

Lecturer of Python – Django at Maktab Sharif

Amirhossein-chegounian

Special thanks to:

Mohammad Amin Rahimi

Content

Introduction to Clean Code Principles

- What’s Clean Code?
  - Clear
  - Readable
  - Understandable
  - Maintainable

Introduction to Clean Code Principles | Why Clean Code Matters

- Better security
- Fewer bugs
- Better teamwork
- Easier maintenance
- Future scalability

Basic Clean Code Practices

- Naming Conventions
- Writing Functions with a Single Responsibility
- Avoiding Hard-Coding Values
- Avoiding Deep Nesting
- Commenting and Documentation
- Using Pythonic Code
- Keeping Code DRY (Don't Repeat Yourself)

Basic Clean Code Practices | Naming Conventions

- Use descriptive and meaningful names for variables, functions, and classes.
- Follow Python’s naming conventions: snake_case for variables and functions, CamelCase for classes.

Basic Clean Code Practices | Naming Conventions | Example

```
def c(p):
    return p * 0.2
```

✗

```
def calculate_discount(price):
    return price * 0.2
```

☑

Basic Clean Code Practices | Writing Functions with a Single Responsibility

- Keep functions focused on a single task or responsibility.
- Tips on writing smaller functions that are easier to understand and test.

Basic Clean Code Practices | Writing Functions with a Single Responsibility | Exmaple

```
def process_order(order):
    total = sum(order)
    print(f"Total: {total}")
    with open("log.txt", "a") as file:
        file.write(str(total) + "\n")
```

✗

```
def calculate_total(order):
    return sum(order)

def log_total(total):
    with open("log.txt", "a") as file:
        file.write(str(total) + "\n")

def process_order(order):
    total = calculate_total(order)
    print(f"Total: {total}")
    log_total(total)
```

☑

Basic Clean Code Practices | Avoiding Hard-Coding Values

- Define constants for fixed values at the top of your code.
- Use clear variable names instead of "magic numbers" or unexplained values.

Basic Clean Code Practices | Avoiding Hard-Coding Values | Example

```
if score > 60:
    print("Pass")
```

✕

```
PASSING_SCORE = 60
if score > PASSING_SCORE:
    print("Pass")
```

☑

Basic Clean Code Practices | Avoiding Deep Nesting

- Break down deeply nested code for readability.
- Use early returns in functions to avoid multiple levels of indentation.

Basic Clean Code Practices | Avoiding Deep Nesting | Example

```
def process(user):
    if user:
        if user.is_active:
            if user.has_permission("edit"):
                print("User can edit")
```

✕

```
def process(user):
    if not user:
        return
    if not user.is_active:
        return
    if not user.has_permission("edit"):
        return

    print("User can edit")
```

☑

Basic Clean Code Practices | Commenting and Documentation

- Write concise, helpful comments where necessary (explain why, not what).
- Use docstrings (""" """) for functions and classes to explain their purpose and usage.

Basic Clean Code Practices | Commenting and Documentation | Example

```
# This function multiplies x by 2
def double(x):
    return x * 2
```

```
# If you are under 18 years old, you are not allowed to register
def can_register(age):
    return age >= 18
```

Basic Clean Code Practices | Using Pythonic Code

- Avoid redundant code by using built-in functions and Pythonic idioms.
- Examples: Using list comprehensions, enumerate() instead of range(len()), and unpacking for multiple assignments.

Basic Clean Code Practices | Using Pythonic Code | Example

```
if fruit == "apple" or fruit == "banana" or
fruit == "orange":
    print("It's a fruit")
```

✗

```
if fruit in {"apple", "banana", "orange"}:
    print("It's a fruit")
```

☑

Basic Clean Code Practices | DRY (Don't Repeat Yourself)

- Refactor repeated code into functions or variables to avoid redundancy.

Basic Clean Code Practices | DRY (Don't Repeat Yourself) | Exmaple

```
price1 = 100
price2 = 150

discount1 = price1 * 0.1
discount2 = price2 * 0.1
```

✕

```
def calculate_discount(price):
    return price * 0.1

discount1 = calculate_discount(100)
discount2 = calculate_discount(150)
```

☑

Advanced Clean Code Tips

- Error Handling
- Consistent Code Style
- Organizing Imports and Modules

Advanced Clean Code Tips | Error Handling

- Use exception handling (try, except) effectively to manage errors gracefully.
- Ensure error messages are clear and descriptive.

Advanced Clean Code Tips | Consistent Code Style

- Follow PEP 8 (Python's style guide) for consistent formatting.
- Using tools like pylint or black to check and format code automatically.

Advanced Clean Code Tips | Organizing Imports and Modules

- Place imports at the top of the file, grouped by standard library, third-party libraries, and project imports.
- Use specific imports rather than import * to avoid namespace conflicts.

Introduction to Virtual Environments

- Introduction to Virtual Environments
- Best Practices for Virtual Environments & Dependency Management

Introduction to Virtual Environments | 1

- Creating a Virtual Environment:
  - Syntax: *python -m venv myenv*
  - Explain how this creates an isolated environment for the project.

Introduction to Virtual Environments | 3

- Activation on different OS:
  - **Windows**: *myenv\Scripts\activate*
  - **macOS/Linux**: *source myenv/bin/activate*
- Deactivation: *deactivate*

Introduction to Virtual Environments | 3

- Installing Packages in a Virtual Environment:
  - **Using pip**: Install packages locally to the environment with pip install package_name.
  - **Listing Dependencies**: Use *pip freeze* to list all installed packages and their versions.

Introduction to Virtual Environments | 4

- Saving and Sharing Dependencies:
  - Create a requirements.txt file with *pip freeze > requirements.txt* to save dependencies.
  - Installing from requirements.txt:
    - Use *pip install -r requirements.txt* to recreate the environment.

Best Practices for Virtual Environments and Dependency Management

- **Avoiding Global Installs**: Always use virtual environments for project-specific packages to prevent conflicts.
- **Using .gitignore for Virtual Environments**: Add virtual environment directories (e.g., myenv/) to .gitignore to keep them out of version control.
- Automating Environment Setup:
  - Share requirements.txt in version control for easy environment setup by others.
  - Optionally introduce pipenv or poetry for more complex dependency and environment management.

Best Practices for Virtual Environments and Dependency Management | 2

Feature	venv (Standard)	pipenv	poetry
Virtual environment management	Manual (python -m venv)	Automatic	Automatic
Package management	With pip	With pipenv install	With poetry add
Config files	None or requirements.txt	Pipfile, Pipfile.lock	pyproject.toml, poetry.lock
Auto dependency installation	Manual	Yes	Yes
Version locking (reproducibility)	Optional (pip freeze)	Yes (Pipfile.lock)	Yes (poetry.lock)
Security checks	No	Yes (pipenv check)	No (but can use external tools)
Build & publish packages	Manual and complex	Not designed for it	Yes (very simple)
Install speed	Fast	Slower	Faster
Beginner friendly	Yes	Yes	Relatively yes (good docs)
Smart dependency resolution	No	Yes	Yes

Best Practices for Virtual Environments and Dependency Management | 2

Command	State
mkdir my_project + cd my_project	Create project
python -m venv venv	Create venv
source venv/bin/activate venv\Scripts\activate	Activate
pip install package_name	Install packages
pip list	List of packages
pip freeze > requirements.txt	Save list of package
pip install -r requirements.txt	Install again
deactivate	Exit
.gitignore venv/	Remove from Git

Any question?

Next course

- Why Linux?
- Installing Linux
- Linux File System Overview
- Basic Linux Commands
- Introduction to Package Managers