

Coding Methodology in Python

Lecturer: Nguyễn Thái Hà (Ph.D)

Supporter: Nguyễn Thọ Anh Khoa (Ph.D Candidate)

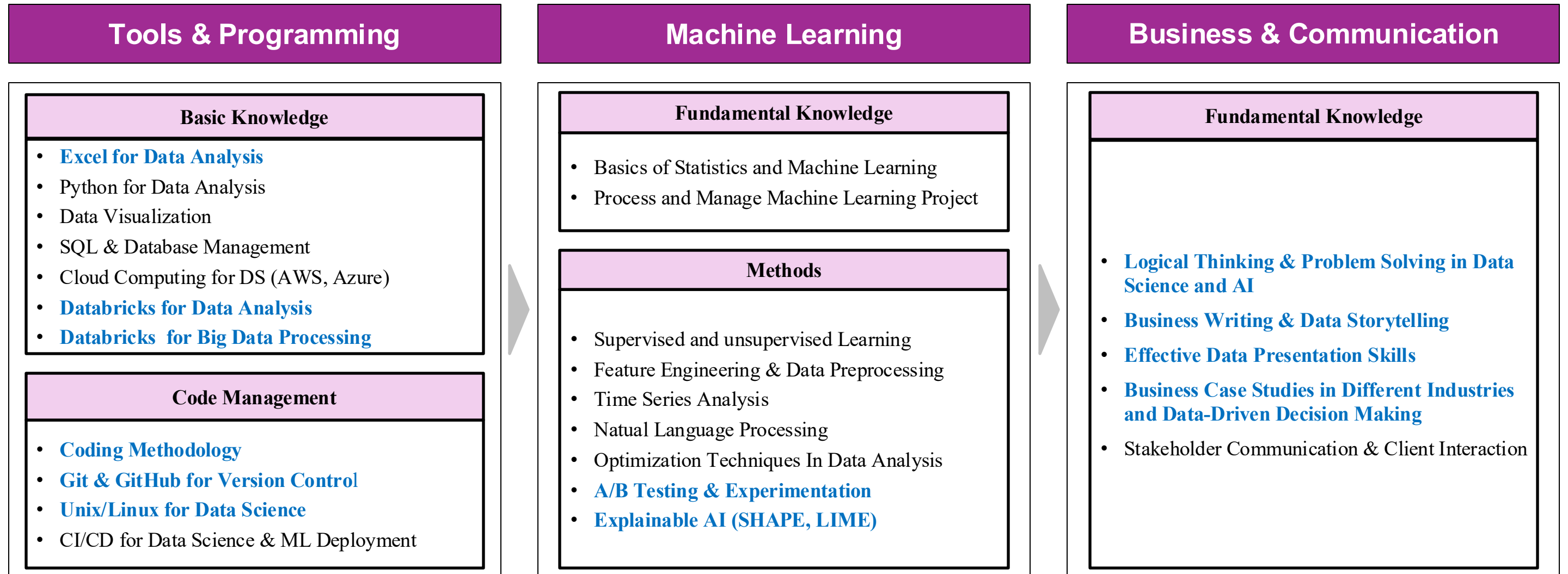
Date: 07/06/2025 (Sat)

AIO2025

[Link Source Code](#)

@AI VietNam

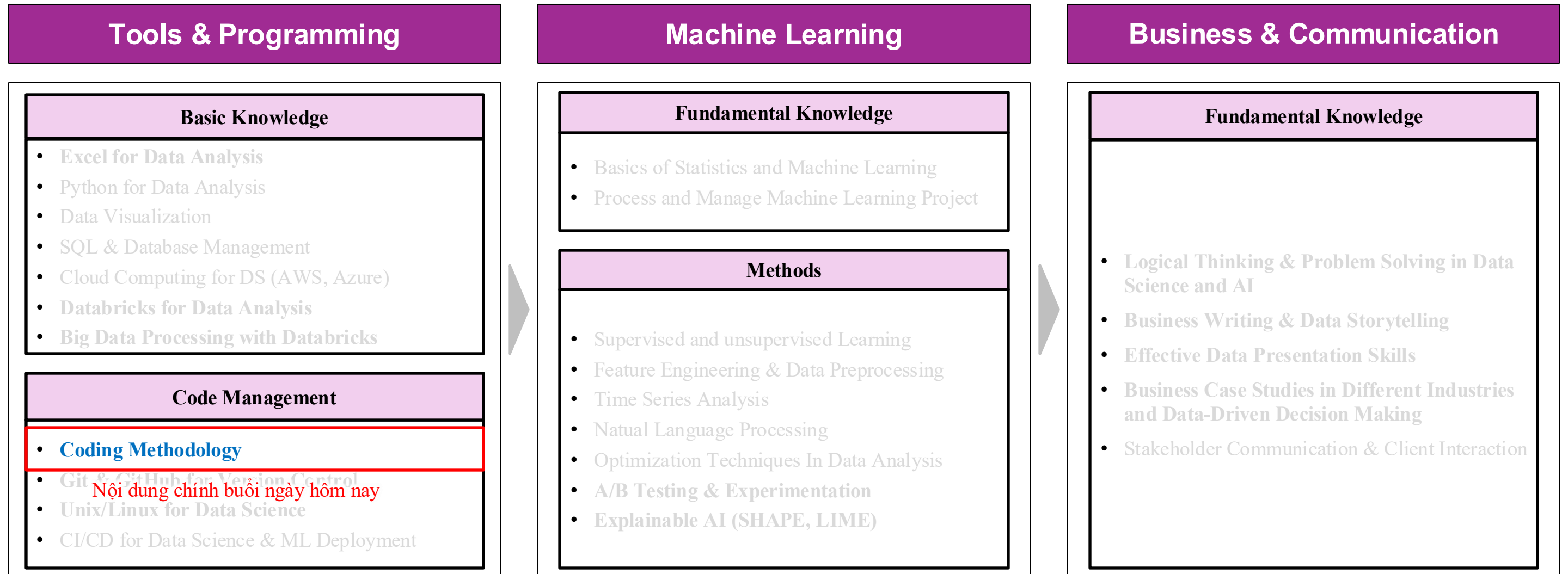
Roadmap for learning Data Science



Practical Projects:

- ✓ **Project 1: Sales Forecasting and Demand Prediction; and applied XAI**
- ✓ **Project 2: Xây dựng công cụ tối ưu hoá quy trình phân bổ hàng hoá cho chuỗi cửa hàng thời trang**
- ✓ **Project 3: Customer Segmentation for Marketing Strategy using Unsupervised Learning - Phân khúc khách hàng cho chiến lược Marketing**

Roadmap for learning Data Science



Practical Projects:

- ✓ Project 1: Sales Forecasting and Demand Prediction; and applied XAI
- ✓ Project 2: Xây dựng công cụ tối ưu hoá quy trình phân bổ hàng hoá cho chuỗi cửa hàng thời trang
- ✓ Project 3: Customer Segmentation for Marketing Strategy using Unsupervised Learning - Phân khúc khách hàng cho chiến lược Marketing



Nắm vững nguyên tắc về Clean Code và tiêu chuẩn PEP-8

Hiểu và áp dụng các nguyên tắc clean code như là định dạng chuẩn PEP-8 để tạo mã nguồn hiệu quả, dễ đọc và dễ bảo trì



Xây dựng tư duy "Pythonic"

Làm quen với phong cách lập trình đặc trưng của Python, sử dụng các tính năng độc đáo của ngôn ngữ một cách hiệu quả.



Áp dụng một số nguyên tắc để lập trình tốt hơn

Thực hành các nguyên tắc DRY, KISS, YAGNI và thiết kế theo hợp đồng để tạo code có chất lượng cao.



Học cách tổ chức mã nguồn có cấu trúc

Học cách tổ chức project, phân chia module và viết code có tính mở rộng cao.

Phần 1: Cơ bản về Clean Code và PEP-8

Phần 2: Viết Code Pythonic

Phần 3: Nguyên lý chung để viết code tốt

Phần 4: Nguyên tắc SOLID và Design Patterns (Nâng Cao)

Phần 1: Cơ bản về Clean Code và PEP-8

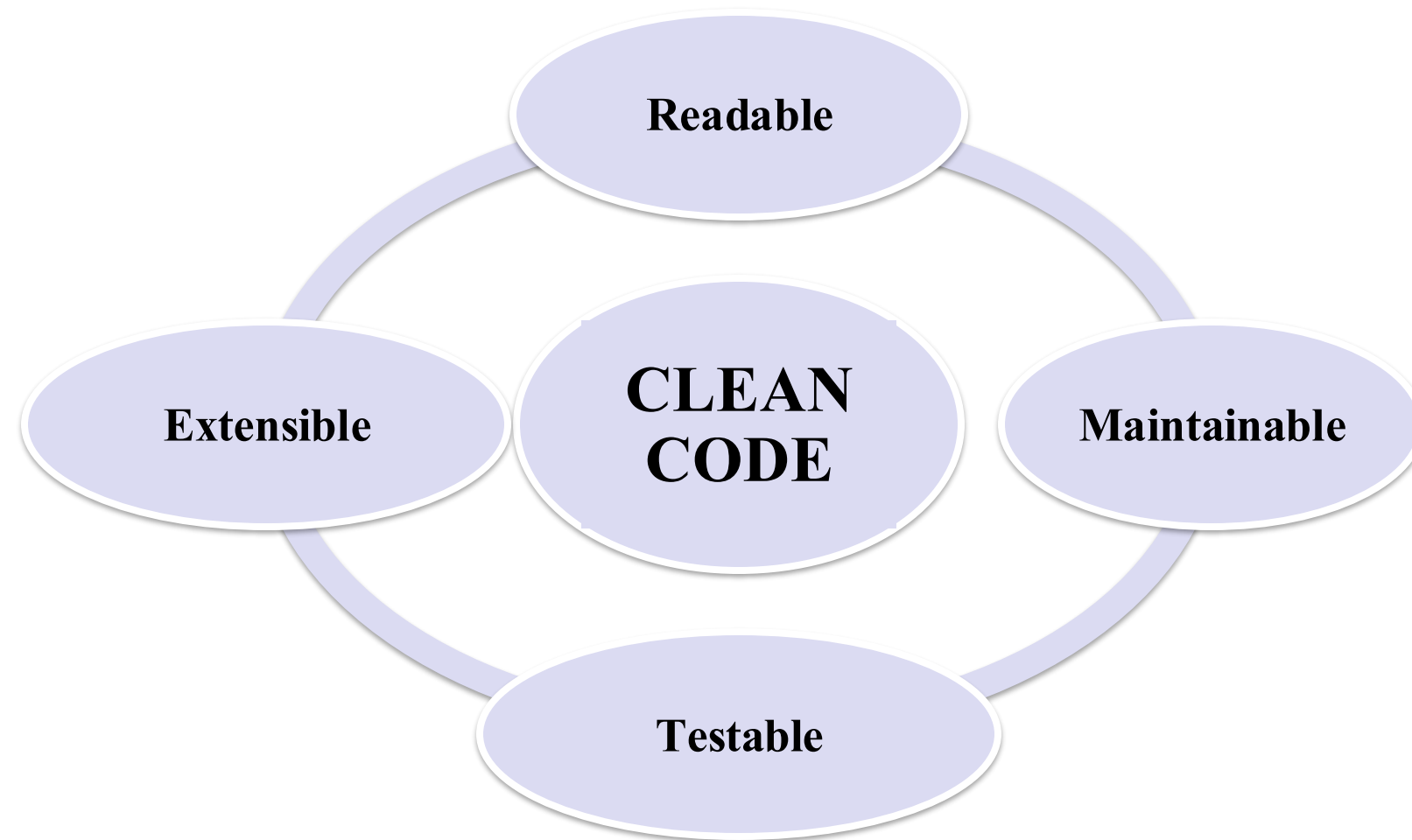
Phần 2: Viết Code Pythonic

Phần 3: Nguyên lý chung để viết code tốt

Phần 4: Nguyên tắc SOLID và Design Patterns (Nâng Cao)

Clean Code là gì?

Clean code là mã nguồn **rõ ràng, dễ đọc, dễ bảo trì, có khả năng mở rộng và dễ dàng kiểm thử**. Nó được viết theo cách mà người khác (đồng nghiệp) hoặc chính bạn trong tương lai có thể dễ dàng hiểu, cải tiến hoặc bảo trì.





Dễ đọc và dễ hiểu (Readable)

Code nên được viết sao cho người khác có thể dễ dàng hiểu được mục đích và cách thức hoạt động



Dễ bảo trì (Maintainable)

Clean code nên dễ dàng sửa đổi, mở rộng và tái sử dụng mà không gây ra lỗi phụ



Dễ kiểm thử (Testable)

Code sạch nên được thiết kế để dễ dàng viết unit test và thực hiện kiểm thử tự động



Có khả năng mở rộng (Extensible)

Code được thiết kế để dễ dàng thêm tính năng mới mà không cần thay đổi code hiện tại

Tầm Quan Trọng Của Clean Code

Clean Code không chỉ là phong cách viết code đơn thuần mà còn là **yếu tố quan trọng giúp tăng hiệu suất làm việc, cải thiện hợp tác nhóm và giảm chi phí bảo trì dài hạn.**



Giảm thiểu lỗi (Bugs)

Clean code giúp dễ dàng phát hiện và ngăn chặn lỗi tiềm ẩn, giảm thời gian debug và sửa lỗi.



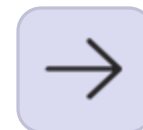
Tiết kiệm thời gian dài hạn

Mặc dù ban đầu có thể tốn thời gian hơn, clean code giúp tiết kiệm rất nhiều thời gian trong quá trình bảo trì và phát triển sau này.



Cải thiện làm việc nhóm

Giúp các thành viên trong nhóm dễ dàng hiểu và làm việc với code của nhau, tăng hiệu quả hợp tác.



Dễ dàng mở rộng

Clean code có cấu trúc rõ ràng, dễ dàng thêm tính năng mới mà không làm ảnh hưởng đến hệ thống hiện tại.

Clean code sử dụng tên biến/hàm có ý nghĩa, thêm tài liệu mô tả và tuân thủ các quy ước định dạng để tăng khả năng đọc hiểu.

Ví dụ 1: Đặt tên biến có ý nghĩa

```
# Bad example - unclear variable names
print("Bad example - unclear variable names:")

a = 5
b = 10
c = a + b
print(f"a + b = {c}")

# Good example - descriptive variable names
print("\nGood example - descriptive variable names:")

apples = 5
oranges = 10
total_fruits = apples + oranges
print(f"Number of apples: {apples}")
print(f"Number of oranges: {oranges}")
print(f"Total fruits: {total_fruits}")
```

Bad example - unclear variable names:
a + b = 15

Good example - descriptive variable names:
Number of apples: 5
Number of oranges: 10
Total fruits: 15

Ví dụ 2: Viết code có formatting

```
# Bad example - poor formatting
print("Bad example - poor formatting:")

def add_numbers(x,y,z):
    result=x+y+z
    return result

print(f"add_numbers(1, 2, 3) = {add_numbers(1,2,3)}")

# Good example - proper formatting with spaces
print("\nGood example - proper formatting with spaces:")

def add_numbers_clean(x, y, z):
    result = x + y + z
    return result

print(f"add_numbers_clean(1, 2, 3) = {add_numbers_clean(1, 2, 3)}")
```

Bad example - poor formatting:
add_numbers(1, 2, 3) = 6

Good example - proper formatting with spaces:
add_numbers_clean(1, 2, 3) = 6

Code dễ mở rộng cho phép **thêm chức năng mới mà không cần sửa đổi code hiện có**. Thiết kế này sử dụng các hàm riêng biệt và cấu trúc dữ liệu như dictionary để quản lý các phép toán, giúp tuân thủ nguyên tắc **"Open-Closed Principle"**.

Khó mở rộng

```
# Bad example - hard to futher addition

def calculate(a, b, operation):
    if operation == 1:
        return a + b
    elif operation == 2:
        return a - b
    elif operation == 3:
        return a * b

print(f"Results: {calculate(1,2, 3)}")

Results: 2
```

Dễ mở rộng

```
# Good example - easy to futher modification

def add(a, b):
    return a + b
def subtract(a, b):
    return a - b
def multiply(a, b):
    return a * b

operations = {
    'add': add,
    'subtract': subtract,
    'multiply': multiply
}

def calculate(a, b, operation):
    func = operations.get(operation)
    if func:
        return func(a, b)
    else:
        raise ValueError("Invalid operation")

print(f"Results: {calculate(1,2, operation='multiply')}")

Results: 2
```

Khi Nào Có Thể "Bỏ Qua" Clean Code?



Làm nguyên mẫu nhanh

Khi cần kiểm chứng ý tưởng trong thời gian ngắn.
Nhưng hãy refactor nếu quyết định dùng code đó.



Tình huống khẩn cấp

Khi hệ thống gặp sự cố nghiêm trọng. Giải quyết vấn đề trước, refactor sau.



Thử nghiệm nhỏ

Code chỉ dùng một lần để khám phá hoặc học. Không sử dụng trong sản phẩm.



Code chỉ mình bạn dùng

Nhưng hãy nhớ: "mình bạn" cũng bao gồm bạn của tương lai!

PEP-8 (Python Enhancement Proposal 8) là **tiêu chuẩn định dạng code Python chính thức**, quy định cách trình bày code để đảm bảo tính nhất quán và dễ đọc trên toàn dự án. Được tạo ra bởi Guido van Rossum - cha đẻ của Python.

Không tuân theo PEP-8

```
def tinhTong(a,b):  
    return a+b  
  
# Sử dụng hàm  
ketQua=tinhTong(5,3)  
print(ketQua)
```

Mục tiêu

- ✓ Làm sao cho code dễ đọc, dễ hiểu
- ✓ Làm sao giảm lỗi
- ✓ Cải thiện việc làm việc theo nhóm hiệu quả hơn

Tuân theo PEP-8

```
def tinh_tong(a: int, b: int) -> int:  
    """  
    Tính tổng hai số.  
  
    Args:  
        a (int): Số thứ nhất  
        b (int): Số thứ hai  
  
    Returns:  
        int: Tổng của hai số  
    """  
    return a + b  
  
# Sử dụng hàm  
ket_qua = tinh_tong(5, 3)  
print(ket_qua)
```


Quy Tắc Đặt Tên – Naming Convention

1 Biến/Hàm: snake_case

Sử dụng chữ thường và dấu gạch dưới để phân tách các từ.

```
# Khai báo biến
my_variable = 10

# Định nghĩa hàm
def my_function():
    print("Hello")
```

2 Class: PascalCase

Viết hoa chữ cái đầu của mỗi từ, không có dấu gạch dưới.

```
# Đặt tên class
class MyClass:
    def __init__(self, name):
        self.name = name
```

3 Hằng Số: UPPER_CASE

Sử dụng tất cả chữ hoa và dấu gạch dưới giữa các từ.

```
# Đặt tên hằng số
PI = 3.14
MAX_VALUE = 100
MIN_VALUE = 5

def check_exam_result(score):
    if score >= PASSING_SCORE:
        return "Pass"
    return "Fail"
```



Thụt lề

Dùng 4 khoảng trắng cho mỗi cấp thụt lề. Không dùng tab!



Khoảng trắng trong các biểu thức

Thêm khoảng trắng hai bên phép toán: $a = b + c$



Tránh khoảng trắng thừa

Sau dấu phẩy: $f(a, b)$ không phải $f(a , b)$



Không thêm khoảng trắng trong ngoặc

Đúng: `list([1, 2, 3])` | Sai: `list([1, 2, 3])`

Quy Tắc PEP-8

- Mã nguồn: tối đa 79 ký tự/dòng
- Docstring/comment: tối đa 72 ký tự/dòng
- IDE hiện đại: Nhiều lập trình viên dùng giới hạn 88-100 ký tự
- Mục đích: Đảm bảo code dễ đọc trên mọi màn hình

Cách Ngắt Dòng Đúng Chuẩn

- Dùng dấu gạch chéo (\) để ngắt dòng trực tiếp
- Ưu tiên dùng ngoặc đơn/vuông/nhọn () [] {} để ngắt dòng ẩn
- Ngắt tại dấu phẩy hoặc trước toán tử binary (and, or, +, *)
- Thụt lề dòng tiếp theo thêm 4 khoảng trắng so với dòng gốc

1 Thư viện chuẩn

Các module từ thư viện chuẩn Python như os, sys, math...

```
import os  
import sys  
import math
```

2 Thư viện bên thứ ba

Các package được cài qua pip như requests, numpy, pandas...

```
import numpy as np  
import pandas as pd  
import requests
```

3 Module nội bộ

Các module tự viết trong dự án của bạn.

```
from myproject.utils import  
parse_data  
from myproject.models import  
User
```

Dòng trống giữa các hàm và class

Sử dụng 2 dòng trống để phân tách các định nghĩa class.

Các hàm bên trong class cách nhau 1 dòng trống.

Dòng trống trong hàm

Sử dụng dòng trống để phân tách các nhóm logic code.

Không nên quá nhiều dòng trống gây khó đọc.

Cấu trúc class chuẩn

Thứ tự: docstring → biến class → `__init__()` → các phương thức khác → các phương thức private (`__method`) → các static method.

Cấu trúc hàm

Thứ tự: docstring → các lệnh validate đầu vào → code chính → return. Mỗi hàm nên có một mục đích duy nhất.

Tài liệu hóa code giúp **nâng cao khả năng bảo trì, giảm thời gian đọc hiểu, và tạo điều kiện cho việc hợp tác** trong đội ngũ phát triển.

Viết docstring hiệu quả

Docstring được đặt trong cặp dấu ba nháy kép (""") ngay sau định nghĩa hàm. Python hỗ trợ nhiều định dạng docstring phổ biến:

- **Google style:** Dễ đọc, được sử dụng rộng rãi, phân chia rõ ràng Args, Returns, Raises
- **NumPy style:** Chi tiết hơn, lý tưởng cho các hàm khoa học dữ liệu
- **reStructuredText:** Tích hợp với Sphinx để tạo tài liệu web

Annotations và type hints

Annotations [\(PEP 484\)](#) cung cấp kiểm tra kiểu dữ liệu tĩnh:

- Kiểu cơ bản: `def add(a: int, b: float) -> float:`
- Kiểu phức tạp: `from typing import List, Dict, Tuple, Optional`
- Kiểu Union: `def process(data: Union[str, bytes]) -> None:`
- Công cụ kiểm tra: mypy, Pyright, PyCharm để phát hiện lỗi trước khi chạy

Tài liệu module và project

Tạo tài liệu hoàn chỉnh cho dự án với nhiều cấp độ:

- Module docstring: Đặt ở đầu file, mô tả mục đích và các hàm chính
- README.md: Hướng dẫn cài đặt, ví dụ đơn giản, cấu trúc project
- CONTRIBUTING.md: Quy trình đóng góp, quy tắc code
- Sphinx: Tự động tạo tài liệu API từ docstring
- Wiki: Tài liệu chi tiết cho các tính năng phức tạp

Docstring không chỉ giúp người đọc hiểu code mà còn hỗ trợ tích hợp với các công cụ như Sphinx để tạo tài liệu tự động và IDE để cung cấp gợi ý thông minh.

Không docstring

```
# Bad example - no documentation
print("Bad example - no documentation:")

def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Alice"))
print(greet("Bob", "Hi"))
```

```
Bad example - no documentation:
Hello, Alice!
Hi, Bob!
```

Có docstring rõ ràng

```
# Good example - with docstring
print("\nGood example - with docstring:")

def greet_documented(name, greeting="Hello"):
    """Create a personalized greeting message.

    This function takes a person's name and creates a greeting.

    Args:
        name: The name of the person to greet
        greeting: The greeting word to use (default is "Hello")

    Returns:
        A greeting message as a string
    """
    return f"{greeting}, {name}!"

print(greet_documented("Alice"))
print(greet_documented("Bob", "Hi"))
```

```
Good example - with docstring:
Hello, Alice!
Hi, Bob!
```

Type annotations giúp kiểm tra kiểu dữ liệu, phát hiện lỗi sớm và tối ưu trải nghiệm IDE.

Annotations cơ bản

```
# Annotation basics
def calculate_total(price: float, quantity: int) -> float:
    """Tính tổng giá trị đơn hàng.

    Args:
        price: Giá sản phẩm (VND)
        quantity: Số lượng sản phẩm

    Returns:
        Tổng giá trị đơn hàng
    """
    return price * quantity
```

Annotations cho kiểu phức tạp





```
# Annotation for complex types
from typing import List, Dict, Optional, Union, Tuple

# Kiểu Union - nhiều kiểu có thể
def process_input(data: Union[str, bytes]) -> str:
    return data.decode() if isinstance(data, bytes) else data

# Kiểu Optional - có thể None
def get_user(id: int) -> Optional[Dict[str, any]]:
    # Trả về None nếu không tìm thấy
    return database.find_user(id)

# Kiểu phức hợp
def analyze_data(
    values: List[float],
    labels: Dict[str, str],
    config: Optional[Tuple[int, int]] = None
) -> Dict[str, List[float]]:
    # xử lý và trả về kết quả
    ...
```

- ✓ Sử dụng mypy để kiểm tra annotations trước khi chạy. IDE hiện đại như VSCode và PyCharm tự động hiển thị lỗi kiểu dữ liệu khi viết code.

Công cụ	Mô tả	Cài đặt	Sử dụng
 Flake8	Kiểm tra code: cú pháp, style (PEP-8). Kết hợp pyflakes, pycodestyle và mccabe để phát hiện lỗi logic, độ phức tạp.	<code>pip install flake8</code>	<code>flake your_file.py</code>
 Black	Format code tự động, chuẩn hóa style, giải quyết tranh chấp để nhất quán trong project	<code>pip install black</code>	<code>black your_file.py</code>
 Pylint	Phân tích tĩnh code: lỗi, chất lượng, điểm số, chỉnh sửa. Phân tích sâu hơn Flake8.	<code>pip install pylint</code>	<code>pylint your_file.py</code>
 Mypy	Kiểm tra kiểu dữ liệu tĩnh dựa trên type annotations để phát hiện lỗi. Hỗ trợ kiểm tra các type phức tạp	<code>pip install mypy</code>	<code>mypy your_file.py</code>

Lưu ý

- Các công cụ trên có thể tích hợp vào CI/CD pipeline, pre-commit hooks hoặc IDEs như PyCharm và VSCode để tự động kiểm tra code trước khi commit.

Ví dụ về Flake8

Tạo 1 file "greeting.py" có nội dung như sau

```
def greeting(name):  
    print("Hello, " + name)  
  
greeting("Alice")  
greeting("Bob")
```

Để khắc phục, bạn cần loại bỏ khoảng cách ở phần đầu của dòng 5 này.

```
def greeting(name):  
    print("Hello, " + name)  
  
greeting("Alice")  
greeting("Bob")
```

Chạy Flake8 với greeting.py

```
flake8 your_file.py
```

Kết quả trả về:

```
greeting.py:5:1: E999 IndentationError: unexpected indent
```

Kết quả này chỉ ra rằng dòng 5 được thụt lề không cần thiết.

Giờ khi chạy Flake8, bạn sẽ nhận được cảnh báo sau:

```
greeting.py:4:1: E305 expected 2 blank lines after class or function  
definition, found 1
```

```
greeting.py:5:16: W292 no newline at end of file
```

Dòng 4 sẽ có 2 dòng trống sau khi định nghĩa hàm greeting, nhưng thực tế chỉ có 1 dòng.

Dòng 5 sẽ có một dòng mới ở cuối file.

PEP-8 (Python Enhancement Proposal 8) là **tiêu chuẩn định dạng code Python chính thức**, quy định cách trình bày code để đảm bảo tính nhất quán và dễ đọc trên toàn dự án.

1 Quy tắc căn lề và khoảng trắng

- Thụt lề: Sử dụng 4 khoảng trắng (không dùng tab)
- Độ dài dòng: Tối đa 79 ký tự cho code, 72 cho docstring
- Khoảng trắng: Không đặt trong ngoặc, chỉ sau dấu phẩy
- Phép toán: Đặt khoảng trắng hai bên ($x = 1$)

3 Cấu trúc lệnh và imports

- Imports: Mỗi module một dòng, theo thứ tự chuẩn
- Nhóm imports: standard library, third-party, local
- Dòng trống: 2 dòng trước class, 1 dòng trước method
- So sánh: Dùng `if x is None` thay vì `if x == None`

2 Quy ước đặt tên

- Biến, hàm: snake_case (`my_variable`, `calculate_total`)
- Class: PascalCase (`MyClass`, `UserAccount`)
- Hằng số: UPPERCASE_WITH_UNDERSCORES (`MAX_SIZE`)
- Method trong class: snake_case (`get_value`, `calculate_tax`)

4 Công cụ kiểm tra PEP-8

- `pycodestyle`: Kiểm tra tuân thủ PEP-8
- `autopep8`: Tự động định dạng code theo PEP-8
- `black`: Định dạng code nghiêm ngặt, thống nhất
- `flake8`: Kết hợp kiểm tra lỗi và định dạng

Yêu cầu: Cho hàm tính chỉ số BMI và phân loại kết quả dựa trên giá trị BMI. Các bạn sử dụng nguyên tắc PEP-8 để viết lại đoạn code ban đầu sao cho tuân thủ các quy tắc về phong cách code của PEP-8 theo các gợi ý sau:

1. Đặt tên biến và hàm:

- Sử dụng phong cách snake_case (chữ thường, từ cách nhau bởi dấu gạch dưới).

2. Thụt lề:

- Thụt lề đúng theo chuẩn PEP-8 (4 khoảng cách cho mỗi cấp thụt lề)

3. Khoảng trắng:

- Thêm khoảng trắng sau dấu phẩy, xung quanh phép toán

4. Cấu trúc:

- Đảm bảo mỗi lệnh return nằm trên một dòng riêng

5. So sánh:

- Không đặt điều kiện trong ngoặc đơn, tuân thủ quy chuẩn if/elif/else

6. Docstring:

- Thêm tài liệu mô tả chức năng hàm, tham số và giá trị trả về

Bài tập thực hành PEP-8

Nhiệm vụ: Sử dụng nguyên lý PEP-8 để refactoring lại đoạn code sau đây

Không chuẩn

```
# Bad example - Unfollow PEP-8
def calculateBMI(height,weight):
    BMI=weight/(height**2)
    if(BMI<=18.5):return"Thiếu cân"
    elif(BMI<=25):return"Bình thường"
    else:return"Thừa cân"

print("\nResults:")
calculateBMI(height=170, weight=70)
```

Tuân thủ PEP-8

Đáp án

Một dự án Python được tổ chức tốt sẽ **đễ dàng bảo trì và mở rộng**. Dưới đây là cấu trúc thư mục chuẩn mực

- **Thư mục gốc (project_name/)**
 - Chứa README, requirements.txt, setup.py
- **Mã nguồn (project_name/src/)**
 - Chứa các module và package chính
- **Tests (project_name/tests/)**
 - Chứa các bộ kiểm thử (unittest, pytest)
- **Tài liệu (project_name/docs/)**
 - Hướng dẫn sử dụng và tài liệu API
- **Tài nguyên (project_name/resources/)**
 - Dữ liệu tĩnh, templates, assets
- Sử dụng các file cấu hình như .gitignore, pyproject.toml, và tập tin README.md để đảm bảo dự án được quản lý hiệu quả và dễ dàng cho người khác hiểu và đóng góp.

Ví dụ: https://github.com/khoanta-ai/python_project_template

Cookiecutter Data Science giúp tạo cấu trúc folder tiêu chuẩn nhanh chóng

Project Organization

```

├── LICENSE
├── Makefile          <- Makefile with commands like `make data` or `make train`
├── README.md         <- The top-level README for developers using this project.
├── data
│   ├── external      <- Data from third party sources.
│   ├── interim       <- Intermediate data that has been transformed.
│   ├── processed     <- The final, canonical data sets for modeling.
│   └── raw           <- The original, immutable data dump.
├── docs              <- A default Sphinx project; see sphinx-doc.org for details
├── models            <- Trained and serialized models, model predictions, or model summaries
├── notebooks         <- Jupyter notebooks. Naming convention is a number (for ordering),
│                       the creator's initials, and a short '-' delimited description, e.g.
│                       `1.0-jqp-initial-data-exploration`.
├── references         <- Data dictionaries, manuals, and all other explanatory materials.
├── reports
│   └── figures        <- Generated analysis as HTML, PDF, LaTeX, etc.
│                       <- Generated graphics and figures to be used in reporting
├── requirements.txt  <- The requirements file for reproducing the analysis environment, e.g.
│                       generated with `pip freeze > requirements.txt`
├── setup.py          <- makes project pip installable (pip install -e .) so src can be imported
├── src               <- Source code for use in this project.
│   ├── __init__.py   <- Makes src a Python module
│   ├── data          <- Scripts to download or generate data
│   │   └── make_dataset.py
│   ├── features      <- Scripts to turn raw data into features for modeling
│   │   └── build_features.py
│   ├── models        <- Scripts to train models and then use trained models to make
│   │                   predictions
│   │   ├── predict_model.py
│   │   └── train_model.py
│   └── visualization <- Scripts to create exploratory and results oriented visualizations
│       └── visualize.py
└── tox.ini           <- tox file with settings for running tox; see tox.readthedocs.io
  
```

1 **Nắm được Clean Code cơ bản**

Hiểu được khái niệm và tầm quan trọng của việc viết code sạch, rõ ràng và dễ bảo trì.

2 **Biết dùng PEP-8 để định dạng code Python**

Áp dụng các quy tắc định dạng chuẩn để code thống nhất và dễ đọc.

3 **Biết cách viết docstring và type hint**

Tài liệu hóa code và sử dụng gợi ý kiểu dữ liệu để code rõ ràng hơn.

4 **Sử dụng được công cụ cơ bản để kiểm tra và tự động format code**

Biết cách dùng flake8, black và tích hợp vào CI để đảm bảo chất lượng code.

Phần 1: Cơ bản về Clean Code và PEP-8

Phần 2: Viết Code Pythonic

Phần 3: Nguyên lý chung để viết code tốt

Phần 4: Nguyên tắc SOLID và Design Patterns (Nâng Cao)

The Zen of Python (Triết lý Python)

Tập hợp 19 nguyên tắc hướng dẫn thiết kế Python, được Tim Peters viết trong PEP 20. Có thể xem bằng cách gõ **import this** trong Python.

Beautiful is better than ugly

Cái đẹp tốt hơn cái xấu

Explicit is better than implicit

Rõ ràng tốt hơn ẩn ý

Simple is better than complex

Đơn giản tốt hơn phức tạp

Complex is better than complicated

Phức tạp vẫn tốt hơn rắc rối

Flat is better than nested

Phẳng tốt hơn lồng nhau

Sparse is better than dense

Thưa tốt hơn dày đặc

Ref: <https://peps.python.org/pep-0020/>

Pythonic nghĩa là tận dụng tối đa tính năng và đặc điểm riêng của Python để viết code. Code Pythonic dễ đọc, dễ hiểu, ngắn gọn như đọc tiếng Anh, đồng thời tuân thủ các quy ước và triết lý của Python.

Ví dụ 1: List Comprehensions

```
# Let's say we want to create a list of squares (12, 22, 32, etc.)

# Non-Pythonic way (the way you might do it in other languages)
print("Non-Pythonic way:")
squares = []
for i in range(1, 11): # Numbers 1 to 10
    squares.append(i * i)

print(f"Squares: {squares}")

# Pythonic way using list comprehension
print("\nPythonic way:")
squares_pythonic = [i * i for i in range(1, 11)]
print(f"Squares: {squares_pythonic}")
```

Non-Pythonic way:
Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Pythonic way:
Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Why this is better:

- One line instead of three
- Easier to read once you're familiar with the syntax
- Often faster than the loop approach
- This is the 'Python way' of creating lists from other sequences

Ví dụ 2: Dictionary Comprehensions

```
# Let's create a dictionary mapping numbers to their squares

# Non-Pythonic way
print("Non-Pythonic way:")
square_dict = {}
for i in range(1, 6):
    square_dict[i] = i * i

print(f"Dictionary of squares: {square_dict}")

# Pythonic way
print("\nPythonic way:")
square_dict_pythonic = {i: i * i for i in range(1, 6)}
print(f"Dictionary of squares: {square_dict_pythonic}")
```

Non-Pythonic way:
Dictionary of squares: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Pythonic way:
Dictionary of squares: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Why this is better:

- Creates the entire dictionary in one line
- Similar pattern to list comprehensions, making it easy to remember
- Very readable once you're familiar with the syntax

Python cung cấp cách truy cập mạnh mẽ vào các phần tử trong sequences (list, tuple, string) thông qua indexes và slicing. Cú pháp slicing **sequence[start:stop:step]** giúp thao tác với dữ liệu linh hoạt và Pythonic.

Cách dùng indexes và slices cơ bản

```
numbers = [1, 2, 3, 4, 5]

# Lấy phần tử theo vị trí
first = numbers[0]    # 1
last = numbers[-1]   # 5

# Lấy nhiều phần tử
first_three = numbers[:3] # [1, 2, 3]
last_three = numbers[-3:] # [3, 4, 5]
middle = numbers[1:4]    # [2, 3, 4]
```

Ứng dụng slices nâng cao

```
data = [10, 20, 30, 40, 50, 60]

# Lấy phần tử chẵn/lẻ (step)
even_indexes = data[::2] # [10, 30, 50]
odd_indexes = data[1::2] # [20, 40, 60]

# Đảo ngược dữ liệu
reverse = data[::-1]    # [60, 50, 40, 30, 20, 10]

# Áp dụng cho strings
message = "Python"
reverse = message[::-1] # "nohtyP"
substring = message[1:4] # "yth"
```

Các kỹ thuật Pythonic phổ biến

```
# Copy list (shallow copy)
original = [1, 2, 3]
copy_list = original[:]

# Thay thế một đoạn trong list
letters = list("abcdef")
letters[1:3] = ["X", "Y"] # ["a", "X", "Y", "d", "e", "f"]

# Xóa phần tử với slice
numbers = [1, 2, 3, 4, 5]
numbers[1:3] = []        # [1, 4, 5]
```

List, Dict, Set Comprehensions

List, Dict, Set Comprehensions giúp code gọn và nhanh hơn.

Không Pythonic

```
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
    squares.append(n**2)
```

Pythonic dùng list comprehension

```
numbers = [1, 2, 3, 4, 5]
squares = [n**2 for n in numbers]
```

Dict comprehension

```
names = ['Alice', 'Bob', 'Charlie']
lengths = {name: len(name) for name in names}
```

Set comprehension

```
numbers = [1, 2, 2, 3, 4, 4]
unique_squares = {n**2 for n in numbers}
# {16, 1, 4, 9}
```

Context Manager là cơ chế quản lý tài nguyên thông qua câu lệnh with, giúp tự động giải phóng tài nguyên (đóng file, đóng kết nối DB, giải phóng lock...) khi khối lệnh kết thúc, kể cả khi có lỗi xảy ra.

Không Pythonic

```
# Non-Pythonic way to read a file
print("Non-Pythonic way:")
file = open("example.txt", "r")
content = file.read()
file.close() # We have to remember to close the file!
print(f"File content:\n{content}")
```

Pythonic (tự động quản lý tài nguyên)

```
# Pythonic way using 'with' (context manager)
print("\nPythonic way:")
with open("example.txt", "r") as file:
    content = file.read()
    # No need to close the file - it happens automatically!
print(f"File content:\n{content}")
```

Ưu điểm

- ✓ Đảm bảo tài nguyên luôn được giải phóng đúng cách, tránh memory leak, giúp code ngắn gọn và an toàn hơn khi xử lý ngoại lệ.
- ✓ Context managers được dùng phổ biến cho: quản lý file, kết nối database, lock thread, benchmark thời gian, tạm thay đổi cấu hình, transaction database...

Tạo context manager riêng với decorator

Decorator `@contextmanager` từ module `contextlib` giúp chúng ta dễ dàng tạo context manager chỉ với một hàm generator đơn giản, thay vì phải định nghĩa class với phương thức `__enter__` và `__exit__`.

```
# Context manager
from contextlib import contextmanager

@contextmanager
def file_reader(filename):
    # Phần setup: mở file (được thực thi trước yield)
    f = open(filename, "r")
    try:
        # Trả về tài nguyên cho khối with
        yield f
    finally:
        # Phần cleanup: luôn đóng file (được thực thi sau khối with)
        f.close()

# Sử dụng context manager tự tạo
with file_reader("file.txt") as file:
    print(file.read())
```

Ứng dụng thực tế

- ✓ Tạm thay đổi cấu hình hệ thống và khôi phục sau khi xong
- ✓ Đo thời gian thực thi một đoạn code
- ✓ Quản lý kết nối database, Redis, hoặc APIs
- ✓ Tạo môi trường tạm thời để test

Trường hợp 1: So sánh với None

Sai (Non-Pythonic)

```
# Non-Pythonic
x = None
if x == None: # SAI! None là singleton
    print("x is None")
```

Đúng (Pythonic)

```
# Pythonic
x = None
if x is None:
    print("x is None")
elif x is not None:
    print("x is not None")
```

Giải thích

- None là singleton object, phải dùng is/is not để so sánh về mặt identity, không phải equality.
- Sử dụng == có thể dẫn đến những kết quả không mong muốn

Trường hợp 2: So sánh Boolean

Sai (Non-Pythonic)

```
# Non-Pythonic
ready = True
if (ready == True): # SAI, quá rườm rà!
    print("Ready")

valid = False
if (valid != False): # SAI!
    print("Valid")
```

Đúng (Pythonic)

```
# Pythonic
ready = True
if ready: # tận dụng truthiness
    print("Ready")

valid = False
if not valid: # ngắn gọn và rõ ràng
    print("Invalid")
```

Giải thích

- Boolean tự thân đã có giá trị truth, không cần so sánh thêm và không cần dấu ngoặc đơn khi không cần thiết. Giúp code đơn giản, rõ ràng và dễ đọc.

Trường hợp 3: Kiểm tra chuỗi/list/dict rỗng

Sai (Non-Pythonic)

```
# Non-Pythonic
name = "Python"
if len(name) > 0:  # SAI, không Pythonic!
    print(name)

items = []
if len(items) == 0:  # SAI, nên dùng: if not items
    print("Empty")
```

Đúng (Pythonic)

```
# Pythonic
name = "Python"
if name:  # Tận dụng truthiness
    print(name)

items = []
if not items:  # Kiểm tra list rỗng
    print("Empty list")
```

Giải thích

- Chuỗi/list/dict rỗng được đánh giá là False, không rỗng là True trong ngữ cảnh boolean. Tận dụng truthiness giúp code ngắn gọn và dễ đọc hơn.

Trường hợp 4: Kiểm tra trong collection

Sai (Non-Pythonic)

```
# Non-Pythonic
active_users = ["alice", "bob", "charlie"]
user = "bob"
found = False # SAI, quá phức tạp
for u in active_users:
    if u == user:
        found = True
        break
if found:
    print("User is active")
```

Đúng (Pythonic)

```
# Pythonic
active_users = ["alice", "bob", "charlie"]
user = "bob"
if user in active_users: # trực tiếp, hiệu quả
    print("User is active")
```

Giải thích

- Sử dụng 'in' để kiểm tra sự tồn tại của một phần tử trong collection là một cách hiệu quả và giúp code dễ đọc hơn. Dùng vòng lặp chậm hơn, làm code phức tạp và khó đọc hơn.

Trường hợp 5: Chaining comparison

Sai (Non-Pythonic)

```
# Non-Pythonic
value = 75
if value >= 0 and value <= 100:  # Dài dòng hơn
    print("Valid percentage")
```

Đúng (Pythonic)

```
# Pythonic
value = 75
if 0 <= value <= 100:  # Pythonic, rõ ràng
    print("Valid percentage")
```

Giải thích

- Python cho phép 'chain' các phép so sánh, khi kiểm tra một biến nằm trong một khoảng giá trị, giúp dễ đọc và hiểu hơn giống như toán học.

Properties và dấu underscore _

Sử dụng **@property** cho phép sử dụng method như thuộc tính, giúp code gọn, dễ kiểm soát truy cập. Tuân theo quy chuẩn sử dụng dấu underscore giúp tránh lỗi khi thiết kế class.

Tính diện tích hình chữ nhật với @property

```
# @property
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

rect = Rectangle(3, 4)
print(rect.area) # 12 (không cần rect.area())
```

Tính diện tích hình chữ nhật với @property

```
# Underscore
class Sample:
    def __init__(self):
        self.public = 1          # Biến public thông thường
        self._private = 2       # Quy ước private, vẫn truy cập được
        self.__mangled = 3      # Tự động chuyển thành _Sample__mangled

obj = Sample()
print(obj.public)              # 1
print(obj._private)            # 2 (cảnh báo: không nên truy cập theo quy ước)
# print(obj.__mangled)         # AttributeError
print(obj._Sample__mangled)    # 3 (truy cập được qua tên đã "mangled")
```

1 _var (Single Underscore)

Quy ước cho biến private hoặc "internal use" trong module, class. Không thực sự ngăn truy cập từ bên ngoài, nhưng là dấu hiệu "không nên sử dụng trực tiếp".

3 __var__ (Double Underscore hai bên)

Dành cho phương thức đặc biệt (magic/dunder methods) như `__init__`, `__str__`. Không nên tự tạo phương thức kiểu này trừ khi cần thiết.

2 __var (Double Underscore)

Name mangling - Python tự động đổi tên thành `_ClassName__var` để tránh xung đột khi kế thừa. Giúp thuộc tính không bị ghi đè bởi lớp con một cách vô tình.

4 _ (Một dấu gạch dưới)

Thường dùng làm biến tạm không quan trọng hoặc kết quả gần nhất trong Python interpreter.

Bài tập thực hành: Chuyển code thành Pythonic

Hãy chuyển đổi các đoạn code sau đây thành phong cách Pythonic để code ngắn gọn, hiệu quả và dễ đọc hơn.

Code ban đầu

```
# Exercises
nums = [1, 2, 3, 4, 5]
result = []
for num in nums:
    if num % 2 == 0:
        result.append(num ** 2)

# Ví dụ 2: Tạo từ điển
names = ['Anna', 'Bob', 'Charlie']
ages = [25, 30, 35]
info = {}
for i in range(len(names)):
    info[names[i]] = ages[i]

# Ví dụ 3: Tìm số lớn nhất
numbers = [10, 5, 8, 20, 3]
max_num = numbers[0]
for n in numbers[1:]:
    if n > max_num:
        max_num = n
```

Code Pythonic

Đáp án

1 Hiểu thế nào là Pythonic code

Viết code theo phong cách Python, tận dụng các đặc trưng của ngôn ngữ.

2 Biết sử dụng slicing, comprehensions, context managers

Áp dụng các kỹ thuật đặc trưng của Python để code ngắn gọn, hiệu quả.

3 Biết cách dùng decorators properties, dấu underscore

Sử dụng các tính năng nâng cao để tạo code linh hoạt, dễ bảo trì.

Phần 1: Cơ bản về Clean Code và PEP-8

Phần 2: Viết Code Pythonic

Phần 3: Nguyên lý chung để viết code tốt

Phần 4: Nguyên tắc SOLID và Design Patterns (Nâng Cao)

Nguyên lý chung để viết code tốt hơn dựa trên các nền tảng cốt lõi như **DRY, YAGNI, KISS, lập trình phòng thủ và phân chia trách nhiệm**. Những nguyên lý này giúp tạo code bền vững và dễ bảo trì.



DRY (Don't Repeat Yourself)

Tránh lặp lại code, mỗi kiến thức nên được định nghĩa một lần duy nhất trong hệ thống



YAGNI (You Aren't Gonna Need It)

Không thêm tính năng cho đến khi thực sự cần thiết, tránh phức tạp hóa không cần thiết



KISS (Keep It Simple, Stupid)

Luôn ưu tiên các giải pháp đơn giản, tránh những thiết kế phức tạp khó hiểu



Defensive Programming

Lập trình phòng thủ, luôn đề phòng những đầu vào không mong muốn và lỗi tiềm ẩn



Separation of Concerns

Phân chia trách nhiệm rõ ràng giữa các thành phần, module trong hệ thống

Nguyên tắc DRY (Don't Repeat Yourself)

Nguyên tắc DRY được phát triển bởi Andy Hunt và Dave Thomas: “Tránh lặp lại code, mỗi phần kiến thức trong hệ thống phải có một biểu diễn duy nhất, rõ ràng, và có thẩm quyền”

Vi phạm DRY

```
def print_morning_greeting(name):  
    print(f"Good morning, {name}!")  
    print("I hope you have a wonderful day.")  
    print("Don't forget to drink water and take breaks.")  
    print("-----")  
  
def print_evening_greeting(name):  
    print(f"Good evening, {name}!")  
    print("I hope you had a wonderful day.")  
    print("Don't forget to drink water and take breaks.")  
    print("-----")
```

Tuân thủ DRY

```
def print_greeting(name, time_of_day):  
    print(f"Good {time_of_day}, {name}!")  
    print(f"I hope you {'have' if time_of_day == 'morning' else 'had'} a wonderful day.")  
    print("Don't forget to drink water and take breaks.")  
    print("-----")  
  
print_greeting("Alice", "morning")  
print_greeting("Bob", "evening")
```

Lợi ích khi áp dụng DRY

- ✓ Giảm lỗi khi cần thay đổi logic
- ✓ Code ngắn gọn, dễ bảo trì hơn
- ✓ Tăng khả năng tái sử dụng code

Cách áp dụng

- ✓ Tách các đoạn code lặp lại thành functions, classes, hoặc modules riêng để tái sử dụng trong toàn bộ dự án.

Nguyên tắc YAGNI (You Aren't Gonna Need It)

Nguyên tắc YAGNI (Bạn sẽ không cần đến nó) được phát triển bởi Ron Jeffries. Nguyên tắc này khuyên lập trình viên **không nên thêm chức năng cho đến khi thực sự cần thiết**.

Vi phạm YAGNI

```
class SuperCalculator:
    """A calculator with many features we might not need."""

    def __init__(self):
        self.history = []
        self.memory = 0
        self.scientific_mode = False
        self.conversion_rates = {
            "USD_to_EUR": 0.85,
            "EUR_to_USD": 1.18,
            "USD_to_GBP": 0.73,
            "GBP_to_USD": 1.37,
            # ... dozens more conversion rates
        }

    def add(self, a, b):
        """Add two numbers."""
        result = a + b
        self.history.append(f"{a} + {b} = {result}")
        return result

    def subtract(self, a, b):
        """Subtract b from a."""
        result = a - b
        self.history.append(f"{a} - {b} = {result}")
        return result

    def toggle_scientific_mode(self):
        """Toggle scientific mode."""
        self.scientific_mode = not self.scientific_mode
        return self.scientific_mode

    # ...
```

Tuân thủ YAGNI

```
class SimpleCalculator:
    """A calculator that starts with just what we need."""

    def add(self, a, b):
        """Add two numbers."""
        return a + b

    def subtract(self, a, b):
        """Subtract b from a."""
        return a - b

    # We can add more methods later when we actually need them

# Using the simple calculator
simple_calc = SimpleCalculator()
result = simple_calc.add(5, 3)
print(f"5 + 3 = {result}")
```

Lợi ích khi áp dụng YAGNI

- ✓ Tránh lãng phí thời gian phát triển các tính năng không cần thiết
- ✓ Giảm thiểu technical debt
- ✓ Giữ code đơn giản, dễ bảo trì hơn

Khi nào nên áp dụng

- ✓ Áp dụng khi bạn đang muốn thêm một tính năng "phòng khi cần" mà chưa có yêu cầu cụ thể. Hãy trì hoãn việc thêm tính năng cho đến khi có nhu cầu thực tế.

Nguyên tắc KISS (Keep It Simple, Stupid)

KISS ưu tiên giải pháp **đơn giản** và **dễ hiểu nhất**. Tránh các giải pháp phức tạp khi không cần thiết.

Quá phức tạp

```
def is_even_complex(number):  
    """Determine if a number is even using a complex algorithm."""  
    # Convert to binary string  
    binary = bin(number)[2:] # Remove '0b' prefix  
  
    # Check last digit of binary (1 for odd, 0 for even)  
    last_digit = binary[-1]  
  
    # Convert to integer and check if it's zero  
    return int(last_digit) == 0
```

Rõ ràng, đơn giản hơn

```
def is_even_simple(number):  
    """Determine if a number is even using the modulo operator."""  
    return number % 2 == 0
```

Lợi ích khi áp dụng KISS

- ✓ Code đơn giản dễ dàng đọc, debug, bảo trì và mở rộng. Giảm thiểu bug và tăng hiệu suất làm việc nhóm.

Khi nào nên áp dụng

- ✓ Python vốn được thiết kế theo triết lý "Đơn giản hơn tốt hơn phức tạp" - tận dụng điều này để viết code rõ ràng, ngắn gọn.

Lập trình phòng thủ là kỹ thuật viết code **luôn giả định rằng sẽ có lỗi xảy ra**. Bao gồm kiểm tra đầu vào, xử lý ngoại lệ và kiểm tra điều kiện biên để tránh lỗi không mong muốn.

Thiếu kiểm tra đầu vào

```
# Non-defensive programming
def divide(a, b):
    return a / b

print(divide(5, 0)) # lỗi ZeroDivisionError

def get_element(lst, index):
    return lst[index] # Lỗi nếu index ngoài range
```

Kiểm tra rõ ràng

```
# Defensive programming
def divide(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Tham số phải là số.")
    if b == 0:
        raise ValueError("Không thể chia cho 0.")
    return a / b

def get_element(lst, index):
    if not isinstance(lst, list):
        raise TypeError("Tham số phải là list.")
    if index < 0 or index >= len(lst):
        return None # Hoặc giá trị mặc định thay vì lỗi
    return lst[index]
```

Nguyên tắc cơ bản

- ✓ Luôn kiểm tra đầu vào, xác thực dữ liệu từ người dùng, file, API và các nguồn bên ngoài. Không bao giờ tin tưởng input từ bất kỳ nguồn nào.

Lợi ích

- ✓ Lập trình phòng thủ giúp tạo ra code bền vững, có khả năng ứng phó với các tình huống không lường trước và dễ dàng debug khi có vấn đề. Trong Python, hãy kết hợp kiểm tra kiểu dữ liệu, assert, và try-except để xây dựng code an toàn.

Luôn bắt lỗi cụ thể thay vì chung chung (Exception). Không bao giờ "nuốt lỗi" mà không xử lý hoặc ghi log.

Bắt lỗi quá chung chung

```
# Bad example
try:
    with open("file.txt") as f:
        data = f.read()
        parsed = json.loads(data)
        result = 10 / parsed["value"]
except Exception:
    pass # lỗi bị bỏ qua hoàn toàn!
```

Bắt lỗi cụ thể rõ ràng hơn

```
try:
    with open("file.txt") as f:
        data = f.read()
        parsed = json.loads(data)
        result = 10 / parsed["value"]
except FileNotFoundError:
    logger.error("Không tìm thấy file.")
    raise ConfigError("File cấu hình không tồn tại")
except json.JSONDecodeError:
    logger.error("File không đúng định dạng JSON.")
except KeyError:
    logger.error("Thiếu trường 'value' trong dữ liệu.")
except ZeroDivisionError:
    logger.error("Giá trị 'value' không thể bằng 0.")
```

- ✓ Xử lý lỗi tốt không chỉ là bắt lỗi mà còn là truyền thông tin lỗi đúng cách.
- ✓ Sử dụng logging thay vì print và tạo custom exceptions khi cần thiết để làm rõ ngữ cảnh lỗi.

Phân chia trách nhiệm (Separation of Concerns)

Phân chia code thành các module, class hoặc hàm riêng biệt, mỗi phần chỉ **đảm nhiệm một chức năng cụ thể**.
Giúp dễ bảo trì, dễ kiểm thử và tăng khả năng tái sử dụng.

Không chia trách nhiệm rõ ràng

```
# Bad example
def process_user_data(user):
    # Xác thực dữ liệu
    if not user.email or '@' not in user.email:
        return False

    # Lưu vào database
    db.connect()
    db.execute("INSERT INTO users VALUES (?)",
               (user.to_dict(),))
    db.commit()

    # Gửi email
    smtp = SMTP('smtp.example.com')
    smtp.login('user', 'password')
    smtp.send_mail(
        to=user.email,
        subject="Chào mừng bạn!",
        body="Cảm ơn đã đăng ký...")

    # Tạo báo cáo
    report = {"user": user.id, "time": time.now()}
    with open('report.json', 'w') as f:
        json.dump(report, f)
```

Tách nhiệm vụ rõ ràng hơn

```
# Good example
def process_user_data(user):
    if validate_user(user):
        save_user(user)
        notify_user(user)
        log_activity(user)

def validate_user(user):
    return bool(user.email and '@' in user.email)

def save_user(user):
    repository = UserRepository()
    repository.add(user)

def notify_user(user):
    email_service = EmailService()
    email_service.send_welcome(user.email)

def log_activity(user):
    reporter = ActivityReporter()
    reporter.create_registration_report(user.id)
```

Logging cung cấp tính năng ghi nhật ký chuyên nghiệp với nhiều cấp độ và tùy chọn cấu hình, trong khi Print là giải pháp đơn giản cho gỡ lỗi nhanh nhưng có nhiều hạn chế trong môi trường sản phẩm.

Print	Logging
<ul style="list-style-type: none">• Đơn giản, dễ sử dụng	<ul style="list-style-type: none">• Cấu hình linh hoạt
<ul style="list-style-type: none">• Khó kiểm soát đầu ra	<ul style="list-style-type: none">• Nhiều cấp độ log (DEBUG, INFO, WARNING...)
<ul style="list-style-type: none">• Không phân loại mức độ nghiêm trọng	<ul style="list-style-type: none">• Dễ dàng bật/tắt theo cấu hình
<ul style="list-style-type: none">• Khó tắt khi triển khai	<ul style="list-style-type: none">• Tự động thêm thời gian, file, dòng code
<ul style="list-style-type: none">• Không lưu lại thông tin như thời gian	<ul style="list-style-type: none">• Có thể điều hướng log đến file, email...

Ví dụ sử dụng logging cơ bản

```
# Chỉ log những mức độ cảnh báo (WARNING) trở lên
logging.basicConfig( level=logging.WARNING,
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
                    filename='app.log')

logging.debug("Thông tin chi tiết cho debug")
logging.info("Thông tin thông thường")
logging.warning("Cảnh báo: có vấn đề nhỏ xảy ra")
logging.error("Lỗi: có vấn đề nghiêm trọng")
logging.critical("NGUY HIỂM: chương trình có thể crash")
2025-05-21 08:55:22,342 - root - WARNING - Cảnh báo: có vấn đề nhỏ xảy ra
2025-05-21 08:55:22,342 - root - ERROR - Lỗi: có vấn đề nghiêm trọng
2025-05-21 08:55:22,342 - root - CRITICAL - NGUY HIỂM: chương trình có thể crash
2025-05-21 08:55:22,342 - root - ERROR - Có lỗi xảy ra: division by zero
```

Tùy chỉnh tắt log tạm thời

```
# Tắt log tạm thời
logging.info("Dòng này sẽ hiện ra")
logging.disable(logging.CRITICAL)
logging.info("Dòng này sẽ không hiện ra")
logging.disable(logging.NOTSET) # Bật lại logging
logging.info("Log đã bật lại")
```

```
2025-05-21 08:56:02,487 - root - INFO - Dòng này sẽ hiện ra
2025-05-21 08:56:02,487 - root - INFO - Log đã bật lại
```

Ví dụ ghi lại thông tin lỗi

```
# Sử dụng exception trong logging (ghi lại thông tin lỗi)
try:
    1 / 0
except Exception as e:
    logging.error("Có lỗi xảy ra: %s", e)
```

```
2025-05-21 08:56:02,487 - root - ERROR - Có lỗi xảy ra: division by zero
```


Nhiệm vụ: Sử dụng nguyên tắc DRY và KISS để refactor đoạn code sau đây

Code ban đầu (Vi phạm DRY)

```
# Vi phạm DRY: Lặp lại logic in lời chào  
# Khó bảo trì khi thêm ngôn ngữ mới  
# Cần sửa nhiều nơi khi thay đổi cách hiển thị  
  
def greeting_vn(name):  
    print(f"Xin chào, {name}")  
  
def greeting_en(name):  
    print(f"Hello, {name}")  
  
def greeting_fr(name):  
    print(f"Bonjour, {name}")
```

Code đã refactor (Theo DRY và KISS)

Đáp án

1 **Nắm vững các nguyên lý cơ bản**

Hiểu và áp dụng các nguyên tắc DRY, YAGNI, KISS trong code.

2 **Hiểu rõ lập trình phòng thủ và xử lý lỗi**

Biết cách kiểm tra đầu vào và xử lý lỗi một cách cụ thể, rõ ràng.

3 **Biết cách tổ chức code theo trách nhiệm**

Phân chia code thành các module, hàm có trách nhiệm rõ ràng.

4 **Giảm sự phụ thuộc giữa các module**

Áp dụng nguyên tắc low coupling, high cohesion để code dễ bảo trì và mở rộng.

Phần 1: Cơ bản về Clean Code và PEP-8

Phần 2: Viết Code Pythonic

Phần 3: Nguyên lý chung để viết code tốt

Phần 4: Nguyên tắc SOLID và Design Patterns (Nâng Cao)

1 Single Responsibility

Mỗi lớp chỉ nên có một lý do để thay đổi. Tập trung vào một nhiệm vụ duy nhất.

2 Open/Closed

Mở để mở rộng, đóng để sửa đổi. Code nên dễ mở rộng mà không cần sửa đổi.

3 Liskov Substitution

Các lớp con phải thay thế được lớp cha. Đảm bảo tính nhất quán.

4 Interface Segregation

Nhiều interface nhỏ tốt hơn một interface lớn. Tránh phụ thuộc không cần thiết.

5 Dependency Inversion

Phụ thuộc vào abstraction, không phụ thuộc vào cụ thể. Giảm sự ràng buộc.

✓ **Tính Linh Hoạt**

Python dynamic typing cho phép dễ dàng thay đổi hành vi object trong runtime. Điều này giúp thiết kế các interface linh hoạt, đặc biệt hữu ích cho Open/Closed Principle khi mở rộng tính năng.

</> **Duck Typing**

Python thực hiện "nếu nó đi như vịt và kêu như vịt, thì nó là vịt". Các objects chỉ cần hiện thực các phương thức tương thích, không cần kế thừa, hỗ trợ tốt cho Liskov Substitution và Interface Segregation.

⚙️ **Abstractions**

Module abc (Abstract Base Classes) cung cấp `@abstractmethod` decorator và ABC class để định nghĩa interface thuần khiết. Điều này đảm bảo các lớp con phải triển khai đúng các phương thức được yêu cầu, tăng cường Dependency Inversion.

🔧 **Composition**

Python khuyến khích "composition over inheritance" thông qua mixins và dependency injection. Thay vì kế thừa nhiều tầng, việc kết hợp các đối tượng nhỏ giúp đạt Single Responsibility và giảm phụ thuộc giữa các module.



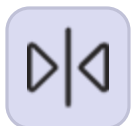
Một Lớp, Một Nhiệm Vụ

Mỗi class chỉ nên có một lý do để thay đổi. Ví dụ: class DatabaseConnector chỉ chịu trách nhiệm kết nối database, không nên xử lý logic nghiệp vụ.



Phân Tách Rõ Ràng

Chia nhỏ các lớp lớn thành các module nhỏ hơn. Thay vì FileProcessor xử lý đọc, xác thực và phân tích dữ liệu, tách thành FileReader, FileValidator và FileParser.



Dễ Bảo Trì Và Test

Khi mỗi class chỉ làm một việc, code dễ hiểu và dễ sửa. Các đơn vị nhỏ cũng dễ kiểm thử hơn, tăng độ tin cậy của phần mềm.

Ví dụ về Single Responsibility Principle

Nguyên tắc trách nhiệm đơn lẻ thể hiện rõ qua việc tách biệt các chức năng trong code

Vi phạm SRP

```
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
        self.grades = []

    def add_grade(self, course, grade):
        """Add a grade for a course."""
        self.grades.append({"course": course, "grade": grade})

    def calculate_gpa(self):
        """Calculate the student's GPA."""
        if not self.grades:
            return 0

        total = sum(item["grade"] for item in self.grades)
        return total / len(self.grades)

    def save_to_database(self):
        """Save student data to the database."""
        print(f"Saving student {self.name} with ID {self.student_id} to database...")
        # Database code would go here

    def print_report_card(self):
        """Print a report card for the student."""
        print(f"\nReport Card for {self.name} (ID: {self.student_id})")
        print("-" * 30)

        for item in self.grades:
            print(f"{item['course']}: {item['grade']}")

        print("-" * 30)
        print(f"GPA: {self.calculate_gpa():.2f}")
```

Tuân thủ SRP

```
class StudentData:
    """Responsible only for storing student data."""
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
        self.grades = []

    def add_grade(self, course, grade):
        """Add a grade for a course."""
        self.grades.append({"course": course, "grade": grade})

class GradeCalculator:
    """Responsible only for calculating grades."""
    @staticmethod
    def calculate_gpa(student):
        """Calculate a student's GPA."""
        if not student.grades:
            return 0

        total = sum(item["grade"] for item in student.grades)
        return total / len(student.grades)

class StudentRepository:
    """Responsible only for saving and loading student data."""
    @staticmethod
    def save(student):
        """Save a student to the database."""
        print(f"Saving student {student.name} with ID {student.student_id} to database...")
        # Database code would go here

    @staticmethod
    def load(student_id):
        """Load a student from the database."""
        print(f"Loading student with ID {student_id} from database...")
        # Database code would go here
        return None # Placeholder
```




Mở Cho Việc Mở Rộng

Code nên được thiết kế sao cho có thể dễ dàng thêm chức năng mới mà không làm thay đổi code hiện có. Ví dụ: thêm loại thanh toán mới vào hệ thống mà không phải sửa code xử lý thanh toán.



Đóng Cho Việc Sửa Đổi

Một khi class đã được kiểm thử và triển khai, nên tránh sửa đổi nội dung bên trong nó. Thay vào đó, hãy mở rộng thông qua kế thừa hoặc composition. Giúp tránh gây ra lỗi ở code đang hoạt động tốt.



Sử Dụng Abstraction

Trong Python, tạo các abstract base class (ABC) hoặc protocol để định nghĩa interface. Ví dụ: định nghĩa PaymentProcessor ABC, rồi thêm các lớp con như CreditCardProcessor, PayPalProcessor mà không cần sửa code xử lý thanh toán.



Mở Rộng Bằng Kế Thừa

Thay vì sửa đổi class Shape để thêm hình mới, hãy tạo subclass Triangle, Square kế thừa từ Shape. Strategy pattern và Plugin architecture là hai kỹ thuật giúp áp dụng nguyên lý này hiệu quả trong Python.

Ví Dụ Về Open/Closed Principle

Open/Closed Principle (OCP) - Mở cho việc mở rộng, đóng cho việc sửa đổi.

Vi phạm OCP

```
class PaymentProcessor:
    def process_payment(self, payment_type, amount):
        if payment_type == "credit_card":
            print(f"Xử lý thanh toán thẻ tín dụng: {amount}")
        elif payment_type == "paypal":
            print(f"Xử lý thanh toán PayPal: {amount}")
        # Cần sửa đổi code mỗi khi thêm phương thức mới
        # elif payment_type == "momo":
        #     print(f"Xử lý thanh toán MoMo: {amount}")
```

Tuân thủ OCP

```
# Following OCP
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process(self, amount):
        pass

class CreditCardProcessor(PaymentProcessor):
    def process(self, amount):
        print(f"Xử lý thanh toán thẻ tín dụng: {amount}")

class PayPalProcessor(PaymentProcessor):
    def process(self, amount):
        print(f"Xử lý thanh toán PayPal: {amount}")

# Dễ dàng mở rộng thêm phương thức mới
class MoMoProcessor(PaymentProcessor):
    def process(self, amount):
        print(f"Xử lý thanh toán MoMo: {amount}")
```

- ✓ Với thiết kế tuân thủ OCP, chúng ta có thể thêm phương thức thanh toán mới mà không cần sửa đổi code hiện có.
- ✓ Mỗi phương thức thanh toán được triển khai trong class riêng, tăng tính mở rộng và bảo trì.



Thay Thế Được

Đối tượng của lớp con phải hoạt động đúng khi được sử dụng thay cho đối tượng lớp cha, đảm bảo hàm `client_code` (Parent p) hoạt động đúng với tham số kiểu Child.



Hành Vi Nhất Quán

Lớp con không được thay đổi hành vi được định nghĩa trong interface/lớp cha. Ví dụ: nếu Bird có phương thức `fly()`, thì Penguin không nên triển khai `fly()` bằng cách ném ra exception.



Tránh Vi Phạm

Không thêm precondition mạnh hơn, postcondition yếu hơn, hoặc ném ra exception mới trong lớp con. Ví dụ: nếu ParentClass chấp nhận tham số có thể là None, ChildClass cũng phải chấp nhận.



Ví Dụ Cổ Điển: Rectangle-Square

Square không thể là lớp con của Rectangle vì `set_width()` và `set_height()` trong Square phải đồng thời thay đổi cả hai kích thước, khiến cho code mong đợi `rectangle.area == width*height` bị phá vỡ sau khi gọi `set_width()`.

Ví Dụ: Liskov Substitution Principle

Nguyên tắc thay thế Liskov: Các đối tượng của lớp con phải có thể thay thế đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình.

Vi phạm LSP

```
# Violated LSP
class Bird:
    def fly(self):
        print("Bird can fly")

class Penguin(Bird):
    def fly(self):
        # Vi phạm LSP - ném exception
        raise Exception("Penguin không thể bay!")

# Client code bị ảnh hưởng
def let_bird_fly(bird):
    bird.fly() # Có thể gây lỗi nếu là Penguin
```

Tuân thủ LSP

```
# Following LSP
class FlyingCreature:
    def fly(self):
        print("Flying creature can fly")

class Bird(FlyingCreature):
    pass

class Penguin: # Không kế thừa FlyingCreature
    def swim(self):
        print("Penguin can swim")

# Client code an toàn
def let_fly(creature: FlyingCreature):
    creature.fly() # Luôn hoạt động đúng
```

- ✓ Thiết kế đúng tách biệt các khả năng thành các interface riêng, đảm bảo lớp con có thể thay thế lớp cha mà không làm thay đổi hành vi mong đợi của chương trình.



Tách Nhỏ Interface

Nhiều interface chuyên biệt tốt hơn một interface lớn. Ví dụ: tách `PrinterInterface` thành các interface nhỏ hơn như `Scanner`, `Printer`, `Fax` thay vì một `AllInOnePrinter` interface.



Tránh "Fat Interface"

Client không nên bị buộc triển khai phương thức không cần. Ví dụ: `class SmartPhone` không cần triển khai tất cả phương thức của một `Device` interface phức tạp (`connect_wifi()`, `make_call()`, `take_photo()`).



Trong Python

Dùng ABC module (từ `abc` import `ABC`, `abstractmethod`) hoặc `typing.Protocol` (Python 3.8+) để định nghĩa interface rõ ràng. Duck typing cho phép các lớp chỉ triển khai những phương thức cần thiết mà không cần kế thừa chính thức.



Lợi Ích

Code linh hoạt hơn với khả năng swap implementation. Dễ unit test với mock objects. Giảm coupling và side-effects khi thay đổi code, đặc biệt trong hệ thống lớn có nhiều module phụ thuộc.

Ví dụ về Interface Segregation Principle

Nhiều interface chuyên biệt tốt hơn một interface lớn. Client không nên bị buộc phải phụ thuộc vào các phương thức mà họ không sử dụng.

Ví dụ không tốt: "Fat Interface"

```
class AllDeviceActions:
    def print(self): pass
    def scan(self): pass
    def fax(self): pass
    def copy(self): pass

class BasicPrinter(AllDeviceActions):
    def print(self):
        print("Printing...")
    # Phải cài đặt cả những phương thức không cần thiết
    def scan(self): raise NotImplementedError()
    def fax(self): raise NotImplementedError()
    def copy(self): raise NotImplementedError()
```

Ví dụ tốt: Tách nhỏ interface

```
# Following ISP
class Printer:
    def print(self): pass

class Scanner:
    def scan(self): pass

class Fax:
    def fax(self): pass

class BasicPrinter(Printer):
    def print(self):
        print("Printing...")

class MultiFunctionDevice(Printer, Scanner, Fax):
    def print(self): print("Printing...")
    def scan(self): print("Scanning...")
    def fax(self): print("Faxing...")
```

Lợi ích

- ✓ Giảm sự phụ thuộc không cần thiết
- ✓ Dễ dàng mở rộng với các tính năng mới
- ✓ Tăng khả năng tái sử dụng code
- ✓ Dễ dàng test và bảo trì
- ✓ Tuân thủ nguyên tắc Single Responsibility Principle



High-level Module

Module cấp cao định nghĩa logic nghiệp vụ và không nên phụ thuộc trực tiếp vào chi tiết triển khai cụ thể. Ví dụ: service không nên phụ thuộc trực tiếp vào database, mà thông qua repository interface.



Abstraction

Cả module cấp cao và cấp thấp đều phụ thuộc vào abstraction (interfaces/protocols). Trong Python, ta có thể dùng abstract base classes (ABC), Protocol từ typing, hoặc đơn giản là duck typing.



Low-level Module

Module cấp thấp (infrastructure, database, external APIs) triển khai các abstractions được định nghĩa bởi module cấp cao, đảm bảo đáp ứng đúng "hợp đồng" interface.

Dependency Inversion Principle - Ví dụ minh họa

DIP khuyến nghị các module cấp cao không nên phụ thuộc trực tiếp vào module cấp thấp, mà cả hai nên phụ thuộc vào abstraction.

Code vi phạm DIP

```
class MySQLDatabase:
    def save(self, data):
        print(f"Lưu {data} vào MySQL")

class UserService:
    def __init__(self):
        # Phụ thuộc trực tiếp vào MySQL
        self.database = MySQLDatabase()

    def create_user(self, user_data):
        # Logic xử lý user
        self.database.save(user_data)

# Khi muốn đổi sang MongoDB sẽ phải
# thay đổi code của UserService
```

Code tuân thủ DIP

```
from abc import ABC, abstractmethod

class Database(ABC):
    @abstractmethod
    def save(self, data): pass

class MySQLDatabase(Database):
    def save(self, data):
        print(f"Lưu {data} vào MySQL")

class MongoDBDatabase(Database):
    def save(self, data):
        print(f"Lưu {data} vào MongoDB")

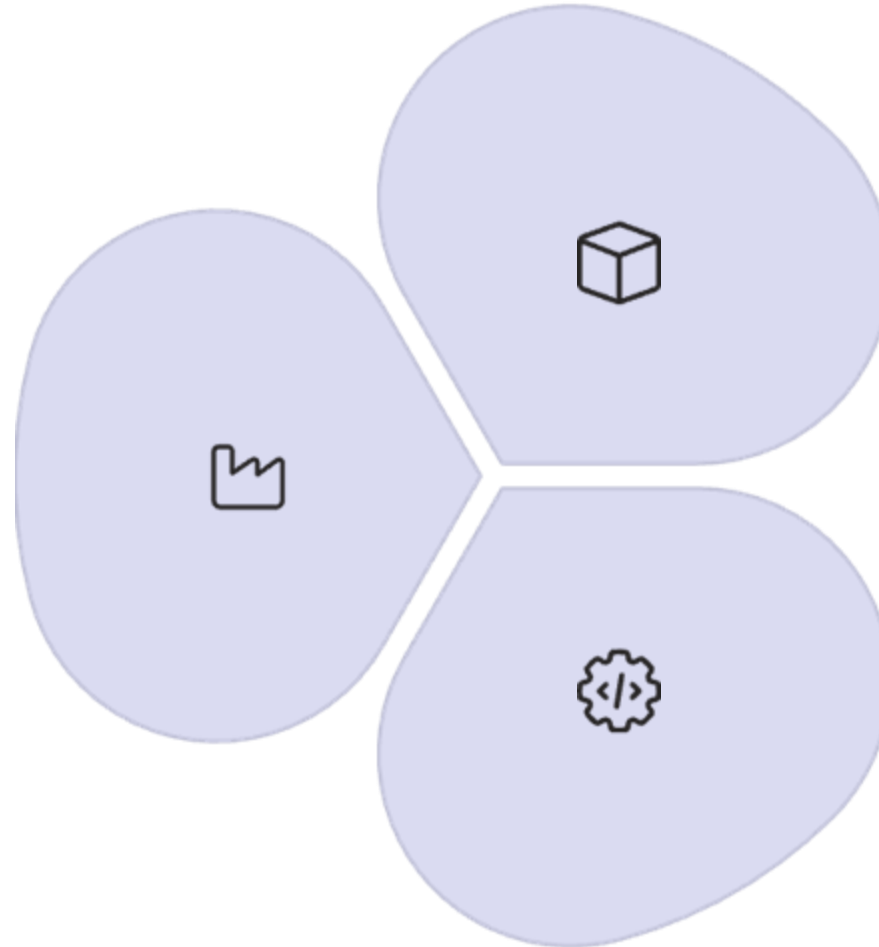
class UserService:
    def __init__(self, database: Database):
        # Phụ thuộc vào abstraction
        self.database = database

    def create_user(self, user_data):
        # Logic xử lý user
        self.database.save(user_data)

# Dễ dàng chuyển đổi database mà không
# cần sửa đổi UserService
service = UserService(MongoDatabase())
```

Creational Patterns

Tạo đối tượng theo cách linh hoạt. Factory và Singleton là phổ biến nhất.



Structural Patterns

Xác định mối quan hệ giữa các đối tượng. Adapter và Decorator rất hữu ích.

Behavioral Patterns

Xác định cách giao tiếp giữa các đối tượng. Command và Template Method thường dùng.

Factory Pattern

Tạo đối tượng mà không cần biết lớp con cụ thể:

- Ẩn logic phức tạp
- Interface chung cho đối tượng liên quan
- Dễ mở rộng mà không sửa client code

```
class ButtonFactory:
    def create_button(self, button_type):
        if button_type == "round":
            return RoundButton()
        elif button_type == "square":
            return SquareButton()
        elif button_type == "toggle":
            return ToggleButton()
        # Dễ dàng thêm button mới

# Sử dụng trong thực tế
factory = ButtonFactory()
button = factory.create_button("round")
button.render() # Client không quan tâm đến chi tiết triển khai
```

Singleton Pattern

Chỉ tạo duy nhất một instance thường dùng cho logging, database.

- Truy cập toàn cục
- Kiểm soát tài nguyên chia sẻ trong hệ thống
- Tiết kiệm bộ nhớ

```
class DatabaseConnection:
    _instance = None
    _is_initialized = False

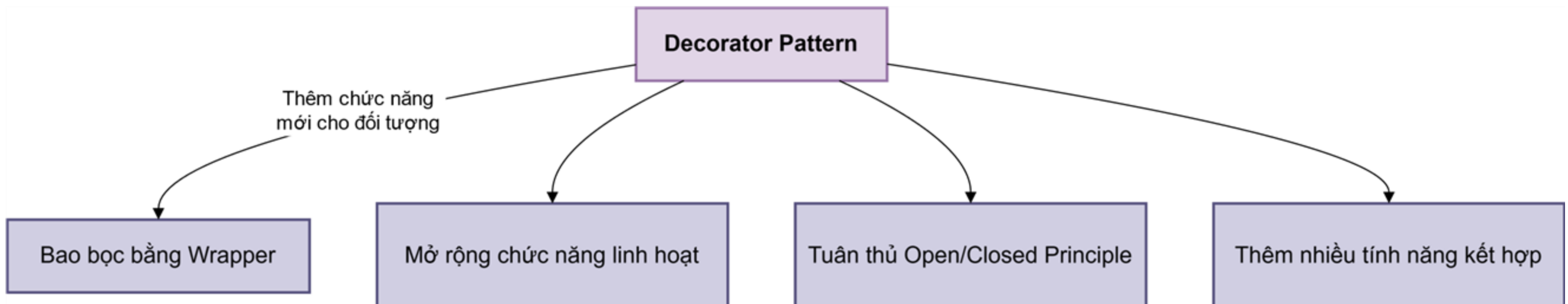
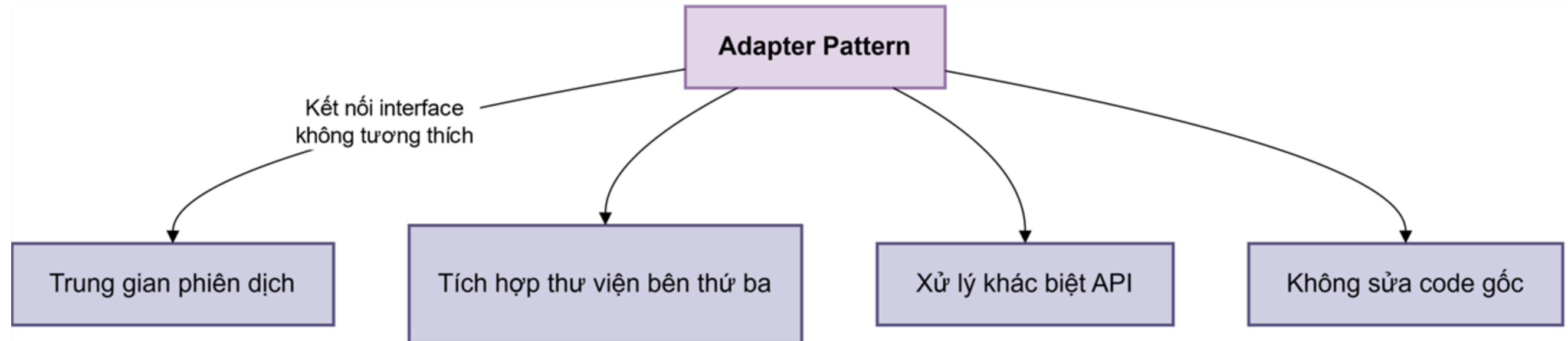
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        # Đảm bảo chỉ khởi tạo một lần
        if not self._is_initialized:
            self.host = "localhost"
            self.connect_db()
            self._is_initialized = True

    def connect_db(self):
        print("Connecting to database once...")

# Luôn trả về cùng một instance
conn1 = DatabaseConnection()
conn2 = DatabaseConnection()
# conn1 is conn2 -> True
```

Structural Patterns: Adapter & Decorator



Adapter Pattern

Cho phép các interface không tương thích làm việc với nhau.

```
class LegacyAPI:
    def old_method(self, data):
        return f"Legacy output: {data}"

class NewInterface:
    def new_method(self, info):
        pass

class Adapter(NewInterface):
    def __init__(self, legacy_obj):
        self.legacy = legacy_obj

    def new_method(self, info):
        # Chuyển đổi từ new_method sang old_method
        return self.legacy.old_method(info)

# Sử dụng adapter
legacy = LegacyAPI()
adapter = Adapter(legacy)
result = adapter.new_method("test data")
# Kết quả: "Legacy output: test data"
```

Decorator Pattern

Thêm chức năng mới cho đối tượng mà không sửa đổi cấu trúc.

```
class Component:
    def operation(self):
        return "Basic operation"

class ConcreteComponent(Component):
    def operation(self):
        return "Concrete operation"

class Decorator(Component):
    def __init__(self, component):
        self._component = component

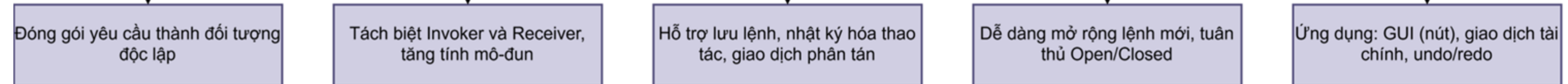
    def operation(self):
        return self._component.operation()

class LoggingDecorator(Decorator):
    def operation(self):
        result = self._component.operation()
        print(f"LOG: Called operation")
        return result

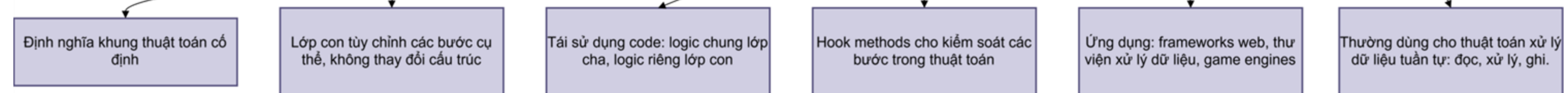
# Tạo component và thêm decorator
component = ConcreteComponent()
decorated = LoggingDecorator(component)
result = decorated.operation()
# In log và trả về "Concrete operation"
```

Behavioral Patterns: Command & Template Method

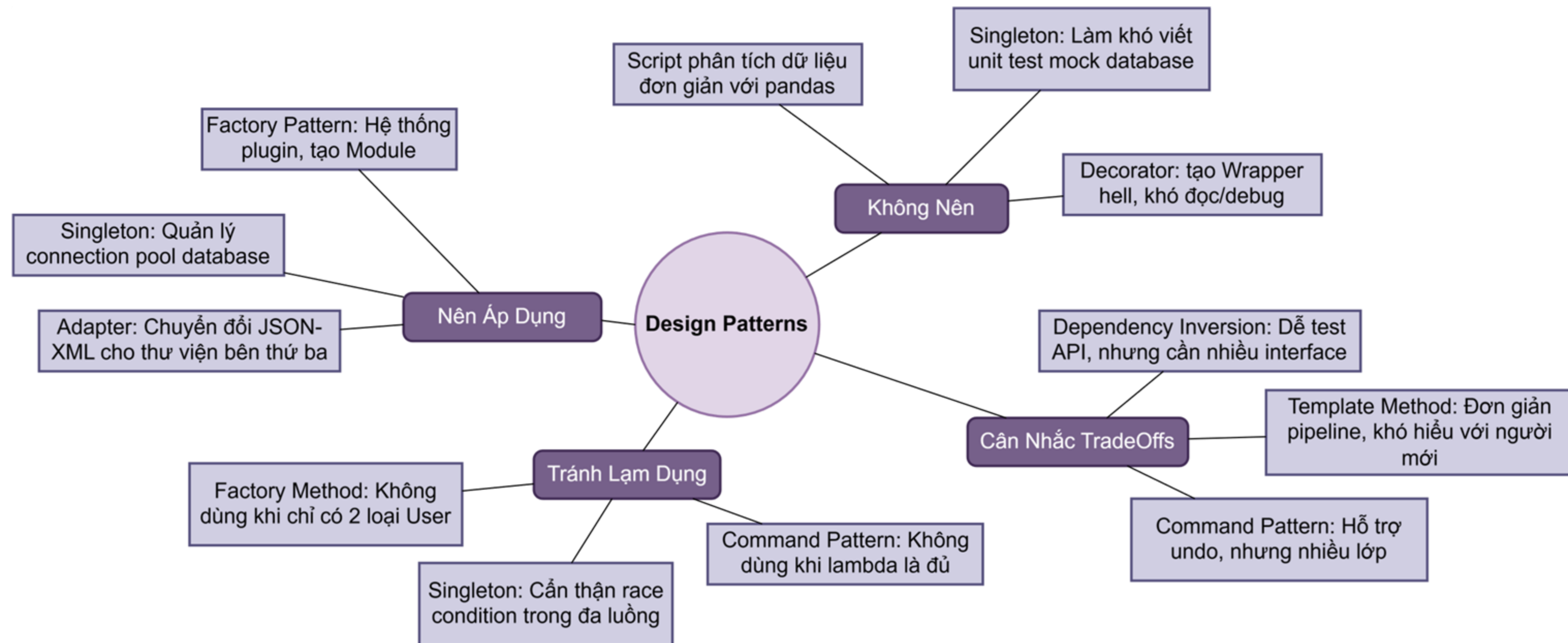
Command Pattern



Template Method Pattern



Cách Dùng Design Patterns Hiệu Quả



1 **Nắm vững 5 nguyên tắc SOLID**

Áp dụng Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion để thiết kế code dễ mở rộng, bảo trì.

2 **Áp dụng SOLID vào Python**

Tận dụng Duck Typing, ABC module, Composition để tuân thủ nguyên tắc SOLID, tránh kế thừa phức tạp.

3 **Sử dụng Design Patterns phù hợp**

Creational: Factory (tạo đối tượng linh hoạt), Singleton (1 class 1 instance).

Structural: Adapter (kết nối interface), Decorator (thêm chức năng linh hoạt).

Behavioral: Command (đóng gói độc lập với tham số và thời gian), Template Method (định nghĩa khung thuật toán).

4 **Sử dụng Design Patterns phù hợp**

Khi nên áp dụng, không nên áp dụng, tránh lạm dụng và cân nhắc trade-offs

Phần 1: Clean Code & Formatting

Tiêu chuẩn PEP-8, tài liệu hóa code với docstring và type annotations, công cụ kiểm tra code và tích hợp CI

Phần 2: Pythonic Code

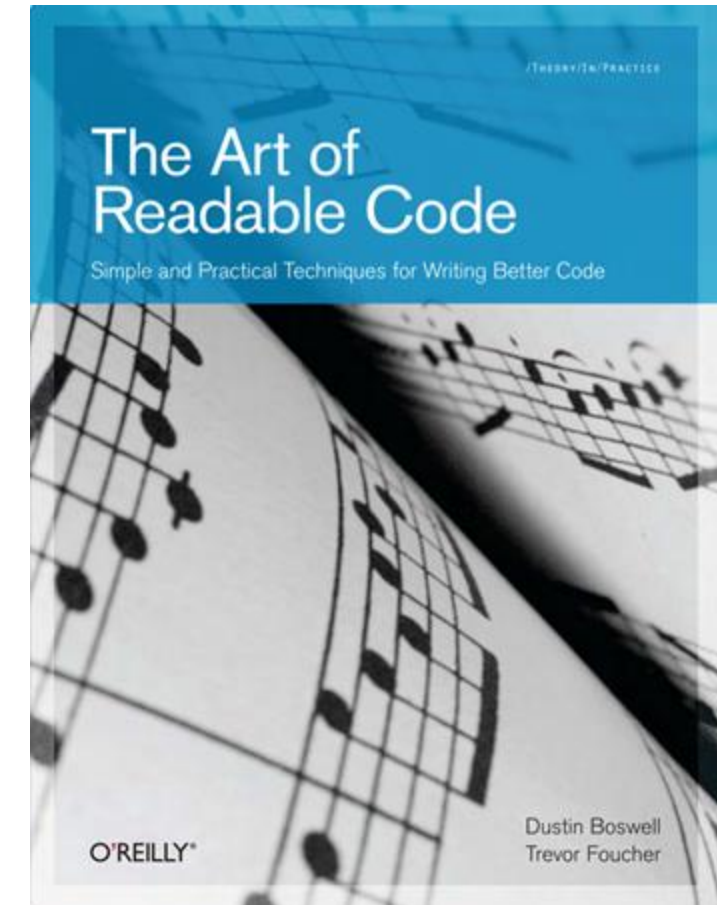
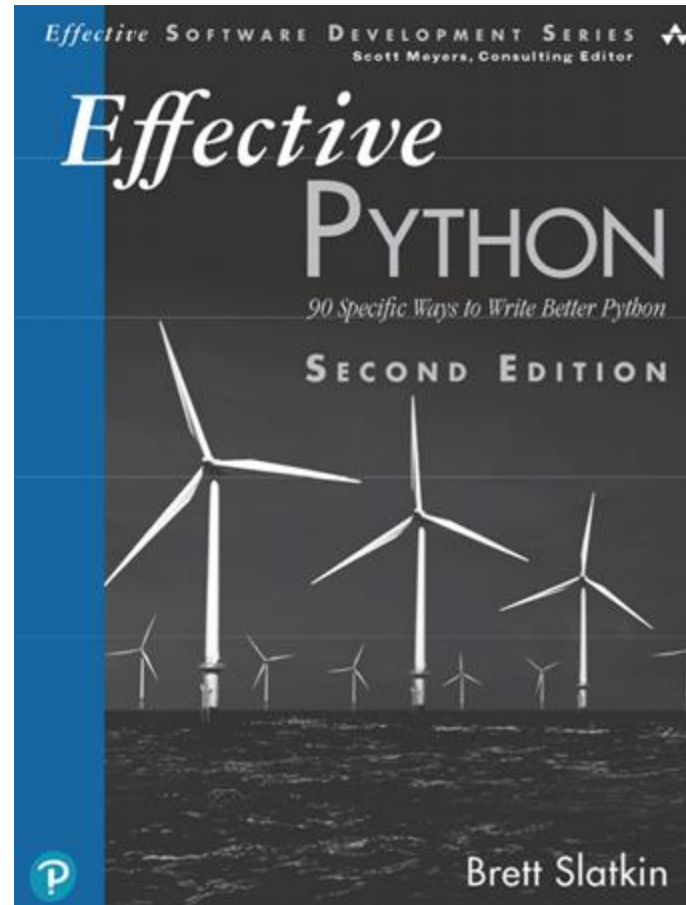
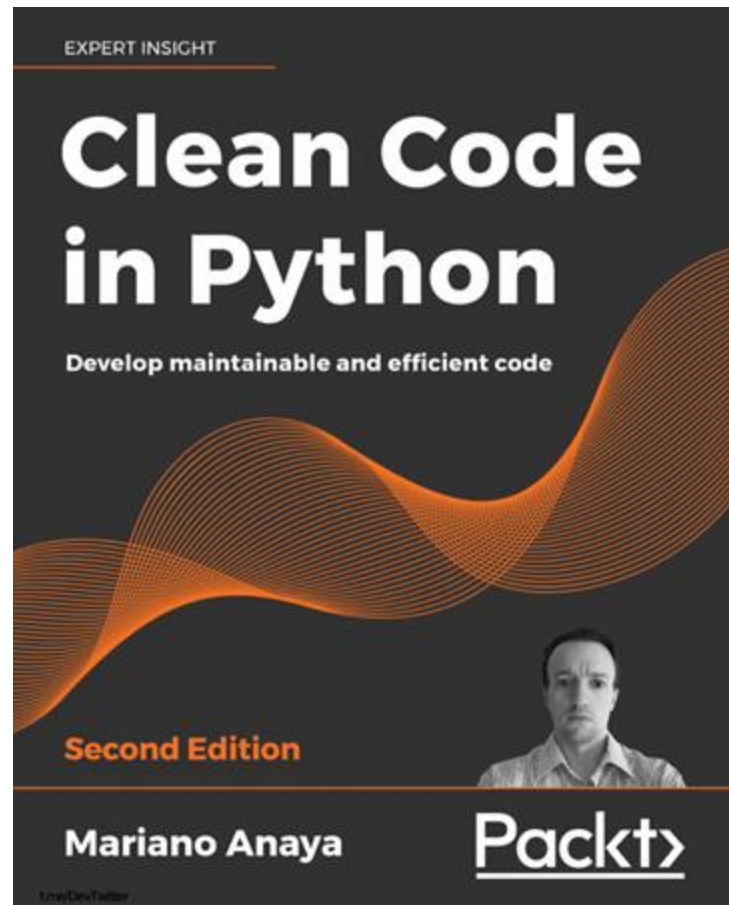
List/Dict comprehensions, context managers, properties, assignment expressions và các kỹ thuật lập trình đặc trưng của Python

Phần 3: Nguyên lý code sạch

DRY, YAGNI, KISS, defensive programming, design by contract và separation of concerns

Phần 4: SOLID & Design Patterns

5 nguyên tắc SOLID và các design patterns phổ biến: Factory, Singleton, Adapter, Decorator, Command



1. Clean code in Python, Mariano Anaya
2. Effective-Python, Brett Slatkin
3. The Art Of Readable Code, Dustin Boswell and Trevor Foucher