Tuần 1 - Tổng hợp kiến thức Buổi học số 5

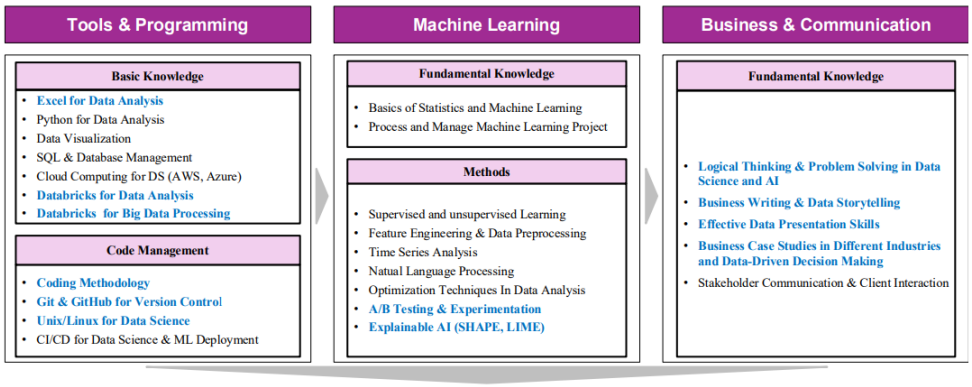
Time-Series Team

Ngày 8 tháng 6 năm 2025

Buổi học số 5 (Thứ 7, 07/06/2025) bao gồm bốn nội dung chính:

- ***Phần I: Kiến thức và Kĩ năng cần cho Data Science***
- ***Phần II: Nguyên tắc Clean Code và PEP-8 trong Python***
- ***Phần III: Pythonic Code trong Python***
- ***Phần IV: Nguyên lý chung để viết code tốt***

Phần I: Kiến thức và Kĩ năng cần cho Data Science



Hình 1: Lộ trình thành Data Science

1. Tools & Programming (Công cụ và Lập trình)

Basic Knowledge (Kiến thức cơ bản)

- **Excel for Data Analysis**: Phân tích dữ liệu bảng với Excel.
- **Python for Data Analysis**: Xử lý và phân tích dữ liệu bằng Python.
- **Data Visualization**: Trực quan hóa dữ liệu (Matplotlib, Seaborn, Power BI...).
- **SQL & Database Management**: Quản lý cơ sở dữ liệu và truy vấn SQL.
- **Cloud Computing for DS (AWS, Azure)**: Dùng dịch vụ đám mây trong phân tích dữ liệu.
- **Databricks for Data Analysis**: Phân tích dữ liệu lớn bằng Databricks.
- **Databricks for Big Data Processing**: Xử lý Big Data bằng Apache Spark trên Databricks.

Code Management (Quản lý mã nguồn)

- **Coding Methodology**: Phương pháp viết mã khoa học và dễ bảo trì.
- **Git & GitHub for Version Control**: Quản lý phiên bản mã nguồn.
- **Unix/Linux for Data Science**: Dùng dòng lệnh để xử lý dữ liệu.
- **CI/CD for DS & ML Deployment**: Tự động hóa triển khai mô hình.

2. Machine Learning (Học máy)

Fundamental Knowledge (Kiến thức nền tảng)

- **Basics of Statistics and Machine Learning**: Thống kê cơ bản và nền tảng ML.
- **Process and Manage Machine Learning Project**: Quản lý quy trình làm dự án ML.

Methods (Phương pháp)

- **Supervised and Unsupervised Learning**: Học có nhãn và không nhãn.
- **Feature Engineering & Data Preprocessing**: Tiền xử lý và tạo đặc trưng.
- **Time Series Analysis**: Phân tích chuỗi thời gian.
- **Natural Language Processing**: Xử lý ngôn ngữ tự nhiên.
- **Optimization Techniques in Data Analysis**: Kỹ thuật tối ưu hoá mô hình.
- **A/B Testing & Experimentation**: Thử nghiệm và kiểm định A/B.
- **Explainable AI (SHAP, LIME)**: Giải thích mô hình ML.

3. Business & Communication (Kỹ năng kinh doanh & giao tiếp)

Fundamental Knowledge (Kiến thức nền tảng)

- **Logical Thinking & Problem Solving in Data Science and AI**: Tư duy logic và giải quyết vấn đề bằng dữ liệu.
- **Business Writing & Data Storytelling**: Viết báo cáo và kể chuyện bằng dữ liệu.
- **Effective Data Presentation Skills**: Trình bày và trực quan hóa hiệu quả.
- **Business Case Studies & Data-Driven Decision Making**: Ứng dụng phân tích trong các ngành để đưa ra quyết định.
- **Stakeholder Communication & Client Interaction**: Giao tiếp với khách hàng và các bên liên quan.

Phần II: Nguyên tắc Clean Code và PEP-8 trong Python

1 Clean Code là gì?

Là **mã nguồn sạch, dễ đọc, dễ hiểu, dễ bảo trì và mở rộng**, được viết có chủ đích để người khác (hoặc chính bạn sau này) dễ dàng tiếp cận.

1.1 Đặc điểm của Clean Code

- **Readable – Dễ đọc, dễ hiểu:** Code phải dễ hiểu ngay cả khi không có bình luận.

```python
# Tên hàm rõ ràng
def calculate_total_price(items: list[float]) -> float:
    return sum(items)

```

- **Maintainable – Dễ bảo trì:** Tách thành các hàm nhỏ, dễ sửa đổi khi có thay đổi yêu cầu.

```python
def get_discount(price: float) -> float:
    return price * 0.1

def calculate_final_price(price: float) -> float:
    return price - get_discount(price)

```

- **Extensible – Dễ mở rộng:** Thiết kế mở rộng dễ dàng mà không sửa đổi quá nhiều code hiện có.

```python
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof"

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

```

- **Testable – Dễ kiểm thử:** Các hàm nhỏ, không phụ thuộc vào trạng thái toàn cục giúp viết unit test dễ dàng.

```python
def is_even(n: int) -> bool:
    return n % 2 == 0

```

1.2 Lợi ích của Clean Code – Quy tắc 3D1C

- **Giảm lỗi** (Bugs)
- **Dễ mở rộng**
- **Dễ làm việc nhóm**
- **Tiết kiệm thời gian dài hạn**

1.3 Khi nào được "tạm bỏ qua" Clean Code?

- Thử nghiệm nhanh (prototyping)
- Tình huống khẩn cấp
- Code chỉ dùng 1 lần
- Code chỉ mình bạn dùng (nhưng nhớ: bạn của tương lai cũng là "người khác")

2 Giới thiệu về PEP-8

PEP-8 là **tiêu chuẩn định dạng code chính thức của Python**, đảm bảo tính nhất quán và dễ đọc.

2.1 Một số quy tắc quan trọng trong PEP-8

1. **Đặt tên:**
   - Biến/Hàm: `snake_case`
   - Class: `PascalCase`
   - Hằng số: `UPPER_CASE`
2. **Thụt lề & khoảng trắng:**
   - 4 khoảng trắng, **không dùng tab**
     Ví dụ:

```
def my_function():
    print("Hello")  # 4 spaces indentation

```

   - Có khoảng trắng **sau dấu phẩy**, **giữa các toán tử** (`a = b + c`)
     Ví dụ:

```
x, y = 5, 10
total = x + y

```

   - **Không đặt khoảng trắng trong ngoặc**
     Ví dụ:

```
print(f(x, y))
my_list = [1, 2, 3]

```

3. **Giới hạn độ dài dòng:**
   - Code: 79 ký tự
     Ví dụ:

```
# There are over 79 characters in one coding row, that is not aligned with PEP-8.
    It is too long to read.
result = some_function_with_many_parameters(param1, param2, param3)

```

- Docstring/comment: 72 ký tự
  Ví dụ:

```
"""
This function calculates the area of a circle
given its radius.
"""

```

4. **Quy tắc import:**

- Nhóm theo thứ tự: thư viện chuẩn → thư viện ngoài → module nội bộ
  Ví dụ:

```
import os
import sys

import numpy as np
import requests

import mypackage.utils

```

- Mỗi dòng chỉ import một module
  Ví dụ:

```
# Correct:
import os
import sys

# Wrong:
import os, sys

```

5. **Dòng trắng:**

- 2 dòng trước class, 1 dòng giữa các hàm
  Ví dụ:

```
class MyClass:

    def method1(self):
        pass

    def method2(self):
        pass

```

- Dòng trắng trong hàm để chia nhóm logic
  Ví dụ:

```
def process_data(data):
    clean_data = clean(data)
    validate(clean_data)

    result = analyze(clean_data)
    return result

```

3 Tài liệu hóa (Docstring)

- Dùng `"""..."""` ngay dưới định nghĩa hàm/class/module
- Các style phổ biến: **Google**, **NumPy**, **reStructuredText**
- Sử dụng **type hints** để rõ ràng hơn:

```python
def add(a: int, b: int) -> int:
    """Sum of two intergers"""
    return a + b
```

Code Listing 1: Ví dụ Docstring với Type Hints

3.1 Công cụ kiểm tra code và định dạng tự động

Công cụ	Mục đích	Cài đặt	Chạy lệnh
Flake8	Kiểm tra lỗi PEP-8, logic	`pip install flake8`	`flake8 file.py`
Black	Format code tự động	`pip install black`	`black file.py`
Pylint	Phân tích sâu, chấm điểm	`pip install pylint`	`pylint file.py`
Mypy	Kiểm tra type hint tĩnh	`pip install mypy`	`mypy file.py`

Lưu ý: Có thể tích hợp vào IDE (VSCode, PyCharm) hoặc CI/CD pipeline.

4 Cấu Trúc Tiêu Chuẩn Cho Dự Án Python

4.1 Cấu trúc thư mục chuẩn

Một dự án Python được tổ chức tốt sẽ dễ dàng bảo trì và mở rộng. Dưới đây là cấu trúc thư mục chuẩn cho một dự án Python:

- **Thư mục gốc (project_name/)**
  Chứa các file quan trọng như `README.md`, `requirements.txt`, `setup.py`, giúp mô tả dự án, quản lý các thư viện phụ thuộc và cấu hình cài đặt.
- **Mã nguồn (project_name/src/)**
  Chứa các module và package chính của dự án.
- **Tests (project_name/tests/)**
  Chứa các bộ kiểm thử, ví dụ như `unittest` hoặc `pytest` để đảm bảo chất lượng code.
- **Tài liệu (project_name/docs/)**
  Chứa các hướng dẫn sử dụng và tài liệu API chi tiết cho người dùng và nhà phát triển.
- **Tài nguyên (project_name/resources/)**
  Bao gồm dữ liệu tĩnh, các mẫu (templates), và tài sản (assets) hỗ trợ cho dự án.

Ngoài ra, dự án nên sử dụng các file cấu hình như `.gitignore`, `pyproject.toml` và `README.md` để đảm bảo dự án được quản lý hiệu quả và dễ dàng cho người khác hiểu cũng như đóng góp.

Ví dụ tham khảo cấu trúc dự án: https://github.com/khoanta-ai/python_project_template

4.2 Cấu Trúc Tiêu Chuẩn Cho Dự Án Data Science

Cookiecutter Data Science là một công cụ giúp tạo cấu trúc folder tiêu chuẩn nhanh chóng và phổ biến trong cộng đồng Data Science.

```
├── 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            <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures        <- 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
```

Hình 2: Cấu trúc folder tiêu chuẩn Data Science

Tài liệu tham khảo: https://cookiecutter-data-science.drivendata.org/

5 Triết lý Python (The Zen of 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.

Đây là 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. Tham khảo: https://peps.python.org/pep-0020/

Phần III: Pythonic Code trong Python

1 Pythonic Code là gì?

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: List Comprehensions và Dictionary Comprehensions

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

Code Listing 2: List Comprehension

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

Code Listing 3: Dictionary Comprehension

2 Indexes và Slices trong Python

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 dữ liệu linh hoạt và Pythonic.

2.1 Cách dùng indexes và slices cơ bản

```
numbers = [1, 2, 3, 4, 5]
first = numbers[0]     # 1
last = numbers[-1]     # 5

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

2.2 Ứng dụng slices nâng cao

```
data = [10, 20, 30, 40, 50, 60]
even_indexes = data[::2]    # [10, 30, 50]
odd_indexes = data[1::2]    # [20, 40, 60]

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

message = "Python"
reverse_msg = message[::-1] # "nohtyP"
substring = message[1:4]    # "yth"
```

2.3 Một số kỹ thuật Pythonic phổ biến với slicing

```
original = [1, 2, 3]
copy_list = original[:]

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

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

3 List, Dict, Set Comprehensions

Comprehensions giúp code ngắn gọn và nhanh hơn so với vòng lặp thông thường.

3.1 Không Pythonic

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

3.2 Pythonic (List Comprehension)

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

3.3 Dictionary Comprehension

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

3.4 Set Comprehension

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

4 Context Managers (with)

4.1 Khái niệm

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

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

Code Listing 4: Context Manager cơ bản với file

4.2 Ưu điểm

Không dùng `with` (không Pythonic):

```
file = open("data.txt", "r")
try:
    data = file.read()
finally:
    file.close()
```

Dùng `with` (Pythonic):

```
with open("data.txt", "r") as file:
    data = file.read()
```

4.3 Tạo Context Manager riêng với `@contextmanager`

Sử dụng decorator `@contextmanager` từ module `contextlib` để tạo context manager với generator.

```
from contextlib import contextmanager

@contextmanager
def open_file(path, mode):
    f = open(path, mode)
    try:
        yield f
    finally:
        f.close()

with open_file("sample.txt", "w") as f:
    f.write("Hello AI Vietnam!")
```

Code Listing 5: Context Manager tùy chỉnh với decorator

4.4 Ứng dụng thực tế

Ví dụ 1: Đo thời gian thực thi

```
import time
from contextlib import contextmanager

@contextmanager
def timer(name):
    start = time.time()
    yield
    end = time.time()
    print(f"[{name}] Elapsed: {end - start:.4f} sec")

with timer("Download task"):
    time.sleep(2)
```

Ví dụ 2: Kết nối tạm thời với SQLite

```
import sqlite3
from contextlib import contextmanager

@contextmanager
def db_connection(path):
    conn = sqlite3.connect(path)
```

```
    try:
        yield conn
    finally:
        conn.close()

with db_connection("mydb.sqlite") as conn:
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT)")
    conn.commit()
```

Ví dụ 3: Tạm thay đổi thư mục làm việc

```
import os
from contextlib import contextmanager

@contextmanager
def change_dir(destination):
    prev_cwd = os.getcwd()
    os.chdir(destination)
    try:
        yield
    finally:
        os.chdir(prev_cwd)

with change_dir("/tmp"):
    print("Working in:", os.getcwd())
```

5 So Sánh và Điều Kiện - Pythonic Style

5.1 Trường hợp 1: So sánh với `None`

Giải thích: `None` là singleton object, phải dùng `is` hoặc `is not` để so sánh về mặt identity, không dùng `==` vì có thể dẫn đến kết quả không mong muốn.

```
# Wrong (Non-Pythonic)
if x == None:
    ...

# Correct (Pythonic)
if x is None:
    ...
```

Code Listing 6: So sánh với None

5.2 Trường hợp 2: So sánh Boolean

Giải thích: Boolean tự thân đã có giá trị truth, không cần so sánh thêm hoặc dùng ngoặc đơn không cần thiết, giúp code đơn giản và rõ ràng hơn.

```
# Wrong (Non-Pythonic)
if flag == True:
    ...

# Correct (Pythonic)
if flag:
    ...
```

Code Listing 7: So sánh Boolean

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

Giải thích: Chuỗi, list, dict rỗng được đánh giá là `False` trong ngữ cảnh Boolean. Tận dụng truthiness giúp code ngắn gọn và dễ đọc hơn.

```
# Wrong (Non-Pythonic)
if len(my_list) == 0:
    ...

# Correct (Pythonic)
if not my_list:
    ...
```

Code Listing 8: Kiểm tra collection rỗng

5.4 Trường hợp 4: Kiểm tra phần tử trong collection

Giải thích: Dùng từ khóa `in` để kiểm tra sự tồn tại của phần tử trong collection giúp code dễ đọc và hiệu quả hơn so với vòng lặp.

```
# Wrong (Non-Pythonic)
found = False
for x in collection:
    if x == target:
        found = True
        break

# Correct (Pythonic)
if target in collection:
    ...
```

Code Listing 9: Kiểm tra phần tử trong collection

5.5 Trường hợp 5: Chaining comparison

Giải thích: Python cho phép nối chuỗi các phép so sánh, giúp code dễ đọc và hiểu hơn, giống như trong toán học.

```
# Wrong (Non-Pythonic)
if x >= 0 and x < 10:
    ...

# Correct (Pythonic)
if 0 <= x < 10:
    ...
```

Code Listing 10: Chaining comparison

6 Properties và dấu gạch dưới _ trong Python

6.1 @property

`@property` cho phép sử dụng method như thuộc tính, giúp code gọn gàng và dễ kiểm soát truy cập.

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

```

```
    @property
    def area(self):
        return self._width * self._height

rect = Rectangle(5, 3)
print(rect.area)  # Không ọgi ưnh hàm, ưnh ộthuc tính
```

6.2 Quy ước sử dụng dấu underscore

- `_var` (Single Underscore)
  Quy ước cho biến private hoặc "internal use" trong module hoặc class. Không thực sự ngăn truy cập từ bên ngoài nhưng là dấu hiệu "không nên dùng trực tiếp".

- `__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 vô tình.

- `__var__` (Double Underscore hai bên)
  Dành cho các phương thức đặc biệt (magic hoặc dunder methods) như `__init__`, `__str__`. Không nên tự tạo kiểu này trừ khi cần thiết.

- `_` (Một dấu gạch dưới)
  Thường dùng làm biến tạm không quan trọng hoặc kết quả gần nhất trong Python interpreter.

Phần IV: Nguyên lý chung để viết code tốt

1 Nguyên lý lập trình tốt: Viết code sạch, dễ bảo trì và mở rộng

Viết code tốt không chỉ là làm cho nó chạy được, mà còn là tạo ra phần mềm bền vững, dễ bảo trì, mở rộng và dễ hiểu đối với các lập trình viên khác (hoặc chính bạn trong tương lai). Một số nguyên lý cốt lõi sau đây là kim chỉ nam cho việc viết code chất lượng.

1.1 DRY – Don't Repeat Yourself

Định nghĩa: Đừng lặp lại chính mình. Mỗi phần kiến thức trong hệ thống chỉ nên được biểu diễn duy nhất một lần.

Lợi ích:

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

Cách áp dụng:

- Tách các đoạn code lặp lại thành hàm hoặc lớp riêng.
- Đưa các logic chung vào module.
- Định nghĩa hằng số hoặc template tại một nơi.

```
# Vi phạm DRY
print("Welcome, John!")
print("Welcome, Alice!")

# Tuân thủ DRY
def greet(name):
    print(f"Welcome, {name}!")

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

1.2 YAGNI – You Aren't Gonna Need It

Định nghĩa: Đừng viết những gì chưa cần.

Lợi ích:

- Tránh lãng phí thời gian vào tính năng không dùng.
- Giữ code đơn giản, dễ bảo trì.
- Giảm technical debt.

Cách áp dụng:

- Không viết các hàm "phòng khi cần".
- Chỉ phát triển tính năng khi có yêu cầu thực tế.

1.3 KISS – Keep It Simple, Stupid

Định nghĩa: Giữ mọi thứ càng đơn giản càng tốt.
Lợi ích:

- Code đơn giản, dễ đọc, dễ bảo trì.
- Dễ debug và tăng hiệu suất nhóm.

```
# Quá phức tạp
def factorial(n): return 1 if n==0 else n*factorial(n-1)

# Đơn giản hơn
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

1.4 Defensive Programming – Lập trình phòng thủ

Định nghĩa: Luôn giả định rằng lỗi có thể xảy ra.
Nguyên tắc:

- Không tin tưởng bất kỳ input nào.
- Xác thực dữ liệu đầu vào.
- Xử lý ngoại lệ và kiểm tra điều kiện biên.

```
def divide(a, b):
    if b == 0:
        raise ValueError("Không thể chia cho 0.")
    return a / b
```

1.5 Separation of Concerns – Phân chia trách nhiệm

Định nghĩa: Chia chương trình thành các phần riêng biệt, mỗi phần đảm nhiệm một vai trò cụ thể.
Lợi ích:

- Dễ bảo trì, mở rộng và test.
- Giảm phụ thuộc giữa các thành phần.
- Tăng khả năng tái sử dụng.

1.6 Error Handling – Xử lý lỗi đúng cách

- Không bắt lỗi quá chung chung như `except Exception`.
- Luôn ghi log lỗi hoặc truyền tiếp thông tin lỗi.
- Sử dụng `logging` thay vì `print`.

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
```

Tuỳ 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
```

Hình 3: Sử dụng Logging

1.7 Logging vs Print – Ghi log đúng cách

Tiêu chí	print()	logging
Dễ sử dụng	✓	Cần cấu hình
Phân loại mức độ	✗	✓
Lưu log vào file/email	✗	✓
Tắt khi deploy	✗	✓

2 Giới Thiệu SOLID Principles

2.1 Áp Dụng SOLID Vào Python

- **Tính linh hoạt:** Python hỗ trợ dynamic typing, giúp thay đổi hành vi đối tượng tại runtime, hữu ích cho nguyên lý Open/Closed.
- **Duck Typing:** Python không yêu cầu kế thừa, miễn là object có phương thức tương thích. Hỗ trợ tốt cho Liskov Substitution và Interface Segregation.
- **Abstractions:** Module `abc` với `@abstractmethod` và `ABC` cho phép định nghĩa interface rõ ràng, đảm bảo nguyên lý Dependency Inversion.
- **Composition:** Ưu tiên composition over inheritance thông qua mixins và dependency injection, hỗ trợ tốt cho Single Responsibility.

2.2 Single Responsibility Principle

- Mỗi lớp chỉ nên có một nhiệm vụ duy nhất và một lý do để thay đổi.
- Chia nhỏ lớp lớn thành các module độc lập như `FileReader`, `FileValidator`, `FileParser`.
- Dễ bảo trì và kiểm thử, nâng cao độ tin cậy của phần mềm.

2.3 Open/Closed Principle

- **Mở cho việc mở rộng, đóng cho việc sửa đổi.**
- Thiết kế code để có thể thêm tính năng mà không cần sửa đổi logic cũ.
- Sử dụng abstract class như `PaymentProcessor`, mở rộng bằng kế thừa: `CreditCardProcessor`, `PayPalProcessor`.
- Áp dụng strategy pattern và plugin architecture để tuân thủ nguyên lý.

2.4 Liskov Substitution Principle

- Các lớp con phải thay thế được lớp cha mà không phá vỡ logic chương trình.
- Không thay đổi hành vi mong đợi trong lớp con.
- Tránh: thêm điều kiện mạnh hơn, giảm điều kiện sau, ném exception mới.
- Ví dụ: `Square` không nên kế thừa từ `Rectangle` nếu phá vỡ logic tính diện tích.

2.5 Interface Segregation Principle

- Tách các interface lớn thành nhiều interface nhỏ chuyên biệt.
- Tránh ép buộc class phải implement các phương thức không sử dụng.
- Áp dụng bằng cách dùng `abc.ABC`, `Protocol` từ `typing`, và duck typing.

2.6 Dependency Inversion Principle

- Module cấp cao không nên phụ thuộc vào module cấp thấp, cả hai nên phụ thuộc vào abstraction.
- Module cấp thấp thực thi các interface được định nghĩa bởi module cấp cao.
- Áp dụng bằng cách tạo các interface bằng `ABC`, `Protocol`, hoặc duck typing.

3 Các Design Patterns Phổ Biến

3.1 Creational Patterns

- **Factory:** Tạo đối tượng mà không cần biết lớp con cụ thể.

Ví dụ: Factory Pattern

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

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

    def get_pet(pet="dog"):
        pets = dict(dog=Dog(), cat=Cat())
        return pets[pet]

    animal = get_pet("cat")
    print(animal.speak())  # Output: Meow!

```

Code Listing 11: Factory Pattern Example

- **Singleton:** Đảm bảo chỉ có một instance được tạo ra trong toàn bộ hệ thống.

Ví dụ: Singleton Pattern

```
    class Singleton:
        _instance = None

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

    s1 = Singleton()
    s2 = Singleton()
    print(s1 is s2)  # Output: True

```

Code Listing 12: Singleton Pattern Example

3.2 Structural Patterns

- **Adapter:** Cho phép các interface không tương thích hoạt động cùng nhau.

Ví dụ: Adapter Pattern

```
class OldPrinter:
    def old_print(self):
        return "Old style print"

class NewPrinter:
    def print(self):
        return "Modern print"

class PrinterAdapter:
    def __init__(self, old_printer):
        self.old_printer = old_printer

    def print(self):
        return self.old_printer.old_print()

printer = PrinterAdapter(OldPrinter())
print(printer.print())  # Output: Old style print

```

Code Listing 13: Adapter Pattern Example

- **Decorator:** Thêm chức năng cho object mà không thay đổi cấu trúc ban đầu.

Ví dụ: Decorator Pattern

```
def make_bold(func):
    def wrapper():
        return "<b>" + func() + "</b>"
    return wrapper

@make_bold
def greet():
    return "Hello"

print(greet())  # Output: <b>Hello</b>

```

Code Listing 14: Decorator Pattern Example

3.3 Behavioral Patterns

- **Command:** Đóng gói yêu cầu dưới dạng đối tượng độc lập.

Ví dụ: Command Pattern

```
class Light:
    def turn_on(self):
        print("Light is ON")

    def turn_off(self):
        print("Light is OFF")

class Command:
    def execute(self): pass

```

```
    class TurnOn(Command):
        def __init__(self, light):
            self.light = light
        def execute(self):
            self.light.turn_on()

    class TurnOff(Command):
        def __init__(self, light):
            self.light = light
        def execute(self):
            self.light.turn_off()

    light = Light()
    on = TurnOn(light)
    off = TurnOff(light)

    on.execute()   # Output: Light is ON
    off.execute()  # Output: Light is OFF

```

Code Listing 15: Command Pattern Example

- **Template Method:** Định nghĩa khung thuật toán và cho phép lớp con tùy biến.

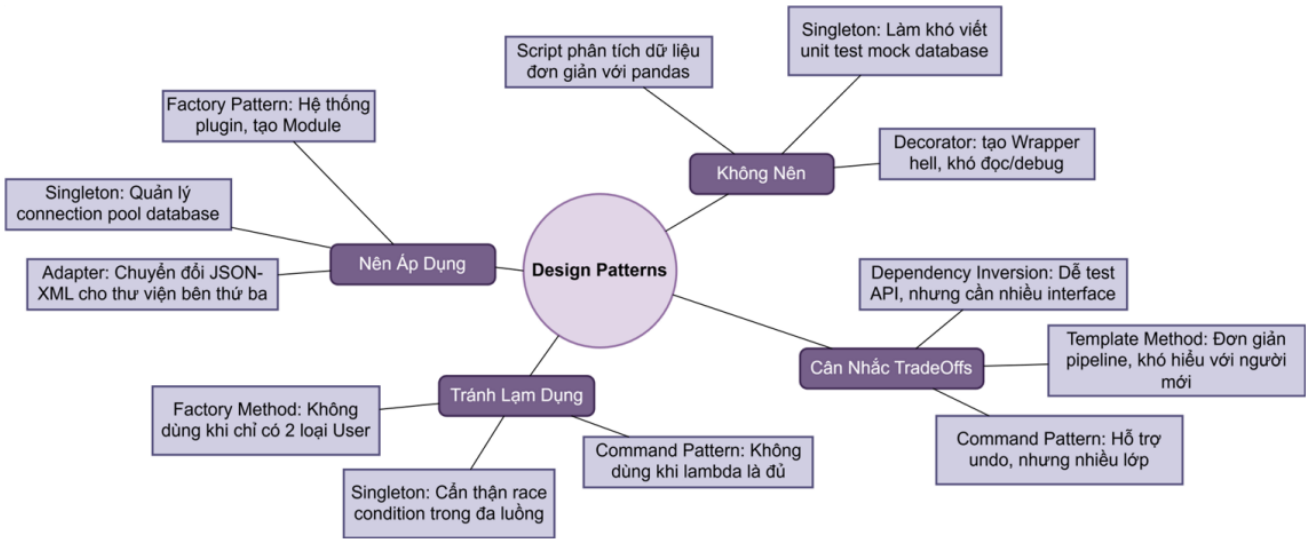
Ví dụ: Template Method Pattern

```
from abc import ABC, abstractmethod

class DataProcessor(ABC):
    def process(self):
        self.load_data()
        self.clean_data()
        self.analyze_data()

    @abstractmethod
    def load_data(self): pass

    def clean_data(self):
        print("Default cleaning...")

    @abstractmethod
    def analyze_data(self): pass

class CSVProcessor(DataProcessor):
    def load_data(self):
        print("Loading CSV")

    def analyze_data(self):
        print("Analyzing CSV")

processor = CSVProcessor()
processor.process()
# Output:
# Loading CSV
# Default cleaning...
# Analyzing CSV

```

Code Listing 16: Template Method Pattern Example

3.4 Cách Dùng Design Patterns Hiệu Quả

1. Nắm vững 5 nguyên lý SOLID.
2. Sử dụng design pattern phù hợp với tình huống cụ thể.
3. Tận dụng duck typing, ABC module, composition trong Python.
4. Cân nhắc lợi ích và tránh lạm dụng pattern không cần thiết.



Hình 4: Cách Dùng Design Patterns Hiệu Quả