

# Một số kỹ thuật hay dùng

- Monkey patch
- Monkey patch phù hợp từng trường hợp trong Django
- Decorator - Gói thêm chức năng cho hàm
- Annotated
- Context Manager (with) - Quản lý tài nguyên
- Metaclass - Tùy biến hành vi của class
- Dynamic import (runtime import)
- Signal/Event Hooks
- Strategy Pattern với Dictionary
- Caching / Memoization
- Metadata
- lambda
- functools
- MyPy
- yield
- yield nâng cao
- Kỹ thuật hay và gọn gàng khi thao tác với danh sách (list)
- Pydantic
- Dataclass

# Monkey patch

**Monkey patch** là kỹ thuật thay đổi hoặc mở rộng hành vi của **một hàm, method, class, hoặc module tại thời điểm runtime** (lúc chương trình đang chạy) **mà không sửa mã gốc**.

Nó được dùng khi:

- Không thể/chưa thể sửa mã nguồn gốc
- Muốn mở rộng thư viện bên thứ 3
- Override tạm thời một behavior

## Nói đơn giản:

“ **Bạn thay thế một hàm hoặc class trong thư viện gốc bằng hàm của bạn**  
– ngay lúc chương trình đang chạy!

Cấu trúc cơ bản

```
# Module gốc (giả sử không thể sửa)
class Animal:
    def speak(self):
        return "Gâu gâu"

# Monkey patch hàm speak
def meow(self):
    return "Meo meo"

Animal.speak = meow

# Test
a = Animal()
print(a.speak()) # [] Meo meo
```

Ví dụ với module tiêu chuẩn Python

Giả sử bạn muốn override `math.sqrt` để log mỗi lần gọi:

```
import math

original_sqrt = math.sqrt

def logged_sqrt(x):
    print(f"√{x} was called")
    return original_sqrt(x)

math.sqrt = logged_sqrt

print(math.sqrt(9)) # √ log + 3.0
```

## Monkey patch method của class

```
class User:
    def greet(self):
        return "Hello"

def custom_greet(self):
    return "Xin chào!"

User.greet = custom_greet

u = User()
print(u.greet()) # Xin chào!
```

## Monkey patch trong unit test

**Rất phổ biến khi test cần thay thế tạm hàm gọi API, DB, hoặc logic tốn thời gian.**

```
import myapp.utils

original = myapp.utils.get_current_time

def fake_time():
    return "2025-01-01"

myapp.utils.get_current_time = fake_time
```

## Cách hiện đại (với `unittest.mock`)

```
from unittest.mock import patch

with patch('myapp.utils.get_current_time', return_value="2025-01-01"):
    print(myapp.utils.get_current_time()) # 2025-01-01
```

# Kỹ thuật monkey patch nâng cao

## 1. Patch theo điều kiện

```
if settings.DEBUG:
    module.func = debug_version
else:
    module.func = prod_version
```

## 2. Chỉ patch một instance (không toàn bộ class)

```
import types

class A:
    def say(self): return "Hello"

a = A()
a.say = types.MethodType(lambda self: "Xin chào", a)

print(a.say()) # Xin chào
```

# Monkey patch và thư viện bên thứ 3

## Ví dụ override hàm trong thư viện requests:

```
import requests

_original_get = requests.get

def my_get(*args, **kwargs):
    print(f"GET called: {args[0]}")
    return _original_get(*args, **kwargs)
```

```
requests.get = my_get
```

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**

# Monkey patch phù hợp từng trường hợp trong Django

## I. Monkey patch một **model method**

Ví dụ: bạn muốn override method `save` của `User` model trong Django mà không sửa trực tiếp `User` gốc

```
from django.contrib.auth.models import User

original_save = User.save

def custom_save(self, *args, **kwargs):
    print(f"[Custom Save] Saving user: {self.username}")
    return original_save(self, *args, **kwargs)

User.save = custom_save
```

## II. Monkey patch một view của app khác (hoặc của Django)

```
from django.contrib.auth.views import LoginView

original_dispatch = LoginView.dispatch

def custom_dispatch(self, request, *args, **kwargs):
    print("[Custom LoginView] User is logging in")
    return original_dispatch(self, request, *args, **kwargs)

LoginView.dispatch = custom_dispatch
```

## III. Monkey patch Django Signals

Ví dụ: override xử lý mặc định của `post_save` signal

```
from django.db.models.signals import post_save
from django.contrib.auth.models import User

def custom_user_post_save(sender, instance, created, **kwargs):
    if created:
        print(f"[Custom Signal] New user created: {instance.username}")

# Ngắt kết nối các signal hiện tại nếu cần
post_save.disconnect(dispatch_uid="default_user_post_save", sender=User)

# Gắn signal monkey patch
post_save.connect(custom_user_post_save, sender=User)
```

## IV. Monkey patch một form method

Ví dụ: bạn muốn thay đổi method `clean_email` trong `UserCreationForm`

```
from django.contrib.auth.forms import UserCreationForm

def custom_clean_email(self):
    email = self.cleaned_data.get("email")
    if not email.endswith("@mycompany.com"):
        raise forms.ValidationError("Email must be a company email")
    return email

UserCreationForm.clean_email = custom_clean_email
```

## V. Monkey patch Django admin

Ví dụ: thêm log mỗi lần object được lưu trong admin

```
from django.contrib import admin
from django.contrib.auth.models import User
```

```
original_save_model = admin.ModelAdmin.save_model

def custom_save_model(self, request, obj, form, change):
    print(f"[Admin Log] {obj} saved by {request.user}")
    return original_save_model(self, request, obj, form, change)

admin.ModelAdmin.save_model = custom_save_model
```

## VI. Gợi ý tổ chức Monkey Patch sạch sẽ

```
myapp/
├─ monkey/
│   ├── __init__.py
│   ├── user_patch.py
│   └── view_patch.py
```

Trong `myapp/monkey/__init__.py`:

```
from .user_patch import patch_user
from .view_patch import patch_login_view

def apply_monkey_patches():
    patch_user()
    patch_login_view()
```

Trong `apps.py` của app bạn:

```
from django.apps import AppConfig

class MyAppConfig(AppConfig):
    name = 'myapp'

    def ready(self):
        from myapp.monkey import apply_monkey_patches
        apply_monkey_patches()
```

## 📦 Mẹo vặt



Tình huống	Lời khuyên
App bạn override chưa load	Hãy đảm bảo thứ tự cài đặt trong <code>INSTALLED_APPS</code>
Gọi lại hàm gốc	Luôn backup lại hàm gốc <code>original_func = Class.func</code>
Muốn patch tạm	Có thể dùng <code>unittest.mock.patch()</code>
Patch không hoạt động	Kiểm tra xem <code>ready()</code> trong <code>apps.py</code> có được gọi không

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**

# Decorator – Gói thêm chức năng cho hàm

Decorator là một khái niệm cực kỳ mạnh mẽ và phổ biến trong Python – đặc biệt khi bạn muốn **gói thêm (wrap) chức năng cho một hàm mà không thay đổi mã gốc**.

“Decorator là một **hàm** dùng để **gói thêm logic** cho một **hàm khác**.”

Bạn có thể hình dung nó giống như việc “bọc thêm lớp áo” cho một người – người đó vẫn là người đó, nhưng có thêm tính năng mới (ví dụ: áo giáp, áo tàng hình,... ).

## I. Cấu trúc cơ bản

```
def my_decorator(func):  
    def wrapper(*args, **kwargs):  
        # Thêm chức năng trước khi gọi func  
        print("Bắt đầu gọi hàm...")  
        result = func(*args, **kwargs)  
        # Thêm chức năng sau khi gọi func  
        print("Đã gọi xong.")  
        return result  
    return wrapper
```

Dùng như sau:

```
@my_decorator  
def say_hello():  
    print("Xin chào!")  
  
say_hello()
```

Kết quả

Bắt đầu gọi hàm...

Xin chào!

Đã gọi xong.

## II. Ví dụ thực tế

### 1. Log thời gian thực thi của một hàm

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Hàm {func.__name__} chạy trong {end - start:.2f} giây")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(2)
    print("Xong việc rồi")

slow_function()
```

### 2. Kiểm tra phân quyền trước khi gọi hàm (giống middleware)

```
def require_admin(func):
    def wrapper(user, *args, **kwargs):
        if user != 'admin':
            print("Không có quyền truy cập")
            return None
        return func(user, *args, **kwargs)
    return wrapper

@require_admin
def view_dashboard(user):
    print(f" Welcome {user}, đây là dashboard.")
```

```
view_dashboard("guest") # Không có quyền truy cập
view_dashboard("admin") # Welcome admin, đây là dashboard.
```

## III. Decorator có tham số (Decorator Factory)

Ví dụ bạn muốn tạo decorator có thể truyền tham số:

```
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def hello():
    print("Hi!")

hello()
```

```
Hi!
Hi!
Hi!
```

## IV. Giữ metadata của hàm gốc (functools.wraps)

Mặc định khi bạn dùng decorator, thông tin gốc của hàm như tên và docstring sẽ bị mất.

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
```

```
def wrapper(*args, **kwargs):  
    print("Bọc hàm nè")  
    return func(*args, **kwargs)  
return wrapper
```

## V. Decorator trong Django và thực tế

Tên Decorator	Chức năng
@login_required	Đảm bảo người dùng đã đăng nhập trước khi truy cập view
@require_POST	Chỉ cho phép method POST (dùng trong view)
@csrf_exempt	Bỏ qua kiểm tra CSRF (thường dùng khi làm API)
@transaction.atomic	Gói các thao tác DB trong một transaction an toàn

```
from django.contrib.auth.decorators import login_required  
  
@login_required  
def dashboard(request):  
    return render(request, 'dashboard.html')
```

## VI. Tóm lại

Khái niệm	Giải thích ngắn
Decorator	Hàm gói hàm khác, thêm chức năng mà không sửa hàm gốc
Dùng @decorator_name	Cú pháp gọn gàng để áp dụng
functools.wraps()	Giữ tên, docstring, v.v. của hàm gốc
Có thể lồng nhau	Decorator A gói decorator B

Tác giả: **Đỗ Ngọc Tú**  
Công Ty Phần Mềm **VHTSoft**

# Annotated

`Annotated` trong Python là cách để **gắn thêm thông tin (metadata)** vào một kiểu dữ liệu.

Cú pháp:

```
from typing import Annotated

x: Annotated[int, "This is some metadata"]
```

- `x` vẫn là `int`
- `"This is some metadata"` không ảnh hưởng đến việc chạy chương trình
- Nhưng các **framework** hoặc **decorator** có thể đọc và xử lý metadata đó để làm điều gì đó đặc biệt

Ví dụ đơn giản: kiểm tra quyền truy cập

```
from typing import Annotated

def view_dashboard(user: Annotated[str, "admin_only"]):
    print(f"Welcome, {user}!")
```

- `user` là một chuỗi (str)
- Nhưng có thêm annotation `"admin_only"` — giả sử bạn dùng một **framework kiểm tra quyền hạn**, nó có thể dùng annotation đó để từ chối người không phải admin

Ví dụ với custom validator

```
from typing import Annotated

def validate_positive(x: int) -> int:
    if x <= 0:
        raise ValueError("Value must be positive")
    return x

PositiveInt = Annotated[int, validate_positive]

def set_age(age: PositiveInt):
    print(f"Age set to: {age}")
```

```
set_age(25) # OK
```

```
set_age(-1) # Sẽ báo lỗi vì validator check
```

“ Ở đây `PositiveInt` là `int` **kèm theo hàm kiểm tra**. Bạn có thể tưởng tượng `Annotated` giống như `int`, nhưng có thêm "hướng dẫn sử dụng".

# Context Manager (with) – Quản lý tài nguyên

Dùng để xử lý logic mở/đóng tự động: file, kết nối, khóa, transaction...

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

Tự động gọi `f.close()` dù có lỗi hay không.

Bạn có thể tự định nghĩa:

```
from contextlib import contextmanager  
  
@contextmanager  
def custom_context():  
    print("Before")  
    yield  
    print("After")  
  
with custom_context():  
    print("Inside")
```



# Metaclass – Tùy biến hành vi của class

Metaclass cho phép bạn thay đổi **cách class được tạo ra**.

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        dct['hello'] = lambda self: print("Hello from Meta")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass

obj = MyClass()
obj.hello()
```

Dùng nhiều trong frameworks như Django, SQLAlchemy

# Dynamic import (runtime import)

Giúp bạn import module hoặc class **dựa theo tên chuỗi**, rất mạnh khi viết plugin hoặc hệ thống mở rộng.

```
import importlib

module = importlib.import_module("math")
print(module.sqrt(16)) # 4.0
```

Hoặc:

```
def dynamic_import(path):
    module_path, class_name = path.rsplit(".", 1)
    module = importlib.import_module(module_path)
    return getattr(module, class_name)
```

# Signal/Event Hooks

Frappe/Django dùng signal để gọi logic khi có sự kiện xảy ra.

```
from frappe.model.document import Document
from frappe import hooks

def my_custom_validate(doc, method):
    print(f"Validating {doc.name}")

# hooks.py
doc_events = {
    "Sales Invoice": {
        "validate": "my_app.custom.my_custom_validate"
    }
}
```

Bạn có thể tự tạo hệ thống signal trong Python bằng cách dùng `blinker` hoặc viết thủ công.

# Strategy Pattern với Dictionary

Dùng dict để map các hàm theo key, tiện xử lý logic thay vì `if-elif` dài dòng:

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

operations = {
    "add": add,
    "sub": sub
}

result = operations["add"](3, 4) # 7
```

# Caching / Memoization

Dùng để lưu kết quả tạm để tránh tính toán lại:

```
from functools import lru_cache
```

```
@lru_cache(maxsize=128)
```

```
def slow_function(x):
```

```
    print(f"Calculating {x}")
```

```
    return x * x
```

# Metadata

**Metadata** nghĩa là "**dữ liệu về dữ liệu**" — tức là thông tin mô tả về một dữ liệu nào đó.

Hay nói cách khác: Metadata **không phải là nội dung chính**, mà là **thông tin bổ sung để mô tả hoặc hướng dẫn cách sử dụng dữ liệu đó**.

## Ví dụ dễ hiểu

### Ví dụ 1: Hình ảnh

- Dữ liệu: một bức ảnh `.jpg`
- Metadata:
  - Kích thước ảnh: 1920x1080
  - Ngày chụp: 2024-04-17
  - Thiết bị: iPhone 13
  - GPS: Hà Nội, Việt Nam

Bạn không **nhìn thấy metadata trong ảnh**, nhưng phần mềm máy ảnh và thư viện ảnh dùng **metadata để sắp xếp, hiển thị, tìm kiếm ảnh**.

### Ví dụ 2: Bài hát MP3

- Dữ liệu: file nhạc `.mp3`
- Metadata:
  - Tên bài hát
  - Ca sĩ
  - Album
  - Thời lượng

Các ứng dụng như Zing MP3, Spotify dùng metadata để hiển thị thông tin bài hát mà không cần phát nó.

Trong Python

```
from typing import Annotated
```

```
Age = Annotated[int, "Độ tuổi phải lớn hơn 0"]
```

- `int` là kiểu dữ liệu chính

- "Độ tuổi phải lớn hơn 0" là metadata — mô tả thêm, nhưng **không ảnh hưởng tới chạy code**
- Framework như **Pydantic**, **LangGraph**, v.v. sẽ đọc phần metadata này để **thực hiện kiểm tra, xử lý tự động, hoặc hiển thị thông tin**

## Ví dụ 3:

Giả sử bạn đang xây dựng một API đăng ký người dùng:

```
from pydantic import BaseModel, Field, EmailStr

class User(BaseModel):
    name: str = Field(..., title="Tên đầy đủ", min_length=3, max_length=50, description="Họ tên người dùng")
    email: EmailStr = Field(..., description="Email hợp lệ để xác thực")
    age: int = Field(..., gt=0, le=120, description="Tuổi phải lớn hơn 0 và không quá 120")
```

**Pydantic sử dụng metadata này để:**

- Tự động kiểm tra dữ liệu
- Sinh thông báo lỗi rõ ràng
- (Khi dùng với FastAPI) tự tạo ra tài liệu Swagger đẹp & chi tiết

**Kết hợp với FastAPI để thấy Metadata hoạt động**

```
from fastapi import FastAPI
from pydantic import BaseModel, Field, EmailStr

app = FastAPI()

class User(BaseModel):
    name: str = Field(..., title="Tên", description="Tên người dùng", min_length=3)
    email: EmailStr = Field(..., description="Email hợp lệ")
    age: int = Field(..., gt=0, le=100, description="Tuổi (0-100)")

@app.post("/register")
def register(user: User):
    return {"message": "Đăng ký thành công", "data": user}
```

FastAPI sẽ:

- Hiển thị các metadata như title, description, min\_length, gt, v.v.
- Kiểm tra lỗi nếu người dùng nhập sai format

- Dùng metadata để gợi ý giao diện form

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**



# lambda

`lambda` trong Python là một **hàm ẩn danh (anonymous function)** – tức là một hàm không cần đặt tên.

Nó thường được dùng khi bạn cần một hàm nhỏ, nhanh gọn, và chỉ dùng **một lần** hoặc **dùng ngay tại chỗ** (ví dụ: trong `sorted`, `map`, `filter`, `reduce`, v.v.)

```
lambda arguments: expression
```

- **arguments**: là danh sách các đối số (giống như trong `def`)
- **expression**: là giá trị sẽ được trả về (chỉ 1 dòng)

Ví dụ cơ bản

```
add = lambda x, y: x + y
print(add(2, 3)) # 5
```

```
#Tương đương với:
def add(x, y):
    return x + y
```

## Ứng dụng phổ biến

### 2. Dùng trong `sorted` để sắp xếp theo key

```
items = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
sorted_items = sorted(items, key=lambda item: item[1])
print(sorted_items) # [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
```

### 2. Dùng với `map()`

```
nums = [1, 2, 3]
squared = list(map(lambda x: x**2, nums))
print(squared) # [1, 4, 9]
```

### 3. Dùng với `filter()`

```
nums = [1, 2, 3, 4, 5]
even = list(filter(lambda x: x % 2 == 0, nums))
print(even) # [2, 4]
```

#### 4. Dùng trực tiếp (one-time use)

```
print((lambda name: f"Hello, {name}!")("VHTSoft")) # Hello, VHTSoft!
```

## Lưu ý

- `lambda` **chỉ dùng cho các hàm ngắn gọn, một dòng**.
- Không thể có nhiều dòng hoặc câu lệnh `if`, `for`, `while` trong thân `lambda`.
- Khi logic phức tạp hơn, bạn nên dùng `def` để code dễ đọc hơn.

## Một ví dụ thực tế kết hợp `lambda` và `Callable`

```
from typing import Callable

def run_operation(func: Callable[[int], int], number: int) -> int:
    return func(number)

print(run_operation(lambda x: x * 10, 5)) # 50
```

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**

# functools

`functools` là một **module chuẩn trong Python**, cung cấp các **công cụ giúp thao tác và cải tiến hàm** – như lưu cache kết quả, tạo decorator, hay cố định đối số cho hàm.

“Nói đơn giản: `functools` giúp bạn **viết code ngắn gọn hơn, tối ưu hơn và linh hoạt hơn** khi xử lý hàm.

## Một số hàm nổi bật trong `functools`

`@lru_cache` - Ghi nhớ kết quả hàm (caching)

```
from functools import lru_cache

@lru_cache(maxsize=100)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

print(fib(35)) # Rất nhanh nhờ cache
```

**Tăng tốc đáng kể** cho các hàm đệ quy hoặc tính toán nặng lặp lại nhiều lần.

`partial()` - Gán sẵn một số đối số cho hàm

```
from functools import partial

def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)

print(square(4)) # 16
```

Tạo các hàm "rút gọn" rất tiện khi dùng với `map`, `filter`, GUI callback,...

## `wraps()` - Bảo toàn metadata khi viết decorator

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Before function")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def greet():
    """This is a greeting function"""
    print("Hello!")

print(greet.__name__) # [] greet (không bị đổi thành 'wrapper')
print(greet.__doc__) # [] This is a greeting function
```

Giữ lại tên hàm, docstring,... khi bạn tạo custom decorators.

## `reduce()` - Áp dụng hàm tích lũy lên phần tử của list

```
from functools import reduce

nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product) # [] 24
```

Dùng cho các phép cộng dồn, nhân dồn,... giống `fold` trong functional programming.

### Khi nào nên dùng `functools`

Tình huống	Dùng gì từ <code>functools</code>
Cần cache hàm đệ quy hoặc gọi lặp	<code>@lru_cache</code>
Tạo decorator tùy chỉnh	<code>@wraps</code>
Cần tạo hàm mới từ hàm cũ, rút gọn tham số	<code>partial()</code>
Muốn tính toán tích lũy trên list	<code>reduce()</code>

Tác giả: **Đỗ Ngọc Tú**  
Công Ty Phần Mềm **VHTSoft**



# MyPy

MyPy là một **trình kiểm tra kiểu tĩnh (static type checker)** cho ngôn ngữ lập trình **Python**. Mặc dù Python là một ngôn ngữ **động (dynamically typed)**, từ Python 3.5 trở đi, bạn có thể sử dụng **type hints (chú thích kiểu)** để khai báo kiểu biến, hàm, v.v. MyPy giúp bạn **phân tích mã nguồn và phát hiện lỗi kiểu dữ liệu** mà không cần phải chạy chương trình.

## Mục tiêu chính của MyPy

- Giúp **phát hiện lỗi sớm** trong quá trình phát triển.
- **Tăng tính an toàn** của mã nguồn.
- **Cải thiện trải nghiệm lập trình**, đặc biệt khi làm việc nhóm hoặc dự án lớn.
- Kết hợp tốt với các công cụ IDE như VSCode, PyCharm.

Ví dụ đơn giản

```
def greet(name: str) -> str:
    return "Hello, " + name

greet(123) # Sai kiểu, nhưng Python vẫn chạy được
```

Python sẽ không báo lỗi khi chạy, nhưng MyPy sẽ phát hiện:

```
error: Argument 1 to "greet" has incompatible type "int"; expected "str"
```

## Cài đặt MyPy

```
pip install mypy
```

## Cách sử dụng

```
def add(a: int, b: int) -> int:
    return a + b
```

kiểm tra

```
mypy example.py
```

# Thiết lập MyPy cho một dự án thực tế

Giả sử bạn có một dự án như sau:

```
my_project/  
├─ main.py  
├─ utils/  
│   └─ __init__.py  
│   └─ math_tools.py  
└─ requirements.txt
```

## Cài đặt MyPy

```
pip install mypy
```

## Viết chú thích kiểu (type hints)

Ví dụ trong `math_tools.py`:

```
# utils/math_tools.py  
  
def multiply(a: int, b: int) -> int:  
    return a * b  
  
def divide(a: float, b: float) -> float:  
    if b == 0:  
        raise ValueError("Cannot divide by zero")  
    return a / b
```

Và trong `main.py`:

```
# main.py  
  
from utils.math_tools import multiply, divide  
  
result1 = multiply(4, 5)  
result2 = divide(10.0, 2.0)  
  
print("Multiply:", result1)  
print("Divide:", result2)
```

Tạo file cấu hình `mypy.ini`

Tại gốc dự án, tạo file `mypy.ini`

```
[mypy]
python_version = 3.10
ignore_missing_imports = True
disallow_untyped_defs = True
check_untyped_defs = True
strict_optional = True
warn_unused_ignores = True
```

- `ignore_missing_imports = True`: Bỏ qua lỗi khi thiếu kiểu từ thư viện bên ngoài (ví dụ: thư viện không có type stub).
- `disallow_untyped_defs = True`: Bắt buộc tất cả các hàm phải có chú thích kiểu.
- `check_untyped_defs = True`: Kiểm tra cả hàm không có type hints.
- `strict_optional = True`: Bật kiểm tra biến có thể là `None`.

Kiểm tra bằng MyPy

```
mypy .
```

# Tích hợp vào quy trình CI hoặc pre-commit

**Bạn có thể thêm vào CI/CD hoặc Git hook để đảm bảo mọi thay đổi đều được kiểm tra type:**

**Ví dụ dùng pre-commit hook:**

```
# .pre-commit-config.yaml
- repo: https://github.com/pre-commit/mirrors-mypy
  rev: v1.8.0 #rev viết tắt của revision, dùng để chỉ phiên bản (version) của repo của bạn
  hooks:
    - id: mypy
```

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**





# yield

`yield` là một **từ khóa** trong Python, dùng để **tạm dừng một hàm và trả về một giá trị**, nhưng không **kết thúc hàm** như `return`.

Khi hàm sử dụng `yield`, nó trở thành một **generator function** – mỗi lần bạn gọi `next()`, hàm tiếp tục chạy từ chỗ đã dừng.

## Ví dụ đơn giản dùng yield

```
def count_up_to(n):  
    i = 1  
    while i <= n:  
        yield i  
        i += 1  
  
gen = count_up_to(3)  
  
for num in gen:  
    print(num)
```

Mỗi lần lặp, hàm `count_up_to` chạy đến `yield`, trả về `i`, rồi "ngủ đông" – đến vòng lặp tiếp theo thì tiếp tục chạy tiếp.

## Kết quả

```
1  
2  
3
```

Ví dụ không dùng yield, chúng ta phải dùng danh sách để lưu vì vậy nếu dữ liệu lớn thì cũng phải dùng nhiều RAM

```
def count_up_to(n):  
    result = []  
    i = 1  
    while i <= n:  
        result.append(i)  
        i += 1
```

```
return result
```

```
nums = count_up_to(3)
```

```
for num in nums:  
    print(num)
```

## `yield` khác gì với `return` ?

<code>return</code>	<code>yield</code>
Trả về <b>một giá trị duy nhất</b>	Trả về <b>một chuỗi giá trị (generator)</b>
<b>Kết thúc</b> hàm ngay lập tức	Tạm dừng, giữ trạng thái và tiếp tục lần sau
Dùng để <b>hoàn thành</b> một tác vụ	Dùng để <b>tạo dòng dữ liệu</b> dần dần

## Lợi ích của `yield`

- **Tiết kiệm bộ nhớ:** Không cần lưu toàn bộ danh sách trong RAM.
- **Hiệu suất cao:** Lười biếng – chỉ tính toán khi cần.
- **Rất hữu ích** khi làm việc với **file lớn, dữ liệu streaming, API phân trang**, v.v.

### Ví dụ thực tế - đọc file lớn

```
def read_large_file(filename):  
    with open(filename) as f:  
        for line in f:  
            yield line.strip()  
  
for line in read_large_file("data.txt"):  
    print(line)
```

Nếu `data.txt` chứa hàng triệu dòng, thì `yield` giúp đọc **từng dòng một** mà không làm đầy bộ nhớ.

### Bạn có thể dùng `next()` để lấy từng giá trị từ generator:

```
gen = count_up_to(3)  
print(next(gen)) # 1  
print(next(gen)) # 2
```

Khi hết giá trị, nó sẽ raise `StopIteration`.

### So sánh tổng quát:

Tiêu chí	Dùng <code>yield</code> (Generator)	Không dùng <code>yield</code> (Trả về list)
Bộ nhớ	Cực kỳ tiết kiệm (chỉ sinh 1 giá trị/lần)	Tốn bộ nhớ (lưu toàn bộ kết quả)
Hiệu suất	Cao khi xử lý dữ liệu lớn hoặc stream	Kém hơn với dữ liệu lớn
Tốc độ khởi tạo	Nhanh, không tính toán ngay	Tính toán toàn bộ trước
Dễ debug/logging	Hơi khó hơn một chút	Dễ quan sát dữ liệu
Tính liên tục (streaming)	Rất phù hợp (ví dụ đọc file, API nhiều trang)	Không phù hợp

**Tác giả: Đỗ Ngọc Tú**  
**Công Ty Phần Mềm VHTSoft**

# yield nâng cao

`yield` **nâng cao** – tức là những cách dùng `yield` ở mức cao hơn, như:

1. `yield` như **kênh giao tiếp 2 chiều**
2. `yield from` để **lồng generator**
3. `generator pipeline` – **xử lý dữ liệu từng bước**
4. `send()` và `close()` – điều khiển luồng nâng cao

## 1. yield như một kênh 2 chiều

Không chỉ dùng để trả về giá trị, `yield` còn có thể nhận giá trị từ bên ngoài bằng cách kết hợp với `send()`.

```
def echo():
    while True:
        x = yield
        print("Nhận được:", x)

g = echo()
next(g)      # Khởi động generator
g.send("Hello") # => Nhận được: Hello
g.send("World") # => Nhận được: World
```

## 2. yield from - Lồng generator

Nếu bạn có nhiều generator con, bạn có thể "gom" chúng lại bằng **yield from**

```
def numbers():
    yield from [1, 2, 3]

def letters():
    yield from ['a', 'b', 'c']

def combo():
    yield from numbers()
    yield from letters()

for val in combo():
```

```
print(val)
```

Kết quả:

```
1
2
3
a
b
c
```

### 3. Pipeline generator - Xử lý dữ liệu từng bước

Bạn có thể xâu chuỗi nhiều generator để tạo pipeline xử lý dữ liệu rất gọn gàng:

```
def read_lines(lines):
    for line in lines:
        yield line

def to_upper(lines):
    for line in lines:
        yield line.upper()

def starts_with_a(lines):
    for line in lines:
        if line.startswith('A'):
            yield line

data = ["apple", "banana", "Avocado", "Apricot", "mango"]
pipeline = starts_with_a(to_upper(read_lines(data)))

for line in pipeline:
    print(line)
```

```
APPLE
AVOCADO
APRICOT
```

Pipeline này giống như filter + map + reduce nhưng **tiết kiệm RAM và rất "Pythonic"!**

### 4. send và close nâng cao

Gửi dữ liệu vào generator

```
def running_total():
    total = 0
    while True:
        x = yield total
        if x is None:
            break
        total += x

gen = running_total()
print(next(gen))    # 0
print(gen.send(5))  # 5
print(gen.send(3))  # 8
gen.close()         # Dừng generator
```

Tổng kết

Tính năng	Mô tả ngắn
yield	Trả về giá trị từng bước
yield from	Kết hợp nhiều generator
send(value)	Gửi dữ liệu vào generator
close()	Đóng generator
Generator pipeline	Kết hợp xử lý dữ liệu theo luồng

Tác giả: **Đỗ Ngọc Tú**  
Công Ty Phần Mềm **VHTSoft**

# Kỹ thuật hay và gọn gàng khi thao tác với danh sách (list)

## 1. **List Comprehension** – Viết gọn vòng lặp trong danh sách

Cơ bản:

```
numbers = [1, 2, 3, 4, 5]
squared = [x * x for x in numbers]
print(squared) # [1, 4, 9, 16, 25]
```

Có điều kiện:

```
evens = [x for x in numbers if x % 2 == 0]
print(evens) # [2, 4]
```

## 2. **enumerate()** – Lặp qua danh sách có chỉ số

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

```
0 apple
1 banana
2 cherry
```

## 3. **zip()** – Lặp qua nhiều danh sách cùng lúc

```
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 90, 95]

for name, score in zip(names, scores):
    print(f"{name} got {score}")
```



```
Alice got 85
Bob got 90
Charlie got 95
```

#### 4. Lặp ngược - `reversed()`

```
nums = [1, 2, 3, 4]
for x in reversed(nums):
    print(x)
```

#### 5. Lặp theo chỉ số cách quãng - `range(start, stop, step)`

```
for i in range(0, 10, 2): # 0 2 4 6 8
    print(i)
```

#### 6. Nested loop (vòng lặp lồng nhau)

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

#### 7. List comprehension nâng cao (nested)

```
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [num for row in matrix for num in row]
print(flattened) # [1, 2, 3, 4, 5, 6]
```

#### 8. Sử dụng `map()` và `filter()` - Tính hàm học functional

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x * x, nums))
evens = list(filter(lambda x: x % 2 == 0, nums))

print(squared) # [1, 4, 9, 16]
print(evens) # [2, 4]
```

Tác giả: **Đỗ Ngọc Tú**  
Công Ty Phần Mềm **VHTSoft**

# Pydantic

**Pydantic** là một thư viện Python dùng để:

- Tạo các **class dữ liệu kiểu an toàn (type-safe)**
- **Tự động validate dữ liệu** (rất mạnh)
- Hỗ trợ dễ dàng **chuyển từ dict → object và ngược lại**
- Được dùng rất phổ biến trong **FastAPI, Django**, các hệ thống lớn

Cài đặt

```
pip install pydantic
```

## So sánh nhanh với dataclass

Tính năng	dataclass	pydantic
Tự validate dữ liệu	❌	❌
Chuyển dict dễ	❌	❌ (tốt hơn)
Bảo vệ kiểu dữ liệu	❌	❌
Xử lý dữ liệu lỏng	Thủ công	❌ Tự động
Hỗ trợ JSON	Thủ công	❌

### Ví dụ cơ bản với BaseModel

```
from pydantic import BaseModel

class Person(BaseModel):
    name: str
    age: int

p = Person(name="Linh", age=25)
print(p)
#name='Linh' age=25
```

### Tự động validate dữ liệu

```
p = Person(name="Linh", age="25")
print(p.age) # ❌ Tự động convert '25' → 25 (int)
```

Nếu không thể convert được, sẽ báo lỗi:

```
p = Person(name="Linh", age="abc") # ❌ ValidationError
```

## Chuyển đổi giữa dict và JSON

```
# Chuyển thành dict
print(p.dict())

# Chuyển thành JSON
print(p.json())
```

## Lồng model bên trong model

```
class Address(BaseModel):
    city: str
    country: str

class User(BaseModel):
    username: str
    address: Address

u = User(username="nam", address={"city": "Hanoi", "country": "VN"})
print(u)
```

Pydantic tự động biến dict thành object `Address` – bạn không cần viết tay như dataclass.

## Tùy chỉnh kiểm tra nâng cao (validators)

```
from pydantic import validator

class Person(BaseModel):
    name: str
    age: int

    @validator('age')
    def age_must_be_positive(cls, v):
        if v <= 0:
            raise ValueError('Age must be positive')
        return v
```



# Dataclass

## 1. Bản chất của `@dataclass` là gì?

`@dataclass` là một decorator dùng để **tự động tạo các phương thức đặc biệt** cho một class:

- `__init__`: hàm khởi tạo
- `__repr__`: biểu diễn đối tượng dưới dạng chuỗi
- `__eq__`: so sánh hai đối tượng
- `__lt__`, `__le__`, `__gt__`, `__ge__` (nếu `order=True`)
- `__hash__` (nếu `frozen=True`)

☐ Điều này giúp bạn tiết kiệm rất nhiều công sức khi làm việc với các lớp dữ liệu thuần túy (plain data objects).

## 2. So sánh class bình thường vs dataclass

### Ví dụ: Class bình thường

```
class Product:
    def __init__(self, name, price, in_stock=True):
        self.name = name
        self.price = price
        self.in_stock = in_stock

    def __repr__(self):
        return f"Product(name={self.name!r}, price={self.price!r}, in_stock={self.in_stock!r})"

    def __eq__(self, other):
        return (self.name, self.price, self.in_stock) == (other.name, other.price, other.in_stock)
```

Với `@dataclass`:

```
from dataclasses import dataclass

@dataclass
class Product:
    name: str
    price: float
```

```
in_stock: bool = True
```

### 3. Cấu hình nâng cao

`frozen=True`: bất biến (immutable)

aaa

```
@dataclass(frozen=True)
class User:
    username: str
    email: str

u = User("alice", "alice@example.com")
u.username = "bob" # ❌ Sẽ raise FrozenInstanceError
```

`order=True`: thêm các toán tử so sánh

```
@dataclass(order=True)
class Point:
    x: int
    y: int

Point(1, 2) < Point(2, 1) # ❌ True
```

`init=False`: loại bỏ `__init__`

```
@dataclass
class Token:
    value: str
    secret: str = "secret"
    valid: bool = field(init=False, default=True)
```

`init=False` giúp loại bỏ trường đó khỏi constructor.

#### 4. `field()` – Tùy chỉnh thuộc tính

```
from dataclasses import dataclass, field
from typing import List

@dataclass
class Order:
```

```
items: List[str] = field(default_factory=list)
```

Lý do dùng `default_factory=list` thay vì `items: List[str] = []` là vì:

- Trong Python, **giá trị mặc định là list (hoặc dict) dùng chung** giữa các thể hiện khác nhau nếu không cẩn thận.
- `default_factory` đảm bảo mỗi instance có list riêng biệt.

## 5. Post-init xử lý: `__post_init__`

Được gọi **ngay sau** `__init__` do `@dataclass` tạo.

```
@dataclass
class Product:
    name: str
    price: float

    def __post_init__(self):
        if self.price < 0:
            raise ValueError("Price must be non-negative")
```

## 6. So sánh sâu (deep comparison)

`dataclass` hỗ trợ `__eq__` mặc định — so sánh theo từng field, không theo địa chỉ bộ nhớ.

```
a = Product("Pen", 1.5)
b = Product("Pen", 1.5)
print(a == b) # True
```

## 7. Kế thừa với `dataclass`

```
@dataclass
class Animal:
    name: str

    @dataclass
    class Dog(Animal):
        breed: str
```

Bạn có thể kế thừa như class bình thường.

## 8. Kiểm soát `repr`, `eq`, `compare`, `hash` từng field

```
@dataclass
```

```
class User:
```

```
    username: str = field(compare=True)
```

```
    password: str = field(repr=False, compare=False)
```

- `repr=False`: ẩn khỏi `__repr__`
- `compare=False`: không dùng trong `__eq__` hay `__lt__`

## 9. Chuyển đổi `dataclass` thành dict

```
from dataclasses import asdict
```

```
p = Product("Mouse", 12.5)
```

```
print(asdict(p)) # {'name': 'Mouse', 'price': 12.5, 'in_stock': True}
```

## 10. Hạn chế của `dataclass`

- Không thay thế hoàn toàn ORM như Django Models (chỉ dùng cho logic nghiệp vụ/data layer)
- Không hỗ trợ property getter/setter tự động
- Không hỗ trợ kế thừa nhiều mức quá phức tạp (vẫn dùng được nhưng cần cẩn thận)

Nếu bạn đang làm dự án lớn với kiến trúc như DDD (Domain Driven Design), `dataclass` thường dùng để:

- Định nghĩa **DTO (Data Transfer Object)**
- Làm **request/response schema**
- Tạo các lớp đại diện cho **giá trị** (value objects)