

# Kiểu dữ liệu

- Literal
- TypedDict
- Enum
- Union (hoặc | từ Python 3.10+)
- Optional
- Any
- Callable
- Annotated
- NewType
- Final
- Self (Python 3.11+)
- Generic

# Literal

Literal là một tính năng được cung cấp bởi module `typing` (hoặc `typing_extensions` nếu bạn dùng Python < 3.8), cho phép bạn **giới hạn giá trị của một biến** chỉ trong một số **giá trị cố định cụ thể**.

## Cú pháp cơ bản:

```
from typing import Literal

def get_drink(choice: Literal["coffee", "tea"]) -> str:
    return f"You chose {choice}"
```

## Cách hoạt động

Biến `choice` trong ví dụ trên **chỉ được phép nhận một trong hai giá trị**: `"coffee"` hoặc `"tea"`. Nếu bạn truyền vào `"beer"`, IDE hoặc trình kiểm tra kiểu (như `mypy`) sẽ báo lỗi.

## Mục đích của Literal

- Tăng **tính an toàn** trong lập trình: tránh nhập nhầm giá trị
- Tăng **tự động gợi ý (autocomplete)** trong IDE
- Rất hữu ích khi kết hợp với `TypedDict`, đặc biệt trong hệ thống như LangGraph, Pydantic, FastAPI, v.v.

```
from typing import Literal

DrinkType = Literal["coffee", "tea", "unknown"]

def order(drink: DrinkType):
    print(f"You ordered: {drink}")

order("coffee") # 
order("beer")   #  IDE sẽ cảnh báo!
```

Ví dụ kết hợp `Literal` + `TypedDict`

```
from typing import TypedDict, Literal

# Định nghĩa một kiểu trạng thái đơn giản cho app chọn đồ uống
class DrinkState(TypedDict):
    preference: Literal["coffee", "tea", "unknown"]
    strength: Literal["light", "medium", "strong"]

# Trạng thái hợp lệ
state: DrinkState = {
    "preference": "coffee",
    "strength": "strong"
}

# Trạng thái sai (sẽ bị mypy hoặc IDE cảnh báo)
invalid_state: DrinkState = {
    "preference": "beer",      # ❌ không phải giá trị hợp lệ
    "strength": "super strong" # ❌ không đúng literal
}
```

# TypedDict

`TypedDict` là một lớp đặc biệt trong module `typing` (hoặc `typing_extensions` cho Python < 3.8), cho phép bạn định nghĩa **dictionary có cấu trúc rõ ràng và kiểm soát kiểu dữ liệu** giống như một class.

## Cú pháp cơ bản:

```
from typing import TypedDict

class Person(TypedDict):
    name: str
    age: int
```

Ở đây, `Person` là một dictionary mà:

- `name` phải là chuỗi (`str`)
- `age` phải là số nguyên (`int`)

```
def greet(person: Person):
    print(f"Hello {person['name']}, you are {person['age']} years old.")

greet({"name": "Alice", "age": 30}) # ✅ hợp lệ
greet({"name": "Bob", "age": "thirty"}) # ❌ IDE hoặc mypy sẽ cảnh báo
```

## So sánh với dict bình thường:

Bình thường	TypedDict
Tự do về kiểu dữ liệu	Có kiểm tra kiểu
Không hỗ trợ autocomplete	Hỗ trợ autocomplete
Dễ gây lỗi do sai tên/kiểu	An toàn hơn

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

# Enum

Enum là một kiểu dữ liệu cho phép bạn định nghĩa **tập hợp các hằng số có tên**. Mỗi giá trị trong enum có một tên rõ ràng và không thể thay đổi.

## Khi nào dùng Enum?

- Khi bạn có một tập giá trị cố định, ví dụ: `"coffee"`, `"tea"`, `"unknown"`
- Để tránh dùng *magic strings* hoặc số không rõ ý nghĩa
- Khi bạn muốn code dễ đọc, dễ bảo trì, và hỗ trợ autocomplete

## Cách dùng Enum trong Python

```
from enum import Enum

class DrinkPreference(Enum):
    COFFEE = "coffee"
    TEA = "tea"
    UNKNOWN = "unknown"
```

Bây giờ, bạn có thể sử dụng:

```
choice = DrinkPreference.COFFEE

if choice == DrinkPreference.TEA:
    print("Bạn chọn trà.")
else:
    print("Không phải trà.")
```

Lấy danh sách giá trị enum

```
for drink in DrinkPreference:
    print(drink.name, "=", drink.value)

COFFEE = coffee
TEA = tea
UNKNOWN = unknown
```

## Enum kết hợp với TypedDict

```
from typing import TypedDict
from enum import Enum

class DrinkPreference(Enum):
    COFFEE = "coffee"
    TEA = "tea"
    UNKNOWN = "unknown"

class DrinkState(TypedDict):
    preference: DrinkPreference
```

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

# Union (hoặc | từ Python 3.10+)

`Union` cho phép **một biến hoặc một tham số có thể có nhiều kiểu giá trị**.

Nó đến từ module `typing`, và được dùng nhiều trong **type hinting** để mô tả biến có thể là `kiểu A` hoặc `kiểu B`.

```
from typing import Union

def func(x: Union[int, str]):
    ...
```

Từ Python 3.10 trở lên, bạn có thể dùng cú pháp ngắn hơn:

```
def func(x: int | str):
    ...
```

## Ví dụ thực tế: Gửi thông báo

```
from typing import Union

def send_notification(user: Union[int, str]):
    if isinstance(user, int):
        print(f"Gửi thông báo đến user ID: {user}")
    elif isinstance(user, str):
        print(f"Gửi thông báo đến username: {user}")
    else:
        print("Kiểu dữ liệu không hợp lệ")
```

## Giải thích:

- `user` có thể là `int` (ID người dùng) hoặc `str` (username).
- Hàm sẽ xử lý tùy theo kiểu dữ liệu được truyền vào.

```
send_notification(101)    # Gửi thông báo đến user ID: 101
```

```
send_notification("alice") # Gửi thông báo đến username: alice
```



# Optional

Giống như `Union[X, None]`, dùng cho giá trị có thể bị bỏ qua:

```
from typing import Optional

def greet(name: Optional[str] = None):
    if name:
        print(f"Chào bạn, {name}!")
    else:
        print("Chào bạn, người lạ!")
```

```
greet("Nam")    # In: Chào bạn, Nam!
greet()         # In: Chào bạn, người lạ!
```

## Kết hợp với TypedDict

```
from typing import TypedDict, Optional

class User(TypedDict):
    username: str
    email: Optional[str] # Email có thể không tồn tại

def display_email(user: User):
    if user["email"]:
        print("Email:", user["email"])
    else:
        print("Không có email.")
```

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

# Any

Biến có thể là bất kỳ kiểu gì – dùng khi không thể đoán trước kiểu hoặc muốn bỏ qua kiểm tra:

```
from typing import Any
```

```
def mystery(x: Any):
```

```
    ...
```

# Callable

Trong Python, `Callable` là một kiểu dữ liệu đặc biệt dùng để đại diện cho **một hàm (function)** hoặc **bất kỳ thứ gì có thể được gọi**, ví dụ: hàm, phương thức, hoặc class có định nghĩa `__call__`.

```
from typing import Callable

def run(callback: Callable[[int, str], bool]):
    ...
```

## Khi nào dùng `Callable`?

- Khi bạn truyền **hàm làm đối số** cho một hàm khác
- Khi bạn muốn đảm bảo rằng đầu vào là **một callable đúng định dạng**
- Giúp **IDE** và trình kiểm tra type hiểu đúng hàm bạn đang xử lý

```
Callable[[ArgType1, ArgType2, ...], ReturnType]
```

### Ví dụ cụ thể

Viết một hàm `run_operation` nhận vào một hàm (callable) và hai số, sau đó gọi hàm đó với hai số đó.

```
from typing import Callable

# Hàm chính nhận vào một Callable
def run_operation(operation: Callable[[int, int], int], a: int, b: int) -> int:
    return operation(a, b)

# Một số hàm cụ thể
def add(x: int, y: int) -> int:
    return x + y

def multiply(x: int, y: int) -> int:
    return x * y

# Gọi thử
```

```
print(run_operation(add, 3, 5))    # 8
print(run_operation(multiply, 3, 5)) # 15
```

## Giải thích

Thành phần	Ý nghĩa
<code>Callable[[int, int], int]</code>	Đây là một kiểu dữ liệu đại diện cho một <b>hàm nhận 2 số nguyên và trả về một số nguyên</b>
<code>operation(a, b)</code>	Hàm được truyền vào sẽ được gọi như bình thường
<code>add</code> & <code>multiply</code>	Là những hàm cụ thể bạn truyền vào

## Mở rộng

Bạn có thể dùng `Callable` với các hàm lambda, hoặc class có `__call__`:

```
class Subtract:
    def __call__(self, x: int, y: int) -> int:
        return x - y

sub = Subtract()
print(run_operation(sub, 10, 3)) # 7
```

## Tổng kết

- `Callable` dùng để **type-hint cho function** hoặc **object có thể gọi**
- Cú pháp là `Callable[[arg1_type, arg2_type], return_type]`
- Hữu ích trong lập trình hướng hàm (functional programming), LangGraph nodes, và callback logic

# Annotated

Dùng để thêm metadata cho type (hữu ích với Pydantic hoặc FastAPI):

```
from typing import Annotated
from pydantic import Field

age: Annotated[int, Field(gt=0, lt=120)]
```

# NewType

Tạo kiểu mới dựa trên kiểu cũ, nhưng giúp rõ ràng hơn về mặt ngữ nghĩa:

```
from typing import NewType

UserId = NewType("UserId", int)

def get_user(user_id: UserId):
    ...
```

# Final

Dùng để đánh dấu rằng biến hoặc class không nên bị ghi đè hoặc kế thừa:

```
from typing import Final
```

```
PI: Final = 3.14159
```

# Self (Python 3.11+)

Cho phép annotate chính class đang được định nghĩa:

```
class Counter:  
    def increment(self) -> Self:  
        ...
```



# Generic

Generics cho phép bạn **xác định kiểu dữ liệu một cách tổng quát**, giúp **viết code linh hoạt và an toàn hơn** về mặt kiểu dữ liệu – đặc biệt hữu ích khi viết class, hàm làm việc với nhiều kiểu dữ liệu khác nhau.

Python **hỗ trợ Generics**, đặc biệt từ phiên bản Python 3.5 trở đi, nhờ vào **typing module**.

## Cú pháp cơ bản với `Generic` trong Python

```
from typing import TypeVar, Generic, List

T = TypeVar("T") # T là một kiểu bất kỳ

class Box(Generic[T]):
    def __init__(self, content: T):
        self.content = content

    def get_content(self) -> T:
        return self.content

# Dùng cụ thể
box_int = Box
box_str = Box[str]("hello")

print(box_int.get_content()) # [] 123
print(box_str.get_content()) # [] "hello"
```

`T` ở đây có thể là `int`, `str`, `List[str]`, hay bất cứ kiểu nào bạn muốn

## Ứng dụng thực tế

### 1. Với hàm:

```
from typing import TypeVar

T = TypeVar("T")

def identity(x: T) -> T:
```

```
return x

print(identity(5))    # int
print(identity("test")) # str
```

2. Với container:

```
from typing import List

def first_item(items: List[T]) -> T:
    return items[0]
```

Một số generic types hay dùng trong `typing`

Type	Ý nghĩa
<code>List[T]</code>	Danh sách chứa các phần tử kiểu <code>T</code>
<code>Dict[K, V]</code>	Dictionary với key là <code>K</code> , value là <code>V</code>
<code>Optional[T]</code>	Có thể là <code>T</code> hoặc <code>None</code>
<code>Union[T1, T2, ...]</code>	Có thể là một trong các kiểu liệt kê
<code>Tuple[T1, T2]</code>	Tuple với các phần tử theo thứ tự kiểu
<code>Callable[[T1, T2], R]</code>	Hàm nhận <code>T1</code> , <code>T2</code> trả về <code>R</code>
<code>Generic[T]</code>	Dùng để tạo class generic

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