Python Programming - Functions (Hàm)

> **Mục tiêu học tập**: Nắm vững cách tạo và sử dụng hàm để tổ chức code hiệu quả, tái sử dụng logic, và quản lý scope của biến.

1. Tại Sao Cần Functions?

1.1. Đặt Vấn Đề

Hãy xem xét chương trình tính diện tích hình chữ nhật:

```
# Tính diện tích lần 1
length1 = 5
width1 = 3
area1 = length1 * width1
print(f"Diện tích 1: {area1}")

# Tính diện tích lần 2
length2 = 8
width2 = 4
area2 = length2 * width2
print(f"Diện tích 2: {area2}")

# Tính diện tích lần 3
length3 = 10
width3 = 6
area3 = length3 * width3
print(f"Diện tích 3: {area3}")
```

Vấn đề:

- Code lặp lại nhiều lần (vi phạm DRY - Don't Repeat Yourself)
- Khó bảo trì: nếu công thức thay đổi, phải sửa ở nhiều chỗ
- Dễ mắc lỗi khi copy-paste

1.2. Giải Quyết: Functions

Function (hàm) là một khối code có thể **tái sử dụng**, thực hiện một nhiệm vụ cụ thể.

```
def calculate_rectangle_area(length, width):
    area = length * width
    return area

# Sử dụng hàm
area1 = calculate_rectangle_area(5, 3)
area2 = calculate_rectangle_area(8, 4)
area3 = calculate_rectangle_area(10, 6)
```

```
print(f"Diện tích 1: {area1}")
print(f"Diện tích 2: {area2}")
print(f"Diện tích 3: {area3}")
```

Lợi ích:

- ☑ Code ngắn gọn, dễ đọc
- ☑ Tái sử dụng nhiều lần
- ☑ Dễ bảo trì và debug
- ☑ Tổ chức code có cấu trúc

---

2. Định Nghĩa Hàm với `def`

2.1. Cú Pháp Cơ Bản

```
def tên_hàm(tham_số_1, tham_số_2, ...):
    """Docstring: Mô tả hàm làm gì"""
    # Khối lệnh
    câu_lệnh_1
    câu_lệnh_2
    return giá_trị_trả_về
```

Thành phần:

1. `def`: Từ khóa khai báo hàm
2. **Tên hàm**: Theo quy tắc snake_case (giống biến)
3. **Parameters (tham số)**: Dữ liệu đầu vào (có thể không có)
4. **Dấu** `:`: Bắt buộc
5. **Docstring**: Chuỗi mô tả hàm (tùy chọn nhưng nên có)
6. **Body**: Khối code thực thi
7. `return`: Trả về kết quả (tùy chọn)

2.2. Ví Dụ Đơn Giản

Hàm không có tham số, không return

```
def greet():
    """In ra lời chào"""
    print("Xin chào!")
    print("Chào mừng đến với Python!")

# Gọi hàm
greet()
# Output:
```

```
# Xin chào!
# Chào mừng đến với Python!
```

Hàm có tham số, không return

```
def greet_person(name):
    """In lời chào với tên cụ thể"""
    print(f"Xin chào, {name}!")
    print(f"Rất vui được gặp bạn!")

# Gọi hàm với tham số
greet_person("Minh Đạo")
# Output:
# Xin chào, Minh Đạo!
# Rất vui được gặp bạn!
```

Hàm có return

```
def add(a, b):
    """Tính tổng hai số"""
    result = a + b
    return result

# Gọi hàm và lưu kết quả
total = add(10, 20)
print(f"Tổng: {total}")  # Tổng: 30

# Hoặc dùng trực tiếp
print(f"5 + 7 = {add(5, 7)}")  # 5 + 7 = 12
```

---

3. Parameters (Tham Số) và Arguments (Đối Số)

3.1. Phân Biệt Parameter vs Argument

```
def calculate_bmi(weight, height):  # weight, height là PARAMETERS
    """Tính chỉ số BMI"""
    bmi = weight / (height ** 2)
    return bmi

# 70, 1.75 là ARGUMENTS
result = calculate_bmi(70, 1.75)
```

- **Parameters**: Biến trong định nghĩa hàm
- **Arguments**: Giá trị thực tế truyền vào khi gọi hàm

3.2. Positional Arguments (Đối số vị trí)

Đối số được truyền theo **thứ tự**.

```
def introduce(name, age, city):
    print(f"Tôi là {name}, {age} tuổi, sống ở {city}")

introduce("Minh Đạo", 25, "TP.HCM")
# Output: Tôi là Minh Đạo, 25 tuổi, sống ở TP.HCM

# ⚠ Thứ tự SAI
introduce(25, "Minh Đạo", "TP.HCM")
# Output: Tôi là 25, Minh Đạo tuổi, sống ở TP.HCM (Sai nghĩa!)
```

3.3. Keyword Arguments (Đối số từ khóa)

Đối số được truyền với **tên tham số**.

```
def introduce(name, age, city):
    print(f"Tôi là {name}, {age} tuổi, sống ở {city}")

# Dùng keyword arguments - KHÔNG phụ thuộc thứ tự
introduce(age=25, city="TP.HCM", name="Minh Đạo")
# Output: Tôi là Minh Đạo, 25 tuổi, sống ở TP.HCM

# Kết hợp cả hai (positional trước, keyword sau)
introduce("Minh Đạo", city="TP.HCM", age=25)
```

3.4. Default Parameters (Tham số mặc định)

Tham số có **giá trị mặc định** nếu không truyền vào.

```
def greet(name, greeting="Xin chào"):
    """In lời chào với greeting tùy chỉnh"""
    print(f"{greeting}, {name}!")

greet("Minh Đạo")                    # Xin chào, Minh Đạo!
greet("Minh Đạo", "Hello")           # Hello, Minh Đạo!
greet("Minh Đạo", greeting="Chào")   # Chào, Minh Đạo!
```

⚠ **Lưu ý:** Tham số có giá trị mặc định phải đặt **SAU** tham số bắt buộc.

```
# ☑ ĐÚNG
def create_user(username, role="user"):
    pass
```

```
# ❌ SAI - SyntaxError
def create_user(role="user", username):
    pass
```

3.5. Ví Dụ Thực Tế: Tính phí ship

```
def calculate_shipping(distance, express=False, insurance=False):
    """
    Tính phí ship dựa trên khoảng cách và dịch vụ

    Args:
        distance: Khoảng cách (km)
        express: Giao hàng nhanh (mặc định False)
        insurance: Bảo hiểm (mặc định False)

    Returns:
        Tổng phí ship (VND)
    """
    base_fee = distance * 5000

    if express:
        base_fee += 20000

    if insurance:
        base_fee += 10000

    return base_fee

# Các cách gọi
print(calculate_shipping(10))                              # 50,000
print(calculate_shipping(10, express=True))                # 70,000
print(calculate_shipping(10, express=True, insurance=True))  # 80,000
print(calculate_shipping(10, insurance=True))              # 60,000
```

---

4. Return Values (Giá Trị Trả Về)

4.1. `return` Đơn Giản

```
def square(n):
    """Tính bình phương"""
    return n ** 2

result = square(5)
print(result)  # 25
```

⚠ **Lưu ý:** Sau `return`, hàm **DỪNG NGAY**, không chạy code phía sau.

```python
def check_age(age):
    if age >= 18:
        return "Trưởng thành"
        print("Dòng này KHÔNG BAO GIỜ chạy")  # Unreachable code
    return "Vị thành niên"

print(check_age(20))  # Trưởng thành
```

4.2. Return Multiple Values (Trả về nhiều giá trị)

Python cho phép return nhiều giá trị dưới dạng **tuple**.

```python
def calculate_rectangle(length, width):
    """Tính diện tích và chu vi hình chữ nhật"""
    area = length * width
    perimeter = 2 * (length + width)
    return area, perimeter  # Trả về tuple

# Nhận cả hai giá trị
a, p = calculate_rectangle(5, 3)
print(f"Diện tích: {a}, Chu vi: {p}")  # Diện tích: 15, Chu vi: 16

# Hoặc nhận dưới dạng tuple
result = calculate_rectangle(5, 3)
print(result)  # (15, 16)
print(f"Diện tích: {result[0]}")
```

Ví dụ thực tế: Phân tích điểm

```python
def analyze_scores(scores):
    """Phân tích danh sách điểm"""
    total = sum(scores)
    average = total / len(scores)
    max_score = max(scores)
    min_score = min(scores)

    return total, average, max_score, min_score

scores = [8, 9, 7, 10, 6]
t, avg, mx, mn = analyze_scores(scores)

print(f"Tổng: {t}")
print(f"TB: {avg:.2f}")
print(f"Cao nhất: {mx}")
print(f"Thấp nhất: {mn}")
```

4.3. Return với Điều Kiện

```python
def get_grade(score):
    """Xếp loại học lực"""
    if score >= 9:
        return "Xuất sắc"
    elif score >= 8:
        return "Giỏi"
    elif score >= 6.5:
        return "Khá"
    elif score >= 5:
        return "Trung bình"
    else:
        return "Yếu"

print(get_grade(8.5))  # Giỏi
print(get_grade(4.0))  # Yếu
```

4.4. Return None (Ngầm định)

Nếu hàm **không có** `return` hoặc `return` không có giá trị, Python tự động return `None`.

```python
def say_hello(name):
    print(f"Hello, {name}!")
    # Không có return

result = say_hello("Minh Đạo")
print(result)  # None

# Tương đương với:
def say_hello_2(name):
    print(f"Hello, {name}!")
    return None
```

---

5. Scope của Biến (Local vs Global)

5.1. Đặt Vấn Đề

```python
x = 10  # Biến này ở đâu?

def test():
    x = 20  # Biến này ở đâu?
    print(x)

test()      # In ra 20 hay 10?
print(x)    # In ra 20 hay 10?
```

Câu hỏi: Hai biến `x` này có phải là một không?

5.2. Local Scope (Phạm vi cục bộ)

Biến được khai báo **bên trong hàm** chỉ tồn tại trong hàm đó.

```
def calculate():
    result = 10 * 5  # Biến LOCAL
    print(f"Trong hàm: {result}")

calculate()
# Output: Trong hàm: 50

print(result)  # ❌ NameError: name 'result' is not defined
```

Giải thích: Biến `result` chỉ tồn tại trong hàm `calculate()`, không thể truy cập từ bên ngoài.

5.3. Global Scope (Phạm vi toàn cục)

Biến được khai báo **ngoài tất cả hàm** là biến global, có thể truy cập từ mọi nơi.

```
PI = 3.14159  # Biến GLOBAL

def calculate_circle_area(radius):
    area = PI * radius ** 2  # Đọc biến global
    return area

print(calculate_circle_area(5))  # 78.53975
print(PI)  # 3.14159 - Vẫn truy cập được
```

5.4. Local "Che" Global

Nếu local và global cùng tên, **local ưu tiên**.

```
x = 10  # Global

def test():
    x = 20  # Local - CHE biến global
    print(f"Trong hàm: {x}")

test()
# Output: Trong hàm: 20

print(f"Ngoài hàm: {x}")
# Output: Ngoài hàm: 10 (Biến global không thay đổi)
```

5.5. Từ Khóa `global`

😳 Đặt vấn đề: Thay đổi biến global từ trong hàm

```
counter = 0  # Global

def increment():
    counter = counter + 1  # ❌ UnboundLocalError
    print(counter)

increment()
```

Lỗi: Python hiểu `counter` bên trái `=` là biến local mới, nhưng bên phải chưa được định nghĩa.

💡 Giải quyết: Dùng `global`

```
counter = 0  # Global

def increment():
    global counter  # Khai báo: tôi muốn dùng biến global
    counter = counter + 1
    print(f"Counter: {counter}")

increment()  # Counter: 1
increment()  # Counter: 2
increment()  # Counter: 3

print(f"Giá trị cuối: {counter}")  # 3
```

Ví dụ thực tế: Game với điểm số

```
score = 0
level = 1

def defeat_enemy():
    """Đánh bại kẻ thù, tăng điểm"""
    global score
    score += 10
    print(f"Đánh bại kẻ thù! Điểm: {score}")

def level_up():
    """Tăng cấp độ"""
    global level, score
    level += 1
    score += 50
    print(f"Lên cấp {level}! Thưởng 50 điểm. Tổng: {score}")

defeat_enemy()  # Điểm: 10
defeat_enemy()  # Điểm: 20
level_up()      # Lên cấp 2! Thưởng 50 điểm. Tổng: 70
```

5.6. ⚠ Lưu Ý Khi Dùng `global`

❌ Nên tránh dùng `global` vì:

- Khó debug (không biết biến thay đổi ở đâu)
- Khó test
- Vi phạm nguyên tắc "pure function"

☑ Thay vào đó:

```
# ❌ Cách không tốt
total = 0

def add_to_total(value):
    global total
    total += value

# ☑ Cách tốt hơn
def add_to_total(current_total, value):
    return current_total + value

total = 0
total = add_to_total(total, 10)
total = add_to_total(total, 20)
print(total)  # 30
```

5.7. Nested Scope (Hàm lồng nhau)

```
def outer():
    x = 10  # Biến của outer()

    def inner():
        x = 20  # Biến LOCAL của inner()
        print(f"Inner: {x}")

    inner()
    print(f"Outer: {x}")

outer()
# Output:
# Inner: 20
# Outer: 10
```

Từ khóa `nonlocal`

Dùng để thay đổi biến của **hàm bọc ngoài** (không phải global).

```python
def outer():
    count = 0

    def inner():
        nonlocal count  # Thay đổi biến của outer()
        count += 1
        print(f"Inner count: {count}")

    inner()
    inner()
    print(f"Outer count: {count}")

outer()
# Output:
# Inner count: 1
# Inner count: 2
# Outer count: 2
```

---

6. Các Loại Hàm Đặc Biệt

6.1. Pure Functions (Hàm thuần túy)

Định nghĩa: Hàm mà:

1. Output chỉ phụ thuộc vào input
2. Không có side effects (không thay đổi biến bên ngoài)

```python
# ☑ Pure function
def add(a, b):
    return a + b

# ❌ Không pure - có side effect
total = 0
def add_to_total(value):
    global total
    total += value
    return total
```

Lợi ích của pure functions:

- Dễ test
- Dễ debug
- Có thể cache kết quả
- An toàn khi chạy song song

6.2. Higher-Order Functions

Hàm nhận **hàm khác** làm tham số hoặc trả về hàm.

```
def apply_operation(x, y, operation):
    """Áp dụng operation lên x, y"""
    return operation(x, y)

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

# Truyền hàm làm tham số
result1 = apply_operation(5, 3, add)
print(result1)  # 8

result2 = apply_operation(5, 3, multiply)
print(result2)  # 15
```

6.3. Lambda Functions (Hàm ẩn danh)

Hàm ngắn gọn, định nghĩa trên 1 dòng, không cần `def`.

Cú pháp:

```
lambda tham_số: biểu_thức
```

Ví dụ:

```
# Function thông thường
def square(x):
    return x ** 2

# Lambda tương đương
square_lambda = lambda x: x ** 2

print(square(5))         # 25
print(square_lambda(5))  # 25
```

Dùng với `map()`, `filter()`, `sorted()`:

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

# map: áp dụng hàm lên từng phần tử
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# filter: lọc phần tử thỏa điều kiện
```

```
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4]

# sorted: sắp xếp theo key
students = [("An", 8), ("Bình", 9), ("Chi", 7)]
sorted_students = sorted(students, key=lambda s: s[1], reverse=True)
print(sorted_students)  # [('Bình', 9), ('An', 8), ('Chi', 7)]
```

6.4. Recursive Functions (Đệ quy)

Hàm **gọi chính nó**.

```
def factorial(n):
    """Tính n! bằng đệ quy"""
    if n == 0 or n == 1:
        return 1  # Base case
    return n * factorial(n - 1)  # Recursive case

print(factorial(5))  # 120 (5 * 4 * 3 * 2 * 1)
```

Ví dụ: Fibonacci

```
def fibonacci(n):
    """Số Fibonacci thứ n"""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

for i in range(10):
    print(fibonacci(i), end=" ")
# Output: 0 1 1 2 3 5 8 13 21 34
```

⚠ **Lưu ý:** Đệ quy phải có **base case** (điều kiện dừng), nếu không sẽ tràn stack.

---

7. Docstrings và Type Hints

7.1. Docstrings

Chuỗi mô tả hàm, dùng `"""..."""` ngay sau định nghĩa hàm.

```
def calculate_bmi(weight, height):
    """
    Tính chỉ số BMI (Body Mass Index).

    Args:
        weight (float): Cân nặng (kg)
```

```
        height (float): Chiều cao (m)

    Returns:
        float: Chỉ số BMI

    Example:
        >>> calculate_bmi(70, 1.75)
        22.857142857142858
    """
    return weight / (height ** 2)

# Xem docstring
print(calculate_bmi.__doc__)
help(calculate_bmi)
```

7.2. Type Hints (Python 3.5+)

Gợi ý kiểu dữ liệu cho tham số và return value.

```
def greet(name: str) -> str:
    """Trả về lời chào"""
    return f"Hello, {name}!"

def add(a: int, b: int) -> int:
    """Tính tổng hai số nguyên"""
    return a + b

# Type hints không bắt buộc, chỉ là gợi ý
result = add(5, 10)  # OK
result = add("5", "10")  # Vẫn chạy, nhưng editor sẽ cảnh báo
```

---

8. Best Practices

8.1. Nguyên Tắc Đặt Tên

```
# ☑ TỐT - Mô tả chức năng
def calculate_total_price(items):
    pass

def is_valid_email(email):
    pass

def get_user_by_id(user_id):
    pass

# ❌ KHÔNG TỐT - Không rõ nghĩa
def calc(x):
    pass
```

```
def do_stuff():
    pass

def func123():
    pass
```

8.2. Single Responsibility Principle

Mỗi hàm chỉ nên làm **một việc**.

```
# ❌ KHÔNG TỐT - Làm quá nhiều việc
def process_order(order):
    # Validate
    if not order.is_valid():
        return False
    # Tính toán
    total = calculate_total(order)
    # Lưu database
    save_to_db(order)
    # Gửi email
    send_confirmation_email(order)
    return True

# ☑ TỐT - Tách thành nhiều hàm nhỏ
def validate_order(order):
    return order.is_valid()

def calculate_order_total(order):
    return sum(item.price for item in order.items)

def save_order(order):
    # Lưu vào database
    pass

def send_order_confirmation(order):
    # Gửi email
    pass

def process_order(order):
    if not validate_order(order):
        return False

    order.total = calculate_order_total(order)
    save_order(order)
    send_order_confirmation(order)
    return True
```

8.3. Tránh Quá Nhiều Tham Số

```python
# ❌ KHÔNG TỐT - Quá nhiều tham số
def create_user(name, email, age, phone, address, city, country, zipcode):
    pass

# ☑ TỐT - Dùng dictionary hoặc object
def create_user(user_data):
    name = user_data['name']
    email = user_data['email']
    # ...
    pass

# Hoặc dùng **kwargs
def create_user(**kwargs):
    name = kwargs.get('name')
    email = kwargs.get('email')
    # ...
    pass
```

---

9. Bài Tập Thực Hành

Bài 1: Kiểm tra số nguyên tố

```python
def is_prime(n):
    """
    Kiểm tra n có phải số nguyên tố không

    Args:
        n (int): Số cần kiểm tra

    Returns:
        bool: True nếu là số nguyên tố
    """
    if n < 2:
        return False

    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False

    return True

# Test
print(is_prime(7))   # True
print(is_prime(10))  # False
print(is_prime(17))  # True
```

Bài 2: Tính tổng các số chẵn trong list

```python
def sum_even_numbers(numbers):
    """
    Tính tổng các số chẵn trong danh sách

    Args:
        numbers (list): Danh sách số

    Returns:
        int: Tổng các số chẵn
    """
    total = 0
    for num in numbers:
        if num % 2 == 0:
            total += num
    return total

# Test
print(sum_even_numbers([1, 2, 3, 4, 5, 6]))  # 12
print(sum_even_numbers([10, 15, 20]))        # 30
```

Bài 3: Đảo ngược chuỗi

```python
def reverse_string(text):
    """
    Đảo ngược chuỗi

    Args:
        text (str): Chuỗi cần đảo ngược

    Returns:
        str: Chuỗi đã đảo ngược
    """
    return text[::-1]

# Hoặc dùng vòng lặp
def reverse_string_v2(text):
    result = ""
    for char in text:
        result = char + result
    return result

# Test
print(reverse_string("Python"))     # nohtyP
print(reverse_string("Hello"))      # olleH
```

Bài 4: Tìm số lớn nhất trong list

```python
def find_max(numbers):
    """
    Tìm số lớn nhất trong danh sách

    Args:
        numbers (list): Danh sách số

    Returns:
        int/float: Số lớn nhất
    """
    if not numbers:
        return None

    max_num = numbers[0]
    for num in numbers[1:]:
        if num > max_num:
            max_num = num

    return max_num

# Test
print(find_max([3, 7, 2, 9, 1]))      # 9
print(find_max([-5, -2, -10]))        # -2
```

Bài 5: Tính số Fibonacci

```python
def fibonacci(n):
    """
    Trả về số Fibonacci thứ n (không dùng đệ quy)

    Args:
        n (int): Vị trí trong dãy Fibonacci

    Returns:
        int: Số Fibonacci thứ n
    """
    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b

# Test
for i in range(10):
    print(fibonacci(i), end=" ")
# Output: 0 1 1 2 3 5 8 13 21 34
```

Bài 6: Đếm số lần xuất hiện của ký tự

```python
def count_character(text, char):
    """
    Đếm số lần xuất hiện của ký tự trong chuỗi

    Args:
        text (str): Chuỗi cần tìm
        char (str): Ký tự cần đếm

    Returns:
        int: Số lần xuất hiện
    """
    count = 0
    for c in text:
        if c == char:
            count += 1
    return count

# Test
print(count_character("Hello World", "l"))  # 3
print(count_character("Python", "n"))       # 1
```

Bài 7: Chuyển nhiệt độ Celsius sang Fahrenheit

```python
def celsius_to_fahrenheit(celsius):
    """
    Chuyển đổi Celsius sang Fahrenheit

    Args:
        celsius (float): Nhiệt độ Celsius

    Returns:
        float: Nhiệt độ Fahrenheit
    """
    return celsius * 9/5 + 32

def fahrenheit_to_celsius(fahrenheit):
    """
    Chuyển đổi Fahrenheit sang Celsius

    Args:
        fahrenheit (float): Nhiệt độ Fahrenheit

    Returns:
        float: Nhiệt độ Celsius
    """
```

```
    return (fahrenheit - 32) * 5/9

# Test
print(f"0°C = {celsius_to_fahrenheit(0)}°F")      # 32.0°F
print(f"100°C = {celsius_to_fahrenheit(100)}°F")  # 212.0°F
print(f"32°F = {fahrenheit_to_celsius(32)}°C")    # 0.0°C
```

Bài 8: Tính giai thừa

```
def factorial(n):
    """
    Tính n! (n giai thừa)

    Args:
        n (int): Số nguyên dương

    Returns:
        int: n!
    """
    if n == 0 or n == 1:
        return 1

    result = 1
    for i in range(2, n + 1):
        result *= i

    return result

# Hoặc dùng đệ quy
def factorial_recursive(n):
    if n <= 1:
        return 1
    return n * factorial_recursive(n - 1)

# Test
print(factorial(5))   # 120
print(factorial(10))  # 3628800
```

---

10. Tổng Kết và Checklist

☑ Kiến thức cần nắm vững

Định nghĩa hàm:

- ☐ Cú pháp `def tên_hàm(tham_số):`
- ☐ Docstring để mô tả hàm
- ☐ Gọi hàm với arguments

Parameters và Return:

- ☐ Positional arguments
- ☐ Keyword arguments
- ☐ Default parameters
- ☐ `return` đơn và return nhiều giá trị
- ☐ Return với điều kiện

Scope:

- ☐ Local scope (biến trong hàm)
- ☐ Global scope (biến ngoài hàm)
- ☐ Từ khóa `global`
- ☐ Từ khóa `nonlocal` (nested functions)

Nâng cao:

- ☐ Pure functions
- ☐ Lambda functions
- ☐ Recursive functions
- ☐ Type hints

🎯 Lưu ý quan trọng

1. **Hàm không có `return`** → tự động return `None`
2. **Sau `return`**, code không chạy nữa
3. **Local scope ưu tiên** hơn global
4. **Tránh dùng `global`** khi có thể
5. **Một hàm nên làm một việc** (Single Responsibility)
6. **Đặt tên hàm rõ ràng**, dùng `snake_case`

📚 Bước tiếp theo

Sau khi nắm vững Functions, bạn sẵn sàng học:

- **List comprehensions** (viết gọn với list)
- **Decorators** (modify hàm)
- **Generators** (lazy evaluation)
- **Modules và packages** (tổ chức code lớn)

---

11. Câu Hỏi Thường Gặp (FAQ)

Q1: Khi nào cần dùng `return`, khi nào không?

A:

- Dùng `return` khi muốn **trả về giá trị** để dùng tiếp
- Không cần `return` khi chỉ **thực hiện hành động** (in ra, ghi file, v.v.)

```python
# Cần return - để tính toán
def add(a, b):
    return a + b

result = add(5, 10)  # Cần lưu kết quả

# Không cần return - chỉ thực hiện hành động
def print_info(name):
    print(f"Tên: {name}")

print_info("Minh Đạo")  # Chỉ in ra, không cần lưu gì
```

Q2: Tham số mặc định có thể là list/dict không?

A: Có thể nhưng **NGUY HIỂM** vì list/dict là mutable.

```python
# ❌ NGUY HIỂM
def add_item(item, items=[]):
    items.append(item)
    return items

print(add_item(1))  # [1]
print(add_item(2))  # [1, 2] - Không phải [2]!

# ☑ AN TOÀN
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

print(add_item(1))  # [1]
print(add_item(2))  # [2] - Đúng!
```

Q3: *args và **kwargs là gì?

A: Cho phép hàm nhận **số lượng tham số không xác định**.

```python
def sum_all(*args):
    """args là tuple chứa tất cả arguments"""
    return sum(args)

print(sum_all(1, 2, 3))        # 6
print(sum_all(1, 2, 3, 4, 5))  # 15

def print_info(**kwargs):
```

```
    """kwargs là dict chứa keyword arguments"""
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Minh Đạo", age=25, city="TP.HCM")
# Output:
# name: Minh Đạo
# age: 25
# city: TP.HCM
```

Q4: Có thể có nhiều `return` trong một hàm không?

A: Có, và đây là pattern phổ biến (early return).

```
def divide(a, b):
    if b == 0:
        return None  # Early return để tránh lỗi

    return a / b

result = divide(10, 2)
print(result)  # 5.0

result = divide(10, 0)
print(result)  # None
```

Q5: Làm sao để hàm thay đổi list được truyền vào?

A: List là **mutable**, nên thay đổi bên trong hàm sẽ ảnh hưởng bên ngoài.

```
def add_item(items, item):
    items.append(item)  # Thay đổi list gốc

my_list = [1, 2, 3]
add_item(my_list, 4)
print(my_list)  # [1, 2, 3, 4] - Đã thay đổi!

# Nếu không muốn thay đổi list gốc
def add_item_safe(items, item):
    new_items = items.copy()  # Tạo bản sao
    new_items.append(item)
    return new_items

my_list = [1, 2, 3]
result = add_item_safe(my_list, 4)
print(my_list)  # [1, 2, 3] - Không đổi
print(result)   # [1, 2, 3, 4]
```

---

12. Thử Thách Cuối Khóa

Challenge 1: Máy tính đơn giản

Viết các hàm `add`, `subtract`, `multiply`, `divide` và hàm `calculator` nhận toán tử để gọi hàm phù hợp.

```
"""
Ví dụ:
calculator(10, 5, "+")  → 15
calculator(10, 5, "*")  → 50
calculator(10, 0, "/")  → "Không thể chia cho 0"
"""
```

Challenge 2: Xác thực mật khẩu

Viết hàm `is_valid_password` kiểm tra mật khẩu hợp lệ nếu:

- Ít nhất 8 ký tự
- Có ít nhất 1 chữ hoa
- Có ít nhất 1 chữ thường
- Có ít nhất 1 số

```
"""
Ví dụ:
is_valid_password("abc123")        → False (thiếu chữ hoa, quá ngắn)
is_valid_password("Abc12345")      → True
is_valid_password("ABCDEFGH")      → False (không có số, không có chữ thường)
"""
```

Challenge 3: Tính tổng các số nguyên tố

Viết hàm `sum_primes(n)` tính tổng các số nguyên tố từ 2 đến n.

```
"""
Ví dụ:
sum_primes(10)  → 17 (2 + 3 + 5 + 7)
sum_primes(20)  → 77
"""
```

Challenge 4: Caesar Cipher

Viết hàm mã hóa/giải mã Caesar Cipher (dịch chuyển ký tự trong bảng chữ cái).

```
"""
Ví dụ:
caesar_encrypt("HELLO", 3)   → "KHOOR"
caesar_decrypt("KHOOR", 3)   → "HELLO"
"""
```
