TÀI LIỆU THAM KHẢO PYTHON - LẬP TRÌNH HÀM

Trần Minh Huy

Ngày 5 tháng 7 năm 2025

Mục lục

1 Lập Trình Hàm (FP) - Python

1.1 FP - Stateless Programming (2 câu)

1.1.1 Định nghĩa và Giải thích

Khái niệm cơ bản

Lập trình không trạng thái (Stateless Programming) là phong cách lập trình trong đó không có trạng thái được chia sẻ hay lưu trữ giữa các lần gọi hàm. Hàm luôn trả về giá trị giống nhau khi được gọi với cùng một đầu vào, bất kể thời điểm hay số lần gọi.
Đặc điểm quan trọng:

- **Tính thuần khiết (Purity)**: Hàm không có tác dụng phụ (side effects) - không thay đổi bất kỳ trạng thái nào bên ngoài phạm vi của nó.
- **Tính xác định (Determinism)**: Cùng đầu vào luôn cho ra cùng đầu ra.
- **Không phụ thuộc thời gian**: Kết quả không phụ thuộc vào thời điểm gọi hàm.
- **Dễ dàng kiểm thử**: Vì hàm thuần khiết luôn cho kết quả xác định.

1.1.2 Pure Functions

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

# Impure function (has side effect)
total = 0
def add_to_total(value):
    global total
    total += value  # Thay    i    b i n  global  l   side effect
    return total

# Pure function with no side effects
def calculate_total(numbers):
    return sum(numbers)
```

1.1.3 Phân tích code

Phân tích chi tiết

1. **Hàm thuần khiết** `add(a, b)`:
   - Hàm này chỉ phụ thuộc vào tham số đầu vào `a` và `b`.
   - Luôn trả về cùng kết quả cho cùng đầu vào (như `add(5, 3)` luôn trả về 8).
   - Không thay đổi bất kỳ trạng thái nào bên ngoài hàm.
2. **Hàm không thuần khiết** `add_to_total(value)`:
   - Thay đổi biến toàn cục `total` (đây là side effect).
   - Gọi `add_to_total(5)` nhiều lần sẽ cho kết quả khác nhau.
   - Kết quả phụ thuộc vào lịch sử gọi hàm trước đó.
3. **Hàm thuần khiết** `calculate_total(numbers)`:
   - Chỉ phụ thuộc vào danh sách `numbers` được truyền vào.
   - Không thay đổi danh sách gốc, chỉ tính tổng các phần tử.
   - Luôn trả về cùng kết quả cho cùng danh sách đầu vào.

Khi gọi `add_to_total(5)` lần đầu tiên, kết quả là 5. Nhưng khi gọi lại `add_to_total(5)` lần thứ hai, kết quả là 10. Đây chính là sự khác biệt giữa hàm thuần khiết và không thuần khiết.

```python
# Vi du thuc te ve loi ich cua ham thuan khiet
import time

# Ham khong thuan khiet - ket qua phu thuoc thoi gian
def get_current_time():
    return time.time()  # Tra ve thoi gian hien tai

# Ham thuan khiet - thuc hien tinh toan ma khong phu thuoc yeu to ben ngoai
def calculate_circle_area(radius):
    return 3.14159 * radius * radius

# Cache ket qua cua ham thuan khiet (memoization)
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

# Ap dung memoization cho ham thuan khiet
circle_area = memoize(calculate_circle_area)

# Demo
print(calculate_circle_area(5))  # 78.53975
print(calculate_circle_area(5))  # 78.53975 - luon cho cung ket qua

print(get_current_time())  # 1620000000.123
time.sleep(1)
print(get_current_time())  # 1620000001.234 - ket qua khac nhau moi lan goi

# Kiem tra hieu suat voi memoization
import time

start = time.time()
for _ in range(1000000):
    circle_area(5)  # Chi tinh toan lan dau, cac lan sau lay tu cache
end = time.time()
print(f"Thoi gian voi memoization: {end - start} giay")
```

1.2 FP - Functional Programming in Python (3 câu)

1.2.1 Giải thích về các tính năng hàm trong Python

Khái niệm cơ bản

Python hỗ trợ lập trình hàm thông qua nhiều tính năng:

- **First-class functions**: Hàm được xem như đối tượng bình thường, có thể:
  - Gán cho biến
  - Truyền làm tham số cho hàm khác
  - Trả về từ hàm khác
  - Lưu trữ trong cấu trúc dữ liệu
- **Lambda expressions**: Hàm ẩn danh, ngắn gọn, thường dùng cho các hàm đơn giản
- **Higher-order functions**: Hàm có thể nhận hàm khác làm tham số hoặc trả về hàm
- **Built-in functions**: `map`, `filter`, `reduce` - các hàm xử lý collection
- **List comprehensions**: Cú pháp tạo list ngắn gọn, thay thế cho map/filter

```python
# 1. First-class functions (Ham la doi tuong hang dau)
def greeting(name):
    return f"Hello, {name}!"

# Gan ham cho bien
say_hello = greeting  # Khong co dau (), chi tham chieu toi ham
print(say_hello("Alice"))  # "Hello, Alice!"

# 2. Higher-order function (ham nhan ham lam tham so)
def apply_twice(func, arg):
    return func(func(arg))  # Goi ham func hai lan

def add_five(x):
    return x + 5

result = apply_twice(add_five, 10)
print(result)  # 10 + 5 + 5 = 20

# 3. Map function voi lambda
numbers = [1, 2, 3, 4, 5]
# Lambda tao ham an danh: x => x^2
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# 4. Filter function voi lambda
# Lambda tao ham kiem tra: x => x % 2 == 0
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]

# 5. Reduce function
from functools import reduce
# Lambda cong hai so: (x, y) => x + y
sum_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_numbers)  # 15

# 6. List comprehension
squared_again = [x ** 2 for x in numbers]  # Tuong duong voi map
print(squared_again)  # [1, 4, 9, 16, 25]

even_squared = [x ** 2 for x in numbers if x % 2 == 0]  # Tuong duong map + filter
print(even_squared)  # [4, 16]
```

1.2.2 Phân tích code

Phân tích chi tiết

1. **First-class Functions:**
   - Trong ví dụ, hàm `greeting` được gán cho biến `say_hello`.
   - Khi gọi `say_hello("Alice")`, thực chất là gọi hàm `greeting("Alice")`.
   - Lưu ý rằng khi gán, không sử dụng dấu ngoặc tròn `()`, vì ta muốn tham chiếu đến hàm, không phải kết quả của hàm.
2. **Higher-order Functions:**
   - Hàm `apply_twice` nhận hai tham số: một hàm `func` và một giá trị `arg`.
   - Nó áp dụng hàm `func` cho `arg`, sau đó áp dụng `func` lại cho kết quả.
   - Khi gọi `apply_twice(add_five, 10)`, quá trình thực hiện là:
     - Bước 1: `add_five(10)` = 15
     - Bước 2: `add_five(15)` = 20
     - Kết quả: 20
3. **Map, Filter, Reduce:**
   - `map(func, iterable)`: Áp dụng `func` cho mỗi phần tử trong `iterable`.
   - `filter(pred, iterable)`: Lấy các phần tử thỏa mãn điều kiện `pred`.
   - `reduce(func, iterable)`: Áp dụng `func` lũy tiến cho các phần tử.
   - Các hàm này trả về iterator (trong Python 3), nên cần chuyển thành list để hiển thị.
4. **Lambda Expressions:**
   - Cú pháp: `lambda arguments: expression`
   - `lambda x: x ** 2` tương đương với:

```
def square(x):
    return x ** 2

```

   - Lambda thích hợp khi cần hàm đơn giản chỉ dùng một lần.
5. **List Comprehension:**
   - Cú pháp: `[expression for item in iterable if condition]`
   - `[x ** 2 for x in numbers]` tương đương với `map(lambda x: x ** 2, numbers)`
   - `[x ** 2 for x in numbers if x % 2 == 0]` tương đương với việc kết hợp `map` và `filter`
   - List comprehension thường ngắn gọn và dễ đọc hơn.

```python

data = [
    {"name": "Alice", "age": 25, "city": "New York"},
    {"name": "Bob", "age": 30, "city": "Boston"},
    {"name": "Charlie", "age": 35, "city": "Chicago"},
    {"name": "Dave", "age": 40, "city": "Denver"},
    {"name": "Eve", "age": 45, "city": "Boston"}
]

# Cach thong thuong (imperative)
def get_names_imperative(people):
    names = []
    for person in people:
        names.append(person["name"])
    return names

# Cach functional voi map
get_names_functional = lambda people: list(map(lambda person: person["name"],
    people))

# Cach functional voi list comprehension
get_names_comprehension = lambda people: [person["name"] for person in people]

# Loc nguoi tu Boston
from_boston_imperative = []
for person in data:
    if person["city"] == "Boston":
        from_boston_imperative.append(person)

# Loc voi filter
from_boston_functional = list(filter(lambda person: person["city"] == "Boston",
    data))

# Loc voi list comprehension
from_boston_comprehension = [person for person in data if person["city"] == "Boston
    "]

# Tinh tuoi trung binh
average_age_imperative = 0
for person in data:
    average_age_imperative += person["age"]
average_age_imperative /= len(data)

# Tinh tuoi trung binh voi reduce
from functools import reduce
average_age_functional = reduce(lambda total, person: total + person["age"], data,
    0) / len(data)

# Tinh tuoi trung binh voi comprehension
average_age_comprehension = sum(person["age"] for person in data) / len(data)

print("Names:", get_names_comprehension(data))
print("From Boston:", [person["name"] for person in from_boston_comprehension])
print("Average age:", average_age_comprehension)
```

1.3 FP - Lambda calculus/Lambda function (7 câu)

1.3.1 Định nghĩa chi tiết

Khái niệm cơ bản

Lambda calculus là một hệ hình thức toán học và là nền tảng của lập trình hàm. Trong Python, lambda expressions (biểu thức lambda) là cách để tạo các hàm ẩn danh đơn giản.
Đặc điểm của lambda trong Python:

- **Cú pháp**: `lambda arguments: expression`
- **Ẩn danh**: Không cần khai báo tên hàm
- **Đơn giản**: Chỉ chứa một biểu thức duy nhất
- **Ngắn gọn**: Thích hợp cho các hàm nhỏ, đơn giản
- **Closure**: Có thể capture và sử dụng biến từ scope bên ngoài

```python
# Cu phap co ban cua lambda
add = lambda x, y: x + y
print(add(5, 3))  # 8

# So sanh voi khai bao ham thong thuong
def add_regular(x, y):
    return x + y

print(add_regular(5, 3))  # 8

# Khi nao dung lambda?
# 1. Khi can ham don gian lam tham so cho higher-order function
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# 2. Sap xep theo tieu chi phuc tap
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 92},
    {"name": "Charlie", "grade": 78}
]

# Sap xep theo diem so
sorted_by_grade = sorted(students, key=lambda student: student["grade"])
print([s["name"] for s in sorted_by_grade])  # ['Charlie', 'Alice', 'Bob']

# 3. Tao ham dong (ham factory)
def power_function(n):
    return lambda x: x ** n

square = power_function(2)
cube = power_function(3)

print(square(4))  # 16
print(cube(4))    # 64

# 4. Closure (lambda bat bien tu scope ben ngoai)
def multiplier(n):
    return lambda x: x * n

double = multiplier(2)
triple = multiplier(3)

print(double(5))  # 10
print(triple(5))  # 15

# 5. Lambda voi nhieu tham so
full_name = lambda first, last: f"{first} {last}"
print(full_name("John", "Doe"))  # "John Doe"
```

```python

# 6. Lambda voi tham so mac dinh
greet = lambda name, greeting="Hello": f"{greeting}, {name}!"
print(greet("Alice"))         # "Hello, Alice!"
print(greet("Bob", "Hi"))     # "Hi, Bob!"

# 7. Lambda voi *args va **kwargs
sum_all = lambda *args: sum(args)
print(sum_all(1, 2, 3, 4))  # 10

format_person = lambda **kwargs: f"Person: {kwargs.get('name', 'Unknown')}, Age: {
    kwargs.get('age', 'Unknown')}"
print(format_person(name="Alice", age=30))  # "Person: Alice, Age: 30"
print(format_person(name="Bob"))            # "Person: Bob, Age: Unknown"
```

1.3.2 Phân tích chi tiết

Phân tích chi tiết

1. **Cấu trúc của Lambda:**
   - `lambda`: từ khóa bắt đầu biểu thức lambda
   - `arguments`: danh sách tham số, phân cách bởi dấu phẩy
   - `::` phân tách tham số và thân hàm
   - `expression`: biểu thức đơn - kết quả của biểu thức này là giá trị trả về
2. **So sánh với hàm thông thường:**
   - Lambda không cần từ khóa `def` và tên hàm
   - Lambda không cần từ khóa `return` - giá trị của biểu thức tự động được trả về
   - Lambda chỉ có thể chứa một biểu thức duy nhất, không thể chứa nhiều câu lệnh
   - Lambda không thể chứa các câu lệnh như `if/else`, `for`, `while` (trừ khi dùng biểu thức điều kiện)
3. **Ứng dụng trong map, filter, sorted:**
   - Trong ví dụ với `map`, lambda nhận vào một số `x` và trả về bình phương của nó
   - Trong ví dụ với `sorted`, lambda trích xuất trường "grade" để làm khóa sắp xếp
   - Các hàm như vậy thường rất ngắn và chỉ dùng một lần, nên lambda rất phù hợp
4. **Lambda và Closure:**
   - Trong ví dụ `power_function` và `multiplier`, lambda bắt giữ biến `n` từ scope bên ngoài
   - Khi gọi `multiplier(2)`, một lambda mới được tạo ra, ghi nhớ giá trị `n=2`
   - Khi gọi `double(5)`, lambda sử dụng `n=2` đã được ghi nhớ và trả về `5*2=10`
   - Đây là ví dụ về closure - hàm "nhớ" môi trường nơi nó được tạo ra
5. **Biểu thức điều kiện trong Lambda:**

```
# Cach dung bieu thuc dieu kin trong Lamda
is_even = lambda x: True if x % 2 == 0 else False

```

   - Lambda không thể chứa câu lệnh `if/else` thông thường
   - Nhưng có thể dùng biểu thức điều kiện dạng `a if condition else b`

Điểm mạnh của Lambda:

- Ngắn gọn, giúp code súc tích hơn
- Thích hợp cho các hàm đơn giản
- Rất hữu ích khi kết hợp với các higher-order functions

Điểm yếu của Lambda:

- Không thể chứa nhiều câu lệnh
- Khó đọc khi logic phức tạp
- Không có docstring, khó debug

1.4 FP - Curry Function (3 câu)

1.4.1 Định nghĩa chi tiết

Khái niệm cơ bản

Currying là kỹ thuật biến đổi một hàm nhận nhiều tham số thành chuỗi các hàm, mỗi hàm nhận một tham số. Partial application là quá trình áp dụng một số tham số của hàm và trả về một hàm mới cần ít tham số hơn.
Giải thích:

- **Currying**: Chuyển đổi hàm `f(x, y, z)` thành `f(x)(y)(z)`
- **Partial Application**: Áp dụng một số tham số, ví dụ `f(x, y, z)` thành `g(y, z) = f(a, y, z)` với `a` là hằng số
- **Lợi ích**: Tăng tính tái sử dụng, tạo ra các hàm chuyên biệt từ hàm tổng quát

```python
# 1. Manual currying
def add(x):
    def add_y(y):
        return x + y
    return add_y

add_5 = add(5)  # Tra ve ham add_y voi x=5
print(add_5(3))  # 8 (5+3)

# Goi truc tiep
print(add(5)(3))  # 8

# 2. Ham curry tong quat
def curry(func, arity):
    """
    Tra ve mot phien ban curried cua ham
    arity: so luong tham so cua ham
    """
    def curried(*args):
        if len(args) >= arity:
            return func(*args)
        return lambda *more_args: curried(*(args + more_args))
    return curried

# Vi du voi ham 3 tham so
def add3(x, y, z):
    return x + y + z

curried_add3 = curry(add3, 3)

# Co the goi theo nhieu cach
print(curried_add3(1, 2, 3))      # 6
print(curried_add3(1)(2)(3))      # 6
print(curried_add3(1, 2)(3))      # 6
print(curried_add3(1)(2, 3))      # 6

# 3. Partial application su dung functools
from functools import partial

def multiply(x, y):
    return x * y

double = partial(multiply, 2)  # Co dinh tham so dau tien la 2
triple = partial(multiply, 3)  # Co dinh tham so dau tien la 3

print(double(5))  # 10 (2*5)
print(triple(5))  # 15 (3*5)

# 4. Vi du thuc te: Khoi tao formatters
def format_string(template, name, value):
    return template.format(name=name, value=value)

```

```python
# Tao cac formatters chuyen biet
html_formatter = partial(format_string, "<p>{name}: {value}</p>")
json_formatter = partial(format_string, "{{\"name\": \"{name}\", \"value\": {value
    }}}")

print(html_formatter("age", 30))  # "<p>age: 30</p>"
print(json_formatter("age", 30))  # "{"name": "age", "value": 30}"
```

1.4.2 Phân tích chi tiết

Phân tích chi tiết

1. **Manual Currying:**
   - Hàm `add(x)` không trả về giá trị tính toán mà trả về một hàm mới `add_y(y)`.
   - Khi gọi `add(5)`, ta nhận được một hàm mới `add_y` với `x=5`. Hàm này chờ nhận thêm một tham số `y`.
   - Khi gọi `add_5(3)`, ta thực sự đang gọi `add_y(3)` với `x=5`, và nhận được kết quả `5+3=8`.
   - Chuỗi gọi hàm `add(5)(3)` là cách viết gọn của quá trình trên.
2. **Generic Curry Function:**
   - Hàm `curry` là một higher-order function, nhận vào một hàm `func` và số lượng tham số `arity`.
   - Nó trả về một hàm curried có thể nhận tham số theo từng phần.
   - Nếu số lượng tham số đã đủ (`len(args) >= arity`), gọi hàm gốc với tất cả tham số.
   - Nếu chưa đủ, trả về một lambda để nhận thêm tham số, rồi gọi đệ quy `curried` với tất cả tham số đã được thu thập.
3. **Partial Application:**
   - Khác với currying, partial application áp dụng một số tham số và cố định chúng.
   - Trong ví dụ, `partial(multiply, 2)` tạo ra một hàm mới với tham số đầu tiên cố định là 2.
   - Hàm `double` chỉ cần nhận một tham số (y), vì x đã được cố định là 2.
   - Đây là cách đơn giản để tạo ra các hàm chuyên biệt từ một hàm tổng quát.
4. **Ví dụ thực tế:**
   - Hàm `format_string` nhận ba tham số: template, name, value.
   - Sử dụng partial application, chúng ta cố định tham số `template`.
   - `html_formatter` và `json_formatter` là các phiên bản chuyên biệt, chỉ cần cung cấp `name` và `value`.
   - Đây là một mẫu thiết kế phổ biến để tạo ra các phiên bản chuyên biệt từ một hàm tổng quát.

So sánh Currying và Partial Application:

- **Currying**: Biến đổi hàm để có thể gọi từng tham số một: `f(x,y,z)` → `f(x)(y)(z)`
- **Partial Application**: Cố định một số tham số: `f(x,y,z)` → `g(y,z) = f(a,y,z)`
- Partial application thường dễ sử dụng hơn trong Python với `functools.partial`
- Currying phổ biến hơn trong các ngôn ngữ hàm thuần túy như Haskell

1.5 FP - List Comprehension (4 câu)

1.5.1 Định nghĩa chi tiết

Khái niệm cơ bản

List comprehension là cú pháp ngắn gọn trong Python để tạo list mới từ list có sẵn, thường kết hợp với các phép biến đổi và lọc. Nó thay thế các vòng lặp truyền thống và các hàm như `map`, `filter`.

Cú pháp cơ bản:

- `[expression for item in iterable]` - tạo list mới với biến đổi
- `[expression for item in iterable if condition]` - tạo list mới với biến đổi và lọc
- Ngoài list, còn có dict comprehension, set comprehension và generator expression

```python
# 1. List comprehension co ban
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
print(squares)  # [1, 4, 9, 16, 25]

# Tuong duong voi:
squares_loop = []
for x in numbers:
    squares_loop.append(x**2)
print(squares_loop)  # [1, 4, 9, 16, 25]

# 2. List comprehension voi dieu kien
even_squares = [x**2 for x in numbers if x % 2 == 0]
print(even_squares)  # [4, 16]

# Tuong duong voi:
even_squares_loop = []
for x in numbers:
    if x % 2 == 0:
        even_squares_loop.append(x**2)
print(even_squares_loop)  # [4, 16]

# 3. List comprehension voi nhieu vong lap
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened = [num for row in matrix for num in row]
print(flattened)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# Tuong duong voi:
flattened_loop = []
for row in matrix:
    for num in row:
        flattened_loop.append(num)
print(flattened_loop)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# 4. List comprehension long nhau
transposed = [[row[i] for row in matrix] for i in range(3)]
print(transposed)  # [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

# Tuong duong voi:
transposed_loop = []
for i in range(3):
    transposed_row = []
    for row in matrix:
        transposed_row.append(row[i])
    transposed_loop.append(transposed_row)
print(transposed_loop)  # [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

# 5. Dictionary comprehension
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
name_to_age = {name: age for name, age in zip(names, ages)}
print(name_to_age)  # {'Alice': 25, 'Bob': 30, 'Charlie': 35}
```

```
53
54 # 6. Set comprehension
55 unique_letters = {letter for letter in "mississippi"}
56 print(unique_letters)  # {'m', 'i', 's', 'p'}
57
58 # 7. Generator expression (lazy evaluation)
59 sum_of_squares = sum(x**2 for x in range(1, 6))
60 print(sum_of_squares)  # 55
```

1.5.2 Phân tích chi tiết

Phân tích chi tiết

1. **List Comprehension Cơ bản:**
   - Cú pháp: `[expression for item in iterable]`
   - Ví dụ: `[x**2 for x in numbers]` tạo list mới chứa bình phương của mỗi phần tử
   - Thay thế vòng lặp for truyền thống và append, giúp code ngắn gọn hơn
   - Tương đương với `map(lambda x: x**2, numbers)` nhưng dễ đọc hơn
2. **List Comprehension với Điều kiện:**
   - Cú pháp: `[expression for item in iterable if condition]`
   - Ví dụ: `[x**2 for x in numbers if x % 2 == 0]` tạo list bình phương của các số chẵn
   - Thay thế kết hợp của `filter` và `map`
   - Điều kiện (`if x % 2 == 0`) được kiểm tra trước khi áp dụng biểu thức (`x**2`)
3. **List Comprehension Lồng nhau:**
   - Với nhiều vòng lặp: `[num for row in matrix for num in row]`
   - Thứ tự vòng lặp giống như thứ tự trong câu lệnh for lồng nhau truyền thống
   - Vòng lặp bên ngoài (`for row in matrix`) viết trước, vòng lặp bên trong (`for num in row`) viết sau
4. **Chuyển vị Ma trận:**
   - `[[row[i] for row in matrix] for i in range(3)]`
   - Đây là list comprehension lồng nhau 2 cấp
   - Vòng lặp ngoài (`for i in range(3)`) tạo từng row mới
   - Vòng lặp trong (`for row in matrix`) lấy phần tử tại cùng vị trí i từ mỗi row gốc
5. **Dictionary và Set Comprehension:**
   - Dict: `{key: value for item in iterable}`
   - Set: `{expression for item in iterable}`
   - Tương tự như list comprehension nhưng tạo ra kiểu dữ liệu khác
6. **Generator Expression:**
   - Cú pháp: `(expression for item in iterable)`
   - Không tạo ra list hoàn chỉnh mà tạo generator, tiết kiệm bộ nhớ
   - Phù hợp khi làm việc với dữ liệu lớn hoặc vô hạn
   - Lazy evaluation: các phần tử chỉ được tính khi cần thiết

Ưu điểm của List Comprehension:

- Ngắn gọn, dễ đọc cho các biến đổi đơn giản
- Hiệu suất tốt hơn so với vòng lặp for (tối ưu hóa nội bộ)
- Thể hiện ý đồ rõ ràng: "tạo list mới từ list cũ"

Nhược điểm:

- Có thể khó đọc khi quá phức tạp (nhiều vòng lặp lồng nhau)

- Không phù hợp cho các thao tác phức tạp hoặc xử lý ngoại lệ

1.6 FP - High-order functions (7 câu)

1.6.1 Định nghĩa chi tiết

Khái niệm cơ bản

Higher-order functions (hàm bậc cao) là các hàm nhận hàm khác làm tham số hoặc trả về một hàm. Chúng là nền tảng của lập trình hàm, cho phép tái sử dụng mã và phân tách logic thành các thành phần nhỏ hơn.
Đặc điểm:

- **Hàm là first-class citizens**: Có thể được truyền, trả về, và gán cho biến
- **Hai dạng chính**:
  - Hàm nhận hàm làm tham số: `map`, `filter`, `reduce`
  - Hàm trả về hàm: function factories, decorators
- **Cho phép abstraction**: Tách biệt "cái gì" (chức năng) và "như thế nào" (cách thực hiện)

```python
# 1. Ham nhan ham lam tham so
def apply_function(func, value):
    return func(value)

def square(x):
    return x * x

def cube(x):
    return x * x * x

print(apply_function(square, 5))  # 25
print(apply_function(cube, 3))    # 27

# 2. Ham tra ve ham
def create_multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

double = create_multiplier(2)
triple = create_multiplier(3)

print(double(5))  # 10
print(triple(5))  # 15

# 3. Ham map tich hop
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# 4. Trien khai map tuy chinh
def my_map(func, iterable):
    result = []
    for item in iterable:
        result.append(func(item))
    return result

cubed = my_map(lambda x: x**3, numbers)
print(cubed)  # [1, 8, 27, 64, 125]

# 5. Ham filter tich hop
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]

# 6. Trien khai filter tuy chinh
```

```python
def my_filter(predicate, iterable):
    result = []
    for item in iterable:
        if predicate(item):
            result.append(item)
    return result

odd_numbers = my_filter(lambda x: x % 2 != 0, numbers)
print(odd_numbers)  # [1, 3, 5]

# 7. Ham reduce tu functools
from functools import reduce

sum_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_numbers)  # 15

product_numbers = reduce(lambda x, y: x * y, numbers)
print(product_numbers)  # 120

# 8. Trien khai reduce tuy chinh
def my_reduce(func, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer

    for element in it:
        value = func(value, element)

    return value

sum_again = my_reduce(lambda x, y: x + y, numbers)
print(sum_again)  # 15

# 9. Function composition
def compose(f, g):
    return lambda x: f(g(x))

def add_one(x):
    return x + 1

square_then_add_one = compose(add_one, square)
add_one_then_square = compose(square, add_one)

print(square_then_add_one(5))  # 5   + 1 = 26
print(add_one_then_square(5))  # (5 + 1)   = 36

# 10. Multiple function composition
def compose_multiple(*functions):
    def compose_two(f, g):
        return lambda x: f(g(x))

    if len(functions) == 0:
        return lambda x: x

    return reduce(compose_two, functions)

def add_one(x):
    return x + 1

def square(x):
    return x * x

def double(x):
    return x * 2

# (((5 + 1)  ) * 2) = 72
composed = compose_multiple(double, square, add_one)
print(composed(5))  # 72
```

1.6.2 Phân tích chi tiết

Phân tích chi tiết

1. **Hàm nhận hàm làm tham số:**
   - `apply_function` nhận hai tham số: một hàm `func` và một giá trị `value`
   - Nó đơn giản gọi hàm được truyền vào với giá trị đã cho
   - Điều này cho phép tách biệt cơ chế gọi hàm và logic của hàm
   - Khi gọi `apply_function(square, 5)`, ta truyền hàm `square` (không phải kết quả của nó)
2. **Hàm trả về hàm:**
   - `create_multiplier` là một function factory - nó tạo ra các hàm mới
   - Nó trả về hàm `multiply` với closure bắt giữ biến `factor`
   - Mỗi lần gọi với `factor` khác nhau, ta sẽ có một hàm mới
   - `double` và `triple` là các hàm khác nhau, mỗi hàm có closure riêng
3. **Map, Filter, Reduce:**
   - **map(func, iterable)**: Áp dụng `func` cho mỗi phần tử, trả về iterator với kết quả
   - **filter(pred, iterable)**: Giữ lại các phần tử thỏa mãn `pred`, trả về iterator
   - **reduce(func, iterable)**: Áp dụng `func` lũy tiến, trả về một giá trị duy nhất
   - Cả ba đều là higher-order functions vì chúng nhận hàm làm tham số
4. **Triển khai tùy chỉnh:**
   - `my_map`, `my_filter`, `my_reduce` cho thấy cách thức hoạt động bên trong
   - Đây là ví dụ về abstraction - tách biệt logic lặp và xử lý phần tử
   - Người dùng chỉ cần cung cấp logic xử lý (qua `func` hoặc `predicate`)
5. **Function Composition:**
   - `compose(f, g)` tạo một hàm mới áp dụng `g` trước, sau đó áp dụng `f`
   - Tương đương với toán học: $f \circ g = f(g(x))$
   - `compose_multiple` mở rộng ý tưởng này để kết hợp nhiều hàm
   - Khi gọi `compose_multiple(double, square, add_one)`, thứ tự thực hiện là:
     - Bước 1: `add_one(5)` = 6
     - Bước 2: `square(6)` = 36
     - Bước 3: `double(36)` = 72
   - Lưu ý thứ tự thực hiện từ phải sang trái (như trong toán học)

Lợi ích của Higher-order Functions:

- **Tái sử dụng code**: Tách biệt các hoạt động phổ biến (như duyệt qua collection)
- **Tính linh hoạt**: Đưa behavior vào như tham số
- **Trừu tượng hóa**: Tập trung vào "cái gì" hơn là "như thế nào"
- **Mã ngắn gọn**: Giảm lặp lại, tăng tính mô-đun

Thách thức:

- **Debugging**: Khó debug hơn khi logic bị phân tán

- **Hiệu suất**: Có thể chậm hơn do overhead của việc gọi hàm
- **Đường cong học tập**: Đòi hỏi tư duy khác so với lập trình mệnh lệnh