Hàm (function)

Nội dung

Định nghĩa Hàm

Truyền tham số

Giá trị trả về

Truyền một danh sách vào hàm

Truyền một đối số tùy ý

Lưu trữ hàm trong module

Định nghĩa Hàm

Cấu trúc đơn giản nhất của một hàm:

Dòng lệnh đầu tiên sử dụng từ khóa def để thông báo cho Python rằng ta đang định nghĩa một hàm.

Bất kỳ dòng thụt lề nào theo sau def greet_user(): tạo nên phần thân của hàm

Dòng """Display a simple greeting.""" là một chú thích được gọi là mỗi chuỗi tài liệu

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

```
Hello!
```

Dòng print("Hello!") Là dòng code duy nhất trong phần thân của hàm này, vì vậy, greet_user() chỉ có một lệnh: print(“Hello!").

Truyền thông tin tới một hàm

Để truyền thông tin tới một hàm, nhập tham số username trong dấu ngoặc đơn định nghĩa hàm

Hàm yêu cầu ta cung cấp một giá trị cho username mỗi khi gọi nó

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")


greet_user('jesse')
```

```
Hello, Jesse!
```

Lệnh greet_user('jesse') sẽ gọi hàm greet_user() và cung cấp cho hàm thông tin cần thiết để thực hiện lệnh gọi print(). Hàm chấp nhận tên được chuyển và hiển thị lời chào cho tên đó

Đối số và tham số

- Biến username của greet_user() là một ví dụ về một *tham số*, một phần thông tin mà hàm cần để thực hiện công việc của nó.
- Giá trị 'jesse' trong greet_user('jesse') là một ví dụ về *đối số*. Đối số là một phần thông tin được truyền từ một lệnh gọi hàm đến một hàm. Khi chúng ta gọi hàm, chúng ta đặt giá trị mà chúng ta muốn hàm hoạt động trong dấu ngoặc đơn.
- Trong trường hợp này, đối số 'jesse' đã được chuyển đến hàm greet_user() và giá trị được gán cho tham số username.

Truyền tham số

Bởi vì một định nghĩa hàm có thể có nhiều tham số, một lệnh gọi hàm có thể cần nhiều đối số. Chúng ta có thể truyền các đối số cho các hàm của mình theo một số cách.

Ta có thể sử dụng các đối số có vị trí, các đối số này cần theo cùng thứ tự mà các tham số đã được viết; các đối số từ khóa, trong đó mỗi đối số bao gồm một tên biến và một giá trị; và danh sách và từ điển các giá trị.

Đối số có vị trí

Khi gọi một hàm, Python phải khớp từng đối số trong lệnh gọi hàm với một tham số trong định nghĩa hàm. Cách đơn giản nhất để làm điều này là dựa trên thứ tự của các đối số được cung cấp

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.”)
    print(f"My {animal_type}'s name is {pet_name.title()}.”)


describe_pet('hamster', 'harry')
```

```
I have a hamster.
My hamster's name is Harry.
```

Nhiều lời gọi hàm

Có thể gọi một hàm nhiều lần nếu cần

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")


describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

```
I have a hamster.
My hamster's name is Harry.


I have a dog.
My dog's name is Willie.
```

Vấn đề thứ tự trong Đối số có vị trí

Ta có thể nhận được kết quả không mong muốn nếu trộn thứ tự của các đối số trong một lệnh gọi hàm khi sử dụng các đối số vị trí

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")


describe_pet('harry', 'hamster')
```

```
I have a harry.
My harry's name is Hamster.
```

Đối số từ khóa

Đối số từ khóa là một cặp tên-giá trị được truyền cho một hàm.

Chúng ta liên kết trực tiếp tên và giá trị bên trong đối số, vì vậy khi ta truyền đối số vào hàm, sẽ không có sự nhầm lẫn nào

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")


describe_pet(animal_type='hamster', pet_name='harry')
```

Thứ tự của các đối số từ khóa không quan trọng vì Python biết mỗi giá trị sẽ đi đến đâu. Hai lệnh gọi hàm sau đây là tương đương

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

Giá trị mặc định

Khi viết một hàm, ta có thể xác định một giá trị mặc định cho mỗi tham số. Nếu một đối số cho một tham số được cung cấp trong lệnh gọi hàm, thì Python sẽ sử dụng giá trị đối số. Nếu không, nó sử dụng giá trị mặc định của thông số.

Khi xác định giá trị mặc định cho một tham số, ta có thể loại trừ đối số tương ứng mà ta thường viết trong lệnh gọi hàm.

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")


describe_pet(pet_name='willie')
```

```
I have a dog.
My dog's name is Willie.
```

Giá trị mặc định

Để mô tả một con vật không phải con chó, ta có thể sử dụng một lệnh gọi hàm như sau:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Bởi vì một đối số rõ ràng cho animal_type được cung cấp, Python sẽ bỏ qua giá trị mặc định của tham số.

Ghi chú: Khi ta sử dụng giá trị mặc định, bất kỳ thông số nào có giá trị mặc định cần được liệt kê sau tất cả các tham số không có giá trị mặc định. Điều này cho phép Python tiếp tục diễn giải các đối số vị trí một cách chính xác.

Lệnh gọi hàm tương đương

Bởi vì các đối số vị trí, đối số từ khóa và giá trị mặc định đều có thể được sử dụng cùng nhau, nên thường ta sẽ có một số cách tương đương để gọi một hàm.

```
def describe_pet(pet_name, animal_type='dog'):


# A dog named Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# A hamster named Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Tránh lỗi đối số

Các đối số không khớp xảy ra khi ta cung cấp ít hoặc nhiều đối số hơn một hàm cần thực hiện công việc của nó

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.”)


describe_pet()
```

```
Traceback (most recent call last):
    File "pets.py", line 6, in <module>
        describe_pet()
TypeError: describe_pet() missing 2 required positional arguments: 'animal_ type' and 'pet_name'
```

Giá trị trả về

Không phải lúc nào một hàm cũng phải hiển thị trực tiếp đầu ra của nó. Thay vào đó, nó có thể xử lý một số dữ liệu và sau đó trả về một giá trị hoặc tập hợp các giá trị.

Giá trị mà hàm trả về được gọi là giá trị trả về. Câu lệnh trả về nhận một giá trị từ bên trong một hàm và gửi trở lại dòng được gọi là hàm.

Giá trị trả về cho phép ta chuyển phần lớn công việc khó khăn của chương trình sang các hàm, điều này có thể đơn giản hóa phần nội dung chương trình của mình.

Trả về giá trị đơn

Hãy xem xét một hàm lấy họ và tên và trả về tên đầy đủ

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted. """
    full_name = f"{first_name} {last_name}"
    return full_name.title()


musician = get_formatted_name('jimi', 'hendrix')
print(musician)
```

Khi ta gọi một hàm trả về một giá trị, ta cần cung cấp một biến mà giá trị trả về có thể được gán cho. Trong trường hợp này, giá trị trả về được gán cho biến musician.

```
Jimi Hendrix
```

Tạo số đối số tùy chọn

Đôi khi, việc đặt một đối số tùy chọn để những người sử dụng hàm có thể chọn cung cấp thêm thông tin chỉ khi họ muốn. Chúng ta có thể sử dụng các giá trị mặc định để làm cho một đối số tùy chọn.

```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name, neatly formatted. """
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()


musician = get_formatted_name('john', 'lee', 'hooker ')
print(musician)
```

```
John Lee Hooker
```

Tạo số đối số tùy chọn

Để đặt tên đệm là tùy chọn, chúng ta có thể cung cấp cho đối số middle_name một giá trị mặc định trống và bỏ qua đối số trừ khi người dùng cung cấp giá trị. Để làm cho get_formatted_name() hoạt động mà không có tên đệm, chúng ta đặt giá trị mặc định của middle_name thành một chuỗi trống và di chuyển nó đến cuối danh sách các tham số:

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Return a full name, neatly formatted."""
    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

```
Jimi Hendrix
John Lee Hooker
```

Trả về một từ điển

Một hàm có thể trả về bất kỳ loại giá trị nào ta cần, bao gồm cả các cấu trúc dữ liệu phức tạp hơn như danh sách và từ điển.

Ví dụ, hàm sau nhận các phần của tên và trả về từ điển đại diện cho một người:

```
def build_person(first_name, last_name):
    """Return a dictionary of information about a person. """
    person = {'first': first_name, 'last': last_name}
    return person


musician = build_person('jimi', 'hendrix')
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix'}
```

Trả về một từ điển

Hàm này nhận thông tin dạng văn bản đơn giản và đưa nó vào một cấu trúc dữ liệu có ý nghĩa hơn cho phép ta làm việc với thông tin ngoài việc in ra. Các chuỗi 'jimi' và 'hendrix' hiện được gắn nhãn là tên và họ.

Chúng ta có thể dễ dàng mở rộng chức năng này để chấp nhận các giá trị tùy chọn như tên đệm, tuổi, nghề nghiệp hoặc bất kỳ thông tin nào khác mà ta muốn lưu trữ về một người.

```
def build_person(first_name, last_name, age=None):
    """Return a dictionary of information about a person.a"""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person


musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix', 'age': 27}
```

Sử dụng một hàm với vòng lặp while

Có thể sử dụng các hàm với tất cả các cấu trúc Python mà ta đã học cho đến thời điểm này

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# This is an infinite loop!
while True:
    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Sử dụng một hàm với vòng lặp while

```
while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
            break
    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()
```

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: Eric
Last name: matthes

Hello, Eric Matthes!
Please tell me your name:
(enter 'q' at any time to quit)
First name: q
```

Truyền một danh sách vào hàm

Giả sử chúng ta có một danh sách người dùng và muốn in lời chào cho mỗi người dùng. Ví dụ sau đây gửi một danh sách các tên đến một hàm có tên gọi là greet_users(), hàm này chào từng người trong danh sách:

```
def greet_users(names):
    """Print a simple greeting to each user in the list. """
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

Sửa đổi danh sách trong một hàm

Khi ta truyền một danh sách vào một hàm, hàm này có thể sửa đổi danh sách.

Mọi thay đổi được thực hiện đối với danh sách bên trong nội dung của hàm là vĩnh viễn, cho phép ta làm việc hiệu quả ngay cả khi ta đang xử lý một lượng lớn dữ liệu

```
# Start with some designs that need to be printed.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []
# Simulate printing each design, until none are left.
#  Move each design to completed_models after printing.
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Printing model: {current_design}”)
    completed_models.append(current_design)
```

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

Sửa đổi danh sách trong một hàm

Có thể tổ chức lại đoạn code này bằng cách viết hai hàm, mỗi hàm thực hiện một công việc cụ thể

Hàm print_models

```
def print_models(unprinted_designs, completed_models):
    """
    Simulate printing each design, until none are left. Move each design to completed_models
after printing.
    """
    while unprinted_designs:
        current_design = unprinted_designs.pop()
        print(f"Printing model: {current_design}")
        completed_models.append(current_design)
```

Sửa đổi danh sách trong một hàm

Hàm show_completed_models

```
def show_completed_models(completed_models):
    """Show all the models that were printed. """
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)
```

Phần chương trình chính:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []
print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Ngăn một hàm sửa đổi danh sách

Ngăn một hàm sửa đổi danh sách bằng cách chuyển cho hàm một bản sao của danh sách, không phải bản gốc.

```
function_name(list_name[:])
```

Kí hiệu lát cắt [:] tạo một bản sao của danh sách để gửi đến hàm. Nếu không muốn làm trống danh sách các thiết kế chưa in, chúng ta có thể gọi print_models() như sau:

```
print_models(unprinted_designs[:], completed_models)
```

Truyền một đối số tùy ý

Đôi khi ta sẽ không biết trước có bao nhiêu đối số mà một hàm cần chấp nhận.

Python cho phép một hàm thu thập một số đối số tùy ý từ câu lệnh gọi

```
def make_pizza(*toppings):
    """Print the list of toppings that have been requested."""
    print(toppings)


make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Dấu hoa thị trong tham số * toppings cho Python biết tạo một bộ giá trị trống được gọi là toppings và đóng gói bất kỳ giá trị nào nó nhận được vào bộ này.

Lời gọi print() trong thân hàm tạo ra kết quả cho thấy Python có thể xử lý một lệnh gọi hàm với một giá trị và một lệnh gọi có ba giá trị.

```
('pepperoni',)
('mushrooms', 'green peppers', 'extra cheese')
```

Truyền một đối số tùy ý

Thay thế lệnh gọi print() bằng một vòng lặp chạy qua danh sách các topping và mô tả bánh pizza đang được đặt hàng

```
def make_pizza(*toppings):
    """Summarize the pizza we are about to make."""
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
make_pizza('pepperoni')
make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
Making a pizza with the following toppings:
- pepperoni

Making a pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Cú pháp này hoạt động bất kể hàm nhận được bao nhiêu đối số.

Kết hợp đối số vị trí và tùy ý

Nếu muốn một hàm chấp nhận một số loại đối số khác nhau, thì tham số chấp nhận một số đối số tùy ý phải được đặt cuối cùng trong định nghĩa hàm.

Python khớp các đối số vị trí và từ khóa trước rồi thu thập bất kỳ đối số nào còn lại trong tham số cuối cùng.

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")

make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Sử dụng đối số từ khóa tùy ý

Đôi khi ta muốn chấp nhận một số lượng đối số tùy ý, nhưng ta sẽ không biết trước loại thông tin nào sẽ được chuyển đến hàm. Trong trường hợp này, ta có thể viết các hàm chấp nhận nhiều cặp khóa-giá trị như câu lệnh gọi cung cấp.

```
def build_profile(first, last, **user_info):
    """Build a dictionary containing everything we know about a user."""
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info


user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')
print(user_profile)
```

Sử dụng đối số từ khóa tùy ý

Trong phần nội dung của build_profile(), chúng ta thêm họ và tên vào từ điển user_info vì chúng ta sẽ luôn nhận được hai phần thông tin này từ người dùng và chúng chưa được đưa vào từ điển. Sau đó, chúng ta trả lại từ điển user_info cho dòng gọi hàm.

Gọi build_profile(), truyền cho nó tên là 'albert', họ 'einstein' và hai cặp khóa-giá trị là location = 'Princeton' và field = 'Physics'. Gán hồ sơ đã trả về cho user_profile và in user_profile:

```
{'location': 'princeton', 'field': 'physics', 'first_name': 'albert',
'last_name': 'einstein'}
```

Từ điển trả về chứa họ và tên của người dùng, trong trường hợp này là cả vị trí và lĩnh vực nghiên cứu. Hàm sẽ hoạt động bất kể có bao nhiêu cặp khóa-giá trị bổ sung được cung cấp trong lệnh gọi hàm.

Lưu trữ hàm trong module

Một ưu điểm của các hàm là cách chúng tách các khối mã nguồn khỏi chương trình chính. Bằng cách sử dụng tên mô tả cho các hàm, chương trình chính sẽ dễ theo dõi hơn nhiều.

Ta có thể lưu trữ các hàm của mình trong một tệp riêng được gọi là mô-đun và sau đó nhập mô-đun đó vào chương trình chính. Một câu lệnh nhập (import) yêu cầu Python làm cho code trong một mô-đun có sẵn trong tệp chương trình hiện đang chạy.

Việc lưu trữ các hàm trong một tệp riêng biệt cho phép ẩn các chi tiết của code chương trình và tập trung vào logic cấp cao hơn của nó. Nó cũng cho phép sử dụng lại các hàm trong nhiều chương trình khác nhau.

Khi ta lưu trữ các hàm của mình trong các tệp riêng biệt, ta có thể chia sẻ các tệp đó với các lập trình viên khác mà không cần phải chia sẻ toàn bộ chương trình của mình. Biết cách import các hàm cũng cho phép ta sử dụng thư viện các hàm mà các lập trình viên khác đã viết.

Import toàn bộ module

Để bắt đầu import các hàm, trước tiên chúng ta cần tạo một mô-đun.

Mô-đun là một tệp kết thúc bằng .py có chứa code muốn import vào chương trình chính.

pizza.py

```
def make_pizza(size, *toppings):
    """Summarize the pizza we are about to make."""
    print(f"\nMaking a {size}-inch pizza with the following toppings:")
    for topping in toppings:
        print(f"- {topping}")
```

make_pizzas.py

```
import pizza


pizza.make_pizza(16, 'pepperoni')
pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Import toàn bộ module

Để gọi một hàm từ một mô-đun đã import, hãy nhập tên của mô-đun ta đã import, pizza, theo sau là tên của hàm, make_pizza(), được phân tách bằng dấu chấm.

```
Making a 16-inch pizza with the following toppings:
- pepperoni

Making a 12-inch pizza with the following toppings:
- mushrooms
- green peppers
- extra cheese
```

Nếu ta sử dụng loại câu lệnh import này để import toàn bộ mô-đun có tên module_name.py, mỗi hàm trong mô-đun có sẵn thông qua cú pháp sau:

```
module_name.function_name()
```

Ví dụ: pizza.make_pizza(16, 'pepperoni')

Import các hàm cụ thể

Có thể import một hàm cụ thể từ một mô-đun. Đây là cú pháp chung cho cách tiếp cận này:

```
from module_name import function_name
```

có thể import bao nhiêu hàm tùy thích từ một mô-đun bằng cách phân tách tên của từng hàm bằng dấu phẩy:

```
from module_name import function_0, function_1, function_2
```

```
from pizza import make_pizza


make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Sử dụng as để cấp cho hàm một bí danh (Alias)

Nếu tên của một hàm ta đang import có thể xung đột với tên tồn tại trong chương trình đang viết hoặc nếu tên hàm dài, có thể sử dụng một bí danh ngắn, duy nhất — một tên thay thế tương tự như biệt hiệu cho hàm.

Đặt biệt hiệu đặc biệt này cho hàm khi import hàm.

```
from pizza import make_pizza as mp


mp(16, 'pepperoni')
mp(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Cú pháp:

```
from module_name import function_name as fn
```

Sử dụng as để cấp cho một mô-đun một bí danh

Có thể cung cấp bí danh cho tên mô-đun. Đặt cho mô-đun một bí danh ngắn, như p cho pizza, cho phép gọi các chức năng của mô-đun nhanh hơn.

```
import pizza as p


p.make_pizza(16, 'pepperoni')
p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Mô-đun Pizza được đặt bí danh p trong câu lệnh import, nhưng tất cả các chức năng của mô-đun vẫn giữ nguyên tên ban đầu.

Cú pháp:

```
import module_name as mn
```

Import tất cả các hàm trong module

Có thể yêu cầu Python import mọi hàm trong một mô-đun bằng cách sử dụng toán tử dấu hoa thị (*):

```
from pizza import *
make_pizza(16, 'pepperoni')
make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

Dấu hoa thị trong câu lệnh import yêu cầu Python sao chép mọi hàm từ mô-đun pizza vào tệp chương trình này. Vì mọi hàm đều được import nên ta có thể gọi từng hàm theo tên mà không cần sử dụng ký hiệu dấu chấm.

Hàm lambda

Nói một cách đơn giản, một hàm lambda cũng giống như bất kỳ hàm python bình thường nào, ngoại trừ việc nó *không có tên khi định nghĩa* nó và ***nó được chứa trong một dòng mã***.

Cú pháp

lambda argument(s): expression

Một hàm lambda đánh giá một biểu thức cho một đối số nhất định. Ta cung cấp cho hàm một giá trị (đối số) và sau đó cung cấp hoạt động (biểu thức).
Từ khóa lambda phải xuất hiện đầu tiên. Dấu hai chấm đầy đủ (:) ngăn cách đối số và biểu thức.

```
#Normal python function
def a_name(x):
        return x+x

#Lambda function
lambda x: x+x
```

Scalar values

```
y =(lambda x: x*4)(12)
print(y)
#48
```

Lists - Filter

Đây là một thư viện có sẵn trong Python chỉ trả về những giá trị phù hợp với các tiêu chí nhất định

Cú pháp: **filter(function, iterable)**

Iterable: có thể là một chuỗi bất kỳ như List, Set, hoặc chuỗi các đối tượng

```
list_1 = [1,2,3,4,5,6,7,8,9]
list_2=list(filter(lambda x: x%2==0, list_1))
print(list_2)
#[2, 4, 6, 8]
```

Map()

Map là một hàm có sẵn (build in) khác trong Python với cú pháp:

map(function, iterable).

Kết quả là một danh sách đã được thay đổi so với danh sách ban đầu, thay đổi được mô tả bởi hàm function.

```
list_1 = [1,2,3,4,5,6,7,8,9]
cubed = map(lambda x: pow(x,3), list_1)
print(list(cubed))
###Results
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Series object

Đối tượng Series là một cột trong khung dữ liệu, hay nói cách khác là một chuỗi các giá trị với các chỉ số tương ứng.

Hàm lambda có thể được sử dụng để thao tác dữ liệu trong mộ Pandas frame

```
import pandas as pd
df = pd.DataFrame({
'Name': ['Luke','Gina','Sam','Emma'],
'Status': ['Father', 'Mother', 'Son', 'Daughter'],
'Birthyear': [1976, 1984, 2013, 2016],
})
```

	Name	Status	Birthyear
0	Luke	Father	1976
1	Gina	Mother	1984
2	Sam	Son	2013
3	Emma	Daughter	2016

Lambda with Apply() function by Pandas.

Để có được tuổi hiện tại của mỗi thành viên, chúng ta lấy năm hiện tại trừ đi năm sinh của họ. Trong hàm lambda bên dưới, x tham chiếu đến một giá trị trong cột năm sinh

```
df['age'] = df['Birthyear'].apply(lambda x: 2021-x)
```

	name	Status	Birthyear	age
0	Luke	Father	1976	45
1	Gina	Mother	1984	37
2	Sam	Son	2013	8
3	Emma	Daughter	2016	5

Lambda with Python's Filter() function.

```
list(filter(lambda x: x>18, df['age']))
```

conditional operations

df['Gender'] = df['Status'].map(lambda x: 'Male' if x=='father' or x=='son' else 'Female')

	Name	Status	Birthyear	age	double_age	Gender
0	luke	father	1976	45	90	Male
1	gina	mother	1984	37	74	Female
2	sam	son	2013	8	16	Male
3	emma	daughter	2016	5	10	Female