Python cho Khoa học dữ liệu

Bài 7: Hàm và Modules

Hà Minh Tuấn

Khoa Toán - Tin học
Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM

Ngày 31 tháng 8 năm 2025

Nội dung bài học

Python Functions

- Hàm (function) là **khối mã tái sử dụng**, thực hiện một tác vụ duy nhất.
- Giúp chương trình có **tính mô-đun** và dễ bảo trì.
- Một hàm có thể nhận **tham số (arguments)** và trả về **giá trị (return value)**.
- Có 3 loại hàm:
  - Built-in functions (`print()`, `len()`, `sum()`, ...)
  - Hàm trong built-in modules (cần `import`)
  - User-defined functions (tự định nghĩa)

Định nghĩa một hàm

Cú pháp:

```
def function_name(parameters):
    "docstring (tùy chọn)"
    function_suite
    return [expression]
```

Ví dụ:

```
def greetings():
    "This is docstring of greetings function"
    print("Hello World")
    return
```

Gọi hàm Python

```
# Định nghĩa hàm
def printme(str):
    "In ra chuỗi được truyền vào"
    print(str)
    return

# Gọi hàm
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Kết quả chạy

```
I'm first call to user defined function!
Again second call to the same function
```

Default Arguments trong Python

- Hàm Python cho phép gán **giá trị mặc định** cho một hoặc nhiều tham số.
- Khi gọi hàm:
  - Nếu không truyền giá trị cho tham số $\Rightarrow$ dùng giá trị mặc định.
  - Nếu truyền giá trị $\Rightarrow$ giá trị mặc định sẽ bị **ghi đè**.
- Cú pháp:

```
def func(arg1, arg2=default):
    statements
```

Ví dụ: Default Argument

```python
# Định nghĩa hàm
def showinfo(name, city="Hyderabad"):
    print("Name:", name)
    print("City:", city)

# Gọi hàm
showinfo(name="Ansh", city="Delhi")
showinfo(name="Shrey")
```

Kết quả:

```
Name: Ansh
City: Delhi
Name: Shrey
City: Hyderabad
```

Ví dụ khác: Tính % với default argument

```
def percent(phy, maths, maxmarks=200):
    val = (phy + maths) * 100 / maxmarks
    return val

print("percentage:", percent(60, 70))
print("percentage:", percent(40, 46, 100))
```

Kết quả:

```
percentage: 65.0
percentage: 86.0
```

Lưu ý: Mutable Default Arguments

- Python chỉ **đánh giá giá trị mặc định một lần** khi hàm được định nghĩa.
- Nếu giá trị mặc định là **mutable object** (list, dict, ...) và bị thay đổi, thì thay đổi đó sẽ được lưu lại cho những lần gọi sau.

Ví dụ

```
def fcn(nums, numericlist=[]):
    numericlist.append(nums + 1)
    print(numericlist)

fcn(66)
fcn(68)
fcn(70)
```

```
[67]
[67, 69]
[67, 69, 71]
```

Keyword Arguments trong Python

- Python cho phép truyền đối số dưới dạng **keyword arguments** (named arguments).
- Khi gọi hàm, bạn có thể chỉ rõ tên tham số và giá trị.
- Ưu điểm:
  - Không cần tuân theo thứ tự các tham số trong định nghĩa hàm.
  - Giúp code dễ đọc, rõ nghĩa.

Ví dụ: Positional vs Keyword Arguments

```python
def printinfo(name, age):
    print("Name:", name)
    print("Age", age)

# Positional arguments
printinfo("Naveen", 29)

# Keyword arguments
printinfo(name="miki", age=30)
```

```
Name: Naveen
Age 29
Name: miki
Age 30
```

Thứ tự Keyword Arguments

```python
def division(num, den):
    q = num/den
    print("num:{} den:{} quotient:{}".format(num, den, q))

# Gọi theo vị trí
division(10, 5)
division(5, 10)

# Gọi theo keyword
division(num=10, den=5)
division(den=5, num=10)
```

```
num:10 den:5 quotient:2.0
num:5 den:10 quotient:0.5
num:10 den:5 quotient:2.0
num:10 den:5 quotient:2.0
```

Lưu ý khi kết hợp Positional và Keyword

- Positional arguments phải đứng **trước** keyword arguments.
- Nếu vi phạm, Python báo lỗi `SyntaxError`.

Ví dụ: Lỗi khi trộn

Vì đối số Vị trí không thể xuất hiện sau đối số từ khóa, Python đưa ra thông báo lỗi sau

```
def division(num, den):
    print(num/den)

# Sai: positional sau keyword
division(num=5, 10)
```

```
SyntaxError: non-keyword arg after keyword arg
```

Keyword-Only Arguments

- Trong Python, bạn có thể **bắt buộc** tham số chỉ được truyền bằng **keyword**.
- Đặt dấu hoa thị (*) trước danh sách tham số keyword-only trong định nghĩa hàm.
- Các tham số sau * không thể gọi theo vị trí.
- Ví dụ: `print()` dùng keyword-only arguments như `sep`, `end`.

Ví dụ: print() với keyword-only argument

```
print("Hello", "World", sep="-")
```

```
Hello-World
```

Sai khi dùng sep như positional argument

```
print("Hello", "World", "-")
```

```
Hello World -
```

Ví dụ: Hàm user-defined với keyword-only

```
def intr(amt, *, rate):
    return amt * rate / 100

print(intr(1000, rate=10))
```

```
100.0
```

Sai khi gọi bằng positional argument

```
def intr(amt, *, rate):
    return amt * rate / 100

print(intr(1000, 10))
```

```
TypeError: intr() takes 1 positional argument
but 2 were given
```

Positional Arguments

- Các biến khai báo trong dấu ngoặc khi định nghĩa hàm gọi là **formal arguments**.
- Khi gọi hàm, các giá trị truyền vào theo đúng thứ tự gọi là **positional arguments**.
- Đặc điểm:
  - Tất cả các đối số đều bắt buộc.
  - Số lượng đối số thực tế = số lượng formal arguments.
  - Giá trị được gán theo **thứ tự** định nghĩa.
  - Kiểu dữ liệu phải khớp.
  - Tên biến trong lời gọi hàm không nhất thiết phải giống tên formal.

Ví dụ 1: Positional Arguments chuẩn

```
def add(x, y):
    z = x + y
    print("x={} y={} x+y={}".format(x, y, z))

a = 10
b = 20
add(a, b)
```

```
x=10 y=20 x+y=30
```

Ví dụ 2: Thiếu đối số

```
def add(x, y):
    z = x + y
    print(z)

a = 10
add(a)
```

```
TypeError: add() missing 1 required positional
argument: 'y'
```

Ví dụ 3: Thừa đối số

```
def add(x, y):
    z = x + y
    print("x={} y={} x+y={}".format(x, y, z))

add(10, 20, 30)
```

```
TypeError: add() takes 2 positional arguments
but 3 were given
```

Ví dụ 4: Sai kiểu dữ liệu

```
def add(x, y):
    z = x + y
    print(z)

a = "Hello"
b = 20
add(a, b)
```

```
TypeError: can only concatenate str (not "int")
to str
```

Positional vs Keyword Arguments

Positional Argument	**Keyword Argument**
Chỉ dùng giá trị theo thứ tự	Truyền theo dạng `name=value`
Phải theo thứ tự định nghĩa	Thứ tự có thể thay đổi
`f(a, b)`	`f(x=10, y=20)`

Positional-Only Arguments

- Python cho phép định nghĩa hàm trong đó một hoặc nhiều tham số chỉ nhận giá trị theo **vị trí**, không thể dùng keyword.
- Để khai báo, dùng dấu gạch chéo / trong định nghĩa hàm.
- Các tham số đứng trước / sẽ là positional-only.
- Ví dụ: `input()` là hàm built-in positional-only.

Ví dụ: input() với keyword argument

```
name = input(prompt="Enter your name ")
```

```
TypeError: input() takes no keyword arguments
```

Ví dụ 1: Hàm chỉ có positional-only arguments

```
def intr(amt, rate, /):
    return amt * rate / 100

print(intr(316200, 4))
```

```
12648.0
```

Ví dụ 2: Gọi sai bằng keyword arguments

```
def intr(amt, rate, /):
    return amt * rate / 100

print(intr(amt=1000, rate=10))
```

```
TypeError: intr() got some positional-only
arguments passed as keyword arguments: 'amt, rate'
```

Ví dụ 3: Kết hợp positional-only, regular và keyword-only

```
def myfunction(x, /, y, *, z):
    print(x, y, z)

myfunction(10, y=20, z=30)
myfunction(10, 20, z=30)
```

```
10 20 30
10 20 30
```

Arbitrary Arguments

- Hàm có thể nhận số lượng đối số tùy ý.
- Dùng `*args` để nhận **arbitrary positional arguments**.
- Dùng `**kwargs` để nhận **arbitrary keyword arguments**.
- Quy tắc:
  - `*args` gom thành **tuple**.
  - `**kwargs` gom thành **dictionary**.
  - Thứ tự trong danh sách tham số:
    bắt buộc → tùy chọn → `*args` → `**kwargs`.

Ví dụ: *args

```
# sum of numbers
def add(*args):
    s = 0
    for x in args:
        s = s + x
    return s

print(add(10,20,30,40))
print(add(1,2,3))
```

```
100
6
```

Kết hợp đối số thường + *args

```python
# avg of first test and best of following tests
def avg(first, *rest):
    second = max(rest)
    return (first + second) / 2

result = avg(40, 30, 50, 25)
print(result)
```

```
45.0
```

Ví dụ: **kwargs

```
def addr(**kwargs):
    for k, v in kwargs.items():
        print("{}:{}".format(k, v))

print("pass two keyword args")
addr(Name="John", City="Mumbai")

print("pass four keyword args")
addr(Name="Raam", City="Mumbai",
     ph_no="9123134567", PIN="400001")
```

```
pass two keyword args
Name:John
City:Mumbai
pass four keyword args
Name:Raam
City:Mumbai
ph_no:9123134567
PIN:400001
```

Kết hợp nhiều loại đối số

```
def percent(math, sci, **optional):
    print("maths:", math)
    print("sci:", sci)
    s = math + sci
    for k, v in optional.items():
        print("{}:{}".format(k, v))
        s = s + v
    return s / (len(optional) + 2)

result = percent(math=80, sci=75,
                 Eng=70, Hist=65, Geo=72)
print("percentage:", result)
```

```
maths: 80
sci: 75
Eng:70
Hist:65
Geo:72
percentage: 72.4
```

Scope của biến trong Python

- Scope = phạm vi mà một biến có thể được truy cập.
- Phụ thuộc vào vị trí và cách biến được định nghĩa.
- Có 3 loại scope chính:
  - **Local**: bên trong hàm.
  - **Global**: toàn bộ chương trình.
  - **Nonlocal**: trong hàm lồng nhau.

Local Variables

```
def myfunction():
    a = 10
    b = 20
    print("a:", a)
    print("b:", b)
    return a+b

print(myfunction())
```

```
a: 10
b: 20
30
```

Global Variables

```
name = 'TutorialsPoint'
marks = 50

def myfunction():
    print("name:", name)
    print("marks:", marks)

myfunction()
```

```
name: TutorialsPoint
marks: 50
```

Nonlocal Variables (Nested Functions)

```
def outer():
    a = 5
    b = 6
    def inner():
        nonlocal a, b
        a, b = 10, 20
        print("a:", a)
        print("b:", b)
    inner()
outer()
```

```
a: 10
b: 20
```

Namespace trong Python

- Namespace = tập hợp các tên (biến, hàm, lớp...).
- 3 loại namespace chính:
  - **Built-in**: hàm & exception có sẵn.
  - **Global**: các tên trong chương trình chính.
  - **Local**: các tên trong một hàm.
- Các namespace lồng nhau (Local $\subset$ Global $\subset$ Built-in).

globals() và locals()

```
name = 'TutorialsPoint'
marks = 50

def myfunction():
    a = 10
    b = 20
    print("globals():", globals().keys())
    print("locals():", locals())

myfunction()
```

```
globals(): ['__name__', ..., 'name', 'marks', 'myfunction']
locals(): {'a': 10, 'b': 20}
```

Namespace Conflict

```
marks = 50    # global

def myfunction():
    marks = 70    # local
    print(marks)

myfunction()
print(marks)
```

```
70
50
```

Cập nhật Global từ trong hàm

```
var1 = 50
var2 = 60
def myfunction():
    globals()['var1'] += 10
    global var2
    var2 += 20

myfunction()
print("var1:", var1, "var2:", var2)
```

```
var1: 60 var2: 80
```

Truy cập Local từ Global → Lỗi

```
def myfunction(x, y):
    total = x + y
    print("Total:", total)

myfunction(50, 60)
print(total)   # lỗi
```

```
Total: 110
NameError: name 'total' is not defined
```

Function Annotations trong Python

- Cho phép gắn thêm **metadata** cho tham số và kiểu trả về của hàm.
- Không ảnh hưởng đến quá trình chạy chương trình (không enforce type).
- Hữu ích cho IDEs, tài liệu, và các công cụ kiểm tra kiểu tĩnh (ví dụ: `mypy`).
- Được giới thiệu từ **PEP 3107**.

Ví dụ cơ bản

```
def myfunction(a: int, b: int):
    return a + b

print(myfunction(10, 20))
print(myfunction("Hello ", "Python"))
```

```
30
Hello Python
```

Annotation với Return Type

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

print(myfunction(56, 88))
print(myfunction.__annotations__)
```

```
144
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int
```

Annotation với Expression tuỳ ý

```
def total(x: 'marks in Physics', y: 'marks in Chemistry'):
    return x + y

print(total(86, 88))
print(total.__annotations__)
```

```
174
{'x': 'marks in Physics', 'y': 'marks in Chemistry'}
```

Annotation với Default Arguments

```python
def myfunction(a: "physics", b: "maths" = 20) -> int:
    return a + b

print(myfunction(10))
print(myfunction.__annotations__)
```

```
30
{'a': 'physics', 'b': 'maths', 'return': <class 'int'>}
```

Arbitrary Arguments

```
def myfunction(*args: "arbitrary args",
               **kwargs: "arbitrary kwargs") -> int:
    pass

print(myfunction.__annotations__)
```

```
{'args': 'arbitrary args',
 'kwargs': 'arbitrary kwargs',
 'return': <class 'int'>}
```

Annotation phức tạp với dict

```
def division(num: dict(type=float, msg='numerator'),
             den: dict(type=float, msg='denominator')) ->
             ↪  float:
    return num / den

print(division.__annotations__)
```

```
{'num': {'type': <class 'float'>, 'msg': 'numerator'},
 'den': {'type': <class 'float'>, 'msg': 'denominator'},
 'return': <class 'float'>}
```

Python Modules

- Module = file `.py` chứa hàm, class, biến, hằng số...
- Giúp **tái sử dụng** và **tổ chức code**.
- Dùng `import` để nạp nội dung module.

Ví dụ Module có sẵn

```
import math
print("Square root of 100:", math.sqrt(100))
```

```
Square root of 100: 10.0
```

Một số Built-in Modules

- `os` – OS interface
- `re` – Regex
- `math, cmath`
- `datetime`
- `asyncio`
- `collections`
- `functools`
- `pickle`
- `socket`
- `sqlite3`
- `statistics`
- `typing`
- `json`
- `unittest`
- `random`
- `sys`

Tạo User-defined Module

File `mymodule.py`:

```
def SayHello(name):
    print(f"Hi {name}! How are you?")
```

Sử dụng:

```
import mymodule
mymodule.SayHello("Harish")
```

Import nâng cao

- `from mymodule import sum, average`
- `from mymodule import *` (không khuyến khích)
- `import mymodule as mm`

```
import mymodule as x
print(x.sum(10,20))
```

Module Search Path

Python tìm module theo thứ tự:

1. Thư mục hiện tại
2. Biến môi trường `PYTHONPATH`
3. Thư mục mặc định (vd: `/usr/local/lib/python/`)

`sys.path` chứa danh sách đường dẫn tìm kiếm.

Namespace và Scoping

- Mỗi hàm có **local namespace**.
- Nếu trùng tên biến, local che global.
- Dùng `global varname` để gán biến toàn cục trong hàm.

Module Attributes

- `__file__` đường dẫn file
- `__doc__` docstring của module
- `__name__` tên module
- `__dict__` scope của module

```
import mymodule
print(mymodule.\_\_name\_\_)
```

__name__ == __main__

```
def sum(x,y): return x+y

if __name__ == "__main__":
    print("sum:", sum(10,20))
```

- Khi chạy trực tiếp → `__name__ = "__main__"`
- Khi import → `__name__ = "mymodule"`

Các hàm tiện ích

```
import math
print(dir(math))    # liệt kê symbol
```

```
import imp, test
imp.reload(test)    # reload module
```

Packages trong Python

Cấu trúc thư mục:

```
Phone/__init__.py
    Pots.py
    Isdn.py
    G3.py
```

```python
# Phone/__init__.py
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

```python
import Phone
Phone.Pots()
```

Built-in Functions trong Python

- **Built-in functions** là các hàm đã được định nghĩa sẵn trong Python.
- Không cần import thư viện để sử dụng.
- Cung cấp nhiều thao tác có sẵn trên chuỗi, số, iterator, cấu trúc dữ liệu.
- Ví dụ quen thuộc: `sum()`, `min()`, `max()`, `len()`, `print()`.

Cách sử dụng Built-in Functions

- Cách gọi rất đơn giản: `ten_ham(tham_so)`.
- Tham số có thể là bắt buộc hoặc tùy chọn.
- Vì là hàm dựng sẵn nên không cần khai báo trước.

Ví dụ cơ bản với Built-in Functions

```
# Sử dụng print() và len()
text = "Tutorials Point"
print(len(text))
```

```
15
```

Một số Built-in Functions toán học

- `abs(x)`: Trị tuyệt đối của `x`.
- `max(iterable)`: Giá trị lớn nhất.
- `min(iterable)`: Giá trị nhỏ nhất.
- `pow(x, y)`: Tính `x**y`, có thể kèm mod.
- `round(x, n)`: Làm tròn số thập phân.
- `sum(iterable)`: Tổng các phần tử.

Ví dụ các hàm toán học

```
numbers = [3, -7, 5, 10]

print(abs(-7))
print(max(numbers))
print(min(numbers))
print(pow(2, 3))
print(round(3.14159, 2))
print(sum(numbers))
```

```
7
10
-7
8
3.14
11
```

Danh sách Built-in Functions (Python 3.12.2)

- `all()`, `any()`, `ascii()`
- `bin()`, `bool()`, `bytearray()`
- `bytes()`, `callable()`, `chr()`
- `dict()`, `dir()`, `divmod()`
- `enumerate()`, `eval()`, `exec()`
- `filter()`, `float()`, `format()`
- `frozenset()`, `getattr()`, `globals()`
- `hasattr()`, `hash()`, `help()`
- `hex()`, `id()`, `input()`
- `int()`, `isinstance()`, `issubclass()`
- `iter()`, `len()`, `list()`
- `locals()`, `map()`, `memoryview()`
- `next()`, `oct()`, `open()`
- `ord()`, `print()`, `property()`

Một số Built-in Functions khác

- `range()`, `repr()`, `reversed()`
- `set()`, `setattr()`, `slice()`
- `sorted()`, `staticmethod()`, `str()`
- `super()`, `tuple()`, `type()`
- `vars()`, `zip()`, `__import__()`

- (Trong Python 2: `unichr()`, `long()`)
- Các hàm này bao phủ:
  - Xử lý dữ liệu.
  - Quản lý class, đối tượng.
  - Đọc ghi file.
  - Chuyển đổi kiểu dữ liệu.

Ưu điểm của Built-in Functions

- Giúp **giảm độ dài code**, tăng tính đọc hiểu.
- Tái sử dụng, nhất quán trong chương trình.
- Đa dạng chức năng: toán học, chuyển kiểu, xử lý iterator...
- Tên hàm mô tả rõ ý nghĩa, dễ nhớ và dễ bảo trì.

Câu hỏi thường gặp

- **Xử lý lỗi khi dùng hàm?**
  $\Rightarrow$ Dùng `try-except`.
- **Có thể mở rộng chức năng hàm dựng sẵn?**
  $\Rightarrow$ Có, kết hợp với phương thức khác, nhưng không thay đổi bản chất.
- **Có tạo được hàm built-in mới không?**
  $\Rightarrow$ Không, nhưng có thể viết hàm do người dùng định nghĩa.
- **Cách dùng?**
  $\Rightarrow$ Gọi trực tiếp bằng tên hàm + tham số.