



TIN HỌC CƠ SỞ

Phần 2: Ngôn ngữ lập trình Python

CHƯƠNG 9.

HÀM (function) TRONG PYTHON

Tin cơ sở (LT): 010100229802 - DHKL16A1HN, - DHKL16A2HN

GV. Cao Diệp Thắng

Mục tiêu chương

Nắm vững:

- ☐ Khái niệm, cách khai báo hàm.
- ☐ Các tham số trong lời gọi hàm
- ☐ Cấp phát và phạm vi hoạt động của các biến
- ☐ Hàm lambda, map, filter và reduce. Built functionc
- ☐ Độ qui



Nội dung

- 1 Khái niệm hàm (function), xây dựng hàm
- 2 Biến cục bộ và biến toàn cục
- 3 Tham chiếu (reference), tham trị (value)
- 4 Tham số
- 5 lambda, map, filter và reduce
- 6 Built function
- 7 Hàm đệ qui



9.1. KHÁI NIỆM HÀM (FUNCTION)

Định nghĩa

Trong Python, **hàm/phương thức (Function)** là tập hợp các dòng code được viết để thực hiện một chức năng một tác vụ cụ thể nào đó. Mục đích xây dựng hàm là để có thể sử dụng nhiều lần và thuận tiện khi viết code. Hàm giúp chia chương trình Python thành những khối/phần/mô-đun nhỏ hơn. Khi chương trình Python quá lớn, hoặc cần mở rộng, thì các hàm giúp chương trình có tổ chức và dễ quản lý hơn.

- ❑ *Python cung cấp rất nhiều hàm xây dựng sẵn (built-in function), lập trình viên cũng có thể tự xây dựng hàm (user-defined function)*



9.2. XÂY DỰNG HÀM

Cú pháp:

```
def function_name(parameters) :  
    #mô tả về function  
    các dòng lệnh  
    return [expression]
```

Chú ý:

- Function có thể không trả về giá trị. Hoặc Function có thể trả về một hoặc nhiều giá trị bằng lệnh **return**.
- Gọi hàm thông qua tên hàm (*function_name*) và các đối số (*parameters*).

Ví dụ 9.1.

Viết function tính điểm trung bình học kỳ 1 và trung bình hk2. In kết quả trực tiếp trong function.

Công thức tính: $dtb = (hk1 + hk2 * 2) / 3$

```
def ham_tinh_diem_TB(hk1,hk2):  
    dtb=float((hk1+hk2*2)/3)  
    print("Diem trung binh nam la: %0.2f"%dtb)  
    return
```

Thực hiện gọi hàm

```
>>> hk1=float(input("Nhập điểm trung bình học kỳ 1 : "))  
Nhập điểm trung bình học kỳ 1 : 6  
>>> hk2=float(input("Nhập điểm trung bình học kỳ 2 : "))  
Nhập điểm trung bình học kỳ 2 : 7.5  
>>> ham_tinh_diem_TB(hk1,hk2)  
Diem trung binh nam la: 7.00
```

Ví dụ 9.2. Viết function tính và trả về chỉ số BMI của một người dựa trên cân nặng và chiều cao được cung cấp. Biết công thức tính BMI:

$$\text{BMI} = \text{cân nặng}/(\text{chiều cao})^2$$

Trong đó, cân nặng: kg, chiều cao : met

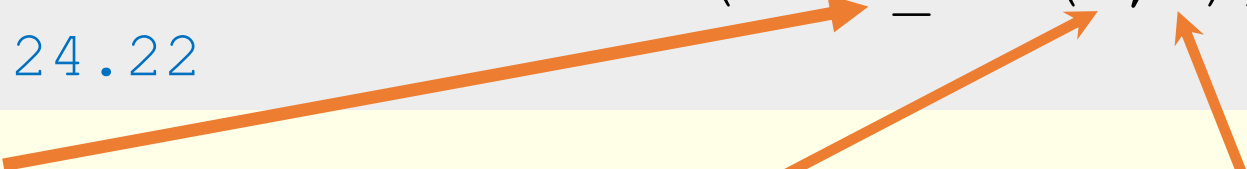
```
>>> import math
>>> def tinh_bmi(can_nang, chieu_cao):
    #Tính chỉ số BMI của một người dựa vào cân nặng và chiều cao
    bmi = can_nang/math.pow(chieu_cao,2)
    return bmi
```

Thực hiện gọi hàm **tinh_bmi()**

Cách 1

```
>>> w=float(input("Cho biet so can (kg) : "))
Cho biet so can (kg) : 70
>>> h=float(input("Cho biet so do chieu cao (m): "))
Cho biet so do chieu cao (m): 1.7
>>> print("Chi so BMI cua ban la: %0.2f"%(tinh_bmi(w,h)))
Chi so BMI cua ban la: 24.22
```

Gọi hàm **tinh_bmi()**, tham số **cân nặng**, tham số **chiều cao**



Cách 2

```
>>> bmi=tinh_bmi(w,h) #Bien bmi nhan ket qua tra ve tu ham tinh_bmi()
>>> print("Chi so BMI cua ban la: %0.2f"%bmi)
Chi so BMI cua ban la: 24.22
```


9.3. BIẾN CỤC BỘ VÀ BIẾN TOÀN CỤC

- ❑ Biến cục bộ (**local variable**) có thể được truy cập chỉ trong hàm(function) có thể được truy cập chỉ trong hàm mà các biến đó được khai báo.
- ❑ Biến toàn cục (**global variable**) có thể được truy cập trên toàn chương trình của tất cả các hàm (function)

Ví dụ 9.3

s là biến toàn cục

```
>>> s = "Hello Python"
>>> def in_s():
    print(s)
    return

>>> print(s)
Hello Python
>>> in_s()
Hello Python
```

Ví dụ 9.4

s là
biến
cục
bộ

```
>>> s = "Hello Python"
>>> def in_s():
>>>     s = "Xin chào Python"
>>>     print(s)
>>>     return
```

```
>>> in_s()
Xin chào Python
>>> print(s)
Hello Python
```

9.4. THAM CHIẾU (REFERENCE), THAM TRỊ (VALUE)

- Các tham số (*parameter/argument*) có kiểu **tham trị** trong hàm: Nếu ta thay đổi giá trị của tham số đó thì vẫn không ảnh hưởng gì đến giá trị của tham số bên ngoài hàm.
- Các tham số (*parameter/argument*) có kiểu **tham chiếu** trong hàm: nếu lập trình viên thay đổi giá trị của các tham số đó bên trong hàm thì các giá trị được thay đổi này vẫn giữ nguyên ở bên ngoài hàm.

Ví dụ 9.5.

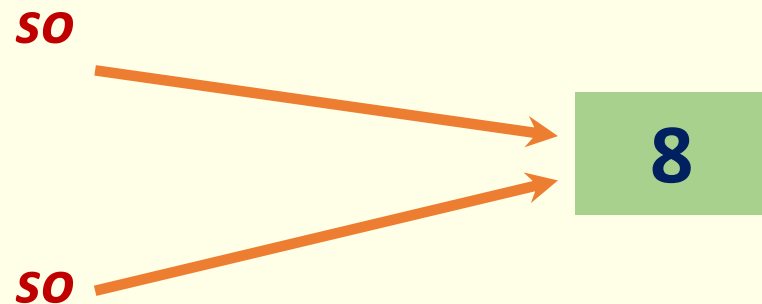
Tham trị

```
>>> def tinh_binh_phuong(so):  
    #Hàm tính bình phương của một số  
    so = so*so  
    print("Giá trị của số trong hàm :%d" %so)  
    return  
  
>>> so = 8  
>>> tinh_binh_phuong(so)  
Giá trị của số trong hàm :64  
>>> print("Giá trị của số ngoài hàm :%d"%so)  
Giá trị của số ngoài hàm :8
```

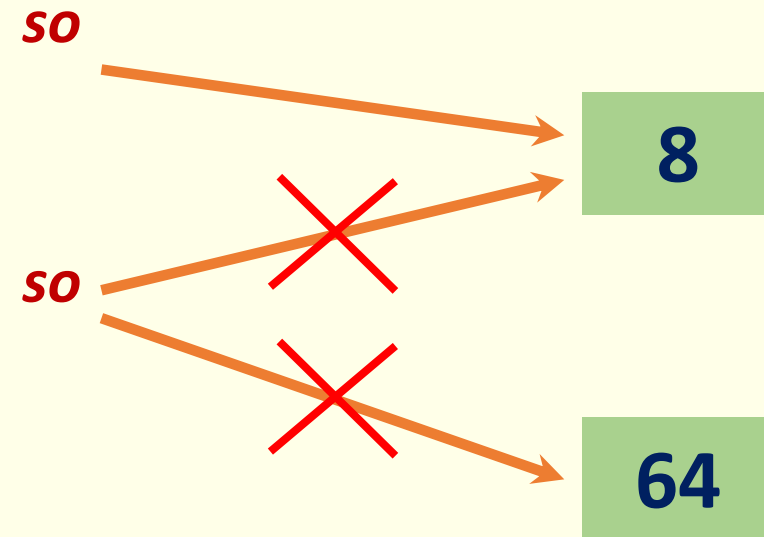
Nhận xét: Như vậy giá trị của biến *so* ngoài hàm vẫn là giá trị ban đầu trước khi gọi hàm

Trong ví dụ 9.5, cơ chế truyền đối số cho hàm **Tinh_binh_phuong(so)** xảy ra như sau

1. Khi gán **so=8**, thì số tham chiếu đến một đối tượng của lớp int, lưu giá trị là 8
2. Gọi hàm **tinh_binh_phuong(so)** với biến **so** là đối số được truyền vào cho hàm **tinh_binh_phuong(so)**
3. Sau đó, một biến **so** nữa được tạo ra, biến này cũng sẽ tham chiếu đến đối tượng int lưu trữ giá trị 8 mà biến **so** tham chiếu đến. Tức là hai biến **so** và **so** khác nhau cùng tham chiếu đến một **object 8**.
4. Trong hàm **tinh_binh_phuong()**, gán **so=so*so** thì lúc này **so** sẽ tham chiếu đến một đối tượng của lớp int lưu giữ giá trị 64. Tức là biến số trong hàm bỏ tham chiếu đến **object 8**. Lúc này chỉ còn biến **so** ngoài hàm vẫn tham chiếu đến **object 8**.
5. Sau khi kết thúc hàm **tinh_binh_phuong()**, biến **so** và object 64 trong hàm bị xóa. Lúc này biến **so** ngoài hàm vẫn tham chiếu đến **object 8** mà không bị thay đổi khi ra khỏi hàm.



*Biến **so** ngoài hàm và **so** trong hàm tham chiếu đến cùng object 8*



*Khi gán **so = so*so**, biến **so** ngoài hàm và **so** trong hàm tham chiếu đến khác object 8*

Ví dụ 9.6. Đối số của hàm là một list

Tham chiếu

```
>>> def change_list(lst):  
    #Thêm vào sau list  
    lst.append(10)  
    lst.append(20)  
    print("list trong hàm :", lst)  
    return  
  
>>> lst = [1,3,7,12]  
>>> print("list ban đầu:", lst)  
list ban đầu: [1, 3, 7, 12]  
>>>  
>>> change_list(lst)  
list trong hàm : [1, 3, 7, 12, 10, 20]  
>>>  
>>> print("List ngoài hàm :", lst)  
List ngoài hàm : [1, 3, 7, 12, 10, 20]
```

Nhận xét: trong ví dụ 9.6 giá trị của danh sách *lst* đã bị thay đổi bên trong hàm *change_list()*, sau khi gọi hàm *change_list()* giá trị của danh sách *lst* không trở lại như ban đầu mà giữ nguyên giá trị đã bị thay đổi bên trong hàm *change_list*.

Ví dụ tham chiếu liên kết dữ liệu chuẩn và list

Ví dụ 9.6

```
>>> x = 5
```

```
>>> y = x
```

```
>>> x = 3
```

```
>>> y
```

```
5
```

```
>>> x is y
```

```
False
```

← **x is y** trả về kết quả là **False**,
vậy x và y nhận các giá trị khác nhau

Lặp lại ví dụ 9.6 nhưng với hai biến nhớ kiểu dữ liệu dạng List là **xlist** và **ylist**.

Ví dụ 9.7

```
>>> xlist = [1, 2]
```

```
>>> ylist = xlist
```

```
>>> xlist[0] = 10
```

```
>>> ylist
```

```
[10, 2]
```

```
>>> xlist is ylist
```

```
True
```

← Thay đổi giá trị
thành phần của **xlist**

← Kết quả **xlist is ylist** trả về True,
vậy xlist và ylist giống nhau
không như trong ví dụ trên 9.6

9.5. THAM SỐ

- ❑ Tham số trong python được phân thành 4 loại
 - **Required** argument (tham số bắt buộc)
 - **Keyword** argument (tham số từ khóa)
 - **Default** argument (tham số mặc định)
 - **Variable-length** argument (tham số thay đổi không xác định)

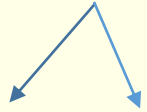
9.5.1. Tham số bắt buộc

- ❑ Là tham số khi truyền vào function phải đúng vị trí
- ❑ Yêu cầu số lượng các tham số khi gọi function cần phải trùng khớp chính xác với danh sách tham số của function đã được xây dựng.

Ví dụ 9.8 Hàm tính BMI trong ví dụ 8.2.2 có 2 tham số kiểu số, vì vậy khi gọi hàm bmi cũng phải truyền vào 2 giá trị kiểu số **đúng theo thứ tự** đã định nghĩa


```
>>> import math
>>> def tinh_bmi(can_nang, chieu_cao):
    bmi = can_nang/math.pow(chieu_cao, 2)
    return bmi
```


Truyền đúng 02 tham số là kiểu số
cho hàm **tingh_bmi()**



```
>>> bmi = tingh_bmi(52, 1.6)
>>> print("BMI:", bmi)
BMI: 20.3124999999999996
>>>
```

Truyền thiếu tham số cho
hàm **tingh_bmi()**



```
>>> tingh_bmi(50)
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    tingh_bmi(50)
TypeError: tingh_bmi() missing 1 required positional
argument: 'chieu_cao'
```

9.5.2. Tham số từ khóa (keyword argument)

- Là tham số liên quan đến lời gọi hàm function.
- Khi lập trình viên sử dụng tham số từ khóa thì lời gọi hàm sẽ ưu tiên việc thực hiện dựa trên tên tham số (không cần phải theo thứ tự).
- Tuy nhiên, số lượng tham số vẫn phải chính xác.

Ví dụ 9.9 Hàm tính BMI trong ví dụ 8.2.2 có 2 tham số kiểu số, vì vậy khi gọi hàm bmi cũng phải truyền vào 2 giá trị kiểu số. tuy nhiên không cần theo đúng thứ tự mà bắt buộc phải theo **đúng tên tham số**.

```
>>> import math
>>> def tinh_bmi(can_nang, chieu_cao):
    bmi = can_nang/math.pow(chieu_cao, 2)
    return bmi
```

```
>>> bmi = tingh_bmi(chieu_cao = 1.6, can_nang = 52)
```

```
>>> print("BMI:", bmi)
```

```
BMI: 20.3124999999999996
```

```
>>>
```

```
>>> bmi = tingh_bmi(cao = 1.6, nang = 52)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#29>", line 1, in <module>
```

```
    bmi = tingh_bmi(cao = 1.6, nang = 52)
```

```
TypeError: tingh_bmi() got an unexpected keyword  
argument 'cao'
```

9.5.3. Tham số mặc định (Default argument)

- ❑ Tham số mặc định là tham số sẽ được function tự động lấy giá trị mặc định để thực hiện nếu khi gọi function người dùng không cung cấp giá trị cho nó.

Ví dụ 9.10: với function **choose_drink()**, nếu người dùng không truyền vào drink thì mặc định là **'coffee'**

```
>>>def choose_drink(price, drink = 'coffe') :  
    print("With", price, "vnd, you can buy", drink)  
    return
```

```
>>>choose_drink(10000, "tea")  
With 10000 vnd, you can buy tea  
>>>  
>>> choose_drink(10000)  
With 10000 vnd, you can buy coffe
```

Chú ý: các tham số có giá trị mặc định phải đứng cuối danh sách tham số

9.7. LAMBDA, MAP, FILTER VÀ REDUCE

9.7.1. Hàm ẩn danh(Anonymous Function) **lambda**

Định nghĩa: Anonymous Function (Phương thức ẩn danh): gọi là ẩn danh vì hàm không được khai báo theo cách tiêu chuẩn sử dụng từ khóa **def** mà được viết ngắn gọn bằng cách sử dụng từ khóa **lambda** để tạo ra các phương thức ngắn (chỉ 1 dòng lệnh)

Cú pháp: **Lambda**[parameter1[, parameter2,...]]: **expression**

Chú ý : biểu thức lambda

- ☐ Lambda có thể có một hoặc nhiều tham số truyền vào nhưng chỉ trả về một giá trị duy nhất
- ☐ Phần thân của lambda chỉ là một biểu thức, không thể viết nhiều dòng hoặc quá nhiều lệnh. Lambda không thể chứa nhiều lệnh hoặc expression
- ☐ Biểu thức lambda có thể gán cho một biến và sử dụng như một hàm.
- ☐ Hàm lambda có thể được khai báo trực tiếp ở nơi cần dùng, vì vậy tránh được nhu cầu xây dựng các hàm con sử dụng một lần.

Ví dụ 9.13: Viết chương trình tính biểu thức $s = (x^2 + 1)^n$, x và n nhập từ bàn phím.

#Áp dụng hàm lambda

```
>>> import math
>>> s = lambda x, n: math.pow(math.pow(x, 2) + 1, n)
>>>
>>> print("s = ", s(2, 3))
s = 125.0
>>>
>>> print("s = ", s(3, 3))
s = 1000.0
```

#Viết hàm f(x,n)

```
>>> def f(x, n):
    return math.pow(math.pow(x, 2) + 1, n)
>>> print("s = ", f(2, 3))
s = 125.0
```

9.7.2. Phép ánh xạ map()

- Hàm “map” cho phép ánh xạ từ kiểu tuần tự sang một danh sách thông qua một hàm ánh xạ nào đó.
- Cho phép tạo sequence mới từ các sequence đã có bằng cách xử lý các phần tử.
- Tạo ra một sequence mới dựa trên một phương thức và sequence cũ, mỗi phần tử trong sequence cũ sẽ áp dụng phương thức để thành phần tử mới.
- Nếu có nhiều hơn một sequence được cung cấp thì phương thức sẽ được gọi kết hợp cho từng phần tử của các sequence.
- Nếu một sequence ngắn hơn so với một sequence khác ➔ kết quả sẽ có số phần tử bằng với sequence ngắn hơn.

Cú pháp:

map(hàm_ánh_xạ, sequence, [sequence...]) ➔ list

Ví dụ 9.14

```
1.>>> def binh_phuong(x):
2.         return x*x
3.>>> List_1 = [1,2,3,4,5]
4.
5.>>> List_2 = list(map(binh_phuong, List_1))
6.>>> print(List_2)
[1, 4, 9, 16, 25]
```

Nhận xét: có thể sử dụng kết hợp hàm **lambda** thay cho hàm ánh xạ **binh_phuong()** ở dòng lệnh 5 ở trên khi áp dụng hàm **map()** như sau:

```
>>> List_2 = list(map(lambda i: i**2, List_1))
>>> print(List_2)
[1, 4, 9, 16, 25]
```


Ví dụ 9.15

```
>>>list_1 = [1,2,3,4,5]
>>>List_1 = [1,2,3,4,5]
>>>List_3 = [6,7,8,9,10]
>>>print("List_1 :", List_1)
List_1 : [1, 2, 3, 4, 5]
>>>
>>>List_4=list(map(lambda i1,i2: i1+i2, List_1, List_3))
>>>print(List_4)
[7, 9, 11, 13, 15]
```

Ví dụ 9.16

```
>>>from operator import add
>>>list_new=list(map(lambda i1,i2:i1+i2, [1,2,3,4], (2,3,4,5)))
>>> print('list_new: ', list_new)
list_new:  [3, 5, 7, 9]
>>>
>>>list_new_1=list(map(lambda i1,i2:i1-i2, [1,2,3], [2,3,4,5,6]))
>>> print('list_new_1: ', list_new_1)
list_new_1:  [-1, -1, -1]
>>>
>>>print("list_add:",list(map(add, [1,2,3,4,5], [6,7,8,9,10])))
list_add: [7, 9, 11, 13, 15]
```

9.7.3. Hàm Filter

Sử dụng để lọc các item trong sequence, tạo ra một sequence (kiểu dữ liệu tuần tự) mới với các item thỏa mãn điều kiện của function

Cú pháp: **filter**(function, sequence) → **list**

Ví dụ 9.17

```
>>>list_number = [1,2,3,4,5,6,7,8,9,10]
>>>list_even_number=list(filter(lambda i: i%2 ==0, list_number))
>>> print('list_even_number', list_even_number)
list_even_number [2, 4, 6, 8, 10]
```

```
>>> list_string=["abc","def", "abf", "stu", "cba"]
>>>list_string_a = list(filter(lambda i: 'a' in i, list_string))
>>> print('list_string_a:', list_string_a)
list_string_a: ['abc', 'abf', 'cba']
```

```
>>>print('list_filter',list(filter(lambda i:i=='a', "hello abababa")))
list_filter ['a', 'a', 'a', 'a']
```

9.7.4. Hàm reduce()

❑ Trả về kết quả là một giá trị đơn bằng cách áp dụng phương thức cho các item trong sequence

Cú pháp:

reduce (<function>, <sequence>, [initializer]) → **value**

Ở đây hàm function phải là một hàm có 2 tham số

↑ Khởi tạo

Lưu ý:

1. <function> phải là hàm có hai tham số
2. Tham số của hàm <function> phải cùng kiểu với giá trị trong <sequence>
3. Kết quả trả về của <function> phải cùng kiểu với giá trị trong <sequence> (và cùng kiểu với tham số đầu vào của chính <function>)
4. Đối số thứ ba là tùy chọn. Giá trị mặc định là None. Nếu chúng ta truyền vào một Initializer, nó sẽ được sử dụng làm giá trị đầu tiên (thay vì phần tử đầu tiên của sequence).
5. Kết quả trả về của reduce() là một giá trị cùng kiểu với kết quả trả về của <function>

Ví dụ 9.18: tìm tổng của các số trong list.

```
>>> from functools import reduce          #import reduce từ gói functools
>>> from operator import add
>>> list_number = [1,2,3,4,5]
>>> sum1 = reduce(lambda x, y: x+y, list_number)
>>> print("Tong : =", sum1)
Tong : = 15

>>>
>>> sum2 = reduce(add, list_number)
>>> print("Tong : =", sum2)
Tong : = 15
```

9.8. BUILT FUNCTION

- ❑ **Build-in functions** là những hàm được cung cấp sẵn bởi ngôn ngữ Python, được hiểu là các function được đánh giá là "sử dụng nhiều" khi lập trình.
- ❑ Khi sử dụng build-in function, lập trình viên sẽ rút ngắn được code và trong một số trường hợp sẽ đảm bảo hiệu năng tính toán (với bài toán số "cực lớn" có thể gây tràn memory thì code tay sẽ cho độ phức tạp tính toán tốt hơn)
- ❑ Có thể xem và tham khảo cách sử dụng các build-in functions bằng cách truy cập vào link từ trang chủ của Python: <https://docs.python.org/3/library/functions.html>

Lưu ý: Các hàm **map()**, **filter()**,...được gọi là built function

9.9. HÀM ĐỆ QUY

9.9.1. Khái niệm

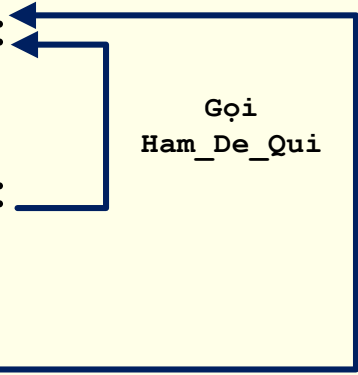
Đệ quy là cách lập trình hoặc code một vấn đề, trong đó **hàm tự gọi lại chính nó** một hoặc nhiều lần trong khối code. Thông thường, nó trả giá trị trả về của lần gọi hàm. Nếu định nghĩa hàm thỏa mãn điều kiện đệ quy thì hàm được gọi là hàm đệ quy.

Cú pháp:

```
def  tên_ham() :  
    #code  
    ...  
    ten_ham()  
return
```

Tên hàm phải xuất hiện (được gọi)
ít nhất một lần trong thân hàm

```
def  Ham_De_Quy() :  
    ...  
    Ham_De_Quy() :  
    ...  
Ham_De_Quy() :
```



Hình 9.1. Minh họa hoạt động của hàm Ham_De_Quy()

9.9.2. Cấu trúc chung của hàm đệ quy

- ❑ **Điều kiện chấm dứt:** Một hàm đệ quy cần phải có điều kiện chấm dứt để dừng việc tự gọi lại nó. Hàm đệ quy chấm dứt khi mỗi lần gọi đệ quy thì số giải pháp của vấn đề được giảm bớt và tiến gần đến ***điều kiện cơ sở***.
- ❑ **Một điều kiện cơ sở** là điểm mà ở đó vấn đề được giải quyết và không cần đệ quy thêm. *Nếu các lần gọi đệ quy không thể đến được điều kiện cơ sở thì hàm đệ quy trở thành một vòng lặp vô hạn.*

Ví dụ 9.23. Viết hàm tính giai thừa theo giải thuật đệ qui.

```
1  #Ví dụ hàm tính giai thừa đệ qui
2  def giai thua(n):
3      if n==1:
4          return 1
5      else:
6          return (n * giai thua(n-1))
7  number1 = 5
8  number2 = int(input("Nhập số cần tính giai thừa: "))
9  print("Giai thừa của", number1, "là", giai thua(number1))
10 print("Giai thừa của", number2, "là", giai thua(number2))
```

Kết quả:

```
Nhập số cần tính giai thừa: 3
Giai thừa của 5 là 120
Giai thừa của 3 là 6
```

Khi gọi hàm **giaithua**(n) với số nguyên dương, nó sẽ gọi đệ quy bằng cách giảm dần số. Mỗi một lần gọi hàm, nó sẽ nhân số với giai thừa của 1 cho đến khi số bằng 1. Giả sử với n=4, việc gọi đệ quy này có thể được giải thích trong các bước sau:

1	giaithua (4)	# Lần gọi đầu tiên với 4
2	4 * giaithua (3)	# Lần gọi thứ hai với 3
3	4 * 3 * giaithua (2)	# Lần gọi thứ ba với 2
4	4 * 3 * 2 * giaithua (1)	# Lần gọi thứ tư với 1
5	4 * 3 * 2 * 1	# Trả về từ lần gọi 4 khi n =1
6	4 * 3 * 2	# Trả về từ lần gọi 3
7	4 * 6	# Trả về từ lần gọi 2
8	24	# Trả về từ lần gọi 1

9.9.3. Ưu/nhược điểm của hàm đệ quy

a. Ưu điểm:

- Các hàm đệ quy làm cho code trông gọn gàng và nhẹ nhàng hơn.
- Những nhiệm vụ phức tạp có thể được chia thành những vấn đề đơn giản hơn bằng cách sử dụng đệ quy.
- Tạo trình tự với đệ quy dễ dàng hơn so với việc sử dụng những vòng lặp lồng nhau.

b. Nhược điểm của đệ quy:

- Đôi khi logic đằng sau đệ quy khá khó để hiểu rõ.
- Gọi đệ quy tốn kém (không hiệu quả) vì chúng chiếm nhiều bộ nhớ và thời gian.
- Các hàm đệ quy rất khó để gỡ lỗi.
- Mỗi một lần hàm đệ quy tự gọi nó sẽ lưu trữ trên bộ nhớ. Vì thế, nó tiêu tốn nhiều bộ nhớ hơn so với hàm truyền thống.

Ví dụ: Theo mặc định, độ sâu (depth) tối đa của đệ quy là 1000. Nếu vượt qua giới hạn, nó dẫn đến lỗi Đệ quy `RecursionError`.

Ví dụ 9.24.

```
1  def giaithua(n):
2      if n==1:
3          return 1
4      else:
5          return (n * giaithua(n-1))
6  number = int(input("Nhập số cần tính giai thừa: "))
7  print("Giai thừa của", number, "là", giaithua(number))
```

Kết quả:

Nhập số cần tính giai thừa: 1001

Traceback (most recent call last):

...

RecursionError: maximum recursion depth exceeded in comparison

RecursionError

Trong ví dụ 9.24, hàm **giaithua()** viết trong ví dụ 9.23 được gọi với $n=1001$ vượt quá 1000 lần, do đó khi chạy chương trình đã xảy ra lỗi đệ qui **RecursionError**.

Để khắc phục điều này chúng ta có thể gọi phương thức **setrecursionlimit()** trong module **sys** có sẵn trong thư viện module của python. Phương thức **setrecursionlimit()** cho phép lập trình viên điều chỉnh độ sâu (depth) tối đa đệ qui đến một giá trị ấn định.

Chẳng hạn chúng ta đặt là 5000, minh họa trong ví dụ 9.25 sau:

Ví dụ 9.25

```
1  import sys
2  sys.setrecursionlimit(5000)
3  def giaithua(n):
4      if n==1:
5          return 1
6      else:
7          return (n * giaithua(n-1))
8  number = int(input("Nhập số cần tính giai thừa: "))
9  print("Giai thừa của", number, "là", giaithua(number))
```

Khi thực hiện ví dụ 9.25 chúng ta có thể gọi được hàm giaithua() trong phạm vi độ sâu mới không vượt quá 5000.

Nhận xét: Để tính giai thừa thì đệ quy không phải là giải pháp tốt nhất. Các giải pháp khác chẳng hạn “Phá đệ qui” sẽ được đề cập trong học phần cấu trúc dữ liệu và giải thuật.

Câu hỏi thảo luận

1. Trình việh hàm toán học trong Python?.
2. Nêu khái niệm tham chiếu, tham trị?
3. Trình bày khái niệm và cấu trúc chung của hàm đệ qui.
4. Nêu khái niệm biến toàn cục, biến cục bộ
5. Trình bày các loại tham số trong Python?
6. Trình bày cấu trúc hàm lambda, map, filter? Cho ví dụ?

Bài tập vận dụng

Làm bài tập chương 9 – TLHT Tin cơ sở

