LẬP TRÌNH PYTHON TỪ CƠ BẢN ĐẾN NÂNG CAO

Thực hiện: TS.Trần Bình Long

Hàm

1. Hàm
2. Hàm Python tích hợp sẵn
3. Hàm do người dùng tự định nghĩa
4. Tham số trong hàm
5. Hàm đệ quy
6. Hàm vô danh, Lambda
7. Biến toàn cục (global), biến cục bộ (local), biến nonlocal
8. Từ khóa global
9. Module
10. Package

Hàm

Hàm là một nhóm các lệnh có liên quan đến nhau dùng để thực hiện một nhiệm vụ cụ thể nào đó.

Hàm chia chương trình thành những phần nhỏ. Khi chương trình quá lớn, hoặc cần mở rộng, thì hàm giúp chương trình có tổ chức và dễ quản lý hơn.

Hàm tránh việc lặp lại các lệnh thực thi những nhiệm vụ tương tự nhau, giúp chương trình gọn hơn và tái sử dụng được.

Cú pháp của hàm

```
>>> def ten_ham(các tham số/đối số):
        Các câu lệnh
```

Thành phần của hàm

Hàm Python bao gồm các thành phần:

1. Từ khóa def: Bắt đầu tiêu đề hàm.
2. ten_ham: Là định danh duy nhất. Việc đặt tên hàm phải tuân theo quy tắc viết định danh trong Python.
3. Các tham số: Tham số là tùy chọn, dùng truyền giá trị cho hàm.
4. Dấu hai chấm(:): Kết thúc tiêu đề hàm.
5. Docstring: Chuỗi văn bản tùy chọn mô tả hàm.
6. Các câu lệnh: Nhiều lệnh tạo thành khối lệnh, có cùng mức thụt lề (4 khoảng trắng).
7. Lệnh return: Lệnh này tùy chọn, trả về giá trị từ hàm

Gọi hàm

Định nghĩa một hàm đơn giản, gồm tên hàm, tham số của hàm, mô tả hàm và câu lệnh:

```
>>> def Chao(ten):
        """Hàm dùng để chào người được truyền vào
        như một tham số"""
        print('Xin chào, ' + ten + '!')
```

Gọi hàm

```
>>> Chao ('Bạn thân yêu')
```

Kết quả:

```
>>>Xin chào, Bạn thân yêu!
```

Gọi hàm

Ví dụ đơn giản về hàm trong Python.

```
>>> # In lời chào ra màn hình:
>>> def Hello():
        print("Chúng tôi xin chào!")
        return
>>> Hello() # Lời gọi hàm
```

Định nghĩa một hàm trong Python và gọi nó.

Kết quả:

```
>>> Chúng tôi xin chào!
```

Lệnh return thường được dùng để thoát hàm và trở về nơi hàm được gọi.

Nên viết hoa tên hàm giúp phân biệt hàm và các câu lệnh.

Gọi hàm

Gọi hàm từ một hàm khác:

```
>>> def Hello():
        print(“Chúng tôi xin chào!")
        NgayNghi()
        return
>>> def NgayNghi():
        print("Hôm nay là ngày nghỉ, đúng không!")
        return
    Hello()
```

kết quả:

```
>>> Chúng tôi xin chào!
    Hôm nay là ngày nghỉ, đúng không!
```

Truyền dữ liệu bằng hàm

Python cho phép gọi hàm trong khi truyền dữ liệu.

Ví dụ:

```
>>> def XinChao(Name):
        print(‘Xin chào ‘ + Name)
        return
>>> XinChao(‘Cả nhà’)
```

Kết quả:

```
>>> Xin chào Cả nhà
```

Hàm có thể thực hiện các chức năng khác nhau, tùy thuộc vào tham số.

Thao tác dữ liệu trong hàm

Hàm nhận tham số truyền vào, thực hiện và trả về kết quả.

```
>>> def PhepNhan(so):
        return so * 10
>>> print(PhepNhan(5))
```

kết quả trả về

```
>>> 50
>>> def DoDaiTen(ten):
        return len(ten)
>>> nhapten = 'LacHong'
    print(DoDaiTen(nhapten))
>>> Output: 7
```

Lưu ý, **hàm len()** sẽ tính cả dấu cách.

Docstring

Chuỗi đầu tiên sau tiêu đề hàm được gọi là docstring (documentation string) dùng để giải thích chức năng của hàm.

Docstring không bắt buộc, nhưng giúp người dùng khi gọi hàm có thể hiểu ngay công việc của hàm.

Docstring được viết trong cặp 3 dấu ngoặc kép, như một thuộc tính (**__doc__**) của hàm.

Vd: kiểm tra docstring của hàm Chao():

```
>>> print (Chao.__doc__)
```

Lệnh return

Lệnh ***return*** dùng để thoát khỏi hàm và trở về nơi hàm được gọi.

Cú pháp của lệnh return:

>>> return [danh_sach_bieu_thuc]

Nếu không có biểu thức nào trong câu lệnh hoặc không có lệnh ***return*** trong hàm thì hàm sẽ trả về None

Lệnh ***return*** chứa biểu thức sẽ được tính toán và trả về giá trị.

Lệnh return

```
>>> def  TuyetDoi(so):
        """Hàm trả về giá trị tuyệt đối của số nhập vào"""
        if so >= 0:
            return so
        else:
            return -so
>>> print(TuyetDoi(5))
    # Output: 5
    print(TuyetDoi(-8))
    # Output: 8
>>> num=int(input("Nhập số cần lấy giá trị tuyệt đối: "))
    print (TuyetDoi(num))
    # Output: Nhập số cần lấy giá trị tuyệt đối: -7
    7
```

Phạm vi và thời gian tồn tại của biến

Phạm vi của biến trong đoạn chương trình mà biến được khai báo.

Các tham số và biến được xác định bên trong hàm, bên ngoài không thể nhìn thấy. Những biến và tham số này chỉ có phạm vi trong hàm.

Thời gian tồn tại của biến là khoảng thời gian khi hàm được thực thi.

Biến sẽ hủy khi thoát khỏi hàm. Hàm không lưu giá trị của biến trong những lần gọi hàm trước đó.

Phạm vi và thời gian tồn tại của biến

```
>>> def HamIn():
        x = 15
        print("Giá trị bên trong hàm:",x)
    x = 30
    HamIn()
    print("Giá trị bên ngoài hàm:",x)
```

Kết quả:

```
>>> Giá trị bên trong hàm: 15
    Giá trị bên ngoài hàm: 30
```

Biến x trong hàm là biến cục bộ, chỉ có tác dụng trong hàm đó. Biến x bên ngoài hàm là biến toàn cục, có phạm vi trên toàn bộ chương trình.

Các loại hàm trong Python

Python có 2 loại hàm:

- Hàm được tích hợp sẵn trong Python
- Hàm do người dùng định nghĩa.

Các hàm tích hợp sẵn

Python có sẵn một số hàm để sử dụng. Các hàm này được gọi là hàm tích hợp: **print()**, **list()**,. ..

Trong Python 3.x có 68 hàm Python được tích hợp sẵn.

Để biết hàm làm gì, có đối số nào, nhập lệnh:

```
>>> print(ten_ham.__doc__)
```

Python sẽ giải thích đầy đủ về hàm, có thể đọc và làm ví dụ để hiểu hàm đó.

Danh sách các hàm cùng với mô tả ngắn gọn:

Hàm	Mô tả
abs()	Trả về giá trị tuyệt đối của một số
all()	Trả về True khi tất cả các phần tử trong iterable là đúng
any()	Kiểm tra bất kỳ phần tử nào của iterable là True
ascii()	Tả về string chứa đại diện (representation) có thể in
bin()	Chuyển đổi số nguyên sang chuỗi nhị phân
bool()	Chuyển một giá trị sang Boolean
bytearray()	Trả về mảng kích thước byte được cấp
bytes()	Trả về đối tượng byte không đổ
callable()	Kiểm tra xem đối tượng có thể gọi hay không
chr()	Trả về một ký tự (một chuỗi) từ Integer
classmethod()	Trả về một class method cho hàm
compile()	Trả về đối tượng code Python

complex()	Tạo một số phức
delattr()	Xóa thuộc tính khỏi đối tượng
dict()	Tạo Dictionary
dir()	Trả lại thuộc tính của đối tượng
divmod()	Trả về một Tuple của Quotient và Remainder
enumerate()	Trả về đối tượng kê khai
eval()	Chạy code Python trong chương trình
exec()	Thực thi chương trình được tạo động
filter()	Xây dựng iterator từ các phần tử True
float()	Trả về số thập phân từ số, chuỗi
format()	Trả về representation được định dạng của giá trị
frozenset()	Trả về đối tượng frozenset không thay đổi
getattr()	Trả về giá trị thuộc tính được đặt tên của đối tượng
globals()	Trả về dictionary của bảng sumbol toàn cục hiện tại
hasattr()	Trả về đối tượng dù có thuộc tính được đặt tên hay không
hash()	Trả về giá trị hash của đối tượng

help()	Gọi Help System được tích hợp sẵn
hex()	Chuyển Integer thành Hexadecimal
id()	Trả về định danh của đối tượng
input()	Đọc và trả về chuỗi trong một dòng
int()	Trả về số nguyên từ số hoặc chuỗi
isinstance()	Kiểm tra xem đối tượng có là Instance của Class không
issubclass()	Kiểm tra xem đối tượng có là Subclass của Classkhông
iter()	Trả về iterator cho đối tượng
len()	Trả về độ dài của đối tượng
list()	Tạo list trong Python
locals()	Trả về dictionary của bảng sumbol cục bộ hiện tại
map()	Áp dụng hàm và trả về một list
max()	Trả về phần tử lớn nhất

memoryview()	Trả về chế độ xem bộ nhớ của đối số
min()	Trả về phần tử nhỏ nhất
next()	Trích xuất phần tử tiếp theo từ Iterator
object()	Tạo một đối tượng không có tính năng (Featureless Object)
oct()	Chuyển số nguyên sang bát phân
open()	Trả về đối tượng File
ord()	Trả về mã Unicode code cho ký tự Unicode
pow()	Trả về x mũ y
print()	In đối tượng được cung cấp
property()	Trả về thuộc tính property
range()	Trả về chuỗi số nguyên từ số bắt đầu đến số kếtthúc
repr()	Trả về representation có thể in của đối tượng
reversed()	Trả về iterator đảo ngược của một dãy
round()	Làm tròn số thập phân

set()	Tạo một set các phần tử mới
setattr()	Đặt giá trị cho một thuộc tính của đối tượng
slice()	Cắt đối tượng được chỉ định bằng range()
sorted()	Trả về list được sắp xếp
staticmethod()	Tạo static method từ một hàm
str()	Chuyển đối tượng đã cho thành chuỗi
sum()	Thêm một mục vào Iterable
super()	Cho phép tham chiếu đến Parent Class bằng super
tuple()	Tạo một Tuple
type()	Trả về kiểu đối tượng
vars()	Trả về thuộc tính __dict__ của class
zip()	Trả về Iterator của Tuple
__import__()	Hàm nâng cao, được gọi bằng import

Hàm do người dùng định nghĩa

Hàm do người dùng tạo để thực hiện một số công việc cụ thể được gọi là hàm do người dùng định nghĩa.

Hàm có sẵn trong Python được gọi là hàm tích hợp.

Hàm được người dùng viết dưới dạng thư viện, gọi là hàm thư viện (library function). Hàm người dùng định nghĩa có thể trở thành hàm thư viện đối với người dùng nào đó.

Ưu điểm hàm do người dùng định nghĩa

Hàm do người dùng định nghĩa giúp phân tích một chương trình lớn thành những phần nhỏ, khiến chương trình dễ hiểu, dễ duy trì và gỡ lỗi hơn.

Khi một đoạn lệnh lặp lại trong chương trình, có thể sử dụng hàm cho đoạn lệnh này và thực hiện bằng lời gọi hàm.

Các lập trình viên cùng làm việc trong dự án lớn, có thể phân chia công việc bằng cách tạo các hàm khác nhau.

Hàm do người dùng định nghĩa

Cú pháp cơ bản khi định nghĩa hàm:

```
>>> def ten_ham(DoiSo1,DoiSo2,...,DoiSon)
        khối lệnh của hàm
```

vi dụ

```
>>> def ThemSo(a,b):
        tong = a + b
        return tong
    so1 = 5
    so2 = 6
    so3 = int(input("Nhập một số: "))
    so4 = int(input("Nhập một số nữa: "))
    print("Tổng hai số đầu là: ", ThemSo(so1, so2))
    print("Tổng hai số sau là: ", ThemSo(so3, so4))
```

Hàm do người dùng định nghĩa

Hàm **int()**, **input()**, **print()** là hàm tích hợp sẵn.

Định nghĩa hàm ThemSo(), có chức năng là thêm hai số, tính tổng 2 số và trả về kết quả.

Kết quả:

```
>>> Nhập một số: 8
    Nhập một số nữa: 10
    Tổng hai số đầu là: 11
    Tổng hai số sau là: 18
```

Nên đặt tên hàm theo chức năng, nhiệm vụ của hàm sẽ giúp người đọc dễ hiểu.

Tham số trong hàm

Hàm do người dùng định nghĩa có tham số:

```
>>> def Chao(ten,loi_chao):
        “””Hàm Chao gửi lời chào đến đối tượng nào đó
        với thông điệp cho trước """
        print("Xin chào",ten + ', ' + loi_chao)
>>> Chao(“Mọi người”, “học KTLT vui vẻ nha!”)
```

Kết quả:

```
>>> Xin chào Mọi người, học KTLT vui vẻ nha!
```

Hàm Chao() có 2 tham số. Do đó, nếu gọi hàm phải truyền 2 tham số.

Nếu gọi hàm với số tham số khác 2, sẽ bị báo lỗi.

Tham số mặc định trong hàm

Tham số của hàm có thể có giá trị mặc định. Dùng toán tử gán (=) cung cấp giá trị mặc định tham số.

```
>>> def Chao(ten, loi_chao = “học KTLT vui vẻ nha!”)
        “””Hàm chào với thông điệp cho trước. Nếu thông điệp không được cung cấp, sẽ mặc định là ‘học KTLT vui vẻ nha!’ “””
        print("Xin chào",ten + ', ' + loi_chao)
>>> Chao(‘Cả nhà’)
>>> Chao(‘Mọi người’, ‘cố lên mọi người’)
```

Kết quả:

```
>>> Xin chào Cả nhà, học KTLT vui vẻ nha!
    Xin chào Mọi người, cố lên mọi người
```

Tham số mặc định trong hàm

Trong hàm, số lượng tham số mặc định không giới hạn, nhưng phải nằm bên phải của tham số không có giá trị mặc định.

Ví dụ:

```
>>> def Chao(loi_chao = “hoc KTLT vui nha!”, ten):
```

sẽ báo lỗi:

```
>>> SyntaxError: non-default argument follows default argument
```

Tham số keyword

Khi gọi hàm, các giá trị sẽ được gán cho các tham số theo vị trí.

Python cho phép gọi hàm bằng tham số keyword. Khi gọi hàm, thứ tự của tham số có thể thay đổi, nhưng kết quả như gọi hàm thông thường.

```
>>> # 2 tham số keyword
    Chao(ten='Cả nhà', loi_chao = 'học KTLT không?')
>>> # 2 tham số keyword, thứ tự tham số thay đổi
    Chao(loi_chao='học KTLT không?',ten = 'Cả nhà')
>>> # 1 tham số keyword, 1 theo vị trí
    Chao('Cả nhà', loi_chao = 'học KTLT không?')
```

Tham số keyword

Có thể kết hợp tham số keyword và vị trí khi gọi hàm. Nhưng tham số keyword phải đi sau tham số vị trí. Nếu tham số vị trí ở sau tham số keyword sẽ báo lỗi:

```
>>> Chao(ten = 'Mọi người', 'học KTLT không?')
```

Thông báo lỗi:

```
>>> SyntaxError: non-keyword argument after
keyword argument
```

Tham số tùy biến trong hàm

Khi không biết trước số lượng tham số sẽ truyền vào hàm, thì sử dụng tham số tùy biến.
Trong định nghĩa hàm, dùng dấu hoa thị * trước tên tham số.

```
>>> def MonHoc(*ten):
        """Hàm này liệt kê danh sách môn học"""
        for i in ten:
            print('Môn sẽ học: ', i)
>>> MonHoc('Triết', 'Anh văn', 'Toán')
```

Kết quả:

```
>>> Môn sẽ học: Triết
    Môn sẽ học: Anh văn
    Môn sẽ học: Toán
```

Hàm đệ quy

Đệ quy là hàm tự gọi chính nó một hoặc nhiều lần.

Điều kiện dừng: hàm đệ quy cần có điều kiện dừng việc tự gọi lại nó. Khi mỗi lần gọi đệ quy, số vấn đề được giảm bớt và tiến gần đến điều kiện cơ sở. Khi đến điều kiện cơ sở, hàm đệ quy dừng

Điều kiện cơ sở: ở đó vấn đề được giải quyết và không cần đệ quy thêm.

Nếu các lần gọi đệ quy không đến được điều kiện cơ sở thì hàm đệ quy trở thành vòng lặp vô tận.

Hàm đệ quy

Tính giai thừa của một số nguyên.

```
>>> def GiaiThua(n):
        “”” Hàm tính giai thừa của một số nguyên”””
        if n == 1:
            return 1
        else:
            return (n * GiaiThua(n-1))
>>> so = 5
    so1 = int(input(‘Nhập số cần tính giai thừa: ‘))
    print(‘Giai thừa của ’, so, ‘là’, GiaiThua(so))
    print(‘Giai thừa của ‘, so1, ‘là’, GiaiThua(so1))
```

Hàm đệ quy

Phép đệ quy trên kết thúc khi số giảm xuống đến 1. Đây được gọi là điều kiện cơ sở. Mỗi hàm đệ quy phải có điều kiện cơ sở để dừng, nếu không sẽ trở thành hàm vô tận, tự gọi mãi đến nó.

Ưu điểm của hàm đệ quy

- 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 vòng lặp lồng nhau.

Nhược điểm của đệ quy

- Đệ quy khá khó hiểu.
- Gọi đệ quy tốn kém (không hiệu quả) vì chiếm nhiều bộ nhớ và thời gian.
- Các hàm đệ quy rất khó để gỡ lỗi.
- Mỗi lần hàm đệ quy tự gọi nó sẽ lưu trữ trên bộ nhớ, nên tốn nhiều bộ nhớ hơn hàm truyền thống. Vì thế, Python sẽ dừng sau 1000 lần gọi hàm.

Giới hạn của Hàm đệ quy

```
>>> def GiaiThua(n):
        “”” Hàm tính giai thừa của một số nguyên”””
        if n == 1:
            return 1
        else:
            return (n * GiaiThua(n-1))
>>> print (GiaiThua(1001))
```

Sẽ nhận báo lỗi:

```
>>> RecursionError: maximum recursion depth
exceeded in comparion
```

Giới hạn của Hàm đệ quy

Có thể điều chỉnh số lần gọi đệ quy.

```
>>> import sys
    Sys.setrecursionlimit(5000)
    def GiaiThua(n):
        “”” Hàm tính giai thừa của một số nguyên”””
        if n == 1:
            return 1
        else:
            return (n * GiaiThua(n-1))
>>> print (GiaiThua(1001))
```

Nên cẩn thận khi dùng đệ quy.

Hàm vô danh, Lambda

Trong Python, hàm vô danh là hàm được định nghĩa mà không có tên.

Hàm thường được định nghĩa bằng từ khóa def, hàm vô danh được định nghĩa bằng từ khóa lambda. Vì vậy hàm vô danh còn được gọi là hàm Lambda.

Cú pháp Hàm Lambda:

```
lambda tham_so: bieu_thuc
```

Hàm Lambda có thể có nhiều tham số nhưng chỉ có 1 biểu thức. Biểu thức sẽ thực hiện và trả về kết quả.

Hàm Lambda được dùng ở bất cứ nơi nào hàm được yêu cầu.

Hàm Lambda

ví dụ: hàm Lambda nhân đôi số nhập vào.

```
>>> nhan_doi = lambda a: a * 2
    print(nhan_doi(10))
    # Kết quả: 20
```

lambda a: a * 2 là hàm Lambda. a là tham số và a * 2 là biểu thức (thực hiện và trả về kết quả). Hàm này không có tên, một đối tượng hàm - được gán định danh là nhan_doi. Có thể gọi hàm như bình thường:

```
>>> nhan_doi = lambda a: a * 2
```

giống như:

```
>>> def nhan_doi(a):
        return a * 2
```

Sử dụng Lambda

Hàm Lambda được sử dụng khi cần dùng trong thời gian ngắn.

Hàm Lambda thường được sử dụng với các hàm tích hợp sẵn như **filter()** hay **map()**,...

Ví dụ dùng hàm **filter()** để lọc các số chẵn trong list.

```
>>> list_goc = [10, 9, 8, 7, 6, 1, 2, 3, 4, 5]
    list_moi = list(filter(lambda a: (a%2 == 0), list_goc))
    print(list_moi)
    # Kết quả: [10, 8, 6, 2, 4]
```

Sử dụng Lambda

Ví dụ dùng hàm **Lambda** với **map():**

```
>>> list_goc = [10, 9, 8, 7, 6, 1, 2, 3, 4, 5]
    list_moi = list(map(lambda a: a*2 , list_goc))
    print(list_moi)
    # Kết quả: [20, 18, 16, 14, 12, 2, 4, 6, 8, 10]
```

Biến toàn cục (global)

Biến khai báo bên ngoài hàm, trong phạm vi toàn cục gọi là biến toàn cục hay biến **global**.

```
>>> x = "Biến toàn cục" #khai báo biến x
      #Gọi x từ trong hàm ViDu1()
      def ViDu1():
          print("x trong hàm ViDu1() :", x)
      ViDu1()
      #Gọi x ngoài hàm ViDu1()
      print("x ngoài hàm ViDu1():", x)
```

Kết quả:

```
>>> x trong hàm ViDu1(): Biến toàn cục
      x ngoài hàm ViDu1(): Biến toàn cục
```

Biến toàn cục

```
>>> y = 2
      def ViDu2():
          y=y*2
          print(y)
      ViDu2()
```

Kết quả thông báo lỗi:

```
>>> UnboundLocalError: local variable ‘y’ referenced
before assignment
```

Lỗi do y trong hàm là biến cục bộ và y không được khai báo trước.

Biến cục bộ (local)

Biến khai báo bên trong hàm, trong phạm vi cục bộ gọi là biến cục bộ hay biến **local**.

```
>>> def ViDu3():
        y = "Biến cục bộ"
    ViDu3()
    print(y)
```

Kết quả báo lỗi:

```
>>> NameError: name 'y' is not defined
```

Lỗi do y là biến cục bộ, y chỉ làm việc trong hàm ViDu3(), phạm vi cục bộ.

Biến cục bộ

Để tạo một biến cục bộ, phải khai báo trong hàm.
ví dụ:

```
>>> def ViDu3():
        y = "Biến cục bộ"
        print(y)
    ViDu3()
```

Kết quả

```
>>> Biến cục bộ
```

Biến cục bộ và biến toàn cục

Cách dùng biến cục bộ và toàn cục

```
>>> x = 2
    def ViDu4():
        global x
        y = "Biến cục bộ"
        x = x * 2
        print(x)
        print(y)
>>> ViDu4()
```

Kết quả:

```
>>> 4
    Biến cục bộ
```

Biến cục bộ và biến toàn cục

Biến toàn cục và cục bộ trùng tên:

```
>>> x = 5
    def ViDu5():
        x = 10
        print("Biến x cục bộ:", x)
>>> ViDu5()
    print("Biến x toàn cục:", x)
```

Kết quả:

```
>>> Biến x cục bộ: 10
    Biến x toàn cục: 5
```

Hai kết quả khác nhau vì biến được khai báo ở cả hai phạm vi, cục bộ (trong hàm ViDu5())và toàn cục (ngoài hàm ViDu5()).

Biến nonlocal

Biến **nonlocal** được sử dụng trong hàm lồng nhau nơi phạm vi cục bộ không được định nghĩa.

Biến **nonlocal** không phải biến **local** cũng không phải biến **global**.

Biến **nonlocal** có phạm vi rộng hơn **local**, nhưng chưa đến mức **global**.

Để khai báo biến **nonlocal** dùng từ khóa **nonlocal**.

Biến nonlocal

```
>>> def HamNgoai():
        x = "Biến cục bộ"
        def HamTrong():
            nonlocal x
            x = "Biến nonlocal"
            print("Bên trong:", x)
        HamTrong()
        print("Bên ngoài:", x)
    HamNgoai()
```

Kết quả:

```
>>> Bên trong: Biến nonlocal
    Bên ngoài: Biến nonlocal
```

Từ khóa global

Từ khóa **global** được sử dụng để tạo biến **global** và thực hiện thay đổi cho biến trong bối cảnh cục bộ.

Quy tắc của từ khóa global

- Khi tạo biến trong hàm, mặc định là biến cục bộ.
- Khi tạo biến ngoài hàm, mặc định là biến toàn cục, không cần sử dụng từ khóa **global**.
- Sử dụng từ khóa **global** để đọc và ghi biến toàn cục trong hàm.
- Sử dụng từ khóa **global** bên ngoài hàm không có tác dụng.

Cách sử dụng từ khóa global

Truy cập biến toàn cục từ trong hàm

```
>>> a = 1 # Biến toàn cục
    def InBien():
        print(a)
    InBien()
```

Kết quả

```
>>> 1
```

Trường hợp cần thay đổi biến toàn cục trong hàm,

```
>>> a = 1 # Biến toàn cục
    def InBien():
        a = a + 9
        print(a)
    InBien()
```

Cách sử dụng từ khóa global

Sẽ bị báo lỗi:
>>> UnboundLocalError: local variable 'a' referenced before assignment

Do biến toàn cục chỉ có thể truy cập mà không thể thay đổi từ bên trong hàm. Muốn thay đổi biến toàn cục từ bên trong hàm, dùng từ khóa **global**

Cách sử dụng từ khóa global

```
>>> a = 1 # Biến toàn cục
    def InBien():
        global a
        a = a + 9
        print("Trong ham InBien(): ", a)
    InBien()
    print("Ngoài hàm InBien(): ", a)
```

Kết quả:
>>> Trong ham InBien(): 10
Ngoai ham InBien(): 10
Kết quả đã thay đổi biến a trong hàm InBien() và biến toàn cục bên ngoài hàm

Biến toàn cục qua mô-đun

Tạo mô-dun(file) config.py để lưu các biến toàn cục và chia sẻ thông qua mô-đun trong cùng chương trình.

Tạo **file config.py** để lưu trữ biến toàn cục:

```
>>> a = 0
    b = ‘ ‘
```

Tạo **file update.py** để thay đổi biến toàn cục:

```
>>> import config
    config.a = 10
    config.b = ‘LacHong’
```

Biến toàn cục qua mô-đun

Tạo **file kiemtra.py** để kiểm tra sự thay đổi.

```
>>> import config
    import update
    print(config.a)
    print(config.b)
```

Kết quả:

```
>>> 10
     LacHong
```

Biến toàn cục trong hàm lồng nhau

```
>>> def ham1():
        x = 20
        def ham2():
            global x
            x = 25
            print(“x của ham2: ", x)
            print(" Gọi ham2")
        ham2()
        print(“x của ham1: ", x)
    ham1()
    print("x ngoai ham: ", x)
```

Kết quả

Biến toàn cục trong hàm lồng nhau

Kết quả:

```
>>> x của ham2:  25
    Gọi ham2
    x của ham1:  20
    x ngoai ham:  25
```

Khai báo biến **global** trong ham2(). Trong ham1(), x không bị ảnh hưởng bởi từ khóa **global**.

Do từ khóa **global** để tạo biến toàn cục trong ham2() sẽ xuất hiện bên ngoài phạm vi cục bộ.

Module

Module là **file** chứa những câu lệnh và các hàm.
Module được sử dụng khi muốn chia chương trình lớn thành những **file** nhỏ hơn để dễ quản lý và tổ chức.
Phổ biến nhất là các hàm thường sử dụng sẽ được định nghĩa trong **module** và **import** vào chương trình thay vì phải viết lại hàm.

Tạo **module** tính tổng 2 số và lưu với tên tinhtong.py

```
>>> def Tong(p, q):
        """Module tính tổng 2 số và trả về kết quả"""
        tong = p + q
        return tong
```

Hàm sẽ nhận 2 số và trả về tổng của chúng.

Nhập (import) module

Để nhập **module** vào **module** khác hoặc vào chương trình, dùng từ khóa **import**. Cú pháp:

```
>>> import tinhtong
```

Không **import** trực tiếp tên của hàm được định nghĩa trong tinhtong.py, chỉ **import** tên của **module**.
Để truy cập vào hàm trong **module** dùng toán tử (.)

```
>>> tinhtong.Tong(5, 8)
    13
```

Tên hàm có phân biệt chữ hoa, chữ thường.
Python có sẵn nhiều module tiêu chuẩn, có thể kiểm tra danh sách tại địa chỉ:
https://docs.python.org/3/pymodindex.html.

Lệnh import

Lệnh **import** để nhập module có sẵn trong Python

```
>>> import math
    print("Giá trị của pi là: ", math.pi)
```

Kết quả:

```
>>> Giá trị của pi là: 3.141592653589793
```

import module và dùng **bí danh**

```
>>> import math as m
    print("Giá trị của pi là: ", m.pi)
```

Dùng bí danh cho module giúp tiết kiệm thời gian.

Khi dùng bí danh tên **module math** không được công nhận trong phạm vi lệnh nữa, mà phải dùng **m**.

Lệnh from...import

Có thể nhập tên cụ thể từ **module** mà không cần nhập toàn bộ **module**.

```
>>> # Nhập thuộc tính pi từ module math
>>> from math import pi
    print("Giá trị của pi là: ",pi)
```

Trong trường hợp này không cần sử dụng toán tử ".".

Có thể nhập nhiều thuộc tính của module

```
>>> from math import pi, e
>>> pi
    3.141592653589793
>>> e
    2.718281828459045
```

Nhập tất cả tên

Nhập tất cả tên (các hàm) từ module dùng *

```
>>> # Nhập tất cả từ module math
    form math import *
    print("Giá trị của pi là: ",pi)
```

Nhập tất cả các hàm từ **module math**, ngoại trừ những tên bắt đầu bằng dấu gạch dưới _.
Không nên nhập mọi thứ với dấu hoa thị * vì có thể dẫn đến sự trùng lặp của các định danh.

Đường dẫn tìm module

Thứ tự tìm:

- Thư mục hiện tại.
- PYTHONPATH (biến môi trường với danh sách thư mục).
- Thư mục mặc định có vị trí phụ thuộc vào chọn lựa trong quá trình cài đặt.

Lệnh xem đường dẫn:

```
>>> import sys
>>> sys.path
```

Có thể điều chỉnh danh sách này để thêm các path theo mong muốn.

Nạp lại module

import module chỉ thực hiện trong một phiên.

Tạo module test_module.py

```
>>> print('import module')
```

Nhập nhiều lần module test_module.

```
>>> import test_module
    import module
>>> import test_module
>>> import test_module
>>> import test_module
>>> import test_module
```

Có thể thấy lệnh trên chỉ thực hiện một lần.

Nạp lại module

Sửa đổi lệnh trong test_module.py

```
>>> print('kiểm tra import module')
```

Hàm reload() được dùng nạp lại module trong module imp.

```
>>> import imp
>>> imp.reload(test_module)
    kiểm tra import module
```

Hàm dir()

Hàm dir() được sử dụng để tìm các tên được định nghĩa trong module.
Nhập lần lượt các lệnh sau:

```
>>> import tinhtong
>>> dir(tinhtong)
```

Kết quả:

```
['Tong', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__spec__']
```

Danh sách các tên được định nghĩa trong module, sắp xếp theo thứ tự, các tên bắt đầu với dấu gạch dưới là thuộc tính mặc định liên kết với module.

Hàm dir()

Thuộc tính __name__ chứa tên module.

```
>>> import tinhtong
>>> tinhtong.__name__
    'tinhtong'
```

Có thể tìm tên bằng dir() không tham số.

```
>>> import tinhtong
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__',
'__warningregistry__', 'imp', 'test_module', 'tinhtong']
```

Package

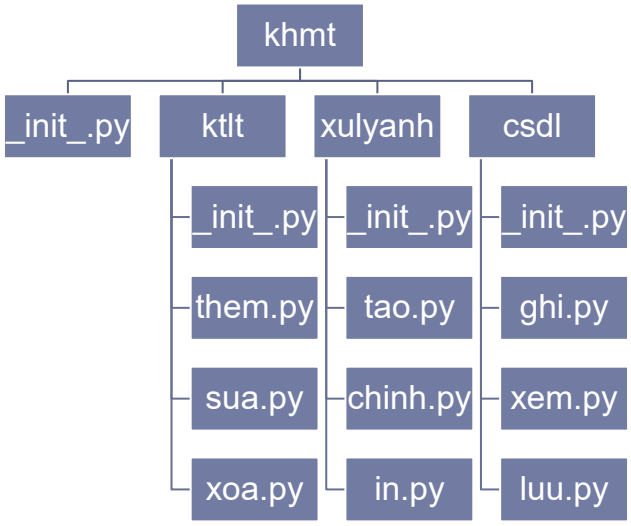
Khi chương trình có nhiều **module**, những **module** sẽ được đặt vào **package** giúp dễ dàng quản lý chương trình.

Package có thể có **package** con và các **module** khác.

Package là một thư mục có chứa file có tên **__init__.py**. File này có thể để trống, nhưng thông thường các lệnh khởi tạo cho **package** đặt ở đây.

Chương trình khmt với các **package** và **module**:

Package



Nhập module từ package

Nhập **module** từ **package** dùng toán tử dấu chấm (.).

```
>>> import khmt.csdl.ghi
```

Nếu trong module ghi.py chứa hàm có tên chon_ghi(), sẽ sử dụng tên đầy đủ để tham chiếu tới:

```
>>> import khmt.csdl.ghi.chon_ghi(ghi1)
```

Có thể nhập module mà không cần package:

```
>>> from khmt.csdl import ghi
```

gọi hàm như sau:

```
>>> ghi.chon_ghi(ghi1)
```

Nhập module từ package

Cách khác để nhập hàm được yêu cầu (lớp hoặc biến) từ **module** trong **package** như sau:

```
>>> from khmt.csdl.ghi import chon_ghi
```

Gọi hàm này:

```
>>> chon_ghi(ghi1)
```

Cách nhập này không được khuyến khích sử dụng.

Việc dùng tên đầy đủ sẽ giúp giảm tình trạng nhầm lẫn và tránh trùng lặp giữa những định danh giống nhau.

Trong khi nhập các package, Python sẽ tìm kiếm danh sách thư mục được định nghĩa trong sys.path, giống như đường dẫn tìm kiếm module.

Bài tập

1. Viết hàm tính giai thừa của một số cho trước. Ví dụ, cho trước số 8 kết quả là 40320 (**đệ qui**)

2. Viết method tính bình phương của một số.

3. Viết chương trình để in tài liệu về một số hàm Python được tích hợp sẵn như abs(), int(),input() và thêm tài liệu cho hàm bạn tự định nghĩa. (**(abs.__doc__), (int.__doc__), (input.__doc__), . . .**)

Bài tập

4. Định nghĩa 1 hàm tính tổng hai số
5. Định nghĩa một hàm chuyển số nguyên thành chuỗi và in ra màn hình. (**str()**)
6. Định nghĩa hàm nhận hai số nguyên dạng chuỗi và tính tổng của chúng, sau đó in ra tổng.
7. Định nghĩa hàm nhận 2 chuỗi, nối chúng lại và in ra màn hình
8. Định nghĩa hàm nhận 2 chuỗi và in ra chuỗi có độ dài lớn hơn. Nếu 2 chuỗi có chiều dài như nhau thì in các chuỗi theo dòng.

Bài tập

9. Định nghĩa hàm nhận số nguyên n và in "Đây là một số chẵn" nếu là chẵn và in "Đây là một số lẻ" nếu là số lẻ

10. Định nghĩa hàm in dictionary chứa key là các số từ 1 đến 3 và các giá trị bình phương của chúng.

11. Định nghĩa hàm in dictionary chứa các key là số từ 1 đến 20 và các giá trị bình phương của chúng.

(**dict[key]=value**)

12. Định nghĩa hàm tạo dictionary, chứa các key là số từ 1 đến 20 và các giá trị bình phương của chúng. Hàm chỉ in các giá trị mà thôi. (**k,v in dict.items()**)

Bài tập

13. Định nghĩa hàm tạo ra một dictionary chứa key là những số từ 1 đến 20 và các giá trị bình phương của key. Hàm chỉ in các key. (**dict.key()**)

14. Định nghĩa hàm tạo và in list chứa các giá trị bình phương của các số từ 1 đến 20.(**list.append()**)

15. Định nghĩa hàm tạo list chứa các giá trị bình phương của các số từ 1 đến 20 và in 5 mục đầu tiên trong list.

16. Định nghĩa hàm tạo list chứa các giá trị bình phương của các số từ 1 đến 20, in 5 mục cuối cùng trong list

Bài tập

17. Định nghĩa hàm tạo list chứa giá trị bình phương của các số từ 1 đến 20, in tất cả các giá trị của list, trừ 5 mục đầu tiên

18. Định nghĩa hàm tạo và in một tuple chứa các giá trị bình phương của các số từ 1 đến 20

19. Viết chương trình lọc các số chẵn trong danh sách sử dụng hàm **filter()**. Danh sách là [1,2,3,4,5,6,7,8,9,10] (**dùng lambda và list(filter())**)

20. Viết chương trình dùng **map()** để tạo list chứa các giá trị bình phương của các số trong [1,2,3,4,5,6,7,8,9,10] (**dùng lambda và list(map())**)

Bài tập

21. Viết chương trình dùng **map()** và **filter()** để tạo list chứa giá trị bình phương của các số chẵn trong [1,2,3,4,5,6,7,8,9,10]

22. Viết chương trình dùng **filter()** để tạo danh sách chứa các số chẵn trong đoạn [1,20]

23. Viết chương trình dùng **map()** để tạo list chứa giá trị bình phương của các số trong đoạn [1,20].

Bài tập

24. Viết chương trình tính: f(n)=f(n-1)+100 khi n>0 và f(0)=1, với n là số được nhập vào (n>0)

(**dùng đệ quy**) (nhập 5, kq = 500)

25. Dãy Fibonacci được tính dựa trên công thức sau:
f(n)=0 nếu n=0
f(n)=1 nếu n=1
f(n)=f(n-1)+f(n-2) nếu n>1
Hãy viết chương trình tính giá trị của f(n) với n là số được người dùng nhập vào. Ví dụ: Nếu n được nhập vào là 7 thì đầu ra của chương trình sẽ là 13.

Bài tập

26. Dãy Fibonacci được tính dựa trên công thức sau:
f(n)=0 nếu n=0
f(n)=1 nếu n=1
f(n)=f(n-1)+f(n-2) nếu n>1
Hãy viết chương trình sử dụng list comprehension để in dãy Fibonacci dưới dạng tách biệt bằng dấu ",", n được người dùng nhập vào

Bài tập

27. Viết hàm tìm kiếm nhị phân để tìm các item trong một list đã được sắp xếp. Hàm sẽ trả lại chỉ số của phần tử được tìm thấy trong list

28. Viết chương trình để giải câu đố:

Vừa gà, vừa chó, bó lại cho tròn

Được 36 con, 100 chân chẵn

Hỏi số gà và chó là bao nhiêu?
(22, 14)