6 Python中的函数与函数式编程（上）

> AI领域中的Python开发 --- by 丁宁

@(SIGAI课程录制)

- 上节课：Python中最常用的数据结构**List**
- 本节课：Python中的函数定义，函数参数，命名空间与作用域解析

认识函数

代码块

- 代码块是一组语句，类似于其他语言中的大括号
- Python中的代码块靠缩进建立，最好是4个空格，而非制表符
- Python中的代码块由`:`来引导，有缩进的提示终止
- 代码块里可以放条件语句，循环语句，函数定义，类定义，上下文管理器等

函数定义

- 函数是一个代码块
- 用`def`定义
- 返回一个值（任何情况下都是）
- Python中的调用符号：`()`
- Python中可调用判断函数：`callable`
- 函数中的`return`语句：可省略；可单独出现；可返回多个变量（终止与返回值）
- 放在函数定义开头的字符串称为**docstring**，作为函数的一部分（`__doc__`）
- 交互式解释器中的`help`函数

```
>>> def print_sigai():
...     print("sigai")
...
>>> print_sigai()
sigai
>>> help(print_sigai)
```

```
Help on function print_sigai in module __main__:

print_sigai()
(END)
```

```
>>> def print_sigai():
...     '''JUST A FUNCTION'''
...     print("sigai")
...
>>> print_sigai()
sigai
>>> help(print_sigai)
```

```
Help on function print_sigai in module __main__:

print_sigai()
    JUST A FUNCTION
(END)
```

```
>>> print_sigai.__doc__
'JUST A FUNCTION'
>>> print_sigai.__doc__ = 'just a function'
>>> print_sigai.__doc__
'just a function'
>>> help(print_sigai)
```

```
Help on function print_sigai in module __main__:

print_sigai()
    just a function
(END)
```

```
>>> def sigai_1():
...     pass
...
>>> def sigai_2():
...     return
...
>>> def sigai_3():
...     return None
...
>>> def sigai_4():
...     return 4
...
>>> def sigai_5():
...     return 1,2,3,4,"5"
...
>>> [eval('sigai_' + str(x) + '()') for x in range(1,6)]
[None, None, None, 4, (1, 2, 3, 4, '5')]
```

函数参数

形参与实参

- 定义函数的时候是**形参**，使用函数的时候传递的是**实参**
- 内部修改实参，不影响外部**同名变量**

```
>>> def sigai(a,b):
...     a += 1
...     b += 1
...     print(a,b)
...     return a + b
...
>>> a,b = 1,1
>>> sigai(a,b)
2 2
4
>>> print(a,b)
1 1
```

参数的传递

先来看个例子：

```
>>> def sigai(a,b):
...     a.append(1)
...     b.append(1)
...     print(a,b)
...     return a + b
...
>>> a,b = [1],[1]
>>> sigai(a,b)
[1, 1] [1, 1]
[1, 1, 1, 1]
>>> print(a,b)
[1, 1] [1, 1]
```

- 实参的传递本质上是赋值
- 既然是赋值，就要看赋的是可变对象还是不可变对象
- 把一个变量和它所在的地址想象成盒子里的东西和盒子外用于标识的标签
- 可变对象赋值，赋的是标签
- 不可变对象赋值，赋的是盒子本身
- 还记得哪些Python变量是可变的，哪些是不可变的吗?
- 如何传递一个可变对象的同时不希望被函数修改呢?

```
>>> a,b = [1],[1]
>>> sigai(a[:],b[:])
[1, 1] [1, 1]
[1, 1, 1, 1]
>>> print(a,b)
[1] [1]
```

> 嵌套的`List`记得使用`copy.deepcopy`哦~

位置参数与关键字参数

- 参数位置解耦
- 默认参数设置

先来看一个反面教材

```
>>> def say_hi(say, name):
...     print(say + ' ' + name + '!')
...
>>> def say_hi_2(name, say):
...     print(name + ' ' + say + '!')
...
>>> say_hi('hello','sigai')
hello sigai!
>>> say_hi_2('hello','sigai')
hello sigai!
```

> 想一下如果有20个位置参数，调用的时候怎么办?

参数名字和位置一起记，实在记不住呀，关键字参数了解一下

```
>>> say_hi(name='sigai',say='hello')
hello sigai!
```

关键字参数更重要的用途是设置默认值（大型程序尤其重要）

```
>>> def say_hi(name, say='hello'):
...     print(say + ' ' + name + '!')
...
>>> say_hi('sigai')
hello sigai!
```

还可以再进一步：

```
>>> def say_hi(name='sigai', say='hello'):
...     print(say + ' ' + name + '!')
...
>>> say_hi()
hello sigai!
```

功能完整的函数：主逻辑用位置参数，配置选项用关键字参数

看一个实际应用中的例子：

```
def __init__(self,
                featurewise_center=False,
                samplewise_center=False,
                featurewise_std_normalization=False,
                samplewise_std_normalization=False,
                zca_whitening=False,
                zca_epsilon=1e-6,
                rotation_range=0,
                width_shift_range=0.,
```

```
                    height_shift_range=0.,
                    brightness_range=None,
                    shear_range=0.,
                    zoom_range=0.,
                    channel_shift_range=0.,
                    fill_mode='nearest',
                    cval=0.,
                    horizontal_flip=False,
                    vertical_flip=False,
                    rescale=None,
                    preprocessing_function=None,
                    data_format='channels_last',
                    validation_split=0.0,
                    dtype='float32'):
```

> 选自`Keras`的预处理库，[链接点我]

两个建议：

- 最好不同时使用多个位置参数与多个关键字参数
- 如果使用了关键字参数，位置参数越少越好，并且集中放在最前面

任意数量的参数与Python中的星号

先来看一个函数，试着想想怎么实现这个函数

```
>>> sum(1,2)
3
>>> sum(1,2,3)
6
>>> sum(1,2,3,4)
10
>>> sum(1,2,3,4,5,6,7)
28
```

怎么接受任意个参数？星号了解一下

```
>>> def sum(*l):
...     result = 0
...     for x in l: result += x
...     return result
...
```

与赋值不同，函数的参数用星号拆包或缝合时，得到的是元组

```
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
>>> a,*b,c = list(range(7))
>>> type(b)
<class 'list'>

>>> def sum(*l):
...     print(type(l))
...
>>> sum(1,2)
<class 'tuple'>
```

还可以接受任意个关键字参数，两个星号了解一下

```
>>> def register(**kw):
...     print("kw's type is",type(kw))
...     print("kw is",kw)
...
>>> register(name='sigai',pi=3.1415926)
kw's type is <class 'dict'>
kw is {'name': 'sigai', 'pi': 3.1415926}
```

可以一起收集位置参数与关键字参数

```
>>> def register(*list,**dict):
...     print(list)
...     print(dict)
...
>>> register(1,2,3,name='sigai',pi=3.14)
(1, 2, 3)
{'name': 'sigai', 'pi': 3.14}
```

刚才的星号是用作封包的，其实也可以用来拆包

```
>>> def sum(a,b,c):
...     return a + b + c
...
>>> x = (1,2,3)
>>> sum(*x)
6
>>> sum(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() missing 2 required positional arguments: 'b' and 'c'
```

对于函数参数中的星号拆包和封包，用一个即可，用两个其实等于没用

```
>>> def sum_1(*l):
...     result = 0
...     for x in l: result += x
...     return result
...
>>> def sum_2(l):
...     result = 0
...     for x in l: result += x
...     return result
...
>>> sum_1(*range(5))
10
>>> sum_2(range(5))
10
```

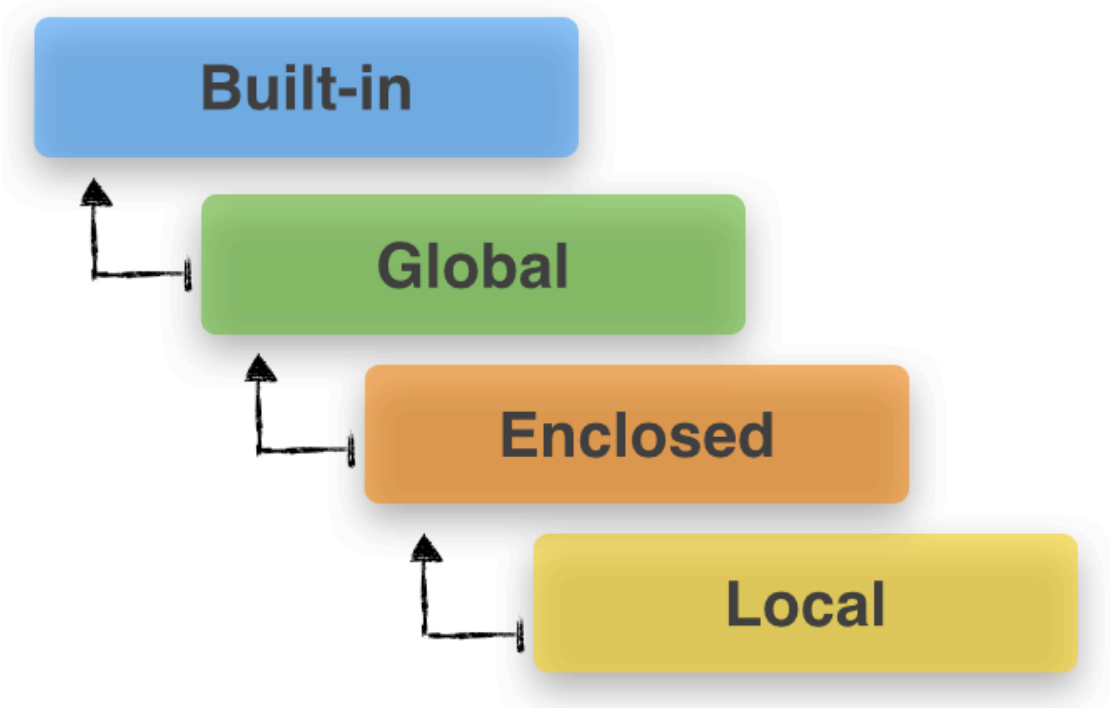
名称空间与作用域解析（Namespace and Scope Resolution）

What is Namespace?

- A namespace is a **mapping** from **names to objects**.
- **Most** namespaces are currently implemented as **Python dictionaries**.

What is Scope?

- Namespaces can **exist independently** from each other
- They are structured in a **certain hierarchy**
- The scope in Python **defines the hierarchy level**
- in which we **search namespaces** for certain “name-to-object” mappings
- Scope resolution for variable names via the **LEGB rule.**



作用域的产生

- **L**ocal：function and class method
- **E**nclosed：its enclosing function, if a function is wrapped inside another function
- **G**lobal：the uppermost level of executing script itself
- **B**uilt-in：special names that Python reserves for itself

作用域的查看（`globals()`与`locals()`）

```
dingnings-Mini:~ dingning$ python3
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 12:04:33)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('\n'.join(globals().keys()))
__name__
__doc__
__package__
__loader__
__spec__
__annotations__
__builtins__
>>> sigai_1 = 1
>>> print('\n'.join(globals().keys()))
__name__
__doc__
__package__
__loader__
__spec__
__annotations__
__builtins__
sigai_1
>>> sigai_2 = 2
>>> import math
>>> def say_hi(): pass
...
>>> class Human: pass
...
>>> print('\n'.join(globals().keys()))
__name__
__doc__
__package__
__loader__
__spec__
__annotations__
__builtins__
sigai_1
sigai_2
math
say_hi
```

```
Human
```

```
from copy import copy

globals_init = copy(list(globals().keys()))
print("globals_init: ", *globals_init, sep='\n')
print()

a, b = 1, 2

def f1(c=3):
    d = 4

    print("globals in f1:", *(globals().keys() - globals_init))
    print("locals in f1:", *(locals().keys()))

    def f2(e=5):
        f = None

        print("globals in f2:", *(globals().keys() - globals_init))
        print("locals in f2:", *(locals().keys()))

    f2()

if __name__ == '__main__':

    f1()
```

```
dingnings-Mini:Desktop dingning$ python3 test.py
Globals_init:
__name__
__doc__
__package__
__loader__
__spec__
__annotations__
__builtins__
__file__
__cached__
copy

globals in f1: globals_init f1 a b
locals in f1: d c
globals in f2: globals_init f1 a b
locals in f2: f e
```

作用域间变量的遮盖

```
name = 'sigai_1'
print(name)

def f1():
    name = 'sigai_2'
    print(name)

    def f2():
        name = 'sigai_3'
        print(name)

    f2()

if __name__ == '__main__':

    f1()
```

```
dingnings-Mini:Desktop dingning$ python3 test.py
sigai_1
sigai_2
sigai_3
```

修改不同作用域里的变量

```
def f1(name):
    name = 'sigai_2'
    print(name)


if __name__ == '__main__':

    name = 'sigai_1'
    print(name)
    f1(name)
    print(name)
```

```
dingnings-Mini:Desktop dingning$ python3 test.py
sigai_1
sigai_2
sigai_1
```

```
def f1():
    global name
    name = 'sigai_2'
    print(name)


if __name__ == '__main__':

    name = 'sigai_1'
    print(name)
    f1()
    print(name)
```

```
dingnings-Mini:Desktop dingning$ python3 test.py
sigai_1
sigai_2
sigai_2
```

```
def f1():
    name = 'sigai_2'
    print(name)

    def f2():
        name = 'sigai_3'
        print(name)

    f2()

    print(name)


if __name__ == '__main__':

    name = 'sigai_1'
    print(name)
    f1()
    print(name)
```

```
dingnings-Mini:Desktop dingning$ python3 test.py
sigai_1
sigai_2
sigai_3
sigai_2
sigai_1
```

```
def f1():
```

```
    name = 'sigai_2'
    print(name)

    def f2():
        nonlocal name
        name = 'sigai_3'
        print(name)

    f2()

    print(name)


if __name__ == '__main__':

    name = 'sigai_1'
    print(name)
    f1()
    print(name)
```

```
dingnings-Mini:Desktop dingning$ python3 test.py
sigai_1
sigai_2
sigai_3
sigai_3
sigai_1
```

作用域的生命周期

- built-in：解释器在则在，解释器亡则亡
- global：导入模块时创建，直到解释器退出
- local：函数调用时才创建

> 下节课讲解函数闭包，还会涉及这部分内容，这里就不展开讲了
