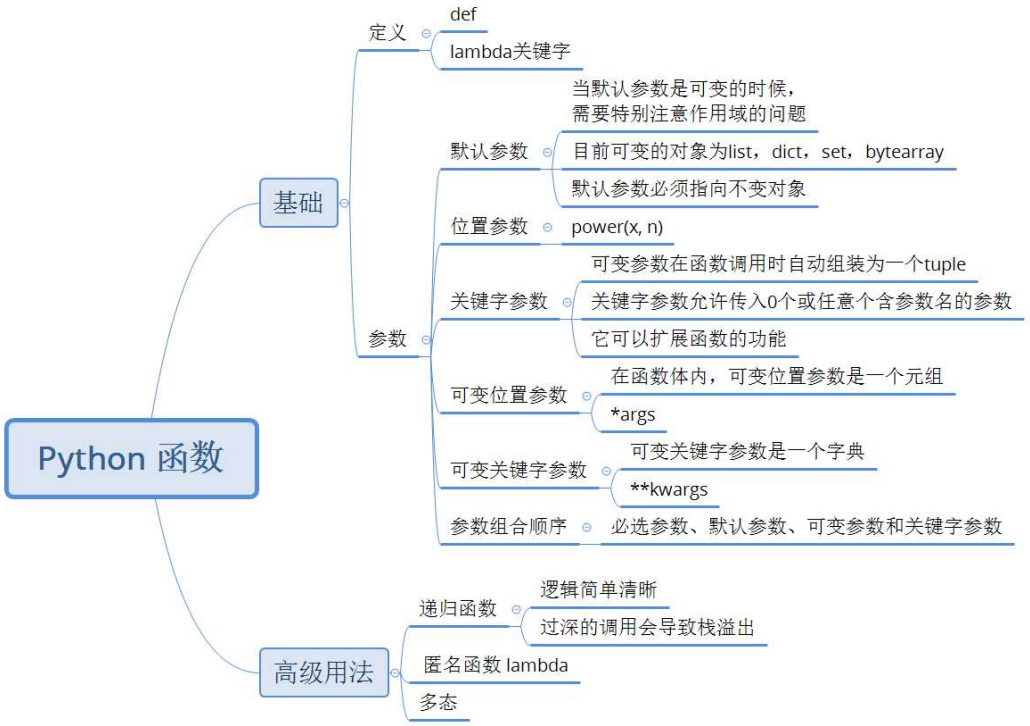
Python 函数基础知识



一、函数基础

简单地说，一个函数就是一组Python语句的组合，它们可以在程序中运行一次或多次运行。Python中的函数在其他语言中也叫做过程或子例程，那么这些被包装起来的语句通过一个函数名称来调用。

有了函数，我们可以在很大程度上减少复制及粘贴代码的次数了（相信很多人在刚开始时都有这样的体验）。我们可以把相同的代码可以提炼出来做成一个函数，在需要的地方只需要调用即可。那么，这样就提高了代码的复用率了，整体代码看起来比较简练，没有那么臃肿了。

函数在Python中是最基本的程序结构，用来最大化地让我们的代码进行复用；与此同时，函数可以把一个错综复杂的系统分割为可管理的多个部分，简化编程、代码复用。

接下来我们看看什么是函数，及函数该如何定义。有两种方式可以进行函数的定义，分别是def及lambda关键字。

1. 函数定义

先总结一下为什么要使用函数?

1. 代码复用最大化及最小化冗余代码;
2. 过程分解（拆解）。把一个复杂的任务拆解为多个小任务。

函数定义的语法为:

```
def func_name(arg1, arg2, arg3, ..., argN):
    statement
    return value
```

根据上面定义，可以简单地描述为：Python中的函数是具有0个或多个参数，具有若干行语句并且具有返回值（返回值可有可无）的一个语句块（注意缩进）。

那么我们就定义一个比较简单的函数，该函数没有参数，进入ipython交互式环境:

```
In[1]: def hello():
  ...:     print('Leave me alone, the world')
  ...:
```

调用（执行）该函数:

```
In[2]: hello()
Leave me alone, the world
```

我们发现hello()函数并没有return语句，在Python中，如果没有显式的执行return语句，那么函数的返回值默认为None。

我们说过，定义函数有两种形式，另外一种形式是使用lambda来定义。使用lambda定义的函数是匿名函数，这个我们在后面的内容进行讲解，这里暂且不表。

二、函数参数

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂的逻辑被封装起来，调用者无需了解。

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

1. 默认参数

默认参数使得API简洁，但不失灵活性。当一个参数有默认值时，调用时如果不传递此参数时，会使用默认值。

```
def inc(init, step=1):
    return init + step
# 调用一下这个函数
>>> inc(3)
4
>>> inc(3, 2)
5
```

默认参数有一个坑，就是非默认参数要放到默认参数的前面（不然Python的解释器会报语法错误）。允许有多个默认参数，但默认参数需要放在参数列表的最后面。

```
def append(x, lst=[]):
    return lst.append(x)
```

此函数有问题。（函数中的形参是全局变量？lst在append函数中叫lst，但在全局作用域中，我们不知道lst具体叫什么名字。）

修改之后的函数为：

```
def append(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst
```

通常来说，当默认参数是可变的时候，需要特别注意作用域的问题，我们需要上述的技巧（不可变的数据类型是值传递，可变的数据类型是引用传递。）。目前可变的对象为list，dict，set，bytearray。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

```
# 先定义一个函数，传入一个list，添加一个END再返回
def add_end(L=[]):
    L.append('END')
    return L
```

当我们正常调用时，结果似乎不错，

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当我们使用默认参数调用时，一开始结果也是对的，

```
>>> add_end()
['END']
```

但是，再次调用add_end()时，结果就不对了，

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

原因解释如下：

Python函数在定义的时候，默认参数L的值就被计算出来了，即[]，因为默认参数L也是一个变量，它指向对象[]，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了。

所以，定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用None这个不变对象来实现，

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

为什么要设计str、None这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

2. 位置参数

我们先写一个计算x^2的函数：

```
def power(x):
    return x * x
```

对于power(x)函数，参数x就是一个位置参数。当我们调用power函数时，必须传入有且仅有的一个参数x：

```
>>> power(5)
25
>>> power(15)
225
```

现在，如果我们要计算x^3怎么办呢？可以再定义一个power3函数，但是如果要计算x^4、x^5、x^n，怎么办？我们不可能定义无限多个函数，我们可以把power(x)修改为power(x, n)，用来计算x^n，说写就写：

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

3. 关键字参数

可变参数允许我们传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。示例如下：

```
def person(name, age, **kwargs):
    print('name:', name, 'age:', age, 'other:', kwargs)
```

函数person除了必选参数name和age外，还接受关键字参数kwargs。在调用该函数时，可以只传入必选参数：

```
>>> person('LavenLiu', 25)
name: LavenLiu age: 25 other: {}
```

也可以传入任意个数的关键字参数：

```
>>> person('LavenLiu', 25)
name: LavenLiu age: 25 other: {}

>>> person('Taoqi', 25, city='Hebei')
name: Taoqi age: 25 other: {'city': 'Hebei'}

>>> person('James', 31, gender='M', job='NBA player')
name: James age: 31 other: {'gender': 'M', 'job': 'NBA player'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

和可变参数类似，也可以先组装出一个dict，然后，把该dict转换为关键字参数传进去：

```
>>> kwargs = {'city': 'Hebei', 'job': 'Test'}

>>> person('Taoqi', 25, **kwargs)
name: Taoqi age: 25 other: {'city': 'Hebei', 'job': 'Test'}
```

4. 位置参数和关键字参数

位置参数和关键字参数是函数调用时的概念。

当默认参数和关键字参数结合起来用的时候，很有用。

关键字参数必须写在位置参数之后，否则会抛出语法错误。

```
def minus(x, y):
    return x - y

minus(3, 5) # 位置参数，位置传参
minus(5, 3) # 位置参数，位置传参
minus(x=5, y=3) # 关键字参数，关键字传参
minus(y=3, x=5) # 关键字参数，关键字传参
```

位置参数和关键字参数可以共存，但是关键字参数必须写到位置参数之后。

5. 可变位置参数

可变位置参数用*定义，在函数体内，可变位置参数是一个元组。

```
In[1]: def fn(*args):
   ...:     print(args)
   ...:

In[2]: fn((1, 2, 3, 4))
((1, 2, 3, 4),)

In[3]: tup01 = (1, 2, 3, 4)

In[4]: fn(tup01)
((1, 2, 3, 4),)

In[5]: fn(*tup01)
(1, 2, 3, 4)
```

在python的函数中，还可以定义可变参数。可变参数就是传入的参数个数是可变的。

```
In[6]: def cacl(*numbers):
   ...:     sum = 0
   ...:     for n in numbers:
   ...:         sum = sum + n * n
   ...:     return sum
   ...:

In[7]: nums = [1, 2, 3]

In[8]: cacl(*nums) # 这里如果不在nums前面加*，有问题吗?
Out[8]: 14
```

6. 可变关键字参数

可变关键字参数使用**定义，在函数体内，可变关键字参数是一个字典。可变关键字参数的key都是字符串，并且符合标识符定义规范。

```
def fn(**kwargs):
    print(kwargs)

dict01 = {'name': 'Laven Liu', 'age': 29}
fn(**dict01)
# fn(dict01)
fn(name='Laven Liu', age=29)

{'name': 'Laven Liu', 'age': 29}
{'name': 'Laven Liu', 'age': 29}
```

- 可变位置参数只能以位置参数的形式调用
- 可变关键字参数只能以关键字参数的形式调用
- 可变位置参数必须在可变关键字参数之前
- 可变参数后置
- 可变参数不和默认参数一起出现

7. 参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数和关键字参数，这4种参数都可以一起使用，或者只用其中某些，但是请注意，参数定义的顺序必须是： 必选参数、默认参数、可变参数和关键字参数

比如定义一个函数，包含上述4种参数：

```
>>> def func(a, b, c=0, *args, **kwargs):
...     print('a =', a, 'b =', b, 'c =', c, 'args = ', args, 'kwargs = ', kwargs)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> func(1, 2)
a = 1 b = 2 c = 0 args =  () kwargs =  {}
>>> func(1, 2, c=3)
a = 1 b = 2 c = 3 args =  () kwargs =  {}
>>> func(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args =  ('a', 'b') kwargs =  {}
>>> func(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args =  ('a', 'b') kwargs =  {'x': 99}
>>>
```

最神奇的是通过一个tuple和dict，我们也可以调用该函数：

```
>>> args = (1, 2, 3, 4)
>>> kwargs = {'x': 99}
>>> func(*args, **kwargs)
a = 1 b = 2 c = 3 args =  (4,) kwargs =  {'x': 99}
```

所以，对于任意函数，都可以通过类似func(*args, **kwargs)的形式调用它，无论它的参数是如何定义的。

8. 参数解构

参数解构发生在函数调用时，可变参数发生函数定义的时候。参数解构分为两种形式，一种是位置参数解构，另一种是关键字参数解构。

参数结构的两种形式：

- 位置参数解构，使用一个星号。解构的对象为可迭代对象，解构的结果为位置参数。
- 关键字参数解构，使用两个星号。解构的对象为字典，解构的结果为关键字参数。

位置参数解构的一个例子：

```
In[23]: def fn(a, b, c):
    ...:     print(a, b, c)
    ...:

In[24]: lst = [1, 2, 3]

In[25]: fn(lst[0], lst[1], lst[2])
1 2 3
# 也可以进行如下形式的调用
In[26]: fn(*lst)  # 这种做法就叫参数解构
1 2 3
# *号可以把线性结构解包成位置参数
lst = [1, 2, 3, 4]
fn(*lst) # -> fn(lst[0], lst[1], lst[2], lst[3])
TypeError: fn() takes 3 positional arguments but 4 were given
# 这里就报错了，本来这个函数只能接收3个位置参数，lst有四个元素，通过参数解构之后，就变成了4个参数，所以就报错了。
```

接下来看字典解构的例子：

```
In[27]: d = {'a': 1, 'b': 2, 'c': 3}

In[28]: fn(**d)
1 2 3
# **可以把字典解构成关键字参数
```

参数解构发生在函数调用时。解构的时候，线性结构的解构是位置参数，字典解构是关键字参数。

传参的顺序：位置参数，线性结构解构；关键字参数，字典解构。尽量的少的同时使用两种解构，除非你真的知道在做什么。

```
In[29]: def fn(a, b, c, d):
    ...:     print(a, b, c, d)
    ...:

In[30]: fn(0, *[2], c=1, **{'d': 3})
0 2 1 3
```

9. 参数槽(keyword-only参数)

Python3中引入的。

```
def fn(a, b, c):
    print(a, b, c)

fn(a=1, b=2, c=3)
```

如果要强制传入的参数为关键字参数：

```
def fn(*, a, b, c):
    print(a, b, c)

>>> fn(1, 2, 3)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    fn(1, 2, 3)
TypeError: fn() takes 0 positional arguments but 3 were given

>>> fn(a=1, b=2, c=3)
1 2 3
# *之后的参数，必须以关键字参数的形式传递，称之为参数槽。
```

参数槽通常和默认参数搭配使用。

```
>>> def fn(a, b, *, x, y):
    print(a, b)
    print(x, y)
>>> fn(1, 2, 3, 4)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    fn(1, 2, 3, 4)
TypeError: fn() takes 2 positional arguments but 4 were given
>>> fn(1, 2, x=3, y=4)
1 2
3 4
>>> fn(1, 2, **{'x': 3, 'y': 4})
1 2
3 4

def fn(a, b, *):
    print(a, b)

def fn(a, b, *):
...     print(a, b)
File "<stdin>", line 1
SyntaxError: named arguments must follow bare *
```

几个例子：

```
def fn01(*, x=1, y=5):
    print(x)
    print(y)
>>> fn01()
1
5

def fn02(x=1, *, y):
    print(x)
    print(y)
>>> fn02(y=3)
1
3
```

参数槽之坑：

1. *之后必须有参数
2. 非命名参数有默认值时，命名参数可以没有默认值
3. 默认参数应该在每段参数的最后
4. 使用参数槽时，不能使用可变位置参数，可变关键之参数必须放在命名参数之后

三、高级用法

1. 递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

```
deffact(n):
    if n==1:
        return1
    return n * fact(n - 1)
```

递归函数

使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。

针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。

2. 匿名函数 lambda

python 使用 lambda 来创建匿名函数。

lambda只是一个表达式，函数体比def简单很多。

lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。

lambda函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。

虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

```
fib=lambda n,x=0,y=1:x if not n elsefib(n-1,y,x+y)

print (fib(20))
```

实例展示

3. Python函数中的多态

一个操作的意义取决于被操作对象的类型：

```
def times(x,y):
  return x*y

>>>times(2,4)
>>>8


times('Python',4) # 传递了与上不同的数据类型
'PythonPythonPythonPython'
```

四、总结

Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

默认参数一定要用不可变对象，如果是可变对象，运行会有逻辑错误！

要注意定义可变参数和关键字参数的语法：

*args是可变参数，args接收的是一个tuple；

**kwargs是关键字参数，kwargs接收的是一个dict。

以及调用函数时如何传入可变参数和关键字参数的语法：

可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过*args传入：func(*(1, 2, 3))；

关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过kwargs传入：func({'a': 1, 'b': 2})。

使用*args和**kwargs是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。