



# Python基础教程

## 第4章 Python函数



# 1.1 什么是函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数是对程序逻辑进行结构化或者过程化的一种编程方法：

（1）将代码隔离成易于管理的小块；

（2）把重复代码放到函数可以节省空间，又有助于保持一致性。只需要改变函数内部代码而无需去寻找再修改大量复制代码的拷贝。



# 1.1 什么是函数

函数能提高应用的模块性，和代码的重复利用率。

Python提供了许多内建函数，比如open()。但也可以自己创建函数，这被叫做用户自定义函数。

Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod() ✓	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval() ✓	int()	open()	str()
bool()	exec() ✓	isinstance() ✓	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes() ✓	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range() ✓	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	



## 1.2 函数返回值

在C语言中默认为“void”的返回类型，意思是没有值返回。

在Python中，对应的返回对象类型是None，解释器会隐式地返回默认值None。

```
>>> def foo():  
        print('test')
```

```
>>> ret = foo()
```

```
test
```

```
>>> ret
```

```
>>> print(ret)
```

```
None
```

```
>>> ret == None
```

```
True
```

Python中使用**def**语句来创建函数，语法如下：

**def** *function\_name* ( *arguments* ):

*“documentation string”* (文档字符串)

*Function body*

标题行由**def**关键字，函数名字，以及参数的集合组成。

**def**子句的剩余部分包括了一个虽然可选，但是强烈推荐的文档字符串（可用**help**函数查看），和必须的函数体。

在某些编程语言里，函数声明和函数定义是区分开的。

一个函数声明包括函数名，参数的名字（可能还有参数类型），但不必给出函数的任何代码，具体的代码通常属于函数定义的范畴。

在声明和定义有区别的语言中，往往是因为函数的定义可能和其声明在不同的文件中。

Python将这两者视为一体：函数子句由声明的标题行以及随后定义的函数体组成。

- 函数名必须以下划线或字母开头，可以包含任意字母、数字或下划线的组合；
- 不能使用任何的标点符号；
- 函数名是区分大小写的。
- 函数名不能是保留字。

Python和其他许多高级语言一样，不允许在函数未声明之前，对其进行引用或者调用。

可以在函数体内创建另外一个函数，这种函数叫做内部/内嵌函数。



## 1.3.4 创建函数-前向引用

Python不允许在函数未声明之前，对其进行引用或者调用：

```
>>> def foo():  
    print('in foo')  
    bar()
```

```
>>> foo()  
in foo  
Traceback (most recent call  
  File "<pyshell#15>", line  
    foo()  
  File "<pyshell#14>", line  
    bar()  
NameError: name 'bar' is not
```

```
>>> def bar():  
    print('in bar()')
```

```
>>> def foo():  
    print('in foo()')  
    bar()
```

```
>>> foo()  
in foo()  
in bar()
```





## 1.3.4 创建函数-前向引用 (续)

```
>>> def foo():  
    print('infoo')  
    bar()
```

```
>>> def bar():  
    print('in bar')
```

```
>>> foo()  
infoo  
in bar  
\\
```

为什么是正确的？

分析：

foo()本身不是在bar()声明前被调用，即调用foo()时，bar()已经存在。

名字错误是当访问没有初始化的标识符时才产生的异常。

- (1) 同大多数语言一样，使用一对圆括号（）调用函数。
- (2) 任何输入的参数都必须放置在括号内，如：`foo( x ,y )`。
- (3) 作为函数声明的一部分，括号也会用来定义那些参数。
- (4) 在Python中，函数的操作符同样用于类的实例化。



Python中的函数调用是让调用者通过名字来区分参数，这样的规范允许参数缺失或者不按顺序，因为解释器能通过给出的名字来匹配参数的值。

```
>>> def net_conn( host , port ) :  
    print( host , port )
```

```
>>> net_conn( '192.168.1.1' , 80 )  
192.168.1.1 80
```

```
>>> net_conn( port = 80 , host = '192.168.1.1' )  
192.168.1.1 80
```

问题：该函数调用时  
port参数能在前面吗？





**定义：**默认参数就是声明了默认值的参数，在调用时，可以不向该参数传入值。如果没有为参数提供值，则使用预先定义的默认值，这些定义在函数声明时给出。

**语法：**所有位置的非默认参数必须出现在任何一个默认参数之前。

**优点：**使用默认参数让程序的健壮性上升到极高的级别，因为它们补充了标准位置参数没有提供的一些灵活性。





## 1.5.2 默认参数 (2)

```
>>> def net_conn(host, port = 80):  
    print(host, port)
```

```
>>> net_conn('192.168.1.23')  
192.168.1.23 80
```

```
>>> def net_conn(host, port = 80, app):  
    print(host, port)
```

SyntaxError: non-default argument follows default argument

(1) 如何使用默认参数

(2) 默认参数出现在了其他参数之前是错误的



有时会遇到需要用函数处理可变数量参数的情况，这时可以使用可变长度的参数列表。

变长的参数在函数声明中不是显式命名的，因为参数的数目在运行之前是未知的。

常规参数都是在函数声明中显示命名的。

由于函数调用提供了关键字以及非关键字两种参数类型，Python用两种方法来支持变长参数。

当函数被调用时，所有的形参（必须的和默认的）都赋值给相对应的局部变量。剩下的非关键字参数按顺序插入到一个元组中便于访问。

可变长度的参数元组必须在确定位置和默认参数之后，带元组的函数普遍的语法如下：

```
def function_name ( [ formal_args , ] *vargs_tuples ):
```

*“documentation string”*

*Function body*



## 1.6.1 可变长参数 (元组)

```
def function_name ( [ formal_args , ] *vargs_tuples ):
```

*“documentation string”*

*Function body*

星号 (\*) 操作符之后的形参将作为元组传递给函数，元组保存了所有传递给函数的“额外的参数”（匹配了所有位置和显示参数后剩余的）。

如果没有给出额外的参数，元组为空。







## 1.6.2 可变长参数（元组）实例

```
>>> def foo( x , y = 'default', *z ):
    print('args 1', x)
    print('args 2', y)
    print('args 3', z)
    for i in z:
        print( 'variable args:', i )
```

```
>>> foo( ' first ', ' second ', '3' , '4' , '5' )
args 1 first
args 2 second
args 3 ('3', '4', '5')
variable args: 3
variable args: 4
variable args: 5
```



## 1.6.3 可变长命名参数变量（字典）

当有不定数量的或者额外集合的关键字的情况下，参数被放入一个字典中，字典中**键**为参数名，**值**为相应的参数值。

区分关键字参数和非关键字非正式参数，使用双星号（\*\*）。

\*\*是被重载了的，以便不与幂运算发生混淆。

关键字变量参数应该为函数定义的最后一个参数，带\*\*。

```
def function_name( [fomal_args, ] [*args, ] **kwargs):
```

*“documentation string”*

*Function body*



# 可变长命名参数变量实例

```
>>> def DictVarArgs( args1, args2, **args3 ):
    print( 'args1:', args1 )
    print( 'args2:', args2 )
    print( 'args3:', args3 )
    for i in args3.keys():
        print( 'args3: %s = %s' % ( i, args3[i]))
```

```
>>> DictVarArgs( 'first', 'second', C = 3, D = 4, E = 5 )
args1: first
args2: second
args3: {'C': 3, 'D': 4, 'E': 5}
args3: C = 3
args3: D = 4
args3: E = 5
```



- 关键字和非关键字可变长参数在同一个函数中的情形：

关键字字典是最后一个参数并且非关键字元组先于它出现之前。

```
def newfoo( arg1, arg2, *nkw, **kw):
```

*“documentation string” 或 “display regular args and all variable args”*

*“Function body:”*

```
    print('arg1 is:', arg1)
```

```
    print ('arg2 is:', arg2)
```

```
    for eachNKW in nkw:
```

```
        print ('additional non-keyword arg:', eachNKW)
```

```
    for eachKW in kw.keys():
```

```
        print (“additional keyword arg ‘s%’: % s” % \ (eachKW, kw[eachKW]))
```



```
>>> def DictVarArgs( args1, args2, *args3, **args4 ):
    print( 'args1:', args1 )
    print( 'args2:', args2 )
    print( 'args3:', args3 )
    print( 'args4:', args4 )
    for i in args3:
        print( 'Non-keyword arg: ', i)
    for j in args4.keys():
        print( "Keyword arg: ' %s ' : ' %s' " % ( j, args4[ j ]))
```

```
>>> DictVarArgs( 'Year', 2020, ' Months', one = 10, two = 11, three = 12 )
args1: Year
args2: 2020
args3: ( ' Months',)
args4: {'one': 10, 'two': 11, 'three': 12}
Non-keyword arg:  Months
Keyword arg: ' one ' : 10'
Keyword arg: ' two ' : 11'
Keyword arg: ' three ' : 12'
```





## 1.6.5 可变长参数组合

```
>>> DictVarArgs( 2, 4, *( 6, 8 ), **{ 'foo': 10, 'bar':12 } )
```

```
args1: 2
```

```
args2: 4
```

```
args3: (6, 8)
```

```
args4: {'foo': 10, 'bar': 12}
```

```
Non-keyword arg: 6
```

```
Non-keyword arg: 8
```

```
Keyword arg: 'foo ':' 10 '
```

```
Keyword arg: 'bar ':' 12 '
```

```
>>> DictVarArgs( 10, 20, 30, 40, foo = 50, bar = 60 )
```

```
args1: 10
```

```
args2: 20
```

```
args3: (30, 40)
```

```
args4: {'foo': 50, 'bar': 60}
```

```
Non-keyword arg: 30
```

```
Non-keyword arg: 40
```

```
Keyword arg: 'foo ':' 50 '
```

```
Keyword arg: 'bar ':' 60 '
```





```
>>> aTuple = ( 6, 7, 8 )
>>> aDict = { 'z': 9 }
>>> DictVarArgs( 1, 2, 3, x = 4, y = 5, *aTuple, **aDict )
args1: 1
args2: 2
args3: (3, 6, 7, 8)
args4: {'x': 4, 'y': 5, 'z': 9}
Non-keyword arg: 3
Non-keyword arg: 6
Non-keyword arg: 7
Non-keyword arg: 8
Keyword arg: ' x ':' 4 '
Keyword arg: ' y ':' 5 '
Keyword arg: ' z ':' 9 '
```



## 1.7 函数式编程-匿名函数 (1)

Python允许使用lambda关键字创造匿名函数（也叫lambda函数，没有函数名）。

匿名是因为不需要以标准的方式来声明（比如使用def语句）。

（1）lambda函数可以接收任意多个参数（包括可选参数）并且返回单个表达式的值。

（2）lambda 函数不能包含命令，包含的表达式不能超过一个。



## 1.7 函数式编程-匿名函数 (1)

(3) lambda语句中，冒号前是参数，可以有多个，用逗号隔开，冒号右边的为表达式。

(4) lambda返回值是一个函数的地址，也就是函数对象。

```
>>> F = lambda x , y , z : x * y * z
>>> type(F)
<type 'function'>
>>> F(2,3,4)
24
```

### 匿名函数特点:

- 能接收任何数量的参数但只能返回一个表达式的值，同时不能包含命令或多个表达式。
- 不能直接调用`print`，因为需要一个表达式，这个表达式本质还是一个函数，被调用时会创建一个框架对象。
- 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。
- 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。

## 函数式编程的内建函数

<code>filter(func,seq)</code>	调用一个布尔函数func来迭代遍历每个seq中的元素;
过滤函数	返回一个使func返回值为True的元素序列

```
def filter( bool_func, seq )
```

```
    filtered_seq = [ ]
```

```
    for eachItem in seq:
```

```
        if bool_func( eachItem ):
```

```
            filtered_seq.append( eachItem )
```

```
    return filtered_seq
```

## 1.7 函数式编程-内建函数filter()

### 函数式编程的内建函数

<code>filter(func, seq)</code>	调用一个布尔函数func来迭代遍历每个seq中的元素；
过滤函数	返回一个使func返回值为True的元素序列

```
>>> def func1( s ):
        if s != 'a':
            return s
        else:
            return None

>>> string = [ 'a' , 'b' , 'c' , 'd' ]
>>> ret = filter( func1, string )
>>> print( ret )
<filter object at 0x0000021C839F7EE0>
>>> ret
<filter object at 0x0000021C839F7EE0>
>>> list( ret )
['b', 'c', 'd']
```

## 函数式编程的内建函数

`map(func, seq1[, seq2...])`

对seq中的item依次执行func(item)，将执行结果组成一个List返回

```
def map( func, seq )
```

```
    mapped_seq = [ ]
```

```
    for eachItem in seq:
```

```
        mapped_seq.append( func( eachItem ) )
```

```
    return mapped_seq
```



```
>>> str = [ 'a', 'b', 'c', 'd' ]
```

```
>>> def func2( s ):
    return s + '.txt'
```

```
>>> ret = map( func2, str )
```

```
>>> print( ret )
```

```
<map object at 0x000001C10A4A53A0>
```

```
>>> list( ret )
```

```
['a.txt', 'b.txt', 'c.txt', 'd.txt']
```

```
>>> def add( x, y ):
    return x + y
```

```
>>> ret = map( add, range( 10 ), range( 10 ) )
```

```
>>> print( ret )
```

```
<map object at 0x000001C10A53FF40>
```

```
>>> list( ret )
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```



## 函数式编程的内建函数

<code>reduce(func,seq[, init])</code>	对seq的元素顺序迭代调用func，如果有init，还可以作为初始值调用
---	--------------------------------------

```
def reduce( bin_func, seq, init = None )
```

```
    lseq = list( seq )
```

```
    if init is None:
```

```
        res = lseq.pop( 0 )
```

```
    else:
```

```
        res = init
```

```
    for item in lseq:
```

```
        res = bin_func( res, item )
```

```
    return res
```



### 函数式编程的内建函数

<code>reduce(func, seq[, init])</code>	对seq的元素顺序迭代调用func，如果有init，还可以作为初始值调用
--	--------------------------------------

```
>>> import functools
```

```
>>> def add1( x, y ):
    return x + y
```

```
>>> functools.reduce( add1, range( 1, 100 ))
4950
```

```
>>> functools.reduce( add1, range( 1, 100 ), 20)
4970
```

- ## 4950 （注：  $1+2+\dots+99$  ）  
## 4970 （注：  $1+2+\dots+99+20$  ）





- 定义函数时，需要确定函数名和参数个数；
- 如果有必要，可以先对参数的数据类型做检查；
- 函数体内部可以用`return`随时返回函数结果；
- 函数执行完毕没有`return`语句时，自动`return None`。
- 函数可以同时返回多个值，但其实就是一个元组。

## 2.1 文档字符串 (1)

Python有一个特性称为文档字符串（documentation strings），  
简称为 docstrings。

docStrings是一个重要的工具，帮助程序文档更加简单易懂，甚至可以在程序运行的时候，从函数恢复文档字符串。

在函数的第一个逻辑行的字符串是这个函数的“文档字符串”。惯例是一个多行字符串，首行以大写字母开始，句号结尾。第二行是空行，从第三行开始是详细的描述。建议在函数中使用文档字符串时遵循这个惯例。

## 2.2 文档字符串 (2)

可以使用\_\_doc\_\_  
(注意双下划线) 调用函数的文档字符串属性(属于函数的名称)。

Python中help()函数，只是抓取函数的\_\_doc\_\_属性，然后整洁地展示。自动化工具也能以同样的方式从程序中提取文档。

```
>>> def Add( x, y ):
    "Add x, y and return
    x, y must be both int/float/string"
    return x + y
```

```
>>> print( Add( 10, 20 ) )
30
```

```
>>> print( Add.__doc__, '\n' )
Add x, y and return
    x, y must be both int/float/string
```

```
>>> help( Add )
Help on function Add in module __main__:

Add(x, y)
    Add x, y and return
    x, y must be both int/float/string
```



# 思考

1. 可变长度参数引入目的?
2. 文档字符串的作用?

