5. 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。

函数能提高应用的模块性，和代码的重复利用率。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

5.1. 定义函数

定义一个由自己想要功能的函数，以下是简单的规则：

- 函数代码块以 **def** 关键词开头，后接**函数标识符名称**和圆括号 **()**接上一个**:**。
- 任何传入参数和自变量必须放在圆括号中间，圆括号之间可以用于定义参数。
- 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。
- 函数内容以冒号 **:** 起始，并且缩进。
- **return [表达式]** 结束函数，选择性地返回一个值给调用方，不带表达式的 return 相当于返回 None。

5.2. 语法

Python 定义函数使用 def 关键字，默认情况下，参数值和参数名称是按函数声明中定义的**顺序匹配**起来的。

```
def 函数名(参数列表):
    函数体
```

示例

```
def hello():
    print("hello world")
hello()
```

带参

```
def max(a, b):
    if a > b:
        return a
    else:
        return b

a = 4
b = 5
print(max(a, b))
```

计算面积

```
def area(width, height):
    return width * height
w = 4
h = 5
print("width =", w, " height =", h, " area =", area(w, h))
```

5.3. 函数调用

定义一个函数：给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从 Python 命令提示符执行。

```
# 定义函数
def printme( str ):
   # 打印任何传入的字符串
   print (str)
   return

# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

5.4. 参数传递

在 python 中，类型属于对象，对象有不同类型的区分，变量是没有类型的：

```
a=[1,2,3]

a="Python"
```

以上代码中，**[1,2,3]** 是 List 类型，**"Python"** 是 String 类型，而变量 a 是没有类型，它仅仅是一个对象的引用（一个指针），可以是指向 List 类型对象，也可以是指向 String 类型对象。

5.4.1 可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- **不可变类型**：变量赋值 **a=5** 后再赋值 **a=10**，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变 a 的值，相当于新生成了 a。
- **可变类型**：变量赋值 **la=[1,2,3,4]** 后再赋值 **la[2]=5** 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

- **不可变类型：**类似 C++ 的值传递，如整数、字符串、元组。如 fun(a)，传递的只是 a 的值，没有影响 a 对象本身。如果在 fun(a) 内部修改 a 的值，则是新生成一个 a 的对象。
- **可变类型：**类似 C++ 的引用传递，如 列表，字典。如 fun(la)，则是将 la 真正的传过去，修改后 fun 外部的 la 也会受影响

python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说**传不可变对象和传可变对象。**

5.4.2 python 传不可变对象实例

通过 **id()** 函数来查看内存地址变化

```
def change(a):
    print(id(a))   # 指向的是同一个对象
    a=10
    print(id(a))   # 一个新对象

a=1
print(id(a))
change(a)
```

在调用函数前后，形参和实参指向的是同一个对象（对象 id 相同），在函数内部修改形参后，形参指向的是不同的 id。

5.4.3 传可变对象实例

可变对象在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。

```
# 可写函数说明
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4])
   print ("函数内取值: ", mylist)
   return

# 调用changeme函数
mylist = [10,20,30]
changeme( mylist )
print ("函数外取值: ", mylist)
```

传入函数的和在末尾添加新内容的对象用的是同一个引用。

5.5. 参数

以下是调用函数时可使用的正式参数类型：

- 必需参数
- 关键字参数
- 默认参数
- 不定长参数

5.5.1. 必需参数

必需参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

```
#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print (str)
   return

# 调用 printme 函数
printme(str="你好")
```

```
#可写函数说明
def printinfo( name, age ):
   "打印任何传入的字符串"
   print ("名字: ", name)
   print ("年龄: ", age)
   return

#调用printinfo函数
printinfo( age=50, name="runoob" )
```

5.5.2. 关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

```
#可写函数说明
def printme( str ):
   "打印任何传入的字符串"
   print (str)
   return

#调用printme函数
printme( str = "菜鸟教程")
```

函数参数的使用不需要使用指定顺序：

```
#可写函数说明
def printinfo( name, age ):
   "打印任何传入的字符串"
   print ("名字: ", name)
   print ("年龄: ", age)
   return

#调用printinfo函数
printinfo( age=50, name="runoob" )
```

5.5.3. 默认参数

调用函数时，如果没有传递参数，则会使用默认参数。以下实例中如果没有传入 age 参数，则使用默认值：

```
#可写函数说明
def printinfo( name, age = 35 ):
   "打印任何传入的字符串"
   print ("名字: ", name)
   print ("年龄: ", age)
   return

#调用printinfo函数
printinfo( age=50, name="runoob" )
print ("------------------------")
printinfo( name="runoob" )
```

5.5.4. 不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号 * 的参数会以元组(tuple)的形式导入，存放所有未命名的变量参数。

```
# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print ("输出: ")
   print (arg1)
   print (vartuple)

# 调用printinfo 函数
printinfo( 70, 60, 50 )
```

如果在函数调用时没有指定参数，它就是一个空元组。我们也可以不向函数传递未命名的变量。

```
# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print ("输出: ")
   print (arg1)
   for var in vartuple:
      print (var)
   return

# 调用printinfo 函数
printinfo( 10 )
printinfo( 70, 60, 50 )
```

还有一种就是参数带两个星号 **基本语法如下:

```
def functionname([formal_args,] **var_args_dict ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了两个星号 ** 的参数会以字典的形式导入。

```
# 可写函数说明
def printinfo( arg1, **vardict ):
   "打印任何传入的参数"
   print ("输出: ")
   print (arg1)
   print (vardict)

# 调用printinfo 函数
printinfo(1, a=2,b=3)
```

声明函数时，参数中星号 * 可以单独出现，例如:

```
def f(a,b,*,c):
    return a+b+c
```

如果单独出现星号 * 后的参数必须用关键字传入。

```
def f(a,b,*,c):
     return a+b+c

f(1,2,3)   # 报错
f(1,2,c=3) # 正常
```

5.6. 匿名函数

5.6.1. 介绍

Python 使用 **lambda** 来创建匿名函数。

所谓匿名，意即不再使用 **def** 语句这样标准的形式定义一个函数。

- **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** 函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。
- 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。

5.6.2. 语法

lambda 函数的语法只包含一个语句。

```
lambda [arg1 [,arg2,.....argn]]:expression
```

5.6.3. 实例

设置参数 a 加上 10:

```
x = lambda a : a + 10
print(x(5))
```

匿名函数设置两个参数:

```
# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2

# 调用sum函数
print ("相加后的值为 : ", sum( 10, 20 ))
print ("相加后的值为 : ", sum( 20, 20 ))
```

我们可以将匿名函数**封装在一个函数内**，这样可以使用同样的代码来创建多个匿名函数。

以下实例将匿名函数封装在 my_fun 函数中，通过传入不同的参数来创建不同的匿名函数:

```
def my_fun(n):
  return lambda a : a * n

my_doubler = my_fun(2)
my_tripler = my_fun(3)

print(mydoubler(11))
print(mytripler(11))
```

5.7. return语句

return [表达式] 语句用于退出函数，选择性地向调用方返回一个表达式。**不带参数值的return语句返回None。**之前的例子都没有示范如何返回数值，以下实例演示了 return 语句的用法：

```
# 可写函数说明
def sum( arg1, arg2 ):
   # 返回2个参数的和."
   total = arg1 + arg2
   print ("函数内 : ", total)
   return total

# 调用sum函数
total = sum( 10, 20 )
print ("函数外 : ", total)
```

5.8. 强制位置参数

Python3.8 新增了一个函数形参语法 / 用来指明函数形参必须使用指定位置参数，不能使用关键字参数的形式。

在以下的例子中，**形参 a 和 b 必须使用指定位置参数，c 或 d 可以是位置形参或关键字形参，而 e 和 f 要求为关键字形参**:

```
def f(a, b, /, c, d, *, e, f):
    print(a, b, c, d, e, f)
```

以下使用方法是正确的:

```
f(10, 20, 30, d=40, e=50, f=60)
```

以下使用方法会发生错误:

```
f(10, b=20, c=30, d=40, e=50, f=60)   # b 不能使用关键字参数的形式
f(10, 20, 30, 40, 50, f=60)           # e 必须使用关键字参数的形式
```

5.9. 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。

举个例子，我们来计算阶乘`n! = 1 x 2 x 3 x ... x n`，用函数`fact(n)`表示，可以看出：

$fact(n) = n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n = (n-1)! \times n = fact(n-1) \times n$

所以，`fact(n)`可以表示为`n x fact(n-1)`，只有n=1时需要特殊处理。

于是，`fact(n)`用递归的方式写出来就是：

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

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229915
608941463976156518286253697920827223758251185210916864000000000000000000000000
```

如果我们计算`fact(5)`，可以根据函数定义看到计算过程如下：

```
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要**注意防止栈溢出。**在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试`fact(1000)`：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，**在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。**这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的`fact(n)`函数由于`return n * fact(n - 1)`引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)`仅返回递归函数本身，`num - 1`和`num * product`在函数调用前就会被计算，不影响函数调用。

`fact(5)`对应的`fact_iter(5, 1)`的调用如下：

```
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的`fact(n)`函数改成尾递归方式，也会导致栈溢出。