函数

In [11]:
```
#设置全部行输出
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

当我们学习完条件和循环语句之后，我们就可以编写一些简单的小程序了。但是随着代码日益增加并且越来越复杂，所以需要一个程序对代码进行重组，而函数就是一些复杂代码的重组。使用函数编写程序将会使代码更简单明了，易于理解并且降低了编写程序的难度。接下来，学习函数的内容将会辅助我们编写更复杂和更高效的程序，教我们将复杂的事情简单化。

为了使程序的代码更加简单，需要把程序分解成较小的部分。函数就是这个较小部分的一种，接下来会学习到的对象和模块其实也是实现这个目的。

函数实际上是：

- 代码的一种组织形式
- 一个函数一般完成一项特定的任务

- 函数使用
  - 函数需要先定义
  - 使用函数，称之为调用
- 函数是组织好的，可以重复使用的，用来实现单一或者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。
- 其实我们已经接触到了很多python的内建函数，比如input(),print(),type()等等。
- 但自己创建函数也是熟练使用python必备的技能之一。这种自建函数也称作自定义函数。

1 函数的定义和调用

除了最初学到的用打印命令（print（）这个内建函数）来打招呼。我们还可以用定义函数，再调用去实现。

1.1 定义一个函数

- 我们可以自己定义一个函数，但是需要遵循以下的规则：
  - 函数的代码块以def关键字开头，后接函数标识符名称和圆括号（）。
  - 圆括号用来存储要传入的参数和变量，这个参数可以是默认的也可以是自定义的。
  - 函数内容以冒号起始，并且有强制缩进。
  - return[表达式]结束函数。选择性的返回一个值给对方调用。不带表达式的return相当于返回None。

1.1.1 函数的调用

定义一个函数：给了函数的一个名称，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就完成了。接下来我们就可以对定义好的函数进行调用执行了。

In [3]:
```
def hehe():
    print("我是一个函数")
```

In [15]:
```
#调用上面定义的函数
#直接函数名后面跟括号就可以实现：
hehe()
```

```
我是一个函数
```

函数的调用和运行机制：当函数func()发生调用时，python会自动找到def func()的定义过程，然后依次执行代码块部分，也就是冒号下有缩进的部分。这时候只需要一条语句就能轻松点额实现函数内所有更能。加入我们想要把上述内容打印三次，只需要调用三次函数即可。

In [16]:
```
hehe()
hehe()
hehe()
```

```
我是一个函数
我是一个函数
我是一个函数
```

1.1.2 练习1

尝试定义一个函数，并调用5次。

In [ ]:
```
```

1.1.3 函数的参数

现在，我们可以去看看括号里面的东西了！

括号里面其实是函数的参数，上面的小练习中都是无参数函数。但是，不含参数的函数就像一个对同样代码的打包程序，这跟使用循环语句没什么区别。所以就有了函数参数。参数可以使每次调用的函数有不同的实现，加入参数的概念，使函数的功能越发的强大，将其与循环语句从本质上区别开来。大大简化了重复编写类似程序的负担。

1、定义打印字符串的函数：

(1)

In [26]:
```
#定义一个函数
def output(aa):
    print(aa)
```

In [28]:
```
#调用一个函数
output('哈哈哈哈')
```

```
哈哈哈哈
```

In [11]:
```
#用循环语句实现多次调用
for i in range(5):
    output("""我要好好学python """"\n")
```

```
我要好好学python

我要好好学python

我要好好学python

我要好好学python

我要好好学python
```

(2)

In [1]:
```
def func_02(num):
    print(num + 100)

func_02(1)
```

```
101
```

注意：定义了一个带参数的函数，在调用时必须带参数调用该函数。

1.1.4 练习2

尝试定义一个带有参数的函数，并调用3次。

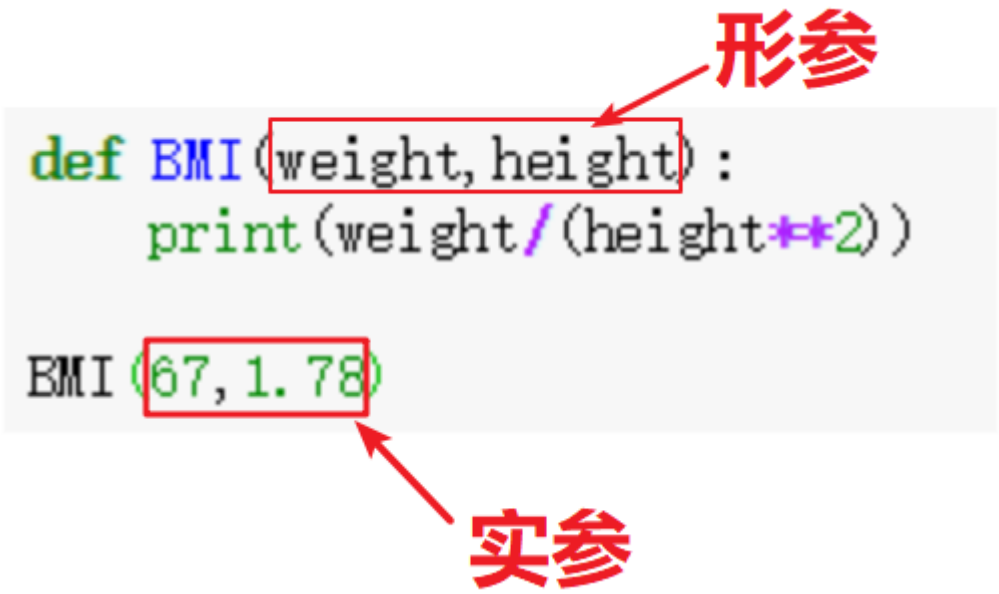
In [ ]:

```
```

1.2 参数其他概念

1.2.1 形参和实参

- 参数从调用的角度来看，分为**形式参数（parameter）**和**实际参数（argument）**
- 跟绝大多数语言类似，**形参**是指函数创建和**定义**过程中的参数，而**实参**则是指函数在**调用**过程中实际传递的参数。
- 定义时小括号中的参数，用来接收参数用的，称为**“形参”**
- 调用时小括号中的参数，用来传递给函数用的，称为**“实参”**



1.2.2 关键字参数

普通参数叫做**位置参数**，通常在调用一个函数的时候，如果一个函数中的参数不止一个，那么记得每一个参数的位置是一个很困难，并且很容易出错的问题。

In [13]:

```
def BMI(weight,height):
    print(weight/(height**2))

BMI(67,1.78)
```

```
21.146319909102385
```

此时，使用**关键字参数**则可以简单的解决潜在的问题，我们可以通过以下的例子进行体会：

In [15]:
```
BMI(weight=67,height=1.78)
BMI(height=1.78,weight=67)
```

```
21.146319909102385
21.146319909102385
```

注意！关键参数是在调用函数的时候使用，而不是在定义函数的时候使用

1.2.3 默认参数

我们学习了关键词参数之后，还需要去理解**默认参数**。因为初学者很容易将二者搞混。

- **默认参数**实际上是在**定义函数的时候**赋予了默认值的参数。
- 一个函数参数的默认值，仅仅在该函数定义的时候，被赋值一次。如此，只有当函数第一次被定义的时候，才讲参数的默认值初始化到它的默认值（如一个空的列表）。
- 使用默认参数时一定是指向不可变对象，这里主要为防止反复调用过程中出现问题。

In [18]:
```
def BMI(weight,height=1.7):
    print(weight/(height**2))
```

In [19]:
```
BMI(67,1.78)
```

```
21.146319909102385
```

- **形参height被定义了默认值1.7。**
- 在调用函数的时候，当height没有传递任何实参时，就会使用默认值，如下：

In [25]:
```
BMI(67)
```

```
他 怎么这么好看
```

关键字参数和默认参数的区别：

- **关键词参数**
  在函数调用的时候必须要带参数调用，通过参数名指定要赋值的参数，这样做就不怕因为该不清楚参数的顺序而导致函数调用出错。

- **默认参数**
  是在参数定义的过程中，为形参赋初值，当函数调用的时候不传递实参时，则默认使用形参的初始值代替。

1.2.4 可变参数

发明可变参数的动机在于：定义函数的作者并不知道这个函数究竟需要多少个参数。虽然听起来费解，但确实会出现这种情况。这时候，仅需要在参数前面加上*号即可：

In [45]:
```
def func05(*hehehe):
    print("参数的个数是：",len(hehehe))
    print(hehehe)
```

In [46]:
```
a="我是a呀"
b="我是b呀"
c="我是c呀"
d="我是d呀"

func05(a,b,c,d)
```

```
参数的个数是： 4
('我是a呀', '我是b呀', '我是c呀', '我是d呀')
```

- python就是把可变参数的参数们打包成一个**元组**。
- 在可变参数后面，如果还需要指定其他参数，在调用函数的时候就需要**使用该参数的关键词**来指定，否则python就会将我们实参都列入到可变参数的额范畴。重复调用的过程中就容易出现问题。

In [49]:
```
a="我是a呀"
b="我是b呀"
c="我是c呀"
d="我是d呀"

def func06(*params,extra_01,extra_02):
    print("可变参数是：",params)
    print("位置参数是：",extra_01)
    print("位置参数是：",extra_02)

func06(a,c,extra_01=b,extra_02=d)
```

```
可变参数是： ('我是a呀', '我是c呀')
位置参数是： 我是b呀
位置参数是： 我是d呀
```

这里报错的原因是，调用函数的时候少一个位置参数，位置参数但凡传入，在调用时就必须要含参数调用，否则就会报错。因此建议大家在定义函数的参数设定时，如果含有可变参数，可以将其设置为默认参数，这样不容易出错：

1.2.5 return语句

return[表达式]语句用于退出函数，选择性地向调用方法返回一个表达式。不带参数值的return语句返回None。之前的例子没有示范返回值，下面我们来演示一下return语句的用法：

In [52]:
```
def sums(arg1 , arg2):
    print("这句话在return语句之前，执行本函数会输出")
    cc=arg1 + arg2
    return cc
    print("这句话在return语句之后，执行本函数不会输出")

sums(10,55)

print("sums()函数调用完毕，这句话不在sums()所属的代码块内")
```

```
这句话在return语句之前，执行本函数会输出
```

Out[52]:
```
65
```

```
sums()函数调用完毕，这句话不在sums()所属的代码块内
```

In [51]:
```
sums(10,55)
```

```
这句话在return语句之前，执行本函数会输出
```

Out[51]:
```
70
```

定义一个计算面积的函数：

In [54]:
```
def area(width, height):
    area = width * height
    return area
```

In [55]:
```
area(4,5)
```

Out[55]:
```
20
```

In [14]:
```
area(height = 6,width = 5)
```

```
30
```

Out[14]:
```
30
```

1.2.6 练习3

定义一个函数，计算a^2+b^3，将a=3，b=5代入定义函数求出。

In [4]:
```

```

Out[4]:
```
134
```

2 变量的作用域

2.1 代码块与作用域

python中只有**模块（module），类(class)以及函数(def ,lamda)**才会引入新的作用域，其他的代码块（如if/else/elif/、for/while、try/except等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码：

In [61]:
```
a=2

if a<3:
    msg = 'I am from CDA'
    print(msg)
```

```
I am from CDA
```

In [62]:
```
msg
```

Out[62]:
```
'I am from CDA'
```

实例中 msg **变量定义在 if 语句块中**，但外部还是**可以访问**的。

如果将 msg **定义在函数中**，则它就是局部变量，外部**不能访问**：

In [63]:
```
def test():
    msg_01 = 'I am from CDA'
    return msg_01
```

In [64]:
```
test()
```

Out[64]:
```
'I am from CDA'
```

In [65]:
```
msg_01
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-65-8cf3c7a919ab> in <module>()
----> 1 msg_01

NameError: name 'msg_01' is not defined
```

2.2 全局变量和局部变量

定义在**函数内部**的变量拥有一个**局部作用域**，定义在**函数外**的拥有**全局作用域。**

局部变量只能在其被声明的**函数内部**访问，而**全局变量**可以在**整个程序**范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

In [68]:
```
total_01 = 0 #这是一个全局变量
#可以写函数说明
def sum_01(arg1,arg2):
    #返回2个参数的和
    total_01 = arg1 + arg2    #total 在这里就是局部变量
    arg3=33
    return arg3

sum_01(2,3)

total_01
```

Out[68]: 33

Out[68]: 0

- 全局变量在整个代码段中都是可以访问到的，但是不要试图在函数内部去修改全局变量。
- 因为全局变量的修改实际上是重新定义了一个新变量，内存地址发生了改变。

2.3 变量在作用域查找顺序

函数变量的作用域其实就是平时我们所说的**变量可见性**。

python当中，程序的变量并不是在任何位置都能被访问，访问权限取决于变量是在什么位置赋值的。
变量的作用域决定了变量在哪一部分程序可以被访问。
python的作用域一共有4种：

- L （Local）局部作用域
- E （Enclosing）闭包函数外的函数中
- G （Global）全局作用域
- B （Built-in）内建作用域

以 L --> E --> G --> B 的规则查找：

即在局部找不到，则会去局部外的局部找，比如闭包，再找不到，就会去全局找，再者则会去内建中找。

In [58]:
```
import os
```

In [61]:
```
a=2
b=2
c=2
#上面是全局作用域

def outer():
    a=1                 #闭包外的作用域
    b=1                 #闭包外的作用域

    def inner():
        msg=1
        a=0             #闭包内的作用域
        print(os)
        return a,b,c  #inner()会返回一个函数
    return inner()

outer()
```

```
<module 'os' from 'C:\\ProgramData\\Anaconda3\\lib\\os.py'>
```

Out[61]: (0, 1, 2)

3 内嵌函数和闭包

3.1 内嵌函数

python的函数定义可以是嵌套的，也就是允许在函数内部创建另一个函数，这种函数叫做内嵌函数或内部函数：

In [58]:
```
def func_1():
    print("func_1()正在被调用...")
    def func_2():
        print("func_2()正在被调用...")
    return func_2()
```

In [57]:
```
func_1()
```

```
func_1()正在被调用...
func_2()正在被调用...
```

如果想在调用func_1()时，将func_2()定义的print也执行了，这就需要在func_1() 这一层就执行func_2()：

In [1]:
```
def func_1():
    print("func_1()正在被调用...")

    def func_2():
        print("func_2()正在被调用...")
    func_2()  #这时调用Func_1() 就会调用Func_2()

func_1()
```

```
func_1()正在被调用...
func_2()正在被调用...
```

如果你想要将内嵌函数中定义的变量返回外层函数作用域中使用：

- 内嵌函数使用return，外层函数调用内嵌函数时候，用一个变量“接住”即可。
  比如下面的tot变量：

In [6]:
```
def func_5(a,b):
    sums = a+b
    print("func_5()正在被调用...")

    def func_6(d):
        print("func_6()正在被调用...")
        tot = sums+d
        return tot

    tot_02=func_6(17)
    return sums,tot_02

func_5(6,7)
```

```
func_5()正在被调用...
func_6()正在被调用...
```

Out[6]: (13, 30)

3.2 global 关键字

- 在函数内部仅仅去访问全局变量就好，尽量不要在函数内部去修改全局。
- 因为这样的话，python会使用屏蔽（Shadowing）的方式“保护”全局变量。
- 一旦函数内部试图修改全局变量，python就会在函数内部自动创建一个名字一样的局部变量。
- 这样修改的结果只会修改到局部变量，而不会影响到全局变量。

In [71]:
```
count = 8.8

def Func2():
    global count
    count= 10
    print(count)    #"本人站在闭包作用域内，在这里，count等于10"

Func2()

count
```

```
10
```

Out[71]:
```
10
```

In [53]:
```
count
```

Out[53]:
```
10
```

通过上述例子，我们可以发现，函数中修改全局变量可能导致程序可读性变差、出现莫名其妙的bug、代码的维护成本高。因此不建议在函数内部修改全局变量。

但是如果你一定要去修改这个全局变量，那么不妨试一试**global关键字**，修改一下上述程序：

In [74]:
```
count = 8.8
def Func():
    global count       #需要使用全局变量用global关键字说明
    count = 10
    print(count)

Func()
count
```

```
10
```

Out[74]:
```
10
```

3.3 nonlocal关键字

修改嵌套作用域中的变量则需要 ***nonlocal*** 关键字了，如下实例：

In [73]:
```
def outer():
    num= 10
    def inner():
        nonlocal num
        num = 200
    inner()
    print(num)

outer()
```

```
200
```

3.4 闭包

闭包是函数式编程的重要语法结构，Pyhon中的闭包从形式上定义为：

- 如果在一个内部函数里，**对在外部作用域（但不是全局作用域）的变量进行引用，那么内部函数就被认为是闭包。**
- 闭包函数的必要条件：
  - 闭包函数必须返回一个函数对象
  - 闭包函数返回的那个函数必须引用外部变量（一般不能是全局变量），而返回的那个函数内部不一定要return
- 用比较容易懂的人话说，就是当某个函数被当成对象返回时，夹带了外部变量，就形成了一个闭包。看例子：

In [13]:
```
def y(b,c):
    def yy(x):
        return b*x+c
    return yy

y(2,3)(4)
```

Out[13]: 11

In [14]:
```
b2_c3=y(2,3)

b2_c3(4)
```

Out[14]: 11

3.5 练习4

用闭包的方式定义线性函数 `y=x**a+x**b+c` ，使得可以任意定义a、b、c，得到一条公式，使得可以通过输入x来的到y。

In [ ]:
```

```

3.6 lamda表达式

python允许使用lambda创建匿名函数。这个匿名函数我们通过字面意思可能不太好理解，但是通过下面的两个例子，相信大家很快就能理解：

1）一个参数的例子：

In [51]:
```
def someFunc(x):
    return 2 * x + 1

someFunc(3)
```

Out[51]: 7

In [18]:
```
aa = lambda x :2 * x + 1
```

In [19]:
```
aa(4)
```

Out[19]: 9

2）两个参数的例子：

In [54]:
```
def someFunc_1(x,y):
    return x + y

someFunc_1(5,6)
```

Out[54]: 11

In [55]:
```
gg = lambda x,y : x + y
```

In [56]:
```
gg(5,6)
```

Out[56]: 11

- lambda函数的应用场景：
  - python编写一些程序的脚本时，使用lambda就可以省下来定义函数的过程。比如写一个简单的脚本管理服务器，就没有必要定义函数，再去调用它。直接使用lambda函数，可以使程序更加简明。
  - 一些只需要调用一两次的函数，就**没有必要为了想个合适的函数名字而费精力**了，直接使用lambda函数就可以省去取名的过程。
  - 阅读普通函数，通常需要跳到开头def定义的位置，使用lambda函数可以省去这样的步骤。
- 下面通过两个高级函数来让大家体会lambda表达式的“匿名”属性。

4 filter()和map()函数

在这里介绍这两个内建函数（BIF）,主要有两方面的原因：首先，这两个函数很实用。另一方面，在这里刚好可以和上面我们介绍的lambda函数结合起来使用。

4.1 filter()函数

我们研究第一个内建函数是一个过滤器。我们每天都会接触到大量的数据，过滤器的作用就显得格外重要，通过过滤器，就可以保留我们关注的信息，把其他不感兴趣的东西直接丢掉。 python对filter（）的解释大致意思是：

- filter（）有两个参数。第一个参数可以是一个函数，也可以是一个None。
  - 如果是第一个参数是函数的话，则将第二个迭代数据里的每一个元素作为函数的参数进行计算，把返回True的值筛选出来；
  - 如果第一个参数为None，则直接将第二个参数中为True的值筛选出来。下面举个例子：

In [78]:
```
temp = filter(None,[1,2,0,False,True])

list(temp)
```

Out[78]: [1, 2, True]

利用filter(),尝试写一个筛选奇数的过滤器：

In [79]:
```
def is_odd(n):
    return n % 2    #如果n是偶数，n%2返回0，则返回假，会在下面filter()函数被过滤掉

tmplist = filter(is_odd,range(10))

newlist = list(tmplist)
newlist
```

Out[79]: [1, 3, 5, 7, 9]

现在，我们使用lambda，简化上述的函数代码：

In [82]:
```
add=lambda x:x % 2==0

add(8)
```

Out[82]: True

In [19]:
```
list(filter(lambda x:x % 2==0 ,range(10)))    #lambda x:x % 2==1的作用就是一个函数，不管返回真还是
```

Out[19]: [0, 2, 4, 6, 8]

4.2 练习5

利用filter(),尝试写一个筛选偶数的过滤器，对10到30的数值进行筛选：

In [4]:
```

```

Out[4]: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]

4.3 map()函数

map在这里不是地图的意思，在编程领域，map一般做“映射”来解释。

map()也有两个参数，任然一个是函数，一个是可迭代序列，将序列的每一个个元素作为函数的参数进行运算加工，直到可迭代序列每个元素加工完毕，返回所有加工后的元素构成的新序列。

有了刚才filter()的经验，在这里举个map()函数的例子：

filter返回的是原象

In [24]:
```
list(filter(lambda x: x %2,range(10)))
```

Out[24]: [1, 3, 5, 7, 9]

map返回的是象

In [25]:
```
list(map(lambda x:x %2,range(10)))
```

Out[25]: [0, 1, 0, 1, 0, 1, 0, 1, 0, 1]

filter()过滤后，返回的是结果为真（即不等于0）的对象，而map()遍历，返回的是计算结果

4.4 练习6

利用map(),尝试写一个筛选奇数的过滤器，对10到15的数值进行筛选,返回布尔值。 生成以下判断奇数的字典效果：

```
{10: False, 11: True, 12: False, 13: True, 14: False, 15: True}
```

In [7]:

Out[7]: {10: False, 11: True, 12: False, 13: True, 14: False, 15: True}

In [11]:

Out[11]: {10: False, 11: True, 12: False, 13: True, 14: False, 15: True}

5 递归

5.1 什么是递归

递归其实不属于基础的范畴，但是对于想要写出漂亮程序的程序员来说，是一个非常好的编程思路。

在程序上，**递归**实质上是**函数调用自身的行为。**

5.2 写一个求阶乘的函数

5.2.1 非递归方法

In [43]:
```
# 写一个非递归的普通求阶乘的函数
def recursion(n):
    result = n
    for i in range(1,n):
        result *= i
    return result

number = int(input('请输入一个整数：'))
result = recursion(number)
print("{}的阶乘是：{}".format(number,result))
```

```
请输入一个整数：3
3的阶乘是：6
```

5.2.2 递归法

In [1]:
```
#写一个递归的高级版本：
# 1）先设置递归限制
import sys
sys.setrecursionlimit(1000)          #设置最大递归次数，根据自己需要设置成任意数字，成千或者上万。
```

In [33]:
```
# 2) 下面我们来写一个递归版本的阶乘函数：
def factorial(n):
    if n == 1 or n==0:
        return 1
    else:
        return n * factorial(n-1)

number = int(input('请输入一个整数：'))
result = factorial(number)
print("{}的阶乘是：{}".format(number,result))
```

```
请输入一个整数：5
5的阶乘是：120
```

我们这个例子实际上是满足了两个条件：

1）调用函数的本身

2）设置了正常的返回条件

具体的我们看成是以下分析步骤：

```
factorial(5) = 5 * factorial(4)
       factorial(4) = 4 * factorial(3)       24
           factorial(3) = 3 * factorial(2)               6
               factorial(2) = 2 * factorial(1)               2
                   factorial(1) = 1                      1
```