1. **Python函数**

**第一部分**

**一、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：

#全局变量

a=23;

sum1=0

def sum(n):

#局部变量sum2

sum2=0

#局部变量i和 n

for i in range(n+1):

sum2=sum2+i

#只可以访问全局变量sum1，不可以修改全局变量sum1

#sum1=sum1+i

return sum2

print(sum(100))

有如下几点要注意：

1、在函数的内部只可以访问全局变量，不可以修改全局变量

2、在函数的内部当局变量和局部变量同名时，访问局部变量

例如：

#全局变量

a=23;

sum2=0

def sum(n):

#局部变量sum2

sum2=0

#局部变量i和 n

for i in range(n+1):

sum2=sum2+i

#只可以访问全局变量sum1，不可以修改全局变量sum1

#sum1=sum1+i

return sum2

print(sum(100))

#访问全局变量sum2

print(sum2)

3、在函数的外部不可以访问局部变量，可以访问全局变量

#全局变量

a=23;

sum1=0

def sum(n):

#局部变量sum2

sum2=0

#局部变量i和 n

for i in range(n+1):

sum2=sum2+i

#只可以访问全局变量sum1，不可以修改全局变量sum1

#sum1=sum1+i

return sum2

print(sum(100))

#访问局部变量sum2，报错

print(sum2)

1. 定义在if语句块、for循环和while循环语句块中的变量，而这些语句不是定义在函数中，变量默认为全局变量。

Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 if/elif/else/、try/except、for/while等）是不会引入新的作用域的，也就是说这这些语句内定义的变量，外部也可以访问。

#全局变量

*#encoding=utf-8*a=23  
sum1=0  
**if** 3<4:  
 sum2=100  
*#访问全局变量sum2，不报错***print**(sum2)

for t in range(100):

sum3=300;

while 4>1:

sum4=500

break

print(sum3)

print(sum4)

5、global 和 nonlocal关键字

当内部作用域想修改外部作用域的变量时，即当一个全局变量在局部范围内，修改数据时，要用到global关键字了。

#全局变量

a=23;

sum1=0

def sum(n):

#局部变量sum2

sum2=0

#全局变量

global sum1

for i in range(n+1):

sum2=sum2+i

sum1=sum1+i

return sum2

sum(100)

print(sum1)

如果要修改(先读取后修改)嵌套作用域(闭包)（enclosing 作用域，外层非全局作用域）中的变量则需要

nonlocal 关键字了，但是只有python3.x以上才支持。

def make\_counter():

count = 0

def counter():

nonlocal count

count += 1

return count

return counter

def make\_counter\_test():

mc = make\_counter()

print(mc())

print(mc())

print(mc())

make\_counter\_test()

结果为：

1

2

3

**第二部分**

**二、变量作用域**

Python 中，程序的变量并不是在哪个位置都可以访问的，变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。Python的作用域一共有4种，分别是：

 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，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内建中找。实际上就是就近原则。

x = int(2.9) # 内建作用域

g\_count = 0 # 全局作用域

def outer():

count = 1 # 闭包函数外的函数中

def inner():

i\_count = 2 # 局部作用域

**三、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 对象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。

**四、python中的可哈希和不可改变性**

1、可哈希（hashable）和不可改变性（immutable）

如果一个对象在自己的生命周期中有一哈希值（hash value）是不可改变的，那么它就是可哈希的（hashable）的，因为这些数据结构内置了哈希值，每个可哈希的对象都内置了\_\_hash\_\_方法，比如比如字符串，元组和整数，可通过help(int)、help(str)、help(tuple)、help(frozenset())，发现他们都包含

\_\_hash\_\_(self, /)

Return hash(self).

而help(list)、help(dict)，help(set),都是不可哈希的，因为\_\_hash\_\_ = None，所以它并不支持哈希运算。

可哈希的对象可以通过哈希值进行对比，也可以作为字典的键值和作为set函数的参数。

所有python中所有不可改变的的对象（imutable objects）都是可哈希的，比如字符串，元组，也就是说可改变的容器如字典，列表不可哈希（unhashable）。我们用户所定义的类的实例对象默认是可哈希的（hashable），它们都是唯一的，而hash值也就是它们的id()。

2、哈希

它是一个将大体量数据转化为很小数据的过程，甚至可以仅仅是一个数字，以便我们可以用在固定的时间复杂度下查询它，所以，哈希对高效的算法和数据结构很重要。

#1、整数的hash值就是它自己  
a=23  
print(a.\_\_hash\_\_())

#2、字符串的哈希值

str=**"12"**print(str.\_\_hash\_\_())

#3、元祖的哈希值

tup1=(1,2,3)  
print(tup1.\_\_hash\_\_())

4、列表的哈希值

list1=[1,2,3]  
print(list1.\_\_hash\_\_())

TypeError: 'NoneType' object is not callable

3、不可改变性

它指一些对象在被创建之后不会因为某些方式改变，特别是针对任何可以改变哈希对象的哈希值的方式

4、 联系：

因为哈希键一定是不可改变的，所以它们对应的哈希值也不改变。如果允许它们改变，那么它们在数据结构如哈希表中的存储位置也会改变，因此会与哈希的概念违背，效率会大打折扣。

**第三部分**

**五、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。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。

def sum(a):

# 打印a的值变化之前的值

print("before a",a)

#a的值变化

a=a+1

#打印a的值变化之后的值

print("after a",a)

b=13;

#传递之前b的值

print("before b",b)

sum(b)

#传递之后的值

print("afert b",b)

执行结果：

('before b', 13)

('before a', 13)

('after a', 14)

('afert b', 13)

形参a值的变化并没有引起实参b值的变化。

可理解为不可变类型传递的是这个参数的值的副本；计算机在内存中新分配一个空间，用来存储这个副本。

 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。

def sum(a):

# 打印a的值变化之前的值

print("before a",a)

#a的值变化

a[0]=12

#打印a的值变化之后的值

print("after a",a)

b=[2,3,4,5];

#传递之前b的值

print("before b",b)

sum(b)

#传递之后的值

print("afert b",b)

执行结果：

('before b', [2, 3, 4, 5])

('before a', [2, 3, 4, 5])

('after a', [12, 3, 4, 5])

('afert b', [12, 3, 4, 5])

形参a值的变化引起实参b值的变化，实际上传递的是b对象的地址给a,这时，a和b指向同一个对象，a的变化必定引起b的变化。

可理解为可变类型传递的是这个参数的地址，从而导致两个参数的地址相同，任何一方的改变必会引起另外一方的改变。