1. 函数基础

提高程序的可读性

函数是一个通用的程序结构的组成部分,使用函数的主要特点有:

1 提高代码的复用性,降低代码的冗余度

我们可以把一个功能的代码抽取为一个函数,如果我们需要使用这个功能直接调用对应的函数就可以完成对应的操作,不在需要编写同样的代码,保证了代码的一次编写,多次运行的特点

2 提高代码的可读性

通过函数我们一个系统拆分为多个功能区编写完成,每一个系统功能都编写一个对应的函数去处理,如果需要使用到对应的功能,直接调用该函数即可.

1.1. 函数的定义和调用

函数是一段代码的集合,并且为这段代码取一个名字,我们称之为函数名,通过函数名可以达到代码多次调用的效果

具体格式:

def 函数名([参数 1,参数 2,...,参数 n]):

函数体

[return 值]

调用:

函数名([参数 1,参数 2,...,参数 n])

```
# 函数的定义
def speak():
    print("你好,兄弟")
# 函数必须调用才会执行
speak()
```

① 函数定义通过 def 关键字定义

② def 创建了一个函数对象并且赋值给了一个变量名(函数名)

③ 函数必须调用才会执行

④ 函数是完成一个功能的代码块,通常情况会接受参数,进行业务处理,然后返回对应的结果

⑤ 在 python 中已经内置一些函数,可以通过对应操作查看所有的内置的函数(比如 len,sum 等)

```
# 导入内置模块 builtins
import builtins
# 打印 builtins 模块中所有的内置函数
print(dir(builtins))
```

1.2. 函数的形参和实参

在函数定义上面的参数列表,我们称之为形式参数,在调用函数的时候传递的参数,我们称之为实际参数

对于形式参数和实际参数的个数应该要保持一致

对于形参和实际参数之间的传递就是赋值操作,把实际参数的对象的引用赋值到形式参数上

```
# 函数的形参和实际参数
# 定义一个用于计算 x*y 并且返回结果的一个函数
def times(x,y):
    print("接受对应的参数")
    print("x:%s,y:%s"%(x,y))
# 调用函数,传递参数 x = 1 y = 6
times(1,6)
# 如果传递的参数个数和形式参数的个数不一致的话,直接报错 TypeError
# times(1,6,10)
a = 1
b = 2
# 对于在函数中改变 x,y 的值,不会对 a,b 进行改变
times(a,b)
print("a:%d,b:%d"%(a,b))
```

```
def times(x,y):
    x[0]=100
    y[0]=200
    print("x:%s,y:%s"%(x,y))
a = [10]
b = [20]
# 如果传递的是可变类型的数据,那么会改变原来的值
times(a,b)
print("a:%s,b:%s"%(a,b))
```

1.3. 函数的返回值

当函数被调用时,在函数完成工作之后,需要将控制权返回给调用者,函数是通过 return 语句将结果传递给调用者,并且接受当前函数的运行.

如果明确的使用 return 进行了返回, 那么函数的返回值是 return 后面的值

如果没有使用 return 进行返回,那么函数的返回值是 None

可以返回多个值, 会使用一个元组进行封装

```
# 计算 x,y 的乘积运算
def times(x,y):
    print("执行函数运算,参数 x:%s,y:%s"%(x,y))
```

```
    # 如果没有 return 语句,那么返回 None
    return x * y
ret = times(2,3)
print(ret)

# 找到列表中的第一个大于 100 的数,并且返回
def times(x,y):
    # 可以返回多个值,会把多个值封装为一个元组 元组的装包功能
    return x**2,y**2

ret = times(1,2)
# a,b = times(1,2)
print(ret)
print(type(ret))
```

1.4. 函数的四种类型

```
# 函数的四种方式
# 1 无参数无返回
# 函数定义
def talk():
    print("很高兴见到你")
# 函数调用
talk()
# 2 有参数无返回
def talk(name):
    print("亲爱的%s,很高兴见到你"%name)
talk("baby")
# 3 无参数有返回
def getName():
    # 使用 return 进行值的返回
    return "王五"
# 定义一个变量接受返回值
name = getName()
print(name)
# 4 有参数有返回
def add(x, y):
    return x+y
ret = add(1,2)
print("结果:",ret)
ret = add("你好","王五")
print("结果:",ret)
```

2. 函数的嵌套

2.1. 函数的嵌套调用

如果在一个函数 A 执行的时候,调用另外一个函数 B,那么我们称之为函数的嵌套调用,在 A 执行到某一个时候的时候,开始执行函数 B,当函数 B 执行完以后继续回来执行函数 A

```
def fun_a():
    print("开始 fun_a 执行函数")
    #嵌套调用
    fun_b()
    print("完成 fun_a 执行函数")
def fun_b():
    print("开始 fun_b 执行函数")
    print("完成 fun_b 执行函数")
# 调用函数 fun_a
fun_a()
```

2.2. 函数的嵌套定义

如果在一个函数 A 中定义一个函数 B,那么对于这样的函数,我们称之为函数的嵌套定义

```
def outer():
    a = 100
    print(a)
    # 在函数里面定义一个函数 inner,函数的嵌套定义
    def inner():
        b = 200
        print(b)
    return a
# 输出 100
outer()
#报错,找不到 inner 函数
inner()
```

3. 函数的作用域

在任何情况下,一个变量的作用于总是有代码中被赋值的地方所决定,并且与函数调用没有关系,通常情况系,变量可以在三个地方分配,分别对应 3 种不同的作用域

① 如果一个变量是定义在函数外面,那么这个变量在整个文件(模块)中都可以访问

② 如果一个变量是定义在函数的里面,那么这个变量只能在函数里面使用,当函数执行完成以后,变量失效,不能访问

③ 如果一个变量 x 是地能以在函数 A 中定义,并且在函数 A 中又定义了一个函数 B,那么对于函数 B 来说,x

是一个非本地的变量

```
# 在文件中定义的变量,在整个文件都可以访问
a = 100
def fun_a():
    b = 200
    # 在函数中访问
    print("函数中",a)
    print("函数中 b:",b)
    def fun_b():
        # 对于函数 fun_b 来说 c 为本地变量, b 为非本地变量 a 为全局变量
        c = 300
        print("b 函数中的 c",c)
    fun_b()
#调用函数
fun_a()
print("函数外面",a)
#在函数外面访问:
#报错 NameError: name 'b' is not defined
# print(b)
```

3.1. 作用域概念

内嵌的模块是全局作用域: 每一个 py 文件就是一个全局作用域,在一个文件中,在同一 py 文件鸿,可以直接访问全局作用域中的变量

全局作用域仅限于单个文件,这里所说的是对于全局作用域只是在当前的 py 文件(模块)中有效,在另外一个文件中不能简单的直接访问变量

每次对函数的调用都参加了一个新的本地作用域,每次调用函数,都会根据定义的函数在去创建对应的对象分配内存等

在函数里面的赋值变量变量除非声明为全局变量或者非本地变量,否则均为本地变量

3.2. 变量的解析规则

变量名引用默认情况会分为四个作用域去查找: 首先是本地作用域,之后是嵌套函数中作用域,然后是全局作用域,最后在内置作用域中寻找,如果都找不到,则报错未定义

默认情况下,变量名的赋值会修改或者创建本地变量

通过全局声明(global)和非本地(nonlocal)什么可以将赋值的变量映射到对应的作用于上去,而不再是本地变量处理

```
# 在文件中定义的变量,在整个文件都可以访问
a = 100
def fun_a():
    # 把变量 a 声明为一个全局变量
```

```
    global a
    #本地变量,只在当前函数中有效
    b = 200
    # 修改全局变量的值
    a = 1000
    def fun_c():
        # 把 b 声明为非本地变量 嵌套函数中 fun_a 中的变量
        nonlocal b
        b = 1
        c = 300
        print("fun_c",b,c)
    #调用函数
    fun_c()
    print("a,b",a,b)

fun_a()
print(a)
print(type(fun_a))
```

4. 函数的参数

4.1. 参数的传递

①参数的传递是自动的将对象赋值给本地变量名来实现.

②在函数内部的参数名的赋值不会影响调用者

③改变函数的可变参数的值可能会对调用者有影响,会影响原来的对象的值

④不可变参数通过值专递

⑤可变参数传递的是引用

```
a = 100
# 打印 a 的引用值
print("a 的引用值",id(a))

def fn(x):
    print("x 的引用值",id(x))
fn(a)
```

```
a = 100
# 打印 a 的引用值
print("a 的引用值",id(a))

def fn(x):
```

```
    x = 200
    print("x 的引用值",id(x))
    print("x 的值",x)
fn(a)
print("调用函数后的 a 的值:",a)
```

```
a = [100,200]
# 打印 a 的引用值
print("a 的引用值",id(a))

def fn(x):
    x[0] = 0
    print("x 的引用值",id(x))
    print("x 的值",x)
fn(a)
print("调用函数后的 a 的值:",a)
```

4.2. 位置参数(命名参数)

在函数调用的时候,传递的参数默认情况是按照位置在进行传递进行赋值的,当然也可以通过参数名来进行指定

```
def fun_a(x,y,z):
    print("x:%s,y:%s,z:%s"%(x,y,z))
# 使用位置参数,安装位置的对应关系进行赋值
fun_a(1,2,3)
# 使用命名参数,给具体的参数名称指定值,这种方式可以看出每个参数的意义
fun_a(y=1,z=27,x=9)
```

4.3. 默认参数

在定义函数的时候,可以给一个参数设定一个默认值,那么当函数调用的时候,如果对应的参数没有传入实际参数,则使用默认参数参与运算

```
def sum(x,y=50,z=100):
    print("x:%s,y:%s,z:%s"%(x,y,z))
sum(1,2)
sum(1,2,3)
sum(1,z=30)
```

4.4. 关键字参数

在调用函数 A 的时候,通常情况我们可以使用位置参数进行值的传递,也可以是名称参数进行传递,但是如果我们要求一个函数的某些参数必须使用 name=value 这种方式来传递的话,我们称之为关键字参数,如果需要使用关键字参数,只需要形参前面添加一个特殊的形式参数*,表示后面的形式参数都需要使用命名参数的

方式进行赋值操作

```
# 其中 y 在*参数后面,必须通过指定名称的方式进行赋值
# y 称为关键字参数
def sum(x,*,y):
    print("x:%s,y:%s"%(x,y))
sum(1,y=2)
sum(1,y=3)
sum(1,y=4)
```

4.5. 不定长参数

在函数定义的时候,不确定函数调用的时候具体传递几个参数过来,可以是 1 个,可以是 2 个,那么在这个时候我们就需要使用不定长参数,在形参中使用一个特殊的参数*args 可以用来接受 0 个或者多个非关键字参数

```
def sum(*args,y):
    print("x:%s,y:%s"%(args,y))
sum(y=2)
sum(1,y=3)
sum(1,2,3,y=4)
```

通过观察可以发现,其实 args 真实的数据类型是一个元组

在关键字参数后面不能在写不同的位置参数,当使用*args 的时候

4.6. 不定长关键字参数

在调用函数的时候如果传递的是命名参数的形式,而且个数长度不固定的话,我们需要使用**keywords 一个特殊的参数来接受,需要注意的是在**keywords 参数必须放到最后一个参数

```
def sum(x,*args,**keywords):
    print("x:%s,args:%s,keywords:%s"%(x,args,keywords))

sum(x=1,y=2)
sum(y=2,x=1,z=3)
sum(1,2,3,4,y=2)
```

需要注意的是,keywords 是一个字典类型的数据,当传递的所有的关键字参数使用其他参数没法接受的时候,我们就使用不定长关键字参数接受

4.7. 参数的解包

```
# 其中 y 在*参数后面,必须通过指定名称的方式进行赋值
# y 称为关键字参数
def sum(x,y,*args):
    print("x:%s,y:%s,args:%s"%(x,y,args))

sum(1,2)
```

```
#使用元组解包操作
sum(*(12,34,444,555))
```

```
def sum(x,y,**keywords):
    print("x:%s,y:%s,keywords:%s"%(x,y,keywords))
sum(**{'x':1,'y':100,'z':200})
```

5. 函数的应用:票务管理系统

```
"""
票务关系系统
每张车票拥有的信息:
    {"开始站点":"广州","目的站点":"北京","日期":"20190909","票价":800,"数量":100}
① 查询所有的车票信息
② 根据开始站点,目的站点,出行日期查询车票信息
③ 购买车票
④ 退票
"""

# 初始化车票信息
tickets =[
    {"开始站点":"广州","目的站点":"北京","日期":"20190909","票价":800,"数量":100},
    {"开始站点":"北京","目的站点":"广州","日期":"20190910","票价":800,"数量":100},
    {"开始站点":"广州","目的站点":"上海","日期":"20190909","票价":600,"数量":100},
    {"开始站点":"上海","目的站点":"广州","日期":"20190910","票价":600,"数量":100}
]

def print_menu():
    print("="*50)
    print("1 查询所有的车票信息")
    print("2 根据开始站点,目的站点,出行日期查询车票信息")
    print("3 购买车票")
    print("4 退票")
    print("5 退出系统")
    print("="*50)

# 打印菜单
print_menu()

# 查询所有的车票信息
def list_all():
    print("*"*50)
    for ticket in tickets:
```

```
            print(ticket)
        print("*"*50)
    # 2 根据开始站点,目的站点,出行日期查询车票信息
    def find_by_param(start,end,date):
        for ticket in tickets:
            if start ==ticket['开始站点'] and end ==ticket["目的站点"] and date == ticket["日期
"]:
                #返回找到的车票信息
                return ticket
        return None
    # 3 购买车票
    def buy_ticket(start,end,date):
            ticket = find_by_param(start,end,date)
            if ticket is None:
                print("没有找到对应的车票信息")
            elif ticket["数量"] ==0:
                print("车票不足:{}".format(ticket))
            else:
                print("购票成功")
                ticket["数量"] -= 1
    # 4 退票
    def return_tickte(start,end,date,price):
            ticket = find_by_param(start,end,date)
            if ticket is None:
                tickets.append({"开始站点":start,"目的站点":end,"日期":date,"票价":price,"数量
":1})
            else:
                ticket["数量"] += 1
            print("退票成功")
    while True:
        model = int(input("请输入你选择的操作序号"))
        if model == 1:
            list_all()
        elif model == 2:
            start = input("请输入开始站点")
            end = input("请输入目的站点")
            date = input("请输入操作日期")
            ticket = find_by_param(start,end,date)
            if ticket is None:
                print("没有找到对应的车票信息")
            else:
                print("车票信息:{}".format(ticket))
        elif model == 3:
            start = input("请输入开始站点")
```

```
        end = input("请输入目的站点")
        date = input("请输入操作日期")
        buy_ticket(start,end,date)
    elif model == 4:
        start = input("请输入开始站点")
        end = input("请输入目的站点")
        date = input("请输入操作日期")
        price = int(input("请输入票价"))
        return_tickte(start,end,date,price)
    elif model == 5:
        break
    else:
        print_menu()
        print("输入的操作有误,请重新输入")
```

6. 函数高级

6.1. 函数文档

函数文档主要用于增强文档的可读性,方便函数调用者知道函数的功能和相关的使用方式,以及参数的传递,返回值的类型

在函数的下面使用三重引号把说明文档包括起来

```
def sum(x,y):
    """
        求和函数
        x,y 必须是同一类型,可以都是整数,或者字符串,或者序列等
        return 如果是数字,则进行相加操作,如果是序列,则进行连接操作
    """
    return x + y
# 通过 help 用于查看文档的使用说明
print(help(sum))
```

6.2. 回调函数

把函数 A 作为函数 B 的参数,然后在函数 B 执行到某一个时机的时候调用函数 A,那么我们把函数 A 称之为回调函数,

```
def send_msg():
    """ 发送短信"""
    print("亲爱的,我到家了")

def go_home(callback):
    """回家"""
    print("打车回家")
```

```
    #调用函数
    callback()
    print("准备睡觉")

# 调用函数 go_home 在函数执行到某一个时机的时候调用函数发送短信
go_home(send_msg);
```

6.3. lambda 表达式

在 Python 中除了 def 可以声明一个函数以外,使用 lambda 表达式也是生成函数对象的表达式形式,对于一个比较简单的,在函数体中只有一小部分代码的时候,使用 lambda 表达式的时候更加清晰

语法格式:

fn = lambda [参数 1,参数 2,参数 3]:执行语句

在 lambda 表达式中可以有多个执行语句,多个执行语句使用分号隔开,如果有返回值,需要把返回值存放到

```
def sum(x,y):
    """ 求和计算"""
    return x + y
# 和上面的功能一样,但是更加清晰
sum = lambda x,y: x+y
```

6.4. 递归函数

所谓递归函数就是指在函数 A 执行到某一个时机的时候,自动的调用函数自己的以中操作方式,因为会涉及到自己调用自己的一种情况,所以需要避免死循环的出现,递归函数一定要有出口

通常情况下,对于递归函数,我们也可以使用循环来实现

```
# 定义一个函数,用于求 100 以内的整数和
def sum(x):
    if x>1:
        return x + sum(x-1)
    else:
        return 1
print(sum(100))
# 对于递归调用,通常情况可以使用循环来解决
```

6.5. 高阶函数

所谓的高阶函数,需要满足下面的两个条件之一

① 接受一个或者多个函数作为参数

② 输出一个函数

常用的高阶函数有 map,reduce,filter 等

```
# map(函数,序列)
ret = map(lambda x:x**2,[1,2,3,4,5])
print(list(ret))
def fn(x):
    return x+10
ret = map(fn,[1,2,3,4,5])
print(list(ret))
# reduce
from functools import reduce
ret = reduce((lambda x,y:x*y),[1,2,3,4])
print(ret)
print("*"*80)
# filter
# 找出所有大于 100 的数
ret = filter(lambda x:x >= 100,[100,1000,90,300,50])
print(list(ret))
```

6.6. 闭包

在函数 A 中定义一个函数 B,在函数 B 中访问函数 A 中的本地变量,并且把函数 B 作为返回值返回的操作

```
def outer():
""" 定义外部函数 """
    start = 0
    def inner():
    """ 定义一个闭包"""
        nonlocal start # 声明为非本地变量
        start += 1
        return start
    # 返回一个内部函数
    return inner
get_id =outer()
print(get_id())
print(get_id())
print(get_id())
print(get_id())
```