Lesson 5: 函数(一)

在Lesson 1中我们提到，我们关注的内容有两个大方向，一是数据，二是用于处理数据的函数。我们也提到封装这一概念，这要求我们尽可能不要暴露过于细致的代码。

函数的一大意义，就是把对数据的处理过程打包，每次需要用到同样流程时直接调用函数即可，起到代码复用和隐藏细节的作用

数学意义上的函数

数学意义上的函数强调集合之间的映射，并对这种映射的性质和集合本身的性质有一定要求。但通俗来讲，就是对每个输入都有固定的输出。

程序意义上的函数

程序意义上的函数一定程度上也包含了数学意义上的函数的含义。函数的参数可以被认为是自变量，函数的返回值可以被认为是因变量。

不同之处在于，程序意义上的函数可能会在执行过程中做一些其他的事情，比如修改外部变量或进行一些IO操作，你无法保证对同个输入总是有相同的返回值。

所以，对于部分函数而言，返回值可能仅仅起标注函数是的执行状态等信息的作用，甚至没有返回值(或说返回值为None)也是被允许的(比如print函数)

函数定义

在你使用你的函数前，你需要先定义你的函数，包括函数名，参数名，函数体，返回值等

- 最普遍的形式

```
# 判断是否为闰年



def is_leap_year(year):
        year = int(year) # 防止传入的是字符串而不是整数
        if year % 4 != 0:
                return False
        if year % 100 != 0:
```

```
                return True
        if year % 400 == 0:
                return True
        else:
                return False
```

- 带类型注解的形式

```
# 两个列表逐元素做加法

# 对于这种参数名没有实际意义的情况，务必做好类型标注来降低他人的理解成本

def add(left: list, right: list)->list:
# 先针对能想到的错误输入情况做处理，避免程序直接崩溃
        if len(left) != len(right):
        print('所有参数长度应相等')
        return
        '''
        用zip生成[(l1, r1),(l2, r2),...]的列表
        再解析for循环得到的元组获得l, r，并据此获得新的列表
        '''
```

```
        return [l + r for l, r in zip(left, right)]
```

类型注解有两个优点，一是便于代码调用者与阅读者理解接口，二是便于代码编写者在代码内部获得编辑器的方法提示，比如你标注了一个变量为list，编译器就会在你输入p时为你匹配pop函数并显示应该填写的参数，这为我们提供了很多方便。

然而，需要注意的是，**类型注解并无强制约束力**，传入不符合标注类型的参数时并不会报错

- 带默认参数的形式

```
def pow(x: list, n: int = 2)->list:
        return [i ** n for i in x]
```

这里有两个注意点，一是默认参数**必须放在必选参数后**，二是默认参数**必须是不可变对象**，否则它在多次调用过程中可能会不一致。

默认参数在实际编程中极为常用，因为它提供了一种便利性和灵活性的平衡，对于普通使用者只需要提供少量非默认参数即可调用函数，对于想调整更多细节的使用者又允许他们修改默认参数来提高灵活性

- 带可变参数的形式

```
def calculate(mode='mean', *args):
        if not mode in ['mean', 'sum']:
                print('暂不支持该模式')
                return
        if len(args) == 0:
                print('请传入参数')
                return
```

```
        sum = 0
        for arg in args:
                sum += arg

        if mode == 'mean':
                sum /= len(args)

        return sum
```

*args表示将传入的多个参数接受并打包为一个list赋值给args，在函数内部只需要把args当列表处理即可。

args并非固定的名称，*nums之类的变量名也是可以的，仅仅是一种习惯。可变参数必须在默认参数后

可变参数的出现主要是为了方便传入多个参数时必须先打包成list的麻烦，如果你先定义了可变参数又需要传入一个list或tuple，只需要在变量名前加*即可

```
calculate(mode='sum', 1, 2, 3)

nums = [1, 2, 3]

calculate(mode='sum', *nums)
```

相比默认参数，可变参数比较少用

- 带关键字参数的形式

可变参数从参数数量上拓展了传参的自由度，但这些参数只能以匿名的形式传递。若你想传递不定量的变量且想给这些变量命名，可以使用关键字参数，它会将传入的变量名与值打包为一个字典

```
def show_message(name, age, **kw):
        print(f'name:{name}\nage:{age}')
        for key, value in kw.items():
                print(f'{key}:{value}')


message('Bob', 19, city = 'Shanghai', job = 'student')
```

在我们定义了关键字参数后，有时可能仍需要传一个字典作为参数，我们可以采用与可变参数类似的方法

```
d = {'city': 'Shanghai', 'job': 'student'}

message('Bob', 19, **d)
```

补充知识: 变量作用域

变量自被定义开始(在python中也就是第一次被赋值)，并不总能存活到全部程序执行完毕，也并不总是能被所有位置的代码访问

我们主要考虑以下两个原则:

1. 函数内部定义的变量无法被外部访问
2. if语句块中定义的变量无法被外部访问，if语句结束后该变量不再存在

参数传递

- 传递的参数是可变对象时，可以在函数内对被传递的参数进行修改
- 传递的参数是不可变对象时，在函数内对该参数变量进行重新构造时(如通过重新赋值来修改)，不会影响原参数
- 函数内部创建的变量与外部变量重名时，内部变量优先级更高

函数调用

- 函数定义与调用在同一文件:

直接通过函数名调用即可

- 函数定义与函数调用在不同文件:

法一

```
import xxx(函数定义所在文件)



...

xxx.函数名(参数)
```

法二

```
from xxx import 函数名1, 函数名2



...

函数名1(参数)

函数名2(参数)
```