Python 程序设计基础

江西财经大学-朱文强

第五章

函 数

函数

函数有时候也称为方法。对于函数，实际上我们并不陌生，在前面一些章节的学习中，我们已经多次使用到函数。

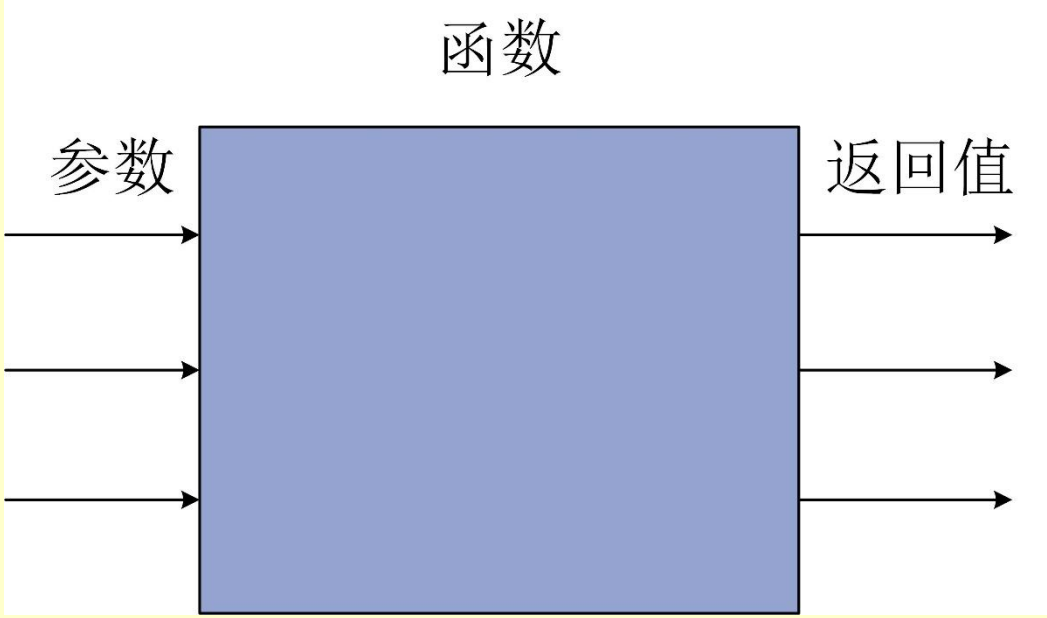
所谓函数，实际上就是具有特定功能的程序代码块。通过函数，我们可以快速调用某些功能，从而不需要从无要有，或者从零开始构建程序。**借助于函数，我们可以很方便地在前人的基础上进行功能的扩展，从而大大提升工作效率**。

函数是编程语言学习过程中非常重要的内容。

1 函数的定义与调用

1.1 函数的概念

函数，是指具有特定功能的代码块，主要是为了代码的重复使用。从使用者角度看，函数就像一个"黑盒子"，只要知道传递了什么参数，能得到什么结果。



使用者角度的函数调用流程图

1.1 函数的概念

函数设计人员在**设计函数时，一般需要考虑以下几个问题**:

(1)函数中哪些部分是动态变化的，即哪些部分应该被定义为函数参数; # 结合实验报告1的代码

(2)函数要实现什么功能，函数最终给使用者返回什么结果;

(3)函数如何实现这些功能，即函数体。

1.2 函数定义及调用

Python定义函数的语法如下:

```
def 函数名([形参列表]):
    '''函数说明文档'''
    函数体
    [return [返回值]]
```

- def: 用来定义函数的关键词;
- 函数名: 函数名应见名知意，**由小写字母组成，多个单词间用下划线隔开**;

1.2 函数定义及调用

- 形参列表：函数可包含0个或多个参数，多个参数用逗号隔开，**即使没有参数，函数名后面的括号也不能少，及括号后面的冒号也不能省略**;
- 函数名之后，应该撰写函数注释，说明函数的作用及调用方式，**函数注释用三双引号或三单引号括起，用于生成函数的说明文档;**
- 函数体，冒号后缩进的由一条或多条语句组成的代码块；
- return语句返回函数结果，可选。函数可以有返回值，也可以没有返回值，还可以有多个返回值，多个返回值以元组形式返回。

1.2 函数定义及调用

通过使用函数, 程序的阅读和测试也将变得更加容易。

```
def  greet_user(username):   # 无返回值
    “””显示简单的问候语”””
    print(“Hello, ” , username.title() + “!”)
greet_user(‘black’)
```

1.2 函数定义及调用

```
def is_leap_year(year):        # 判断一个年份是否为闰年
    if (year % 4 == 0 and year % 100 !=0) or (year % 400 == 0):
        return True        # 函数返回值, 也可直接返回’is’ or ‘is not’
    else:
        return False
year = int(input("Please input year:"))
yes_or_not = "is" if isleapyear(year) else "is not"
print("The {} {} a leap year.".format(year, yes_or_not))
```

1.2 函数定义及调用

函数调用的方式是：函数名(实参列表)，**实参列表中的参数个数要与形参个数相同，参数类型也要一致，否则会抛出TypeError错误**。

函数的调用一定要放在函数定义之后，否则解释器将找不到函数，会抛出NameError错误。**当存在多个同名函数时，调用的是最近一次定义的函数**。

1.2 函数定义及调用

根据函数是否有返回值, 调用函数有两种方式:

(1)**带有返回值的函数调用。通常将函数的调用结果作为一个值处理**。

(2)**没有返回值的函数调用。通常将函数调用作为一条语句来处理**。如果函数没有指定的返回值，那么它将返回None。

注意：不论return语句出现在函数体的什么位置，一旦得到执行, 将直接结束函数的运行。

2 参数类型与参数传递

2.1 形参与实参

在介绍函数参数传递前，要先明确两个概念，即形参和实参。**形参是在函数定义时，写在圆括号里面的变量**。形参不代表任何具体的值，只是作为一种占位符参与函数体的业务逻辑。而**实参是在函数调用时，实际传递给形参的值**。

在Python中，定义函数时不需要指定形参的类型，形参的类型由函数调用时，传递的实参类型决定。实参与形参通常数量相同，当形参是可变长度参数时，实参可以有多个，此时实参和形参的数量可以不相同。

2.1 形参与实参

根据函数参数传递的特点和形式，可大致将**函数参数分为**：位置参数、关键字参数、默认值参数、可变长度参数、序列解包参数等等。

2.2 位置参数

位置参数是指函数调用时，**根据函数定义时形参的位置顺序依次将实参的值赋值给形参。要求实参和形参的个数必须一致，而且实参和形参必须一一对应**。位置参数是函数调用最常见、最简单的参数传递方式。

2.2 位置参数

```
def  describe_pet(pet_type, pet_name):
    “””显示宠物的信息”””
    print(“\nI have a ” + pet_type + “ .”)
    print(“\nMy ” + pet_type + “’s name is ” + pet_name.title() + “ .”)
describe_pet(‘hamster’, ‘harry’)
```

2.3 关键字参数

关键字参数是指函数调用时，以"形参名=实参值"的形式，指定形参的实参值。此时**形参出现的顺序，可以和函数定义时不一致**。

关键字参数灵活、方便，调用者不用关注形参定义的顺序和位置。

2.3 关键字参数

```
def  describe_pet(animal_type, pet_name):
    print(“\nI have a ” + animal_type + “ .”)
    print(“\nMy ” + animal_type + “’s name is ” + pet_name.title() + “ .”)
describe_pet(pet_name=‘harry’, animal_type=‘hamster’)
```

2.4 默认值参数

默认值参数是指函数定义时，在形参列表中，直接为形参指定默认值。函数调用时，对于有默认值的参数，可传值也可不传值。未传值时，将采用默认值；传值时，将用新的值替换默认值。默认值参数方便函数调用，可减少参数传递数量。

注意：定义带有默认值参数的函数时，默认值参数必须出现在函数形参列表的最右端，任何一个默认值参数右边都不能再出现非默认值参数。

2.4 参数默认值

```
def  describe_pet(pet_name, animal_type='dog'):
    print("\nI have a " + animal_type + " .")
    print("\nMy " + animal_type + "'s name is " + pet_name.title() + ".")
describe_pet(pet_name='harry')
```

2.4 参数默认值

对于函数的参数默认值，**可使用'函数名.__defaults__'查看函数所有参数的默认值**，其返回值为一个元组，其中的元素依次表示每个参数的默认值。

```
>>>def say(message, times=10):
        print((message+" ")*times)
>>>say.__defaults__     # (10,)
```

2.5 可变长度参数

可变长度参数是指函数定义时，无法确定参数的个数。此时可将函数的形参设为可变长度参数，根据调用者传递的实际参数数量来确定参数的长度。

可变长度参数有两种形式：*形参名和形参名。**

***形参名：表示该形参是一个元组类型，可接受多个实参，并将传递的实参依次存放到元组中，主要针对以位置传值的实参。**

2.5 可变长度参数

```
def make_pizza(*toppings):
    print("\nMaking a pizza with the following toppings:")
    for topping in toppings:
        print("-" + topping)
make_pizza('pepper')
make_pizza('rushroom', 'pepper', 'raddish')
```

2.5 可变长度参数

在有多种类型参数的情况下，**Python解释器会先匹配关键字实参或位置实参，再匹配任意数量实参，所以必须将任意数量实参放在函数定义的形参列表最右边**。

```
def make_pizza(size, *toppings):
    print("\nMaking a" + str(size) + "-inch pizza with the following toppings:")
    ……

make_pizza(12, 'pepper')
make_pizza(16, 'mushroom', 'pepper', 'extra cheese')
```

2.5 可变长度参数

**形参名：表示该形参是一个字典类型，可接受多个实参，主要针对以关键字传值的实参，并将传递的键值对保存到字典中。

```
def fruit_info(name, grade, **other_info):
    profile = {}
    profile["name"] = name
    profile["grade"] = grade
    for key, value in other_info.items():
```

2.5 可变长度参数

```
    profile[key] = value
return profile
fruit = fruit_info("apple", "A++", location=
"shandong", color="red")
print(fruit)
```

注意：这里的形参other_info中的两个星号让Python创建了一个名为other_info的空字典，并将收到的所有键值对都封装在这个字典中。**

2.6 多种参数混用(进阶)

Python支持定义函数时几种不同形式的参数混用，但**初学者要谨慎使用**，因为使用不当将导致代码混乱且降低可读性，使得程序查错困难。

多种类型参数混用注意事项:

(1)实参传值时，既可通过位置传值，也可通过关键字传值，但可变参数后面的参数只能通过关键字传值，一般建议可变参数放在形参的最后;

2.6 多种参数混用(进阶)

(2)函数定义时，不能同时包含多个相同类型的可变参数，即多个*参数或多个**参数，但可同时包含一个*参数和一个**参数，且*参数要放在**参数前面；

(3)当既有可变参数又有普通参数时，会先给普通参数赋值，最后将多余的值存放在可变参数中，可变参数可以不进行赋值，此时可变参数为空；

(4)带有默认值的参数后面不能包含没有默认值的参数，但可以包含可变参数。

2.6 多种参数混用(进阶)

```
>>>def func4(a, b, *d, c=4, **e):
        print((a, b, c))    # (1, 2, 3)
        print(d)      # (4, 5, 6)
        print(e)     # {'x': '1', 'y': '2', 'z': '3’}
# c可赋值, 也可不赋值
>>>func4(1,2,3,4,5, c= 4, x=1, y=2, c=3)  # 不能使用a, b
```

2.7 参数传递的序列解包(进阶)

参数传递序列解包：实参为序列对象，传值时将序列中的元素依次取出，然后按照一定规则赋值给相应变量。主要有两种形式：*序列对象、**字典对象。

为多个形参传递参数时，可使用列表、元组、字典等可迭代对象作为实参，并**在实参前加上'*' 或"**"号**，Python解释器将对其自动解包，但**务必保证形参和实参数量相等**。如果需要将字典的键或值作为实参，则需使用keys()或values()方法。

2.7 参数传递的序列解包(进阶)

```
>>>def demo(a, b, c):
        print(f"a = {a}, b = {b}, c = {c}")
>>>seq = [1, 2, 3]
>>>demo(*seq)   # 6
```

2.7 参数传递的序列解包(进阶)

如果使用字典作为函数实参，并在前**加上’**’号进行解包**时，会把字典解包成关键字参数进行传递，**字典的键作为参数名，字典值作为参数值**。

```
>>>def demo(a, b, c):
        print(f"a = {a}, b = {b}, c = {c}")
#   字典键与参数名要一致
>>> d1 = {'a':2, 'b':3, 'c':4}
>>> demo(**d1)  # 9
```

2.7 参数传递的序列解包(进阶)

当实参为序列对象，但**没有"*"时，函数传值会将其看成一个整体赋值**给某个形参。

```
def demo2(a, *b):
    print(f"a = {a}")
    print(f"b = {b}")


demo2([20, 15, 30], 40, 50)  # 实参为序列对象
demo2(*[20, 15, 30], 40, 50)
```

2.7 参数传递的序列解包(进阶)

当实参为**字典对象，不加"**"时，函数传值会将其看成一个整体**赋值给某个形参。

```
def demo3(a, **b):
    print(f"a = {a}")
    print(f"b = {b}")
demo3({"a": 5, "b": 0}, c=20, d=30)  # 实参为字典对象
demo3(**{"a": 5, "b": 0}, c=20, d=30)  # 实参为**字典对象
```

2.8 参数传递对实参变量的影响

在函数参数传递时，根据实参对象是否可变，可将实参分为可变类型和不可变类型。**对于不可变类型实参**，例如整型、字符串、元组、浮点型等，函数调用时传递的只是实参的值，相当于是将实参的值复制一份给形参，**函数内部对形参的修改，不会影响到实参**。

对于可变类型实参，例如列表、字典、集合等，函数调用时，传递的是实参所指对象的引用，此时形参和实参指向同一对象，**函数内部对形参的修改可能会影响到实参(**这让你可以高效的处理大量的数据**)**。

2.8 参数传递对实参变量的影响

```
def double1(x):   # 调用这两个函数演示
    x = x * 2
    print("函数内部的值：", x)


def double2(x):    # 对列表的修改
    for i in range(len(x)):
        x[i] = x[i] * 2
    print("函数内部的值：", x)
```

2.8 参数传递对实参变量的影响

有时候，我们需要保留原列表的内容，禁止对原列表进行修改。**为了解决这个问题，可向函数传递列表的副本，而不是原件**。

要将列表的副本传递给函数, 可以像下面这样做:

function_name(list_name[:])

在处理大型列表时，除非有必要保留副本，否则还是应该将原始列表传递给函数，以避免花时间和内存创建副本，从而提高程序运行效率。

3 变量的作用域与递归

3.1 变量作用域

根据变量定义的位置，可将**变量分为全局变量和局部变量。全局变量是指定义在函数外面的变量，可以在多个函数中进行访问**，但不能在函数中执行赋值操作。**如果在函数中有赋值语句，相当于创建了一个同名的局部变量。局部变量是指定义在函数内部的变量，只能在它被定义的函数中使用**，在函数外面无法直接访问。

注意：当局部变量和全局变量同名时，在函数内部使用的变量，通常是局部变量，如果确实需要对全局变量进行修改，需要**使用global关键字对变量进行声明，此时操作的就是全局变量**。

3.1 变量作用域

```
def fun(x):   # x不能为全局变量
    global a   # 局部变量的处理比全局变量略快
    print('a =', a)
    a = 5
    return a + x    # 可能的情况下, 应优先使用局部变量
a = 2
print(fun(3))
print('a =', a)
```

3.2 递归函数

递归函数是指在设计函数结构时，又直接或间接调用该函数本身。这常用来解决结构相似的问题。所谓结构相似，是指构成原问题的子问题与原问题在结构上相似，可以用类似的方法求解。整个问题的求解可分为两部分：第一部分是一些特殊情况，有直接的解法；第二部分与原问题相似，但比原问题的规模小，并且依赖于第一部分的结果。

递归函数通常包含两部分：基线条件(针对最小的问题)，满足这种条件时函数将直接返回一个值，这也是递归的出口；**递归条件，包含自身的调用，旨在解决函数的一部分**，即大问题是如何分解为小问题的。

3.2 递归函数

```
def factorial(n):    #阶乘
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)  # n * (n-1)!
print(factorial(10))
```

3.2 递归函数

```
def power(x, n):  #幂
    if n == 0:
        return 1
    else:
        return x * power(x, n-1)
print(power(2, 3))
```

```
def pw(a, n):
    t = 1
    for i in range(1, n+1):
        t = t * a
    return t
```

一般来讲，使用递归完成的任务，都可以使用循环来完成，而且使用循环的效率会普遍更高。但是，很多情况下，递归会让程序的可读性更好些。

3.2 递归函数

为了防止递归函数陷入无限制的递归状态，消耗系统的资源，**Python解释器设置了默认的递归深度**，超过这个深度，解释器将终止程序的运行，当然，**也可以通过代码来人为设置递归深度**。

```
>>> import sys
>>> sys.setrecursionlimit(200)      #设置递归深度为200层
```

4 特殊函数

Python中有些**特殊的函数，功能强大，可极大提高程序运行效率或减少程序员的开发时间**，在编写程序代码时经常会用到。

常用的特殊函数有map函数、 lambda匿名函数、 filter函数、eval函数、 callable函数等。以下将依次介绍它们。

4.1 map函数

map函数是Python的内置函数，**用于多次调用某一函数，并将可迭代对象中的元素作为实参传入，最终返回结果为函数运行结果的迭代器(1个map对象)**。方法声明为:

map(func, iterables)

其中，func参数表示需调用的函数，可以是已定义好的函数名，也可以是lambda表达式；iterables参数表示可迭代对象，即每次调用时传递的实参，如果函数需要传递多个参数，此时需要多个可迭代对象。

4.1 map函数

```
f = lambda x, y: x + y

a = list(map(f, [1, 2, 3], [3, 2, 1]))   # 实现列表参数映射

print(a)

def proc1(x, y):

    return x * y

c = map(proc1, [1, 2, 3], [3, 2, 1])   # 实现列表参数映射
```

```
In [23]: b = ['ab', 'abc', 'abcd', 'abcde']

In [24]: d = map(len, b)

In [25]: print(list(d))
```

4.1 map函数

```
>>> import random  # 随机生成5个可重复元素的列表
>>> num = random.choices(range(1, 10), k=5)
>>> num
[5, 4, 2, 9, 1]    # 生成这5个数字所能组成的最大数
>>> int(''.join(sorted(map(str, num), reverse=True)))
95421
```

4.1 map函数

```
import math    # 求三角形的第三边长

x = input("请输入两条边的长度及夹角:")

a, b, theta = map(float, x.split())

c = math.sqrt(a**2+b**2-2*a*b*math.cos(theta*math.pi/180))

print('c =', c)
```

4.2 lambda函数

Python使用lambda表达式创建匿名函数，即没有函数名称的、临时使用的函数。可以将lambda函数看成是函数的简写形式。lambda函数的**语法如下:**

lambda 参数列表: 表达式

lambda函数与函数的区别: ①关键字不同，函数使用的是def定义，lambda函数使用的是lambda；②函数有函数名，而lambda函数没有名称；③函数参数列表和lambda函数参数列表的含义完全一样，但是lambda函数的参数列表不需要一对圆括号，而且lambda函数体只能包含一个表达式语句，不能包含多条语句(if, while, for语句也不能有)。

4.2 lambda函数

在lambda函数中可以调用其它函数，并支持默认值参数、关键字参数、可变长度参数等等，**表达式的结果相当于函数的返回值**。此外，可以直接把lambda定义的函数赋值给一个变量，用变量名来表示lambda表达式所创建的匿名函数，这样就可以多次使用该函数。

注意：所有通过lambda函数实现的功能，都可以通过相应的普通函数形式实现，反之则不一定。

4.2 lambda函数

```
>>> a, b = [2, 3, 4, 5], [6, 7, 8, 9]

>>> list(map(lambda x,y: x*y, a, b))

>>>strs = ['radish', 'foo', 'card', 'abea', 'dish', 'bathe', 'hello']

>>>strs.sort(key=lambda x: len(set(x)))

>>>strs   # ['foo', 'bar', 'abea', 'card', 'dish’]

# 根据列表元素字符串的不同字母数量进行排序

>>>ls = [(lambda x: x**2), (lambda x: x**3), (lambda x: x**4)]

>>>print(ls[0](2), ls[1](2), ls[2](2))     # 4 8 16

# ls = [ls[0](2),ls[1](3), ls[2](4)]
```

4.3 callable()函数

一般而言，**要判断某个函数、方法是否可调用，可使用内置函数callable()**。

```
>>> import math
>>> x = 1
>>> y = math.sqrt
>>> callable(x)
False
>>> callable(y)
True
```

4.4 exec()与eval()函数

函数exec用于将字符串作为代码执行。

>>>exec(“print(‘Hello, world!’)”)

Hello, world!

4.4 exec()与eval()函数

eval()是一个类似于exec()的函数。exec()执行一系列的Python语句, 而**eval()计算用字符串表示的表达式的值, 并返回结果(exec()什么都不返回, 因为它本身是条语句)**。

```
>>> eval("4+2")   # 6

>>> a = input('Input:')   # 启动记事本程序

# Input:__import__('os').startfile(r'c:\windows\notepad.exe')

>>> eval(a)
```

注意: eval()和exec()都是不安全的函数, 一种替代解决方案是使用Jpython, 以使用Java沙箱等原生机制

4.4 exec()与eval()函数

```
>>> a, b = eval(input("请输入两个数, 用逗号隔开:"))
请输入两个数, 用逗号隔开:1, 2.5
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float’>
```

然而，**调用函数exec很容易带来本地安全风险**。因此，**应该向它传递一个命名空间(类似于一个字典)，用于放置exec函数所产生的变量, 以免和本地同名变量产生冲突**。

4.4 exec()与eval()函数

```
>>> from math import sqrt
>>> exec('sqrt=1')
>>> sqrt(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable。
>>> sqrt
1
```

4.4 exec()与eval()函数

```
>>> from math import sqrt
>>> scope = {}
>>> exec('sqrt=1', scope)
>>> sqrt(4)
2.0
>>> scope['sqrt']
1
```

4.4 exec()与eval()函数

向exec或eval提供命名空间时, 可在这个命名空间中添加一些变量值。

```
>>> scope = {}
>>> scope['x'] = 2
>>> scope['y'] = 3
>>> eval("x*y", scope)
6
```

4.4 exec()与eval()函数

同样, 同一命名空间可用于多次调用exec()或eval()函数。

```
>>> scope = {}
>>> exec(‘x=2’, scope)  #等价于 scope['x'] = 2
>>> eval('x*x', scope)
4
```

4.5 filter()函数(进阶)

filter函数也是Python的内置函数，**用于过滤可迭代对象中的元素，只保留使得函数调用结果为True或结果可转化为True的元素，最终结果为符合要求的元素组成的迭代器，1个filter对象**。方法声明为:

filter(func, iterable)

其中，func参数表示需调用的函数，可以是已定义好的函数名,也可以是lambda表达式，函数返回值通常为True或False; iterable参数表示可迭代对象，用于每次调用时传递实参。

4.5 filter()函数(进阶)

```
>>>words = ['', 'a', 'b', '', []]
>>>result = filter(len, words)          # 对列表空元素过滤
>>>print(list(result))
>>>seq = ['foo', 'x42', '?!', '00*']
>>>def func(x):
        return x.isalnum()
>>>list(filter(func, seq))    # ['foo', 'x42']
>>>list(filter(str.isalnum, seq))   #  ['foo', 'x42']
```

4.6 reduce()函数(进阶)

标准库函数reduce()可将一个接受2个参数的函数以累积的方式从左到右依次作用到一个序列或迭代器对象的所有元素上。即**先对序列中的第 1、2 个元素进行操作, 得到结果再与第3个数据运算, 最后返回一个汇总结果**。

```
>>> from functools import reduce

>>> reduce(lambda x,y:x*y, range(1, 11))  # 3628800
```

4.6 reduce()函数(进阶)

```
>>>def add(x, y):
        return x + y
>>>reduce(add, map(str, range(10)))
#  '0123456789'
```

5 函数的导入与函数编写指南

5.1 函数的导入

函数的优点之一，是使用它们可将函数代码块和主程序分离。通过**将函数存储在被称为模块的独立文件中，然后就可以将模块中的函数导入到主程序中**。

要让函数是可导入的，得先创建模块。**模块是扩展名为.py的文件**，用以包含要导入的函数和类，模块的设计使众多不同的程序可以重用函数。

使用import语句可以在当前运行的程序文件中导入模块的函数。

5.1 函数的导入

导入整个模块: import module

使用函数: module.function()

```
import make_pizza    # make_pizza为模块名称

make_pizza.make_pizza(16, 'pepper')

make_pizza.make_pizza(12, 'mushroom', 'green peppers',
'extra cheese')
```

注意: import导入模块时, 源文件名称中不能有空格或.号

5.1 函数的导入

导入特定函数: from module import function0, function1, function2

from make_pizza import make_pizza

第一个make_pizza为模块名称, 第二个为函数名称

make_pizza(16, ‘pepper’)

make_pizza(12, ‘mushroom’, ‘green peppers’, ‘extra cheese’)

附：import sys

sys.path.append('D:\Python\BasicPython\chapter 1’)

sys.path.append(os.path.realpath(os.path.dirname(os.path.realpath(‘*.py’))))

5.1 函数的导入

有时候为了方便或避免命名冲突, 可对导入的模块或函数指定别名。

指定模块别名: import module as p

指定函数别名: from module import function1 as mp1, function2 as mp2

设置函数别名(举例): foo = math.sqrt (在import math后)

5.1 函数的导入

有时候为了提高函数导入效率，可使用*号来导入模块中的所有函数。

导入所有函数: from module import *

一般来讲，**不建议使用这种方式导入函数，它可能会导致不同模块的函数名冲突，而产生无法预知的结果。最佳的做法是，只导入你需要使用的函数内容。**

5.2 函数编写指南

1、应给函数指定描述性名称, 且只使用小写字母和下划线;

2、每个函数应包含简要描述其功能的注释，注释应紧跟在函数定义后面, 并采用文档字符串格式;

3、给形参指定默认值时，等号两边不要有空格；对于函数调用的关键字实参, 也应遵守这种规定;

4、如果程序包含多个函数, 应使用一个空行将其分开;

5、所有的import语句, 都应放在文件开头;

5.2 函数编写指南

6、如果函数形参过多，导致函数定义超过了80个字符，这时可**在函数定义输入左括号后按回车键，并在下一行空8个字符，从而将形参和只缩进4个字符的函数体区分开来**。

```
def function_name(
        parameter_0, parameter_1,….
        parameter_n-1, parameter_n)
    function_body……
```

练习

01 编写教材上的例题和练习

02 完成课后编程题3、4、5

谢 谢