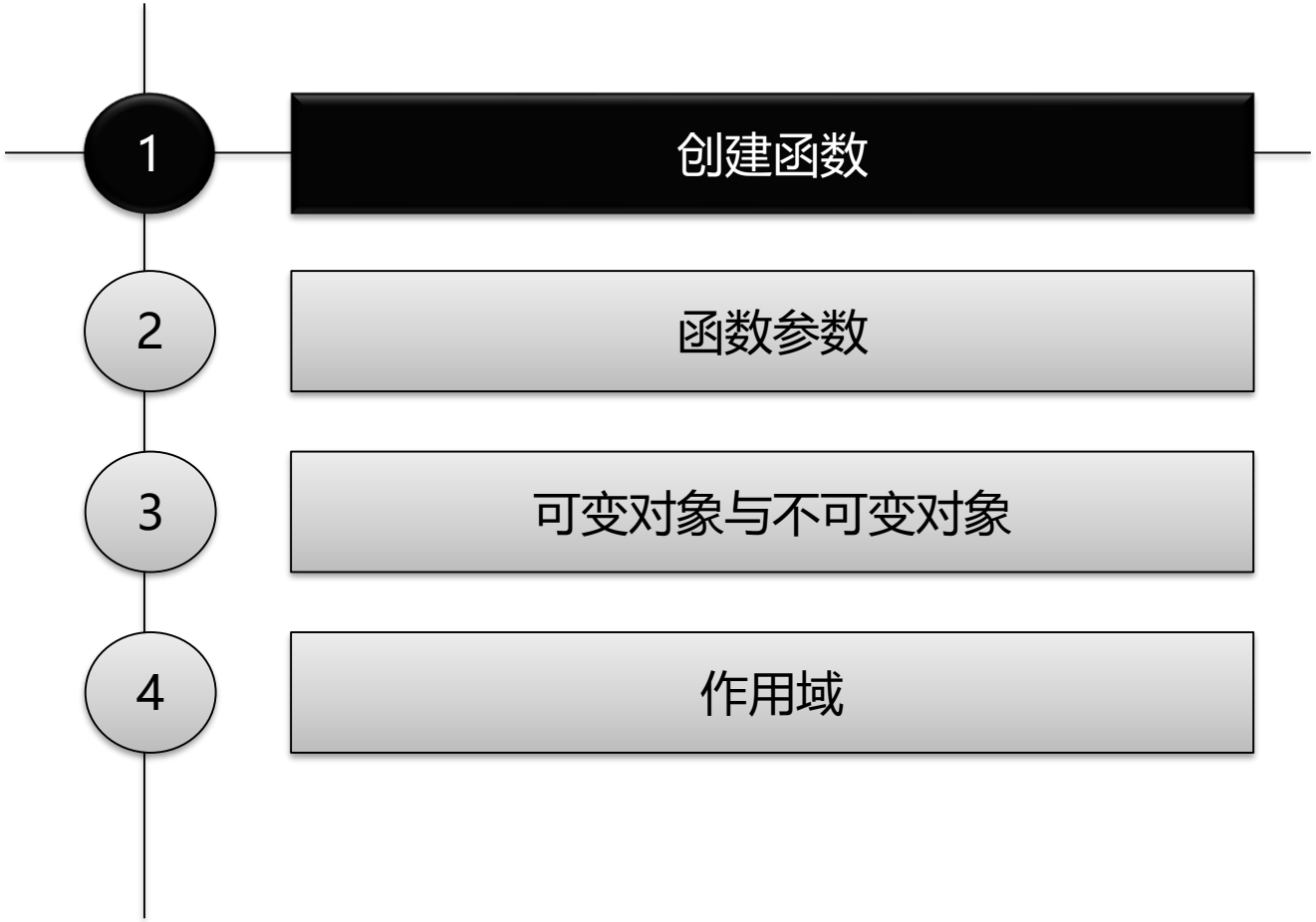
《Python与数据挖掘》

第3章 函数

讲师：武永亮

目录



函数概述

- 函数是Python为了代码效率的最大化，减少冗余而提供的最基本的程序结构。在上一章中，我们学会了众多流程控制的语句，在中大型的程序中，同一段代码可能会被使用多次，如果程序由一段又一段冗余的流程控制语句组成，那么程序的可读性会变差。
- 所以，我们需要使用函数去封装这些重复使用的程序段，并加以注释，下次使用的时候就直接调用即可，使得代码清晰明白。
- 在这里第一次讲到函数封装的概念，实际上我们在前面已经接触到了。如果不封装成函数，每次添加元素都要输入这段代码，这显得非常繁琐。

函数概述

- 函数能使程序变得抽象。抽象节省了工作，并且加大了程序的可读性。例如写一个求一列数据的极差的程序，我们可以分解成如下工作：
  - 1 求最大值
  - 2 求最小值
  - 3 求极差，极差=最大值-最小值
- 在第一和第二步中，我们编写函数max()和函数min()，然后第三步直接调用函数求极差即可。虽然这样做的速度不是最快的，但我们使得程序变得抽象，如果不知道极差的概念，但看到如下的代码：range = max(list1) – min(list1) ,相信你们已经明白程序的输入和输出是什么了。

def语句

- 我们可以用def语句创建函数，格式为：def fun_name(par1,pa2,...):
  由def关键字，函数名和参数表组成。先举一个简单的例子：
  - def fun():
  - print ‘hello,world’
- 这样就定义了一个fun函数，它没有参数，也没有返回值，仅仅打印出hello,world。下面再定义一个有参数也有返回值的函数：
  - def hello(your_name):# your_name表示你的名字，格式是字符串
  - return ‘Hello ’ +your_name
- 这个函数称为hello, 输入参数是your_name, 返回加上hello的字符串。
  程序创建函数后，执行 s = hello(‘Tom’) 即得到一个新的字符串
  ’ Hello Tom’ 并赋值给s。

def语句

- Python的简洁性可以从函数中体现，Python的参数也不需要声明数据类型，但这也有一定的弊端，程序员可能会因不清楚参数的数据类型而输入错误的参数。
- return语句用于返回一个结果对象。Python可以没有返回值，可以有一个返回值，也可以有多个返回值，返回值的数据类型没有限制。
- 当程序执行到函数中的return语句时，就会将指定的值返回并结束函数，如果return后面还有语句，那些语句将不会被执行。
- 举一个Python有多个返回值函数的例子：
  - Def maxmin(a,b) # a,b为两个数值数据，返回它们从大到小排列的结果
  - 	if a>b:
  - 		return a,b
  - 	else:
  - 		return b,a
  - 执行big,small = maxmin(2,4) 后，big=4，small=2。

lambda语句

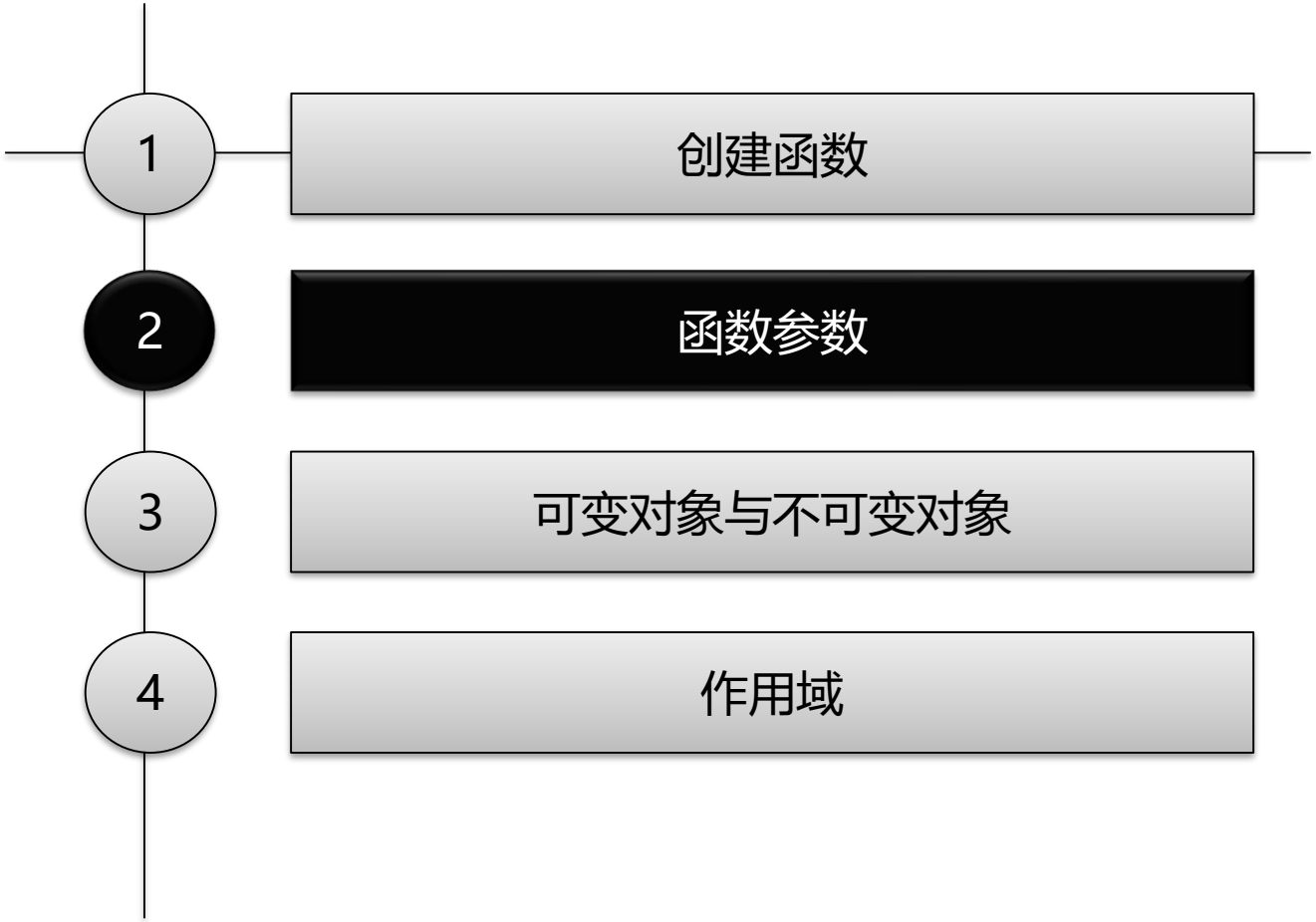
- Python允许使用lambda语句创建匿名函数，也就是说函数没有具体的名称。可能会产生疑惑，函数没有了名称应该不会是一件好事。
- 实际上，使用Python编写一些执行脚本时，使用lambda省去了定义函数的过程，代码变得精简。对于一些抽象的，不会在其他地方复用的函数，有时候给函数命名也是个难题（需要避免函数重名），使用lambda不需要考虑函数命名的问题。
- lamber语句中，冒号前是函数参数，若有多个函数使用逗号分隔，冒号右边是返回值。如此便构建了一个函数对象，def语句也是创建一个函数对像，只是lambda创建的函数对象没有名字。

lambda语句

- >>>g = lambda x : x+1
- >>>print g
- <function <lambda> at 0x030EAEF0>
- >>>g(1)
- 2

- 使用lamber函数应该注意下面几点：
  - lambda定义的是单行函数，如果需要复杂的函数，应使用def语句
  - lamdda参数列表可以包含多个函数，如lambda x , y : x + y
  - lambda语句有且只有一个返回值
  - lambda语句中的表达式不能含有命令，而且仅限一条表达式

目录



函数参数

- Python中的函数参数主要有3种形式，分别是：
  - 位置或关键字参数
  - 任意数量的位置参数
  - 任意数量的关键字参数
- 我们在阅读函数时，需要注意函数的参数列表，没有带默认值的参数需要我们往函数传递值，而无带默认值的参数可以不传递值。

位置或关键字参数

- 这种参数是Python默认的参数类型，函数的参数定义为该种参数后，可以通过位置参数，或者关键字参数的形式传递参数。
- 例如：
  - def fun2(a,b,c):
  - print a,b,c
  - # 可以使用位置参数
  - >>>fun2(1,2,3) # 输出1,2,3
  - #可以使用关键字参数，关键字参数间的顺序没有关系
  - >>>fun2(a=1,c=3,b=2) # 输出1,2,3
  - #也可以混合使用位置参数和关键字参数，但位置参数必须在关
  - 键字参数的前面
  - >>>fun2(1,c=3,b=2) # 输出1,2,3
  - >>>func(a=1,2,3) # 报错

位置或关键字参数

- 函数参数列表中可以定义默认参数，但Python同样不允许带默认值的参数定义在没有默认值的参数之前，因为这样写是有歧义的。
- 假设允许定义：
  - def fun3(a=1,b):
  - print a,b
- 那么我调用 fun3(2)，虽然程序员希望a=1,b=2但Python的位置参数是按顺序赋值的，程序会先把2赋值给a，但已经没有参数赋值给b了，所以程序会报错。
- 如果改成：
  - def fun3(a,b=2):
  - print a,b
- 调用fun3(1)时，按照顺序，先将1赋值给a，虽然后面没有参数传入，但b已经有默认值，因此这样写程序没有歧义，输出1,2。

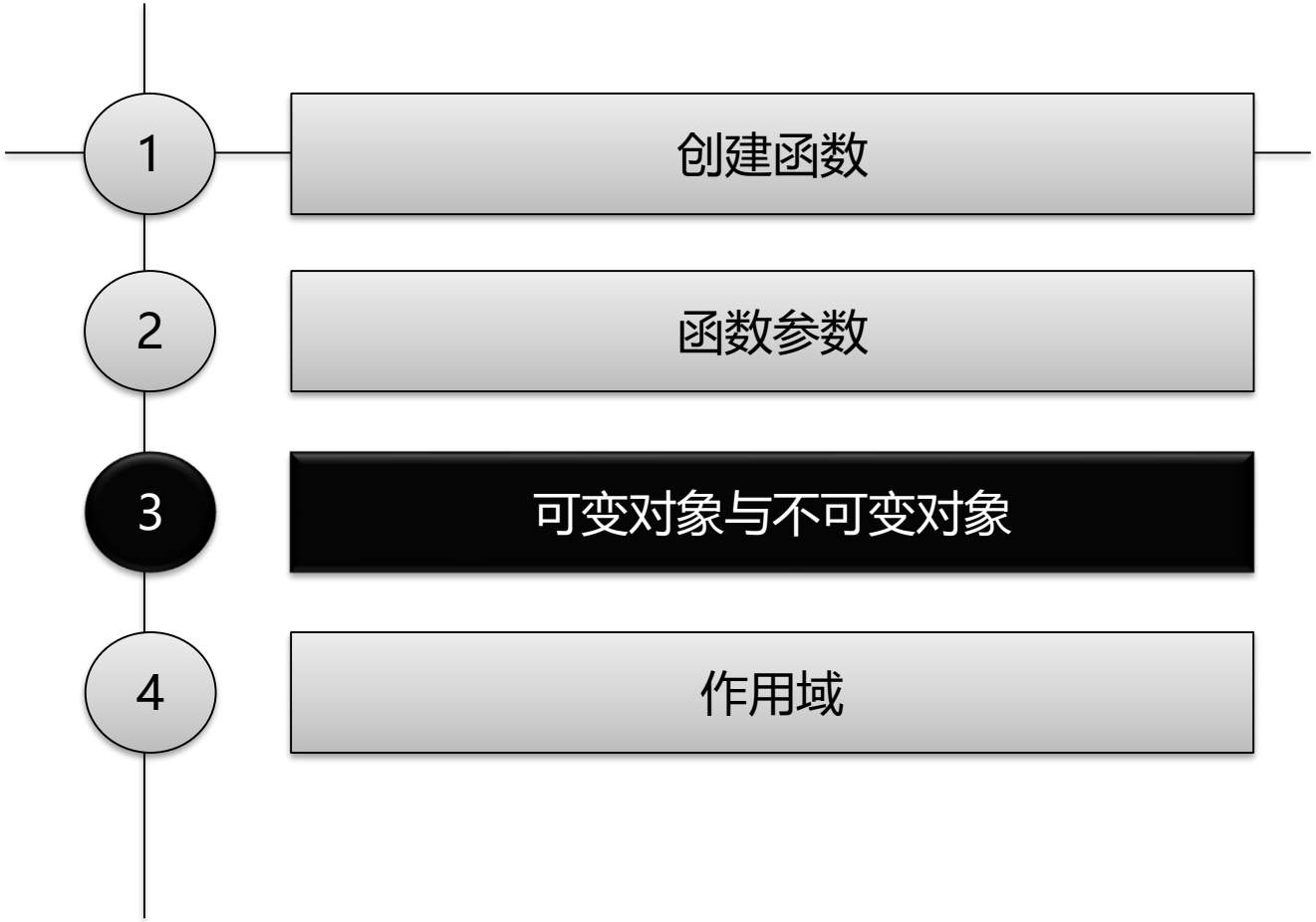
任意数量的位置参数

- 任意数量的位置参数在定义的时候是需要一个星号前缀来表示，在传递参数的时候，可以在原有参数的后面添加零个或多个参数，这些参数将会被放在元组内并传入到函数。带星号前缀的参数必须定义在不带两个星号的参数之后。如：
  - def fun4(str1,*numbers):
  - print fun4 , numbers
  - >>>fun4(“numbers:” ,1,2,3,4) #输出numbers: (1, 2, 3, 4)
- def fun4(*numbers,str1)这样定义参数列表是不允许的，因为同样有歧义。

任意数量的关键字参数

- 任意数量的关键字参数在定义的时候，参数名称前面需要有两个星号(**)作为前缀，这样定义出来的参数，在传递参数的时候，可以在原有的参数后面添加任意零个或多个关键字参数，这些参数会被放到字典内并传入到函数中。带两个星号前缀的参数必须定义在所有带默认值的参数之后。
  - def fun4(a=1,*numbers,**kwargs):
  - print a,numbers,kwargs
  - >>>fun4(4,2,3,4,b=2,c=3)
- #输出4 (2, 3, 4) {'c': 3, 'b': 2}

目录



可变对象和不可变对象

- Python的所有对象可分为可变对象和不可变对象。所谓可变对象是指，对象的内容可变，而不可变对象是指对象内容不可变。如下表说明：

不可变对象	数值类型，字符串，元组
可变对象	字典，列表

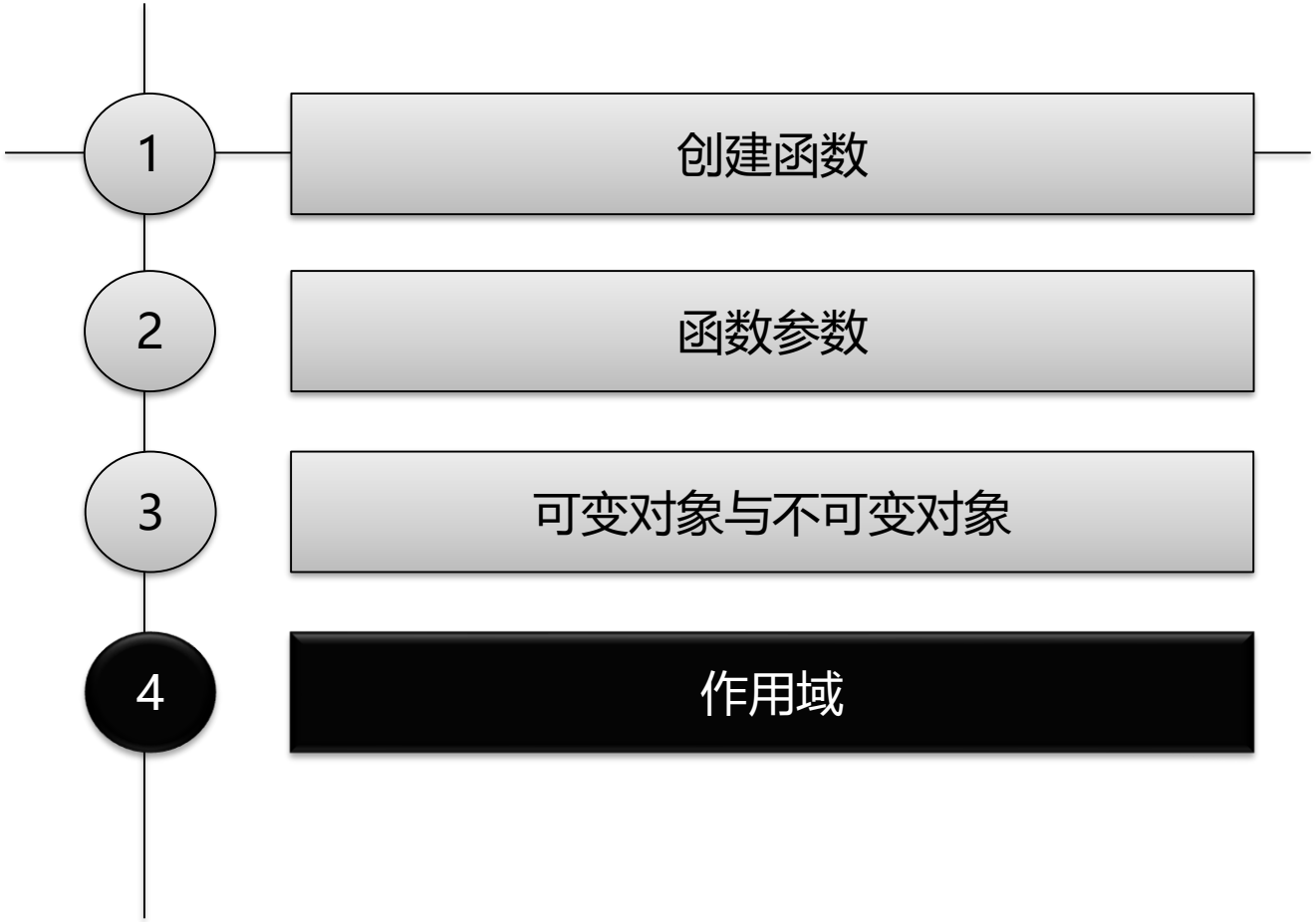
- 我们在第二章已经介绍过数值类型是不可变对象，当程序尝试改变数据的值时，程序会重新生成新的数据，而不是改变原来的数据。
- 之所以本书要将这部分内容放到函数这一章，是因为Python函数的参数都是对象的引用。
- 如果在引用不可变对象中，当尝试修改对象时，程序会在函数中生成新的对象，函数外被引用的对象则不会被改变。

函数

- def add1(num):
- num +=1

- 执行num = 1 ,add1(num),然后再输出num的值，发现num的值还是1。这是因为主程序中的num与函数中的num是不一样的，所以改变函数中的num值时并不会改变函数外的num。但如果参数是一个列表：
  - def add_ele(list):
  - list.append(3)
  - >>>L= [1,2]
  - >>>add_ele(L)
- 如果希望赋值时可变对象不进行引用，而是重新分配地址空间并将数据复制，可以Python的copy模块。其中主要函数有copy.copy和copy.deepcopy
- 1 copy.copy 仅仅复制父对象，不会复制父对象内部的子对象
- 2 copy.deepcopy 复制父对象和子对象

目录



作用域

- Python在创建、改变或查找变量名都是在命名空间进行的，更准确地说，是在特定作用域下进行的。
- 所以我们需要使用某个变量名时，应清晰知道其引用域。由于Python不能声明变量，所以变量第一次被赋值的时候已经与一个特定作用域绑定了。
- 首先举一个函数的例子，如果有这样的函数：
  - def defin_x():
  - x = 2
- 然后执行命令：
  - >>> x = 1
  - >>>defin_x()
  - >>>print x
  - >>>1

全局变量、局部变量

- 执行函数defin_x后函数外的x的值没有变化。这是因为整段程序中存在两个x，起初在函数体外创建了一个x，接着执行defin_x()时又在函数内部创建了一个新的x和一个新的命名空间。
- 第二个x作用域defin_x()函数的内部代码快，赋值语句x = 2仅在局部作用域（即函数内部）起作用。所以它不会使得函数外的x发生改变。我们把函数内的变量称为局部变量而在主程序中的变量称为全局变量。
- 在函数内部是可以访问到全局变量的：
  - def print_x():
  - print x
  - >>>x = 1
  - >>>print_x()
  - >>>1
- 程序没有发生报错并正确返回了1，所以在函数内部同样可以使用全局变

作用域

- 通过我们前面的例子我们已经知道，函数内既可访问局部变量也可访问全部变量。如果局部变量和全局变量出现重名，那最终会访问哪一个呢？
- 实际上，第一个例子已经说明了这个问题，在局部作用域中，如果全局变量与局部变量重名，那么全局变量会被局部变量屏蔽。如果想访问全局变量，可以使用globals函数：
  - def print_x():
  - x = 2
  - print globals()[‘x’]
  - >>> x = 1
  - >>>print_x()
  - >>>1

作用域

- 再考虑另一个方向的问题：我们如何在函数内创建全局变量呢？可以使用global进行声明：
  - def defin_x():
  - 　global x
  - 　x = 2
  - >>>x = 1
  - >>>defin_x()
  - >>>print x
  - >>>2
- 函数内部使用global声明了变量名x的引用域是全局的，因而程序访问的全局变量x。虽然global似乎很好用，但我建议程序中尽量少用global，它会代码变得混乱，可读性变差。相反局部变量使得代码更加抽象，封装性更好。一个好的函数只有输入和输出能够和函数外的程序进行联系。

Thank You!