def a():  
 *"""这是一个测试函数"""* print(1)  
  
  
a()  
  
# 使用\_\_doc\_\_内置属性可以显示函数的说明文档  
# 注意，必须是六个双引号的注释才被采取为说明文档，而#的注释就不行  
print(a.\_\_doc\_\_)  
# 输出：  
# 1  
# 这是一个测试函数

# 函数def  
  
  
#函数的上和下必须各空两行  
def my\_abs(x):  
 if x >= 0:  
 return x  
 else:  
 return -x  
  
  
print(my\_abs(-50))  
# 输出： 50

# 利用isinstance验证，如果不是数字，则raise一个错误  
def my\_abs(x):  
 if not isinstance(x, (int, float)):  
 raise TypeError('bad operand type')  
 if x >= 0:  
 return x  
 else:  
 return -x  
  
  
print(my\_abs('中'))  
# 输出：  
# Traceback (most recent call last):  
# File "D:/python-learning/python-tests/test1/t1.py", line 25, in <module>  
# print(my\_abs('中'))  
# File "D:/python-learning/python-tests/test1/t1.py", line 18, in my\_abs  
# raise TypeError('bad operand type')  
# TypeError: bad operand type

返回多个值的函数

# 函数可以返回多个值，其返回的是一个元组tuple  
def r2():  
 a = 2  
 b = 3  
 return a, b  
  
print(r2())  
# 输出：(2, 3)  
  
print(type(r2()))  
# 输出：<class 'tuple'>  
  
# 也可以这样取值  
x, y = r2()  
print(x)  
# 输出：2  
print(y)  
# 输出：3

■递归函数

# 在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。  
# 举个例子，我们来计算阶乘n! = 1 x 2 x 3 x ... x n，用函数fact(n)表示，可以看出：  
# fact(n) = n! = 1 x 2 x 3 x ... x (n-1) x n = (n-1)! x n = fact(n-1) x n  
# 所以，fact(n)可以表示为n x fact(n-1)，只有n=1时需要特殊处理。  
# 于是，fact(n)用递归的方式写出来就是：  
  
  
def fact(n):  
 if n == 1:  
 return 1  
 return n \* fact(n - 1)  
  
  
print(fact(1))  
# 输出：1  
print(fact(5))  
# 输出：120

●递归函数 分型树实例

import turtle  
  
  
def draw\_branch(branch\_length):  
 if branch\_length > 5:  
 # 右侧树枝  
 turtle.forward(branch\_length)  
 # print('向前', branch\_length)  
 turtle.right(20)  
 # print('右转', 20)  
 draw\_branch(branch\_length - 15)  
  
 # 左侧树枝  
 turtle.left(40)  
 # print('左转', 40)  
 draw\_branch(branch\_length - 15)  
  
 # 返回之前树枝  
 turtle.right(20)  
 # print('右转', 20)  
 turtle.backward(branch\_length)  
 # print('倒退', branch\_length)  
  
  
  
def main():  
 turtle.left(90)  
 turtle.penup()  
 turtle.backward(150)  
 turtle.pendown()  
 turtle.color('brown')  
 draw\_branch(40)  
 turtle.exitonclick()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

■函数的参数

●位置参数

# 计算乘方的函数  
def power(x, n):  
 s = 1  
 while n > 0:  
 n = n - 1  
 s = s \* x  
 return s  
  
  
print(power(2, 3))  
# 输出： 8

●默认参数

# 一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；  
# 二是如何设置默认参数。  
# 当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。  
# 使用默认参数有什么好处？最大的好处是能降低调用函数的难度。  
# 默认参数n=2  
def power(x, n=2):  
 s = 1  
 while n > 0:  
 n = n - 1  
 s = s \* x  
 return s  
  
# 只输入一个参数时，默认第二个参数等于2  
print(power(2))  
# 输出： 4  
print(power(2, 3))  
# 输出： 8

●可变参数

# 计算若干个参数的平方相加 ，例如1的平方加2的平方加3的平方  
def calc(numbers):  
 sum = 0  
 for n in numbers:  
 sum = sum + n \* n  
 return sum  
  
# 这样传入的参数必须是一个数组或元组  
print(calc([1, 2, 3]))  
# 输出： 14  
  
# 使用可变参数\* 就可以随意传入参数了  
def calcs(\*numbers):  
 sum = 0  
 for n in numbers:  
 sum = sum + n \* n  
 return sum  
  
  
print(calcs(1, 2, 3))  
# 输出： 14

●关键字参数

# 关键字参数有什么用？它可以扩展函数的功能。  
# 比如，在person函数里，我们保证能接收到name和age这两个参数，  
# 但是，如果调用者愿意提供更多的参数，我们也能收到。  
# 试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，  
# 利用关键字参数来定义这个函数就能满足注册的需求。  
def person(name, age, \*\*kw):  
 print('name:', name, 'age:', age, 'other:', kw)  
  
# 可以只输入必填参数  
person('Tom', 22)  
# 输出： name: Tom age: 22 other: {}  
  
# 而关键字参数允许传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。  
person('Alice', 21, city='北京')  
# 输出： name: Alice age: 21 other: {'city': '北京'}  
  
# 可以把所有关键字参数先存到一个dict里，然后再把dict传入person，注意前面要加\*\*  
extra = {'city': 'Beijing', 'job': 'Engineer'}  
person('Jack', 24, \*\*extra)  
# 输出： name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

●命名关键字参数

# 如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收city和job作为关键字参数。  
# 和关键字参数\*\*kw不同，命名关键字参数需要一个特殊分隔符\*，\*后面的参数被视为命名关键字参数。  
def person(name, age, \*, city, job):  
 print(name, age, city, job)  
  
  
person('Tom', 22, city='北京', job='程序员')  
# 输出：Tom 22 北京 程序员

■高阶函数

# 既然变量可以指向函数，函数的参数能接收变量，  
# 那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。  
  
  
def add(x, y, f):  
 return f(x) + f(y)  
  
  
# 当我们调用add(-5, 6, abs)时，参数x，y和f分别接收-5，6和abs，根据函数定义，我们可以推导计算过程为：  
# x = -5  
# y = 6  
# f = abs  
# f(x) + f(y) ==> abs(-5) + abs(6) ==> 11  
# return 11  
  
print(add(-5, 6, abs))  
# 输出： 11

●may和reduce 和filter和sorted

Python内建了map()和reduce() 和filter() 和 sorted() 函数

map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

reduce的用法。reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，

和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序：

# Python内置的sorted()函数就可以对list进行排序：  
print(sorted([36, 5, -12, 9, -21]))  
# 输出： [-21, -12, 5, 9, 36]  
  
# 此外，sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序：  
print(sorted([36, 5, -12, 9, -21], key=abs))  
# 输出： [5, 9, -12, -21, 36]  
  
# 默认情况下，对字符串排序，是按照ASCII的大小比较的，由于'Z' < 'a'，结果，大写字母Z会排在小写字母a的前面。  
# # 现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，  
# # 只要我们能用一个key函数把字符串映射为忽略大小写排序即可。  
# # 忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。  
# # 这样，我们给sorted传入key函数，即可实现忽略大小写的排序：  
print(sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower))  
# 输出： ['about', 'bob', 'Credit', 'Zoo']  
  
# 要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True：  
print(sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True))  
# 输出： ['Zoo', 'Credit', 'bob', 'about']

■返回函数和闭包

# 高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。  
# 我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：  
  
  
def calc\_sum(\*args):  
 ax = 0  
 for n in args:  
 ax = ax + n  
 return ax  
  
  
print(calc\_sum(1, 3, 5, 7, 9))  
# 输出： 25  
  
# 但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？  
# 可以不返回求和的结果，而是返回求和的函数：  
  
  
def lazy\_sum(\*args):  
 def sum():  
 ax = 0  
 for n in args:  
 ax = ax + n  
 return ax  
 return sum  
  
  
# 当我们调用lazy\_sum()时，返回的并不是求和结果，而是求和函数：  
f = lazy\_sum(1, 3, 5, 7, 9)  
print(f)  
# 输出： <function lazy\_sum.<locals>.sum at 0x000001F616454400>  
  
# 调用函数f时，才真正计算求和的结果：  
print(f())  
# 输出： 25  
  
# 在这个例子中，我们在函数lazy\_sum中又定义了函数sum，  
# 并且，内部函数sum可以引用外部函数lazy\_sum的参数和局部变量，  
# 当lazy\_sum返回函数sum时，相关参数和变量都保存在返回的函数中，  
# 这种称为“闭包（Closure）”的程序结构拥有极大的威力。  
  
# 注意到返回的函数在其定义内部引用了局部变量args，  
# 所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，  
# 所以，闭包用起来简单，实现起来可不容易。  
# 另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了f()才执行。

●闭包的特性

# 闭包的特性  
  
  
def count():  
 fs = []  
 for i in range(1, 4):  
 def f():  
 return i\*i  
 fs.append(f)  
 return fs  
  
  
f1, f2, f3 = count()  
  
  
# 在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。  
# 你可能认为调用f1()，f2()和f3()结果应该是1，4，9，但实际结果是：  
  
print(f1())  
# 输出：9  
print(f2())  
# 输出：9  
print(f3())  
# 输出：9  
  
# 全部都是9！原因就在于返回的函数引用了变量i，但它并非立刻执行。  
# 等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。  
  
# 返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量  
  
# 如果一定要引用循环变量怎么办？  
# 方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，  
# 无论该循环变量后续如何更改，已绑定到函数参数的值不变：  
  
  
def count():  
 def f(j):  
 def g():  
 return j\*j  
 return g  
 fs = []  
 for i in range(1, 4):  
 fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()  
 return fs  
  
  
f1, f2, f3 = count()  
print(f1())  
# 输出：1  
print(f2())  
# 输出：4  
print(f3())  
# 输出：9  
  
# 缺点是代码较长，可利用lambda函数缩短代码。

●匿名函数

# 匿名函数lambda  
  
  
f = lambda x: x \* x  
  
  
print(f(5))  
# 输出：25  
  
# 关键字lambda表示匿名函数，冒号前面的x表示函数参数。  
# 匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。  
# 用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。  
# 此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：  
  
# 以上函数 f = lambda x: x \* x 等同于下面用def定义的函数  
# def f(x):  
# return x \* x  
  
# 同样，也可以把匿名函数作为返回值返回，比如：  
def build(x, y):  
 return lambda: x \* x + y \* y

■装饰器decorator

<https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014318435599930270c0381a3b44db991cd6d858064ac0000>

■偏函数 functools.partial

<https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/00143184474383175eeea92a8b0439fab7b392a8a32f8fa000>

■函数的作用域

# 函数内定义的变量叫做局部变量，只有函数内部能调用  
  
def b():  
 a = 1  
 c = a + 1  
 return c  
  
  
print(b())  
# 输出： 2  
  
# 在函数外调用a会报错  
print(a)  
# 输出：NameError: name 'a' is not defined

a = 1  
  
  
def b():  
 # 函数内不能直接对全局变量a进行操作  
 a += 1  
 return a  
  
  
# 由于函数b没能操作a，所以a还是原来的值  
print(a)  
# 输出：1  
print(b())  
# 报错：UnboundLocalError: local variable 'a' referenced before assignment  
  
  
# 在函数内部对全局变量进行操作，需要写上global加变量名  
def c():  
 global a  
 a += 1  
 return a  
  
  
print(c())  
# 输出： 2  
  
# 全局变量a，在执行c()时已经被修改  
print(a)  
# 输出：2