在Python中，通过def去定义函数。def语句是实时执行的代码：函数并不存在，直到Python运行了def后，其会生成一个新的函数对象并赋值给函数名这个变量。同之前学习的变量与对象之间的关系类似。

例：程序function\_detail/function\_detail1.py

def pets(pet):  
 print("pet is ", pet)  
  
  
# 函数名pets是一个变量，  
# 其引用了一个函数对象  
# 将其赋值给pets\_1后，  
# 可以通过pets\_1调用函数  
pets\_1 = pets  
pets\_1('cat')

输出为：

pet is cat

Python函数中，return语句返回的是一个结果对象。

Python中的多态：依赖类型的行为。

例：function\_detail/function\_detail2.py

def times(x, y):  
 return x \* y  
  
  
print(times(2, 4)) # 8  
print(times('Ni', 4)) # NiNiNiNi

根据输入对象类型的不同，执行不同的操作，这是Python中的多态。如果输入的对象无法执行定义的操作，Python就会抛出错误。

例：function\_detail/function\_detail2.py

def intersect(seq1, seq2):  
 res = []  
 for x in seq1:  
 if x in seq2:  
 res.append(x)  
 return res  
  
  
s1 = 'SPAM'  
s2 = 'SCAM'  
print(intersect(s1, s2))  
  
# 列表中对象的类型有可能不相同  
list1 = [1, 2, 3, 'a']  
list2 = (1, 'a')  
print(intersect(list1, list2))

输出为：

['S', 'A', 'M']

[1, 'a']

Python中的参数传递：

1. 参数传递是通过自动将对象赋值给本地变量来实现的，遵循赋值的规则，不自动拷贝对象。
2. 在函数内部，参数名的赋值不会影响调用者，类似C/C++中函数内部修改了指针本身的效果。
3. 在函数内部，可变对象参数的修改有可能对调用者有影响，类似C/C++中修改了指针指向的对象。

参数解包

例：程序function\_detail/function\_detail3.py

# 可变参数  
# 可变参数是一个元组  
def args(\*params):  
 print(isinstance(params, tuple))  
  
  
args(1) # True  
args({'a': 1}) # True  
  
  
# 任意关键字参数  
# 任意关键字参数是一个字典  
def args1(\*\*key\_words):  
 print(isinstance(key\_words, dict))  
  
  
args1(a=1, b=2) # True  
  
  
def func(a, b, c, d):  
 print('a = {0}, b = {1}, c = {2}, d = {3}'.format(a, b, c, d))  
  
  
args = (1, 2, 3, 4)  
# func(args) # 参数不匹配  
# 使用\*运算符解包成可变参数  
# 参数个数需要匹配  
func(\*args) # Ok a = 1, b = 2, c = 3, d = 4  
  
keyWords = {'a': 1, 'b': 2, 'd': 4, 'c': 3}  
keyWords1 = {'a1': 1, 'b1': 2, 'c1': 3, 'd1': 4}  
# 使用\*\*解包成任意关键字参数  
# 注意：字典的keys需要与函数  
# 参数名对应并且个数匹配  
# 顺序可以不同  
func(\*\*keyWords)  
# func(\*\*keyWords1) # Error  
  
# \*与\*\*的混合运用  
func(\*(1, 2), \*\*{'d': 5, 'c': 4}) # a = 1, b = 2, c = 4, d = 5  
func(1, \*[2, 3], \*\*{'d': 6}) # a = 1, b = 2, c = 3, d = 6  
func(1, c=99, \*(2, ), \*\*{'d': 101}) # a = 1, b = 2, c = 99, d = 101  
func(1, \*\*{'d': 101, 'b': 99}, c=0) # a = 1, b = 99, c = 0, d = 101  
  
# \*\*解析为关键字参数，后面不能再跟位置参数  
# func(1, \*\*{'d': 101, 'b': 99}, 0) # Error

通用函数

例：程序function\_detail/function\_detail4.py

# 不知道函数名和参数  
def tracer(func, \*args, \*\*kargs):  
 print("call func: ", func.\_\_name\_\_)  
 return func(\*args, \*\*kargs)  
  
  
def func1(a, b, c, d):  
 return a + b + c + d  
  
  
def func2(a, b):  
 return a \* b  
  
  
print(tracer(func1, 1, 2, c=3, d=4))  
print(tracer(func2, \*\*{'a': 5, 'b': 6}))  
print(tracer(func2, 10, \*(5,)))

输出为：

call func: func1

10

call func: func2

30

call func: func2

50

命名关键字参数：

例：程序function\_detail/function\_detail5.py

# \*后的b和c均为命名关键字参数  
# 如果没有默认值，就必须使用关键字参数赋值  
def kwonly1(a, \*, b, c):  
 print('a = {0}, b = {1}, c = {2}'.format(a, b, c))  
  
  
kwonly1(1, b=2, c=3) # a = 1, b = 2, c = 3  
# kwonly1(4, 5, b=5) # Error  
  
  
# 有可变参数，不需要单独的\*  
def kwonly2(a, \*b, c, d=5):  
 print('a = {0}, b = {1}, c = {2}, d = {3}'.format(a, b, c, d))  
  
  
kwonly2(1, c=8) # a = 1, b = (), c = 8, d = 5  
kwonly2(6, 6, c=8, d=9) # a = 6, b = (6,), c = 8, d = 9

递归函数：

例：程序function\_detail/function\_detail7.py

def my\_sum(L):  
 if not L:  
 return 0  
 else:  
 return L[0] + my\_sum(L[1:])  
  
  
def my\_sum1(L):  
 return 0 if not L else L[0] + my\_sum1(L[1:])  
  
  
print(my\_sum([1, 2, 3, 4, 5])) # 15  
print(my\_sum1([1, 2, 3, 4, 5])) # 15  
  
  
# 一般不提倡递归，但递归可以出来嵌套的序列  
def sumtree(L):  
 tot = 0  
 for x in L:  
 if not isinstance(x, list):  
 tot += x  
 else:  
 tot += sumtree(x)  
 return tot  
  
  
L = [1, [2, [3, 4], 5], 6, [7, 8]]  
print(sumtree(L)) # 36  
print(sumtree([1, [2, [3, [4, [5]]]]])) # 15

函数对象：在Python中，函数本身就是一个对象。

例：程序function\_detail/function\_detail8.py

# 间接函数调用  
def echo(message):  
 print(message)  
  
  
def indirect(func, arg):  
 func(arg)  
  
  
indirect(echo, "Argument Call") # Argument Call  
print()  
  
# 函数本身是对象，可以存储在序列中  
schedule = [(echo, 'Spam!'), (echo, 'Ham!')]  
for (func, arg) in schedule:  
 func(arg)  
 # func.count += 1 # Error，echo没有属性count  
print()  
  
# 函数属性，类似类对象的属性  
echo.count = 0  
schedule1 = [(echo, 'Spam1!'), (echo, 'Ham1!')]  
for (func, arg) in schedule1:  
 func(arg)  
 func.count += 1 # OK, echo有count属性

输出为：

Argument Call

Spam!

Ham!

Spam1!

Ham1!

函数中如果没有return或yield返回时，默认返回None。