Python中不支持类似C++的函数重载：

例：程序function\_test/function\_test1.py

def greet\_user():  
*"""显示简单的问候语"""*print('Hello!')  
  
  
def greet\_user(username):  
*"""显示简单的问候语"""*print('Hello! ' + str(username).title())  
  
  
greet\_user()  
greet\_user('Jame')

Error:

Traceback (most recent call last):

File "C:/study/PythonStudy/function\_test/function\_test1.py", line 17, in <module>

greet\_user()

TypeError: greet\_user() missing 1 required positional argument: 'username'

但可以像如下方式来处理：

def greet\_user():  
*"""显示简单的问候语"""*print('Hello!')  
  
  
greet\_user()  
  
  
def greet\_user(username):  
*"""显示简单的问候语"""*print('Hello! ' + str(username).title())  
  
  
greet\_user('Jame')

输出为：

Hello!

Hello! Jame

在定义了def greet\_user(username):  
后，之前定义的def greet\_user():就失效了。

位置参数：基于实参的顺序，与C语言的函数参数类似

例：程序function\_test/function\_test2.py

def describe\_pet(animal\_type, pet\_name):  
 *"""显示宠物信息"""* print("\nI have a " + animal\_type + ".")  
 print("My " + animal\_type + "'s name is " + pet\_name)  
  
  
describe\_pet('hamster', 'harry')  
describe\_pet('dog', 'willie')  
  
# 带有关键字的位置实参，传递给函数key-value对  
# 如果使用了关键字实参，则需要放置在所有的位置参数之后

# 关键字实参不允许重复，但顺序可以随意  
# describe\_pet(animal\_type='cat', 'Tom') # Error  
# describe\_pet(pet\_name='cat', pet\_name='Tom') # Error  
describe\_pet(animal\_type='cat', pet\_name='Tom')  
describe\_pet(pet\_name='Mickey', animal\_type='mouse')

输出为：

I have a hamster.

My hamster's name is harry

I have a dog.

My dog's name is willie

I have a cat.

My cat's name is Tom

I have a mouse.

My mouse's name is Mickey

函数中参数的传递

例：function\_test/function\_test4.py

def modify(para):  
if isinstance(para, int):  
print("para is int.")  
 para = 5  
elif isinstance(para, str):  
print("para is string.")  
 para = 'Jam'  
elif isinstance(para, list):  
print("para is list.")  
# para = ['hamster']  
para.append('cat')  
print("In modify() para is modified: " + str(para))  
  
  
parameter = 1  
print("before modify(): " + str(parameter))  
modify(parameter)  
print("after modify(): " + str(parameter))  
print()  
  
parameter = 'Tom'  
print("before modify(): " + str(parameter))  
modify(parameter)  
print("after modify(): " + str(parameter))  
print()  
  
parameter = ['dog']  
print("before modify(): " + str(parameter))  
modify(parameter)  
print("after modify(): " + str(parameter))

输出为：

before modify(): 1

para is int.

In modify() para is modified: 5

after modify(): 1

before modify(): Tom

para is string.

In modify() para is modified: Jam

after modify(): Tom

before modify(): ['dog']

para is list.

In modify() para is modified: ['dog', 'cat']

after modify(): ['dog', 'cat']

从输出可以看出，传递给modify()数字和字符串时，函数内部修改，不会影响到外部实参，传递列表给函数时，函数内部修改了列表的元素，会影响外部的实参。

参考：Python学习3-对象与类型.docx：

传递实参给函数的形参时，实际是将两个变量引用到同一个对象，Python中的引用类似C语言中的指针，对于不可变对象，函数内部的修改相当于改变了其引用的对象，类似C语言修改了指针的指向，所以不会影响外部的实参；对于可变对象，如果在函数内部修改了其指向的元素，就会影响到外部的实参，如果直接改变其引用的对象，也不会影响外部的实参。当传递列表给modify()函数，如果直接使用语句：

para = ['hamster']

就不会影响外部的实参parameter。

根据以上情况，在使用默认参数时，尽量不要使用默认的可变对象，这有可能会产生预料之外的结果。

例：function\_test/function\_test3.py

def add\_end(animals=[]):  
 animals.append('End')  
 return animals  
  
  
print(add\_end(['dog']))  
print()  
print(add\_end(['cat']))  
print()  
print(add\_end())  
print(add\_end())

add\_end()函数使用了一个默认的空列表参数。当调用：

add\_end(['dog'])

add\_end(['cat'])

输出为：

['dog', 'End']

['cat', 'End']

一切比较正常，但调用两次默认参数：

add\_end()

add\_end()

输出为：

['End']

['End', 'End']

可以看出，输出的结果不正确。

产生这种结果的原因是：Python函数在定义时，默认参数animals的值就被计算出来（类似C/C++函数的默认参数是在编译时确定），它引用一个空[]对象，如果在调用函数时，更改了该对象，例如通过append()增加了元素（相当于C语言中更改了指针指向的元素），那么下次调用时，默认值就发生了改变，不再是空[]。所以，尽可能的不要把函数的默认值设置为可变对象。

可变参数：任意长度的参数

例：程序function\_test/function\_test7.py

def make\_pizza(\*toppings):  
 # 可变参数是一个元组  
 if isinstance(toppings, tuple):  
 print("toppings is tuple")  
 print("Making a pizza with the following toppings:")  
 for topping in toppings:  
 print("- " + topping)  
  
  
make\_pizza('pepperoni')  
print()  
make\_pizza('mushrooms', 'green peppers', 'extra cheese')  
print()  
  
  
def calc\_sum(\*numbers):  
 result = 0  
 for n in numbers:  
 result += n  
 return result  
  
  
print(calc\_sum(1, 2))  
print(calc\_sum(1, 2, 3))  
  
# 如果存在一个列表或元组  
# 可以使用\*将其转换为可变参数  
nums\_1 = [1, 2, 3, 4, 5]  
nums\_2 = (1, 3, 5, 7, 9)  
print(calc\_sum(\*nums\_1))  
print(calc\_sum(\*nums\_2))

输出为：

toppings is tuple

Making a pizza with the following toppings:

- pepperoni

toppings is tuple

Making a pizza with the following toppings:

- mushrooms

- green peppers

- extra cheese

3

6

15

25

从输出可以看出，可变参数被自动封装成了一个元组tuple。

任意数量的关键字实参：

例：程序function\_test/function\_test8.py

# 关键字参数user\_info要放在最后  
# 程序自动将user\_info封装成字典  
def build\_profile(first, last, \*\*user\_info):  
 profile = {}  
 profile['first\_name'] = first  
 profile['last\_name'] = last  
 for key, value in user\_info.items():  
 profile[key] = value  
 return profile  
  
  
# 以下两种调用等效  
user\_profile = build\_profile('albert', 'einstein', location='princeton', field='physics')  
user\_profile = build\_profile(first='albert', last='einstein', location='princeton', field='physics')  
print(user\_profile)

输出为：

{'first\_name': 'albert', 'last\_name': 'einstein', 'location': 'princeton', 'field': 'physics'}

命名关键字实参：

例：程序function\_test/function\_test8.py

# 命名关键字实参  
# 使用\*运算符，只接收city和job关键字实参  
def person(name, age, \*, city, job):  
 print(name, age, city, job)  
  
  
person('Jack', 24, city='Beijing', job='Engineer')  
# person('Tom', 23) # Error缺少2个关键字参数  
# person('Tom', 23, addr='Beijing', code='123') # Error关键字不匹配

输出为：

Jack 24 Beijing Engineer

参数组合：

例：程序function\_test/function\_test9.py

# 位置参数与可变参数的组合  
def make\_pizza(size, \*toppings):  
 print("\nMaking a " + str(size) + " pizza with the following toppings:")  
 for topping in toppings:  
 print("- " + topping)  
  
  
make\_pizza(16, 'pepperoni')  
# make\_pizza(size=16, topping='pepperoni') # Error可变参数无法使用关键字

输出为：

Making a 16 pizza with the following toppings:

- pepperoni

在Python中，可以使用位置参数（必选参数）、默认参数、可变参数、任意关键字参数和命名关键字参数5种参数，但顺序必须是：

1. 位置参数
2. 默认参数
3. 可变参数
4. 命名关键字参数
5. 任意关键字参数

例：程序function\_test/function\_test9.py

# python中参数的组合与顺序  
# c为默认参数  
def parameter\_sequence1(a, b, c=0, \*args, \*\*kw):  
 print('a = ', a, 'b = ', b, 'c = ', c, 'args = ', args, 'kw = ', kw)  
  
  
# c为命名关键字参数  
# 如果参数中有可变参数，就不需要  
# 使用\*来定义命名关键字参数  
def parameter\_sequence2(a, b, \*args, c=0, \*\*kw):  
 print('a = ', a, 'b = ', b, 'args = ', args, 'c = ', c, 'kw = ', kw)  
  
  
parameter\_sequence1(1, 2)  
parameter\_sequence1(1, 2, c=3)  
parameter\_sequence1(1, 2, 3, 'a', 'b', key=99)  
print()  
parameter\_sequence2(1, 2, 'a', 'b', c=99, key=1001)  
parameter\_sequence2(1, 2, 'a', 'b', x=99, key=1001)

输出为：

a = 1 b = 2 c = 0 args = () kw = {}

a = 1 b = 2 c = 3 args = () kw = {}

a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'key': 99}

a = 1 b = 2 args = ('a', 'b') c = 99 kw = {'key': 1001}

a = 1 b = 2 args = ('a', 'b') c = 0 kw = {'x': 99, 'key': 1001}