运算符重载：

1. 以双下划线命名的方法（\_\_X\_\_）
2. 当实例出现在内置运算时，这类方法会自动调用，并会被子类继承
3. 运算符重载方法没有默认值，如果实例进行某种内置运算，而未定义相应的运算符重载方法，就会抛出异常

例：程序class\_detail/class\_detail4.py

import class\_detail3  
  
  
class ThirdClass(class\_detail3.SecondClass):  
def \_\_init\_\_(self, value):  
print('\_\_init\_\_ called')  
self.data = value  
  
def \_\_add\_\_(self, other):  
print('\_\_add\_\_ called')  
return ThirdClass(self.data + other)  
  
def \_\_str\_\_(self):  
print('\_\_str\_\_ called')  
return '[ThirdClass: {0}]'.format(self.data)  
  
def mul(self, other):  
self.data \*= other  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 a = ThirdClass('abc')  
 a.display()  
print(a) # \_\_str\_\_被调用  
print()  
  
 b = a + 'xyz'  
b.display()  
print(b)  
print()  
  
 a.mul(3)  
print(a)

输出为：

\_\_init\_\_ called

Current value = abc

\_\_str\_\_ called

[ThirdClass: abc]

\_\_add\_\_ called

\_\_init\_\_ called

Current value = abcxyz

\_\_str\_\_ called

[ThirdClass: abcxyz]

\_\_str\_\_ called

[ThirdClass: abcabcabc]

从输出可以看出，创建实例对象时，\_\_init\_\_会自动调用，当print或str()操作实例对象时，\_\_str\_\_会自动调用。实例对象参与+运算时，\_\_add\_\_会自动调用。

类的属性：

例：程序class\_detail/class\_detail5.py

# 最简单的Python类  
class Rec:  
pass  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 Rec.name = 'Bob' # 类的属性  
Rec.age = 40  
print(Rec.name)  
print(list(Rec.\_\_dict\_\_.keys()))  
print()  
  
 x = Rec()  
 y = Rec()  
# 实例会继承类的属性  
print(x.name, y.name)  
print(list(x.\_\_dict\_\_.keys())) # []，实例本身没有属性  
print(list(y.\_\_dict\_\_.keys()))  
print()  
  
x.name = 'Sue'  
print(x.name)  
print(list(x.\_\_dict\_\_.keys())) # ['name']，实例x有自身的属性name

输出为：

Bob

['\_\_module\_\_', '\_\_dict\_\_', '\_\_weakref\_\_', '\_\_doc\_\_', 'name', 'age']

Bob Bob

[]

[]

Sue

['name']

类的实例分析：

例：程序class\_detail/person.py

class Person:  
def \_\_init\_\_(self, name, job=None, pay=0):  
self.name = name  
self.job = job  
self.pay = pay  
  
def last\_name(self):  
return self.name.split()[-1]  
  
def give\_raise(self, percent):  
self.pay = int(self.pay \* (1 + percent))  
  
def \_\_str\_\_(self): # 运算符重载  
return '[Person: {0}, {1}]'.format(self.name, self.pay)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 bob = Person('Bob Smith', 'dev', 10000)  
print(bob.name, bob.pay)  
print(bob.last\_name())  
 bob.give\_raise(0.1)  
print(bob.pay)  
print(bob)

输出为：

Bob Smith 10000

Smith

11000

[Person: Bob Smith, 11000]

程序class\_detail/manager.py

import person  
  
  
class Manager(person.Person):  
def \_\_init\_\_(self, name, pay):  
super().\_\_init\_\_(name, 'mgr', pay)  
  
# 重载超类的方法，除非功能完全不同，  
# 一般尽可能使用超类的方法来扩展，  
# 以方便后序维护  
def give\_raise(self, percent, bonus=0.1):  
super().give\_raise(percent+bonus)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 tom = Manager('Tom Jones', 50000)  
 tom.give\_raise(0.1)  
print(tom) # 子类会继承超类的运算符重载

输出为：

[Person: Tom Jones, 60000]

类接口技术：

例：程序class\_detail/class\_detail6.py

class Super:  
def method(self):  
print('in Super.method')  
  
def delegate(self):  
self.action()  
  
def action(self):  
assert False, 'action must be defined!'  
  
  
class Inheritor(Super):  
pass  
  
  
class Replacer(Super):  
def method(self):  
print('in Replacer.method')  
  
  
class Extender(Super):  
def method(self):  
print('starting Extender.method')  
super().method()  
print('ending Extender.method')  
  
  
class Provider(Super):  
def action(self):  
print('in Provider.action')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
for klass in (Inheritor, Replacer, Extender):  
print('\n' + klass.\_\_name\_\_ + '...')  
 klass().method()  
print('\nProvider...')  
 x = Provider()  
 x.delegate()  
  
 y = Super()  
# y.delegate() # AssertionError: action must be defined!

输出为：

Inheritor...

in Super.method

Replacer...

in Replacer.method

Extender...

starting Extender.method

in Super.method

ending Extender.method

Provider...

in Provider.action

Provider类中重载了action()方法，所以可以通过实例对象调用delegate()方法，类似C#/Java中的接口，如果子类不实现接口，则无法使用。

用户自定义的迭代器：

例：程序class\_detail/class\_detail8.py

class Squares:  
def \_\_init\_\_(self, start, stop):  
self.value = start - 1  
self.stop = stop  
  
def \_\_iter\_\_(self): # 自定义迭代器  
return self # 返回实例本身，支持单个迭代器  
  
def \_\_next\_\_(self):  
 if self.value == self.stop:  
raise StopIteration  
self.value += 1  
return self.value \*\* 2  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
for i in Squares(1, 5):  
print(i, end=' ')  
print()  
  
 X = Squares(1, 5)  
 it1 = iter(X)  
 it2 = iter(X)  
print(next(it1)) # 1  
print(next(it2)) # 4  
print(next(it1)) # 9

输出为：

1 4 9 16 25

1

4

9

从上述输出可以看出，由于是单个迭代器，\_\_iter\_\_返回对象本身，所以两个迭代器指向同一个对象时，会相互影响对方。

多个迭代器

程序class\_detail/class\_detail9.py

class SkipIterator:  
def \_\_init\_\_(self, wrapped):  
self.wrapped = wrapped  
self.offset = 0  
  
def \_\_next\_\_(self):  
if self.offset >= len(self.wrapped):  
raise StopIteration  
else:  
 item = self.wrapped[self.offset]  
self.offset += 1  
return item  
  
  
class SkipObject:  
def \_\_init\_\_(self, wrapped):  
self.wrapped = wrapped  
  
def \_\_iter\_\_(self): # 返回新的迭代对象，支持多个迭代  
return SkipIterator(self.wrapped)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 alpha = 'abcedf'  
skipper = SkipObject(alpha)  
 it1 = iter(skipper)  
 it2 = iter(skipper)  
print(next(it1)) # a  
print(next(it2)) # a  
print(next(it1)) # b  
print(next(it2)) # b

输出为：

a

a

b

b

例：程序class\_detail/class\_detail10.py

class Iters:  
def \_\_init\_\_(self, value):  
self.data = value  
  
def \_\_getitem\_\_(self, i): # 索引，切片  
print('get[{0}]:'.format(i), end='')  
  
# 优先级高于\_\_getitem\_\_  
def \_\_iter\_\_(self):  
print('iter=> ', end='')  
self.ix = 0  
return self  
  
def \_\_next\_\_(self):  
print('next:', end='')  
if self.ix == len(self.data):  
raise StopIteration  
item = self.data[self.ix]  
self.ix += 1  
return item  
  
# 优先级高于\_\_iter\_\_  
def \_\_contains\_\_(self, x): # in  
print('contains: ', end='')  
return x in self.data  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 X = Iters([1, 2, 3, 4, 5])  
print(3 in X) # 调用\_\_contains\_\_, contains: True  
for i in X: # 先执行\_\_iter\_\_，然后\_\_next\_\_  
print(i, end=' | ')

输出为：

contains: True

iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:

\_\_call\_\_的重载

例：程序class\_detail/class\_detail11.py

class Callee:  
def \_\_call\_\_(self, \*args, \*\*kwargs):  
print('Called:', args, kwargs)  
  
  
C = Callee()  
C(1, 2, 3) # 自动调用\_\_call\_\_  
C(1, 2, 3, x=4, y=5)  
# \_\_call\_\_支持所有参数传递方式  
C(\*[1, 2], \*\*dict(c=3, d=4))  
C(1, \*(2,), c=3, \*\*dict(d=4))

输出为：

Called: (1, 2, 3) {}

Called: (1, 2, 3) {'x': 4, 'y': 5}

Called: (1, 2) {'c': 3, 'd': 4}

Called: (1, 2) {'c': 3, 'd': 4}

析构函数\_\_del\_\_：当实例空间被收回时，\_\_del\_\_会自动调用。

例：程序class\_detail/class\_detail12.py

class Life:  
def \_\_init\_\_(self, name='unknown'):  
print('Hello', name)  
self.name = name  
  
def \_\_del\_\_(self):  
print('Goodbye', self.name)  
  
  
brain = Life('Brain')  
brain = 1 # 失去Life的最后一个实例，触发\_\_del\_\_函数

输出为：

Hello Brain

Goodbye Brain