# 运算符重载

## 基础知识

当类中提供某个特殊名称方法，在该类的实例出现在它们相关的表达式时，python自动调用它们。

* 不能重载内置类型运算符
* 不能新建运算符，只能重载现有的
* 某些运算符不能重载——is、and、or和not

### 中缀运算符





### 比较运算符



### 构造函数和表达式：\_\_init\_\_和\_\_sub\_\_

class **Number**(object):

def **\_\_init\_\_**(*self*, start):

*self*.data = start

def **\_\_sub\_\_**(*self*, other):

return Number(*self*.data - other)

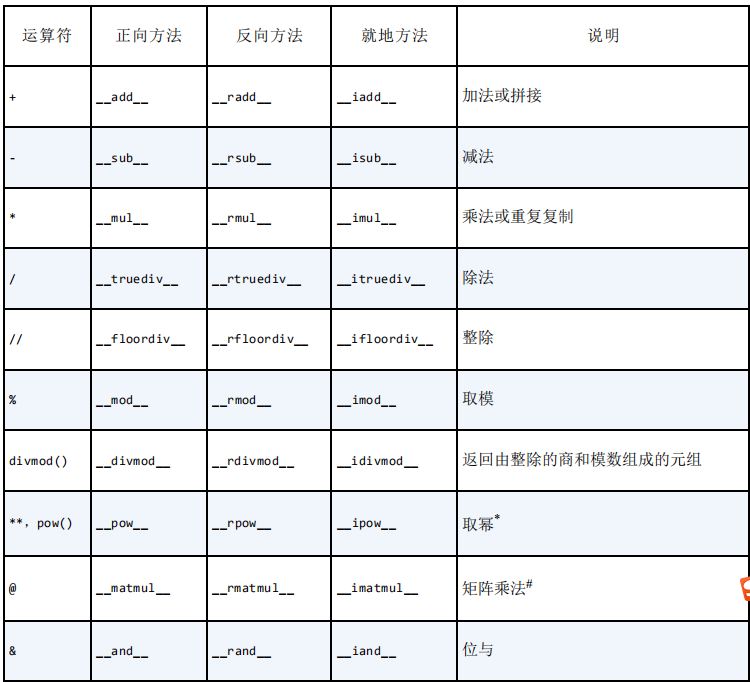
X = Number(5)

Y = X - 2

print Y.data //3

### 常见的运算符重载方法









|  |  |  |
| --- | --- | --- |
| 方法 | 重载 | 调用 |
| \_\_init\_\_ | 构造函数 | 对象建立：X=Class(args) |
| \_\_del\_\_ | 析构函数 | X对象收回 |
| \_\_add\_\_ | 运算符+ | 如果没有\_iadd\_,X+Y,X+=Y |
| \_\_or\_\_ | 运算符|（位or） | 如果没有\_ior\_,X|Y,X|=Y |
| \_\_repr\_\_,\_\_str\_\_ | 打印、转换 | print (X)、repr(X、str(X), |
| \_\_call\_\_ | 函数调用 | X(\*args, \*\*kargs) |
| \_\_getattr\_\_ | 点号运算 | X.undefined |
| \_\_setattr\_\_ | 属性赋值语句 | X.any = value |
| \_\_delattribute\_\_ | 属性获取 | X.any |
| \_\_getitem\_\_ | 索引运算 | X[key],X[i:j],没\_\_iter\_\_时的for循环和其他迭代器 |
| \_\_setitem\_\_ | 索引赋值语句 | X[key]=value, X[i:j]=sequence |
| \_\_delitem\_\_ | 索引和分片删除 | del X[key], del X[i:j] |
| \_\_len\_\_ | 长度 | len(x) |
| \_\_bool\_\_ | 布尔测试 | bool(X),真测试 |
| \_\_lt\_\_,\_\_gt\_\_ | 特定的比较 | X<Y,X>Y |
| \_\_le\_\_,\_\_ge\_\_ |  | X<=Y,X>=Y |
| \_\_eq\_\_,\_\_ne\_\_ |  | X==Y,X!=Y |
| \_\_radd\_\_ | 右侧加法 | Other+X |
| \_\_iadd\_\_ | 实地（增强的）加法 | X+=Y |
| \_\_iter\_\_,\_\_next\_\_ | 迭代环境 |  |
| \_\_contains\_\_ | 成员关系测试 | item in X |
| \_\_index\_\_ | 整数值 | hex(X),bin(X),oct(X),O[X],O[X:] |
| \_\_enter\_\_,\_\_exit\_\_ | 环境管理器 | with obj as var: |
| \_\_get\_\_,\_\_set\_\_ | 描述符属性 | X.attr,X.attr=value,del X.attr |
| \_\_delete\_\_ |  |  |
| \_\_new\_\_ | 创建 | 在\_\_init\_\_之前创建对象 |

### 索引和分片：\_\_getitem\_\_和\_\_setitem\_\_

class **Indexer**:

def **\_\_getitem\_\_**(*self*, index):

return index \*\* 2

X = Indexer()

for i in range(5):

print X[i], #0 1 4 9 16

除了索引分片表达式也调用\_\_getitem\_\_

class **Indexer**:

data = [5,6,7,8,9,10]

def **\_\_getitem\_\_**(*self*, index):

print *'getitem:'*, index

return *self*.data[index]

print X[2:4] #getitem: slice(2, 4, None) [7, 8]

print X[::2] #getitem: slice(None, None, 2) [5, 7, 9]

\_\_setitem\_\_索引赋值方法类似地拦截索引和分片赋值，它为后来者接受了一个分片对象。

def **\_\_setitem\_\_**(*self*, index, value):

*self*.data[index] = value

X[2:4] = [5, 5, 5, 5, 5, 5]

print X.data #setitem: slice(2, 4, None) [5, 6, 5, 5, 5, 5, 5, 5, 9, 10]

索引迭代：

class **stepper**:

def **\_\_getitem\_\_**(*self*, index):

return *self*.data[index]

X = stepper()

X.data = *'spam'*

print X[1] #p

for item in X:

print item, #s p a m

### 迭代器对象\_\_iter\_\_和\_\_next\_\_

class **Squares**:

def **\_\_init\_\_**(*self*, start, stop):

*self*.value = start - 1

*self*.stop = stop

def **\_\_iter\_\_**(*self*):

return *self*

def **next**(*self*):

if *self*.value == *self*.stop:

raise StopIteration

*self*.value += 1

return *self*.value \*\* 2

X = Squares(1, 5)

for i in X:

print i #1 4 9 16 25

### 属性引用：\_\_getattr\_\_和\_\_setattr\_\_

\_\_getattr\_\_方法是拦截属性点号运算，当通过对**未定义**（不存在）属性名称和实例进行点号运算时，就会用属性名称作为字符串调用这个方法。

class **Empty**:

def **\_\_getattr\_\_**(*self*, attrname):

if attrname == *"age"*:

return 40

else:

raise AttributeError, attrname

X = Empty()

print X.age #40

print X.name #AttributeError: name

\_\_setattr\_\_会拦截所有属性的赋值语句：

class **Accesscontrol**:

def **\_\_setattr\_\_**(*self*, attr, value):

if attr == *"age"*:

*self*.\_\_dict\_\_[attr] = value

else:

raise AttributeError, attr + *'not allowed'*

Y = Accesscontrol()

Y.age = 24

print Y.age

### \_\_repr\_\_和\_\_str\_\_会返回字符串表达形式

class **Adder**:

def **\_\_init\_\_**(*self*, value = 0):

*self*.data = value

def **\_\_add\_\_**(*self*, other):

*self*.data += other

class **Addrepr**(Adder):

def **\_\_repr\_\_**(*self*):

return *"Addrepr(%s)"* %*self*.data

X = Addrepr(2)

X+2

print X //Addrepr(4)

* 打印操作会首先尝试\_\_str\_\_和str内置函数。
* \_\_repr\_\_用于所有其他环境：用于交互模式下提示回应以及repr函数。

\_\_repr\_\_用于任何地方，除了当定义一个\_\_str\_\_的时候。

### call表达式:\_\_call\_\_

当调用实例时，使用\_\_call\_\_方法。传递给实例的任何内容都会传递给该方法，包括通常隐式的实例参数。

class **Prod**:

def **\_\_init\_\_**(*self*, value):

*self*.value = value

def **\_\_call\_\_**(*self*, other):

return *self*.value \* other

X = Prod(2)

print X(3) //6

### 对象析构函数：\_\_del\_\_

每当实例空间被收回时，就会执行\_\_del\_\_

class **Life**:

def **\_\_init\_\_**(*self*, name = *'unkown'*):

print *'Hello'*, name

*self*.name = name

def **\_\_del\_\_**(*self*):

print *"Goodbye"*, *self*.name

X = Life(*"Brian"*)

X = *"lorette" //*Goodbye Brian

python在实例回收时，会自动回收该实例所拥有的所有空间，对于空间管理来说是不需要析构函数的。原因之二在于无法轻易地预测何时收回空间。

### with/as环境管理器

with/as语句的设计，是为了让必须在程序代码块周围发生的启动和终止活动一定会发生。和try/finally语句（无论异常是否发生，其离开动作都会执行）类似，但是with/as有更丰富的对象协议，可以定义进入和离开的动作。

**with语句的基本格式**：

with expression [as variable]:

With-block

with语句实际的工作方式：

1. 计算表达式，所得到的对象称为环境管理器，它必须有\_\_enter\_\_和\_\_exit\_\_方法。
2. 环境管理器的\_\_enter\_\_方法会被调用。如果as子句存在，其返回值会赋值给ss子句中的变量，否则丢弃。
3. 代码块中嵌套的代码会执行。
4. 如果with代码块引发异常，\_\_exit\_\_(type, value, traceback)方法就会被调用。
5. 如果with代码块没有引发异常，\_\_exit\_\_方法依然会被调用，其type、value以及traceback参数会以None传递。

class **TraceBlock**:

def **message**(*self*, arg):

print *'running'*, arg

def **\_\_enter\_\_**(*self*):

print *'starting with block'*

return *self*

def **\_\_exit\_\_**(*self*, exc\_type, exc\_value, exc\_tb):

if exc\_type is None:

print *'exited normally\n'*

else:

print *'raise an exception!'*, exc\_type

return False

with TraceBlock() as action:

action.message(*'test 1'*)

print *'reached'*

with TraceBlock() as action:

action.message(*'test 2'*)

raise TypeError

print *'not reached'*

*Output：*

Traceback (most recent call last):

File "C:\E\workspace\PythonLearning\chapter17\withas.py", line 20, in <module>

raise TypeError

TypeError

starting with block

running test 1

reached

exited normally

starting with block

running test 2

raise an exception! <type 'exceptions.TypeError'>

## 常用重载运算符

### \_\_dict\_\_

\_\_dict\_\_获取模块命名空间字典。

例如文件M.py

import N

X = 100

class A:

X = 100

def func():

print "M"

>>> M.\_\_dict\_\_

{'\_\_builtins\_\_': {'bytearray': <type 'bytearray'>, 'IndexError': <type 'exceptions.IndexError'>, 'all': <built-in function all>,......（省略） 'WindowsError': <type 'exceptions.WindowsError'>}, '\_\_file\_\_': 'C:\\E\\M.pyc', '\_\_package\_\_': None, 'func': <function func at 0x0000000002760AC8>, '\_\_name\_\_': 'M', '\_\_doc\_\_': None}

**获取对象命名空间属性的字典。**

>>> M.A.\_\_dict\_\_

{'X': 100, '\_\_module\_\_': 'M', '\_\_doc\_\_': None}

>>> M.func.\_\_dict\_\_

{}

### \_\_all\_\_

\_\_all\_\_列表是指当包或模块名称使用from \*的时候，应该导入的模块或变量清单。

例如：

dir0

——dir1

——M.py

——N.py

——\_\_init\_\_.py

——dir2

——O.py

——\_\_init\_\_.py

dir1目录下\_\_init\_\_.py：

\_\_all\_\_ = ['M']

>>> from dir1 import \*

>>> dir()

['M', 'X', 'Y', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'sys'] # 只导入了M模块，没有导入N模块

文件N.py:

\_\_all\_\_ = ['Y']

Y = 100

Z = 200

>>> from dir1.N import \*

>>> dir()

['M', 'N', 'X', 'Y', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'dir1', 'sys'] # 变量Z没有被导入

>>> from dir1.N import Y

>>> from dir1.N import Z

Traceback (most recent call last):

File "<pyshell#48>", line 1, in <module>

from dir1.N import Z

ImportError: cannot import name Z

*注意：把下划线放在变量之前（例如，\_X），可以防止客户端使from \*语句导入模块名时，把其中的那些变量名复制进去。*

### \_\_name\_\_

* 如果文件是以顶层文件执行，在启动时，\_\_name\_\_就会设置为字符串”\_\_main\_\_”.
* 如果文件被导入，\_\_name\_\_就会改设成客户端所了解的模块名。

### \_\_future\_\_

from \_\_future\_\_ import *featurename*

python中启用未来的语言特性。

例如：

>>> 10/3

3

>>> from \_\_future\_\_ import division

>>> 10/3

3.3333333333333335

或者：

>>> from \_\_future\_\_ import print\_function

>>> print("Hello", "My", "Love", end='!', sep=' ')

Hello My Love!

### \_\_class\_\_

实例C对应的类（仅新式类）.

>>> class A:

pass

>>> A.\_\_class\_\_

Traceback (most recent call last):

File "<pyshell#3>", line 1, in <module>

A.\_\_class\_\_

AttributeError: class A has no attribute '\_\_class\_\_'

>>>

>>> a = A()

>>> a.\_\_class\_\_

<class \_\_main\_\_.A at 0x0000000002EAB2E8>

>>>

>>> class A(object):

pass

>>> a = A()

>>> A.\_\_class\_\_

<type 'type'>

>>> a.\_\_class\_\_

<class '\_\_main\_\_.A'>

### \_\_bases\_\_

类C的所有父类构成的元组.

>>> C.\_\_bases\_\_

(<class \_\_main\_\_.A at 0x0000000002EAB2E8>, <class \_\_main\_\_.B at 0x0000000002EABB88>)

>>> c.\_\_bases\_\_

Traceback (most recent call last):

File "<pyshell#29>", line 1, in <module>

c.\_\_bases\_\_

AttributeError: C instance has no attribute '\_\_bases\_\_'

### \_\_mro\_\_

类继承的查找顺序元组。

### \_\_slots\_\_

将字符串属性名称赋值给特殊的\_\_slots\_\_类属性，只有\_\_slots\_\_列表内的这些变量名可被赋值为**实例属性**。

>>> class Limter(object):

\_\_slots\_\_ = ['a', 'b']

>>> x = Limter()

>>> x.a

Traceback (most recent call last):

File "<pyshell#90>", line 1, in <module>

x.a

AttributeError: a

>>> x.a = 1

>>> x.b = 2

>>> x.d = 3

Traceback (most recent call last):

File "<pyshell#93>", line 1, in <module>

x.d = 3

AttributeError: 'Limter' object has no attribute 'd'

>>> Limter.\_\_dict\_\_

dict\_proxy({'a': <member 'a' of 'Limter' objects>, '\_\_module\_\_': '\_\_main\_\_', 'b': <member 'b' of 'Limter' objects>, '\_\_slots\_\_': ['a', 'b'], '\_\_doc\_\_': None})

使用\_\_slots\_\_时，实例通常没有\_\_dict\_\_属性。

>>> x.\_\_dict\_\_

Traceback (most recent call last):

File "<pyshell#96>", line 1, in <module>

x.\_\_dict\_\_

AttributeError: 'Limter' object has no attribute '\_\_dict\_\_'

通过在\_\_slots\_\_中包含\_\_dict\_\_仍然可以容纳额外的属性。

>>> class D:

\_\_slots\_\_ = ['a', 'b', '\_\_dict\_\_']

c = 3

def \_\_init\_\_(self):

self.d = 4

>>> X = D()

>>> X.d

4

>>> X.\_\_dict\_\_

{'d': 4}

>>> D.\_\_dict\_\_

{'c': 3, '\_\_module\_\_': '\_\_main\_\_', '\_\_slots\_\_': ['a', 'b', '\_\_dict\_\_'], '\_\_init\_\_': <function \_\_init\_\_ at 0x0000000002F28048>, '\_\_doc\_\_': None}

>>> X.c

3

>>> X.a = 1

>>> getattr(X, 'a',)

1

>>> getattr(X, 'c')

3

>>> getattr(X, 'd')

4

>>> for attr in list(X.\_\_dict\_\_)+X.\_\_slots\_\_:

print attr, '=>', getattr(X, attr, None)

a => 1

d => 4

a => 1

b =>None

* 如果一个子类继承自一个没有\_\_slots\_\_的超类，那么超类的\_\_dict\_\_属性总是可以访问的，使得子类的\_\_slots\_\_无意义。

>>> class A:pass

>>> class B(A):

\_\_slots\_\_ = ['a', 'b']

>>> X = B()

>>> X.a = 1

>>> X.b = 2

>>> X.d = 3 # 子类的\_\_slots\_\_无意义

* 如果一个类定义了与超类相同的slot名称，超类slot定义的名称版本只有通过直接从超类获取其描述符才能访问。

>>> X = B()

>>> X.a = 1

>>> X.b = 2

>>> X.c = 3

>>> A.\_\_dict\_\_

{'\_\_module\_\_': '\_\_main\_\_', '\_\_slots\_\_': ['a', 'b'], '\_\_doc\_\_': None}

>>> A.\_\_slots\_\_

['a', 'b']

>>> B.\_\_slots\_\_

['a', 'c']

>>> B.\_\_dict\_\_

{'\_\_module\_\_': '\_\_main\_\_', '\_\_slots\_\_': ['a', 'c'], '\_\_doc\_\_': None}

>>> X.\_\_dict\_\_

{'a': 1, 'c': 3, 'b': 2}

在类中定义 \_\_slots\_\_ 属性的目的是告诉解释器：“这个类中的所有实例属性都在这儿

了！”这样，Python 会在各个实例中使用类似元组的结构存储实例变量，从而避免使用消

耗内存的 \_\_dict\_\_ 属性。如果有数百万个实例同时活动，这样做能节省大量内存。