第十三章 面向对象编程

介绍

Python中的面向对象主要有两个主题:**类和类实例**

```
class MyNewObjectType(bases):
    'define MyNewObjectType class'
    class_suite #类体
bases可以是一个单继承或多重继承用于继承的父类
```

实例化

```
myFirstObject = MyNewObjectType()
```

类名使用我们所熟悉的函数操作符(())，以“函数调用”的形式出现。

只要你需要，类可以很简单，也可以很复杂。**最简单的情况， 类仅用作名称空间（ namespaces ）,这意味着你把数据保存在变量中，对他们按名称空间进行分组**，使得他们处于同样的关系空间中——所谓的关系是使用标准 Python 句点属性标识。

示例

```
class MyData(object):
    pass

mathObj = MyData()
mathObj.x = 4
mathObj.y = 5
print mathObj.x+mathObj.y #9
#这些属性x,y实质上是动态的：不需要在构造器中,或其它任何地方为它们预先声明或者赋值。
```

方法的定义

```
class MyDataWithMethod(object): # 定义类
    def printFoo(self): # 定义方法
        print 'You invoked printFoo()!'
```

self 参数，它在所有的方法声明中都存在。(实例方法一般需要`self`参数，而静态方法或类方法不会)。`self`(实例对象)参数自动由解释器传递.

类似于C++中的`this`另外，函数调用约定(即规定了方法参数的入参规则和堆栈清理的规则)
有`__cdecl,__fastcall,__pascal,__,__thiscall,__stdcall`等.

`__cdecl`调用约定，又称C调用约定，是 C/C++ 语言缺省的调用约定。参数按照从右至左的方式入栈，函数本身不清理栈，此工作由调用者负责。

`__stdcall`调用月id那个，参数按照从右至左的方式入栈，函数自身清理堆栈

特殊的方法

`__init__()`类似于类构造器。
Python 创建实例后，在实例化过程中，调用`__init__()`方法,主要是在实例被创建后，实例化调用**返回这个实例之前，去执行某些特定的任务或设置。**

类定义

```
class AddrBookEntry(object): # 类定义
    'address book entry class'
    def __init__(self, nm, ph): # 定义构造器
        self.name = nm # 设置 name
        self.phone = ph # 设置 phone
        print 'Created instance for:', self.name
    def updatePhone(self, newph): # 定义方法
        self.phone = newph
        print 'Updated phone# for:', self.name
```

创建子类

```
class EmplAddrBookEntry(AddrBookEntry):
    'Employee Address Book Entry class'#员工地址本类
    def __init__(self, nm, ph, id, em):
        AddrBookEntry.__init__(self, nm, ph)
        self.empid = id
        self.email = em
    def updateEmail(self, newem):
        self.email = newem
        print 'Updated e-mail address for:', self.name
```

如果需要，每个子类最好定义它自己的构造器，不然，基类的构造器会被调用。
然而，如果子类重写基类的构造器，基类的构造器就不会被自动调用了

面向对象编程

常用术语
抽象/实现，
封装/接口–Python中所有的类属性都是公开的
合成–组合
派生/继承/继承机构
泛化/特化
多态
自省/反射–**如何在运行期取得自身信息的**

类属性

数据属性
称为静态属性或者静态数据,这种类型的数据相当于在一个变量声明前加上`static`关键字。

方法
方法，仅仅是一个作为类定义一部分定义的函数. (这使得方法成为类属性)。

Python 严格要求， 没有实例， 方法是不能被调用的。

不管是否绑定，方法都是它所在的类的固有属性，即使它们几乎总是通过实例来调用的。

特殊的类属性

`C.__name__`是给定类的字符串名字

`C.__doc__`是文档字符串

`C.bases`包含了一个由所有父类组成的元组

`C.__dict__`由类的数据属性组成的(包含方法),访问一个类属性的时候，Python 解释器将会搜索字典以得到需要的属性。 如果在`__dict__`中没有找到， 将会在基类的字典中进行搜索，采用“深度优先搜索”顺序。基类集的搜索是按顺序的， 从左到右，按其在类定义时，定义父类参数时的顺序。

`C.module`–类 C 定义所在的模块。Python 支持模块间的类继承,类名就完全由模块名所限定

`C.__class__`–实例C对应的类。发现它就是一个类型对象的实例。换句话说，一个类已是一种类型了。

实例

Python2.2类和类型做了同一，即类型type包括类类型class,整型int,长整型long,浮点型float等。

`__init__()`构造器方法

`__init__(self)`方法是在创建实例(对构造器`__init__()`的调用)后调用的第一个方法。

Called after the instance has been created (by **new**()), but before it is returned to the caller.

`__new__()`‘构造器’方法

If **new**() returns an instance of cls, then the new instance’s **init**() method will be invoked like **init**(self[, …]), where self is the new instance and the remaining arguments are the same as were passed to **new**().

If **new**() does not return an instance of cls, then the new instance’s **init**() method will not be invoked.

new() is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

> 关于 `__new__()` 和 `__init__()` 的总结
> 构造一个对象(实例化一个对象)分为两步:(1)分配内存-与C++中的 `::operator new` 的作用类似,底层调用 `std::alloc` 来为对象分配内存(2)构造对象–这类似于C++，Java中的默认构造函数
>
> 那么在Python中(1)对应于 `__new__()` (2)对应于 `__init__()` 。即先分配内存(并返回成功与否)，然后在已经分配的内存上初始化对象。这也就是为什么 `__init__()` 必须在 `__new__()` 之后调用的原因。

`__del__()` '解构器'方法

解构器(析构器)在类 C 实例所有的引用都被清除掉后，才被调用的
`del x` 仅仅表示减少x的一个引用计数，只有在引用计数为0时才调用解构器

实例属性

实例仅拥有数据属性(**方法严格来说是类属性**)，后者只是与某个类的实例相关联的数据值，并且可以通过句点属性标识法来访问。这些值独立于其它实例或类。

设置实例的属性可以在实例创建后任意时间进行，也可以在能够访问实例的代码中进行

能够在“运行时”创建实例属性，是 Python 类的优秀特性之一。

Python 不仅是动态类型，而且在运行时，允许这些对象属性的动态创建。

在构造器中设置实例属性

`__init__()`

默认参数提供默认的实例安装

在`__init__()`实例化方法中提供含有默认值参数的方法

`__init__()`应当返回 None

如果定义了构造器，它不应当返回任何对象，因为实例对象是自动在实例化调用后返回的。相应地，`__init__()`就不应当返回任何对象应当为`None`；

13.6.2 查看实例属性

内建函数`dir()`可以显示类属性

实例也有一个`__dict__`特殊属性，它是实例属性构成的一个字典，键是属性名，值是属性相应的数据值。

实例仅有两个特殊属性，对于任意对象I，
`I.__class__`实例化I类
`I.__dict__`I的类属性

字典中仅有实例属性，没有类属性或特殊属性。

> 如果需要修改`__dict__`的值可以通过重载`__setattr__()`特殊方法来实现。不过不推荐使用。

内建类型中不包含`__dict__`属性

13.6.5 实例属性和类属性

类属性仅是与类相关的数据值，和实例属性不同，类属性和实例无关。
类和实例都是名字空间。类是类属性的名字空间，实例则是实例属性的。

访问类属性

类属性的访问可以通过`C.attrname-类属性访问`或者`c.attrname-实例属性访问`都可以访问。
只有当使用类引用 version 时(`C.attrname`访问)才能更新它的值。如果使用实例属性访问的方式去更新类属性时，是不成功的，因为，实例属性的访问会阻止类属性的访问，这样实例属性的访问有效地遮蔽了类属性。
类属性和实例属性作用域访问规则:`c.attrname`方式对变量的访问是现在实例中查找，然后在类中查找。

给一个与类属性同名的实例属性赋值，我们会有效地“隐藏” 类属性

赋值语句右边的表达式计算出原类的变量，增加 0.2，并且把这个值赋给新创建的实例属性
`foo.x = Foo.x + 0.2`–一切的原因是作用域的访问规则使然。

怎样使类属性不可变？

13.7 从这里开始校对–绑定和方法调用

关于方法的说明

1. 方法仅仅是类内部定义的函数。(这意味着方法是类属性而不是实例属性)。
2. 方法只有在其所属的类拥有实例时，才能被调用。
   当存在一个实例时，方法才被认为是绑定到那个实例了(通过方法声明中的第一个参数`self`实现)。没有实例时方法就是未绑定的。
3. 任何一个方法定义中的第一个参数都是变量 self，它表示调用此方法的实例对象。

调用绑定方法

`self`–`self`变量用于在类实例方法中引用方法所绑定的**实例**。

当你还没有一个实例并且需要调用一个非绑定方法的时候你必须传递`self`参数。

调用绑定方法只需方法生命中第一个参数为`self`参数，然后有一个类实例就可以实现调用绑定方法。

调用非绑定方法

需要调用一个还没有任何实例的类中的方法的一个主要的场景
是:你在派生一个子类,而且你要覆盖父类的方法，这时你需要调用那个父类中想要覆盖掉的构造方法。

```
class EmplAddrBookEntry(AddrBookEntry):
    'Employee Address Book Entry class' # 员工地址记录条目
    def __init__(self, nm, ph, em):
        AddrBookEntry.__init__(self, nm, ph)
        self.empid = id
        self.email = em
```

在子类构造器中调用父类的构造器并且明确地传递（父类）构造器所需要的 self 参数（因为我们没有一个父类的实例）。
子类中`__init__()`的第一行就是对父类`__init__()`的调用。

这里的`self`暗指AddrBookEntry类
`AddrBookEntry.__init__(self, nm, ph)`–通过类绑定方法–这里的实现类似于多态？待确认
`add.update(nm, ph)`–通过`self`通过实例绑定方法

静态方法和类方法

对于类方法而言，需要类而不是实例作为第一个参数，它是由解释器传给方法。类不需要特别地命名, 类似 self，不过很多人使用 cls 作为变量名字。

在经典类中创建静态方法和类方法

```
class TestStaticMethod:
    def foo():
        print 'calling static method foo()'
        foo = staticmethod(foo)

class TestClassMethod:
    def foo(cls):
        print 'calling class method foo()'
        print 'foo() is part of class:', cls.__name__
        foo = classmethod(foo)

# 对应的内建函数被转换成它们相应的类型，并且重新赋值给了相同的变量名。
```

13.8.2 使用函数修饰符–装饰器

在 Python2.4 中加入的新特征，你可以用它把一个函数应用到另个函数对象上, 而且新函数对象依然绑定在原来的变量。我们正是需要它来整理语法。
通过使用 decorators，我们可以避免像上面那样的重新赋值。

```
class TestStaticMethod:
    @staticmethod
    def foo():
        print 'calling static method foo()'

class TestClassMethod:
    @classmethod
    def foo(cls):
        print 'calling class method foo()'
        print 'foo() is part of class:', cls.__name__
```

组合–has a 关系

使用类的两种方式

1. 组合–类之间的关系比较弱时
2. 派生–类之间的关系比较强时

子类和派生

如果你的类没有从任何祖先类派生，可以使用 object 作为父类的名字。

```
class SubClassName(ParentClass1[, ParentClass2, ...]):
    'optional class documentation string'
    class_suite
```

继承

一个子类可以继承它的基类的任何属性，不管是数据属性还是方法。(除了文档字符串`__doc__`)

`__bases__`类属性

`__bases__`类属性，对任何(子）类,它是一个包含其父类的集合的元组。没有父类的类，它们的`__bases__`属性为空。

通过继承覆盖overriding方法

能否调用那个被我覆盖的基类方法呢？
使用内建法方法`super()`

```
# 在Derived类中调用一下方法
super(D, self).method()
```

当从一个带构造器`__init()__`的类派生，如果你不去覆盖`__init__()`，它将会被继承并自动调用。
但是子类重写`__init__()`构造器方法后基类的`__init__()`方法就不会被自动调用。

```
# 在Derived类中调用父类的构造方法
super(D, self).__init__()
```

从标准类型派生

覆盖了`__new__()`特殊方法来定制我们的对象，使之和标准 Python 浮点数（float）有一些区别： 我们使用 round()内建函数对原浮点数进行舍入操作

```
class RoundFloat(float):
    # 通过调用父类的构造器来创建真实的对象的， float.__new__()
    def __new__(cls, val):
        return float.__new__(cls, round(val, 2))

class RoundFloat(float):
    def __new__(cls, val):
        return super(RoundFloat, cls).__new__(cls, round(val, 2))
```

所有的`__new()__`方法都是类方法(`@staticmethod`修饰)我们要显式传入类为第一个参数。

多重继承

当使用多重继承时,有两个不同的方面

1. 找到合适的属性
2. 当重写方法时，如何调用相应父类方法;并在子类中处理好自己的义务

Method Resolution Order(MRO)–方法解释顺序

在 Python 2.2 以前的版本中，算法非常简单：**深度优先，从左至右进行搜索**，取得在子类中使用的属性。

新的MRO算法是采用广度优先，而不是深度优先。
论文：Python 2.3 方法解释顺序
http://python.org/download/releases/2.3/mro/

新式类也有一个**mro**属性，告诉你查找顺序是怎样的

多重继承只限用在对两个完全不相关的类进行联合。这就是术语 mixin 类（或者“mix-ins”）的由来。

多重继承导致了一个菱形继承的问题
C++中采用了虚拟继承`virtual`来处理

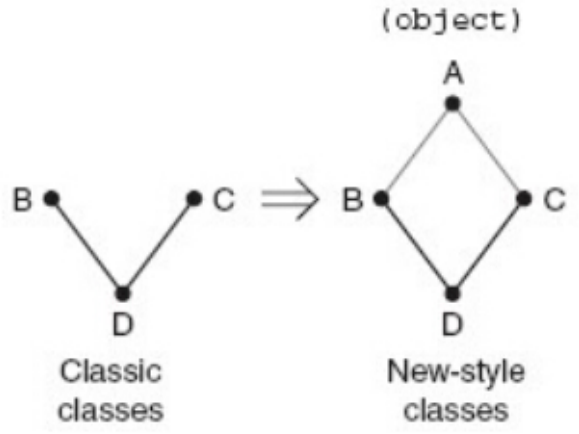

图 13.3 继承的问题是由于在新式类中，需要出现基类，这样就在继承结构中，形成了一个菱形。D 的实例上溯时，不应当错过 C，但不能两次上溯到 A（因为 B 和 C 都从 A 派生）。去读读 Guido van Rossum 的文章中有关"协作方法"的部分，可以得到更深地理解。

在处理菱形继承问题上，Python是怎样处理不能两次上溯到基类的?因为这样的问题就会产生(MRO的问题)
Guido van Rossum的文章中有关**协作方法**

经典类，使用深度优先算法。因为新式类继承自 object，新的菱形类继承结构出现，问题也就接着而来了，所以必须新建一个MRO。

类、实例、和其他对象的内建函数

`issubclass(sub,sup)`–判断一个类是另一个类的子类或子孙类
一个类可视为其自身的子类–此判断也会返回`True`

`isinstance(obj1, obj2)`–在 obj1 是类 obj2 的一个实例，或者是 obj2 的子类的一个实例时，返回 `True`

> 第二个参数应当是类

调用 Python 的 `isinstance()` 不会有性能上的问题，主要是因为它只用来**快速搜索类族集成结构**，以确定调用者是哪个类的实例，还有更重要的是，它是用 C 写的！

`hasattr(), getattr(),setattr(), delattr()`

*attr()系列函数可以在各种对象下工作，不限于类(class)和实例(instances)。

当使用这些函数时，你传入你正在处理的对象作为第一个参数，但属性名，也就是这些函数的第二个参数，是这些属性的字符串名字。
换句话说，在操作 `obj.attr` 时，就相当于调用 `*attr(obj,'attr'....)` 系列函数

```
hasattr(instance, 'foo')
getattr(instance, 'foo')
setattr(instance, 'bar', 'my attr')
```

`super()`

找出相应的父类

`vars()`

`vars()` 内建函数与 `dir()` 相似，只是给定的对象参数都必须有一个 `__dict__` 属性。
vars()返回一个字典，它包含了对象存储于其 `__dict__` 中的属性(键)及值。

如果没有提供对象作为 vars()的一个参数，它将显示一个包含本地
名字空间的属性（键）及其值的字典， 也就是， `locals()`

用特殊方法定制类

构造器和析构器–`__init__(),__del()__`

自定义特殊方法可以实现:

- 模拟标准类型
- 重载操作符

特殊方法允许类通过重载标准操作符`+`,`*`,甚至包括分段下标及映射操作操作`[]`来模拟标准类型。
这些方法都是以双下划线开始以及结尾的

用来定制类的特殊方法（只列出一点P526）

特殊方法	描述
`C.__init__(self[,arg1, ...])`	构造器–实例化
`C.__new__(self[,arg1, ...])`	构造器–内存分配
`C.__del__(self)`	析构器
`C.__str__(self)`	内建 str()及 print 语句
`C.__repr__(self)`	内建 repr() 和‘ ‘ 操作符
`C.__unicode__(self)`	内建 unicode()

`C.__call__(self, *arg1)`	表示可调用的实例
`C.__nonzero__(self)`	为object 定义 False 值；内建 bool()
`C.__len__(self)`	内建 len()

属性组帮助管理类的实例属性。

简单定制

```
class RoundFloatManual(object):
    '''重写了__init()__,__str()__'''
    def __init__(self, val):
        assert isinstance(val, float), "Value must be a float!"
        self.value = round(val,2)

    def __str__(self):
        #return str(self.value)
        return '%.2f' % self.value

    __repr = __str__
```

迭代器

```
# 任意项迭代器
from random import choice
class AnyIter(object):
    def __init__(self, data, safe=False):
        self.safe = safe
        self.iter = iter(data)

    def __iter__(self):
        return self

    def next(self, howmany = 1):
        retval = []
        for each in range(howmany):
            try:
                retval.append(self.iter.next())
            except StopIteration:
                if self.safe:
                    break
                else:
                    raise
        return retval
```

多类型定制 `NumStr`

一些方法的重写

```
class NumStr(object):
    def __init__(self, num=0, string=''):
        self.__num = num
        self.__string = string

    def __str__(self):
        return '[%d::%r]' % (self.__num, self.__string)

    __repr__ = __str__  #调用__repr__()是会调用__str__()

    def __add__(self, other):
        if isinstance(other, NumStr):
            return self.__class__(self.__num+other.__num,self.__string+other.__string)
        else:
            raise  TypeError,'Illegal argument type for built-in operatation'

    def __mul__(self, num):
        if isinstance(num, int):
            return self.__class__(self.__num*num,self.__string*num)
        else:
            raise  TypeError,'Illegal argument type for built-in operatation'
    def __nonzero__(self):
        return self.__num or len(self.__string)

    def __norm_cval(self, cmpres):
        return cmp(cmpres, 0)

    def __cmp__(self, other):
        return self.__norm_cval(cmp(self.__num,other.__num)+self.__norm_cval(cmp(self.__string,othe
```

```
    r.__string)))
```

私有化

默认情况下，属性在 Python 中都是`public`。
Python中对数据的访问控制。

双下划线–属性

Python 为类元素（属性和方法）的私有性提供初步的形式。由双下划线开始的属性在运行时被“混淆”，所以直接访问是不允许的。实际上，会在名字前面加上下划线和类名。比如，以例`self.__num`属性为例，被“混淆”后，用于访问这个数据值的标识就变成了`self._NumStr__num`–以此形成了类的作用域。
把类名加上后形成的新的“混淆”结果将可以防止在祖先类或子孙类中的同名冲突。

> 对于类中的属性和方法采用语言自由的一套规则，在运行时将属性和方法进行重命名，从而实现对数据的访问控制。

单下划线–属性钱使用单下划线表示模块级私有化

都是基于作用域的

Python可以按你的需要严格地定制访问权

13.15 授权

授权的过程，即是所有更新的功能都是由新类的某部分来处理，但已存在的功能就授权给对象的默认属性。–定制化一样

```
from time import time, ctime
class TimeWrapMe(object):
    def __init__(self, obj):
        self.__data = obj
        self.__ctime = self.__mtime = self.__atime = time()

    def get(self):
        self.__atime = time()
        return self.__data

    def __getattr__(self, attr): # delegate
        self.__atime = time()
        return getattr(self.__data, attr)

    def gettimeval(self,t_type):
        if not isinstance(t_type,str) or t_type[0] not in 'cma':
            raise TypeError,"argument of 'c','m' or 'a' req'd"
        return getattr(self,'_%s__%stime' % (self.__class__.__name__,t_type[0]))

    def gettimestr(self, t_type):
        return ctime(self.gettimeval(t_type))

    def set(self, obj):
        self.__data = obj
        self.__mtime = self.__atime = time() # 更新修改时间和访问时间

    def __repr__(self):
        self.__atime = time()
        return  `self.__data`
```

```
    def __str__(self):
        self.__atime = time() # 更新访问时间
        return str(self.__data)
```

新式类的高级特性

13.16.2 `__slots__`类属性

`__dict__`属性跟踪所有实例属性
举例来说，你有一个实例 inst.它有一个属性 foo，那使用 `inst.foo` 来访问它与使用 `inst.__dict__['foo']`来访问是一致的。

字典会占据大量内存，如果你有一个属性数量很少的类，但有很多实例，那么正好是这种情况。
为内存上的考虑，用户现在可以使用`__slots__`属性来替代`__dict__`。

基本上，`__slots__`是一个类变量，由一序列型对象组成，由所有合法标识构成的实例属性的集合来表示。

任何试图创建一个其名不在`__slots__`中的名字的实例属性都将导致 `AttributeError` 异常。

这种特性的主要目的是节约内存。其副作用是某种类型的”安全”,它能防止用户随心所欲的动态增加实例属性。

13.16.3 `__getattribute__()`特殊方法

Python 类有一个名为`__getattr__()`的特殊方法，它仅当属性不能在实例的`__dict__`或它的类（类的`__dict__`），或者祖先类（其`__dict__`）中找到时， 才被调用。

如果类同时定义了`__getattribute__()`及`__getattr__()`方法,除非明确从`__getattribute__()`调用，或_`_getattribute__()`引发了`AttributeError`异常,否则后者不会被调用.

为了安全地访问任何它所需要的属性,你总是应该调用祖先类的同名方法；比如， `super(obj,self).__getattribute__(attr)`。

13.16.4 描述符

描述符是 Python 新式类中的关键点之一。它为对象属性提供强大的 API。你可以认为**描述符是表示对象属性的一个代理。**

明确删除掉某个属性时会调`__delete__()`方法,很少被实现。

那些同时覆盖`__get__()``及__set__()`的类被称作**数据描述符**

整个描述符系统的心脏是`__getattribute__()`，因为对每个属性的实例都会调用到这个特殊的方法。

`x.foo`由`__getattribute__()`转化成：
`type(x).__dict__['foo'].__get__(x,type(x))`

如果类调用了`__get__()`方法，那么`None`将作为对象被传入(对于实例， 传入的是`self`)：
`X.__dict__['foo'].__get__(None, X)`

优先级别

`__getattribute__()`方法的执行方式有以下优先级

- 类属性
- 数据描述符(实现了`__get__()`和`__set__()`方法的描述符)
  *实例属性(`___dict__`对象的值)
- 非数据描述符(非数据描述符的目的只是当实例属性值不存在时，提供一个 值 而已。)
- 默认为`__getattr__()`

> 函数是非数据描述符，实例属性有更高的优先级

静态方法、类方法、属性(见下面一节)，甚至所有的函数都是描述符。

有内置的函数、用户自定义的函数、类中定义的方法、静态方法、类方法。这些都是函数的例子。 它们之间唯一的区别在于调用方式的不同。

属性和`property()`内建函数

属性是一种有用的特殊类型的描述符。它们是用来处理所有对实例属性的访问。
当你使用点属性符号来处理一个实例属性时，其实你是在修改这个实例的`__dict__`属性。

表面上来看，你使用`property()`访问和一般的属性访问方法没有什么不同，但实际上这种访问的实现是不同的 — 它使用了函数(或方法)。

如果要处理大量的实例属性时，使用那些特殊属性时会使代码变得臃肿。

`property()`内建函数的语法
`property(fget=None, fset=None, fdel=None, doc=None)`

实际上，property()是在它所在的类被创建时被调用的，这些传进来的(作为参数的)方法是非绑定的，所以这些方法其实就是函数。

13.16.5 Metaclass和metaclass

创建的元类用于改变类的默认行为和创建方式。

在执行类定义的时候，将检查此类正确的(一般是默认的)元类，元类(通常)传递三个参数(到构造器):**类名，从基类继承数据的元组，和(类的)属性字典。**

创建一个新风格的类或传统类的通用做法是使用系统自己所提供的元类的默认方式。

用户一般都不会觉察到元类所提供的创建类(或元类实例化)的默认模板方式。
关于元类–PEPs 252 和 253

相关模块和文档

`types` 模块
`operator` 模块