描述器使用指南

发布 *3.9.21*

Guido van Rossum
and the Python development team

十二月 09, 2024

Python Software Foundation
Email: docs@python.org

Contents

作者 Raymond Hettinger（译者：wh2099 at outlook dot com）

联系方式 <python at rcn dot com>

目录

描述器让对象能够自定义属性查找、存储和删除的操作。

本指南主要分为四个部分：

1) “入门”部分从简单的示例着手，逐步添加特性，从而给出基本的概述。如果你是刚接触到描述器，请从这里开始。
2) 第二部分展示了完整的、实用的描述器示例。如果您已经掌握了基础知识，请从此处开始。
3) 第三部分提供了更多技术教程，详细介绍了描述器如何工作。大多数人并不需要深入到这种程度。
4) 最后一部分有对内置描述器（用 C 编写）的纯 Python 等价实现。如果您想了解函数如何变成绑定方法或对 `classmethod()`，`staticmethod()`，`property()` 和 __slots__ 这类常见工具的实现感兴趣，请阅读此部分。

1 入门

现在，让我们从最基本的示例开始，然后逐步添加新功能。

1.1 简单示例：返回常量的描述器

类 `Ten` 是一个描述器，它的 `__get__()` 方法总是返回常量 `10`：

```
class Ten:
    def __get__(self, obj, objtype=None):
        return 10
```

要使用描述器，它必须作为一个类变量存储在另一个类中：

```
class A:
    x = 5                            # Regular class attribute
    y = Ten()                        # Descriptor instance
```

用交互式会话查看普通属性查找和描述器查找之间的区别：

```
>>> a = A()                          # Make an instance of class A
>>> a.x                              # Normal attribute lookup
5
>>> a.y                              # Descriptor lookup
10
```

在 `a.x` 属性查找中，点运算符会找到存储在类字典中的键 `x` 及对应的值 `5` 。在 `a.y` 查找中，点运算符会根据描述器实例的 `__get__` 方法将其识别出来，调用该方法并返回 `10` 。

请注意，值 `10` 既不存储在类字典中也不存储在实例字典中。相反，值 `10` 是在调用时才取到的。

这个简单的例子展示了一个描述器是如何工作的，但它不是很有用。在查找常量时，用常规属性查找会更好。

在下一节中，我们将创建更有用的东西，即动态查找。

1.2 动态查找

有趣的描述器通常运行计算而不是返回常量:

```
import os

class DirectorySize:

    def __get__(self, obj, objtype=None):
        return len(os.listdir(obj.dirname))

class Directory:

    size = DirectorySize()              # Descriptor instance

    def __init__(self, dirname):
        self.dirname = dirname          # Regular instance attribute
```

交互式会话显示查找是动态的，每次都会计算不同的，经过更新的返回值:

```
>>> s = Directory('songs')
>>> g = Directory('games')
>>> s.size                              # The songs directory has twenty files
20
>>> g.size                              # The games directory has three files
3
>>> os.remove('games/chess')            # Delete a game
>>> g.size                              # File count is automatically updated
2
```

除了说明描述器如何运行计算，这个例子也揭示了 `__get__()` 参数的目的。形参 *self* 接收的实参是 *size*，即 *DirectorySize* 的一个实例。形参 *obj* 接收的实参是 *g* 或 *s*，即 *Directory* 的一个实例。而正是 *obj* 让 `__get__()` 方法获得了作为目标的目录。形参 *objtype* 接收的实参是 *Directory* 类。

1.3 托管属性

描述器的一种流行用法是托管对实例数据的访问。描述器被分配给类字典中的公开属性，而实际数据作为私有属性存储在实例字典中。当访问公开属性时，会触发描述器的 `__get__()` 和 `__set__()` 方法。

在下面的例子中，*age* 是公开属性，*_age* 是私有属性。当访问公开属性时，描述器会记录下查找或更新的日志:

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAgeAccess:

    def __get__(self, obj, objtype=None):
        value = obj._age
        logging.info('Accessing %r giving %r', 'age', value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', 'age', value)
        obj._age = value
```

(下页继续)

(续上页)

```
class Person:

    age = LoggedAgeAccess()                # Descriptor instance

    def __init__(self, name, age):
        self.name = name                   # Regular instance attribute
        self.age = age                     # Calls __set__()

    def birthday(self):
        self.age += 1                      # Calls both __get__() and __set__()
```

交互式会话展示中，对托管属性 *age* 的所有访问都被记录了下来，但常规属性 *name* 则未被记录：

```
>>> mary = Person('Mary M', 30)          # The initial age update is logged
INFO:root:Updating 'age' to 30
>>> dave = Person('David D', 40)
INFO:root:Updating 'age' to 40

>>> vars(mary)                           # The actual data is in a private attribute
{'name': 'Mary M', '_age': 30}
>>> vars(dave)
{'name': 'David D', '_age': 40}

>>> mary.age                             # Access the data and log the lookup
INFO:root:Accessing 'age' giving 30
30
>>> mary.birthday()                      # Updates are logged as well
INFO:root:Accessing 'age' giving 30
INFO:root:Updating 'age' to 31

>>> dave.name                            # Regular attribute lookup isn't logged
'David D'
>>> dave.age                             # Only the managed attribute is logged
INFO:root:Accessing 'age' giving 40
40
```

此示例的一个主要问题是私有名称 *_age* 在类 *LoggedAgeAccess* 中是硬耦合的。这意味着每个实例只能有一个用于记录的属性，并且其名称不可更改。

1.4 定制名称

当一个类使用描述器时，它可以告知每个描述器使用了什么变量名。

在此示例中，`Person` 类具有两个描述器实例 *name* 和 *age*。当类 `Person` 被定义的时候，他回调了 *LoggedAccess* 中的 `__set_name__()` 来记录字段名称，让每个描述器拥有自己的 *public_name* 和 *private_name*：

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
```

(下页继续)

(续上页)

```
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r', self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', self.public_name, value)
        setattr(obj, self.private_name, value)

class Person:

    name = LoggedAccess()                # First descriptor instance
    age = LoggedAccess()                 # Second descriptor instance

    def __init__(self, name, age):
        self.name = name                 # Calls the first descriptor
        self.age = age                   # Calls the second descriptor

    def birthday(self):
        self.age += 1
```

交互交互式会话显示类 Person 调用了 __set_name__() 方法来记录字段的名称。在这里，我们调用 vars() 来查找描述器而不触发它：

```
>>> vars(vars(Person)['name'])
{'public_name': 'name', 'private_name': '_name'}
>>> vars(vars(Person)['age'])
{'public_name': 'age', 'private_name': '_age'}
```

现在，新类会记录对 *name* 和 *age* 二者的访问：

```
>>> pete = Person('Peter P', 10)
INFO:root:Updating 'name' to 'Peter P'
INFO:root:Updating 'age' to 10
>>> kate = Person('Catherine C', 20)
INFO:root:Updating 'name' to 'Catherine C'
INFO:root:Updating 'age' to 20
```

这两个 *Person* 实例仅包含私有名称：

```
>>> vars(pete)
{'_name': 'Peter P', '_age': 10}
>>> vars(kate)
{'_name': 'Catherine C', '_age': 20}
```

1.5 结束语

descriptor 就是任何一个定义了 `__get__()`，`__set__()` 或 `__delete__()` 的对象。

可选地，描述器可以具有 `__set_name__()` 方法。这仅在描述器需要知道创建它的类或分配给它的类变量名称时使用。（即使该类不是描述器，只要此方法存在就会调用。）

在属性查找期间，描述器由点运算符调用。如果使用 `vars(some_class)[descriptor_name]` 间接访问描述器，则返回描述器实例而不调用它。

描述器仅在用作类变量时起作用。放入实例时，它们将失效。

描述器的主要目的是提供一个挂钩，允许存储在类变量中的对象控制在属性查找期间发生的情况。

传统上，调用类控制查找过程中发生的事情。描述器反转了这种关系，并允许正在被查询的数据对此进行干涉。

描述器的使用贯穿了整个语言。就是它让函数变成绑定方法。常见工具诸如 `classmethod()`，`staticmethod()`，`property()` 和 `functools.cached_property()` 都作为描述器实现。

2 完整的实际例子

在此示例中，我们创建了一个实用而强大的工具来查找难以发现的数据损坏错误。

2.1 验证器类

验证器是一个用于托管属性访问的描述器。在存储任何数据之前，它会验证新值是否满足各种类型和范围限制。如果不满足这些限制，它将引发异常，从源头上防止数据损坏。

这个 `Validator` 类既是一个 abstract base class 也是一个托管属性描述器。

```
from abc import ABC, abstractmethod

class Validator(ABC):

    def __set_name__(self, owner, name):
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        return getattr(obj, self.private_name)

    def __set__(self, obj, value):
        self.validate(value)
        setattr(obj, self.private_name, value)

    @abstractmethod
    def validate(self, value):
        pass
```

自定义验证器需要从 `Validator` 继承，并且必须提供 `validate()` 方法以根据需要测试各种约束。

2.2 自定义验证器

这是三个实用的数据验证工具：

1) `OneOf` 验证值是一组受约束的选项之一。
2) `Number` 验证值是否为 `int` 或 `float`。根据可选参数，它还可以验证值在给定的最小值或最大值之间。
3) `String` 验证值是否为 `str`。根据可选参数，它可以验证给定的最小或最大长度。它还可以验证用户定义的 predicate。

```
class OneOf(Validator):

    def __init__(self, *options):
        self.options = set(options)

    def validate(self, value):
        if value not in self.options:
            raise ValueError(f'Expected {value!r} to be one of {self.options!r}')

class Number(Validator):

    def __init__(self, minvalue=None, maxvalue=None):
        self.minvalue = minvalue
        self.maxvalue = maxvalue

    def validate(self, value):
        if not isinstance(value, (int, float)):
            raise TypeError(f'Expected {value!r} to be an int or float')
        if self.minvalue is not None and value < self.minvalue:
            raise ValueError(
                f'Expected {value!r} to be at least {self.minvalue!r}'
            )
        if self.maxvalue is not None and value > self.maxvalue:
            raise ValueError(
                f'Expected {value!r} to be no more than {self.maxvalue!r}'
            )

class String(Validator):

    def __init__(self, minsize=None, maxsize=None, predicate=None):
        self.minsize = minsize
        self.maxsize = maxsize
        self.predicate = predicate

    def validate(self, value):
        if not isinstance(value, str):
            raise TypeError(f'Expected {value!r} to be an str')
        if self.minsize is not None and len(value) < self.minsize:
            raise ValueError(
                f'Expected {value!r} to be no smaller than {self.minsize!r}'
            )
        if self.maxsize is not None and len(value) > self.maxsize:
            raise ValueError(
                f'Expected {value!r} to be no bigger than {self.maxsize!r}'
            )
        if self.predicate is not None and not self.predicate(value):
            raise ValueError(
                f'Expected {self.predicate} to be true for {value!r}'
```

(下页继续)

(续上页)

```
            )
```

2.3 实际应用

这是在真实类中使用数据验证器的方法：

```
class Component:

    name = String(minsize=3, maxsize=10, predicate=str.isupper)
    kind = OneOf('wood', 'metal', 'plastic')
    quantity = Number(minvalue=0)

    def __init__(self, name, kind, quantity):
        self.name = name
        self.kind = kind
        self.quantity = quantity
```

描述器阻止无效实例的创建：

```
>>> Component('Widget', 'metal', 5)      # Blocked: 'Widget' is not all uppercase
Traceback (most recent call last):
    ...
ValueError: Expected <method 'isupper' of 'str' objects> to be true for 'Widget'

>>> Component('WIDGET', 'metle', 5)      # Blocked: 'metle' is misspelled
Traceback (most recent call last):
    ...
ValueError: Expected 'metle' to be one of {'metal', 'plastic', 'wood'}

>>> Component('WIDGET', 'metal', -5)     # Blocked: -5 is negative
Traceback (most recent call last):
    ...
ValueError: Expected -5 to be at least 0
>>> Component('WIDGET', 'metal', 'V')    # Blocked: 'V' isn't a number
Traceback (most recent call last):
    ...
TypeError: Expected 'V' to be an int or float

>>> c = Component('WIDGET', 'metal', 5)  # Allowed:  The inputs are valid
```

3 技术教程

接下来是专业性更强的技术教程，以及描述器工作原理的详细信息。

3.1 摘要

定义描述器，总结协议，并说明如何调用描述器。提供一个展示对象关系映射如何工作的示例。

学习描述器不仅能提供接触到更多工具集的途径，还能更深地理解 Python 工作的原理。

3.2 定义与介绍

一般而言，描述器是一个包含了描述器协议中的方法的属性值。这些方法有 `__get__()`，`__set__()` 和 `__delete__()`。如果为某个属性定义了这些方法中的任意一个，它就可以被称为 descriptor。

属性访问的默认行为是从一个对象的字典中获取、设置或删除属性。对于实例来说，`a.x` 的查找顺序会从 `a.__dict__['x']` 开始，然后是 `type(a).__dict__['x']`，接下来依次查找 `type(a)` 的方法解析顺序（MRO）。如果找到的值是定义了某个描述器方法的对象，则 Python 可能会重写默认行为并转而发起调用描述器方法。这具体发生在优先级链的哪个环节则要根据所定义的描述器方法及其被调用的方式来决定。

描述器是一个强大而通用的协议。它们是属性、方法、静态方法、类方法和 `super()` 背后的实现机制。它们在 Python 内部被广泛使用。描述器简化了底层的 C 代码并为 Python 的日常程序提供了一组灵活的新工具。

3.3 描述器协议

```
descr.__get__(self, obj, type=None) -> value

descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None
```

描述器的方法就这些。一个对象只要定义了以上方法中的任何一个，就被视为描述器，并在被作为属性时覆盖其默认行为。

如果一个对象定义了 `__set__()` 或 `__delete__()`，则它会被视为数据描述器。仅定义了 `__get__()` 的描述器称为非数据描述器（它们经常被用于方法，但也可以有其他用途）。

数据和非数据描述器的不同之处在于，如何计算实例字典中条目的替代值。如果实例的字典具有与数据描述器同名的条目，则数据描述器优先。如果实例的字典具有与非数据描述器同名的条目，则该字典条目优先。

为了使数据描述器成为只读的，应该同时定义 `__get__()` 和 `__set__()`，并在 `__set__()` 中引发 `AttributeError`。用引发异常的占位符定义 `__set__()` 方法使其成为数据描述器。

3.4 描述器调用概述

描述器可以通过 `d.__get__(obj)` 或 `desc.__get__(None, cls)` 直接调用。

但更常见的是通过属性访问自动调用描述器。

表达式 `obj.x` 在命名空间的链中查找 `obj` 的属性 `x`。如果搜索在实例 `__dict__` 之外找到描述器，则根据下面列出的优先级规则调用其 `__get__()` 方法。

调用的细节取决于 `obj` 是对象、类还是超类的实例。

3.5 通过实例调用

实例查找通过命名空间链进行扫描，数据描述器的优先级最高，其次是实例变量、非数据描述器、类变量，最后是 `__getattr__()`（如果存在的话）。

如果 `a.x` 找到了一个描述器，那么将通过 `desc.__get__(a, type(a))` 调用它。

点运算符的查找逻辑在 `object.__getattribute__()` 中。这里是一个等价的纯 Python 实现：

```
def object_getattribute(obj, name):
    "Emulate PyObject_GenericGetAttr() in Objects/object.c"
    null = object()
    objtype = type(obj)
    cls_var = getattr(objtype, name, null)
    descr_get = getattr(type(cls_var), '__get__', null)
    if descr_get is not null:
        if (hasattr(type(cls_var), '__set__')
            or hasattr(type(cls_var), '__delete__')):
            return descr_get(cls_var, obj, objtype)     # data descriptor
    if hasattr(obj, '__dict__') and name in vars(obj):
        return vars(obj)[name]                          # instance variable
    if descr_get is not null:
        return descr_get(cls_var, obj, objtype)         # non-data descriptor
    if cls_var is not null:
        return cls_var                                  # class variable
    raise AttributeError(name)
```

请注意，在 `__getattribute__()` 方法的代码中没有调用 `__getattr__()` 的钩子。这就是直接调用 `__getattribute__()` 或调用 `super().__getattribute__` 会彻底绕过 `__getattr__()` 的原因。

相反，当 `__getattribute__()` 引发 `AttributeError` 时，点运算符和 `getattr()` 函数负责调用 `__getattr__()`。它们的逻辑封装在一个辅助函数中：

```
def getattr_hook(obj, name):
    "Emulate slot_tp_getattr_hook() in Objects/typeobject.c"
    try:
        return obj.__getattribute__(name)
    except AttributeError:
        if not hasattr(type(obj), '__getattr__'):
            raise
    return type(obj).__getattr__(obj, name)             # __getattr__
```

3.6 通过类调用

像 `A.x` 这样的点操作符查找的逻辑在 `type.__getattribute__()` 中。步骤与 `object.__getattribute__()` 相似，但是实例字典查找改为搜索类的 method resolution order。

如果找到了一个描述器，那么将通过 `desc.__get__(None, A)` 调用它。

完整的 C 实现可在 Objects/typeobject.c 中的 `type_getattro()` 和 `_PyType_Lookup()` 找到。

3.7 通过 super 调用

super 的点操作符查找的逻辑在 `super()` 返回的对象的 `__getattribute__()` 方法中。

类似 `super(A, obj).m` 形式的点分查找将在 `obj.__class__.__mro__` 中搜索紧接在 `A` 之后的基类 `B`，然后返回 `B.__dict__['m'].__get__(obj, A)`。如果 `m` 不是描述器，则直接返回其值。

完整的 C 实现可以在 Objects/typeobject.c 的 `super_getattro()` 中找到。纯 Python 等价实现可以在 Guido's Tutorial 中找到。

3.8 调用逻辑总结

描述器的机制嵌入在 `object`，`type` 和 `super()` 的 `__getattribute__()` 方法中。

要记住的重要点是：

- 描述器由 `__getattribute__()` 方法调用。
- 类从 `object`，`type` 或 `super()` 继承此机制。
- 由于描述器的逻辑在 `__getattribute__()` 中，因而重写该方法会阻止描述器的自动调用。
- `object.__getattribute__()` 和 `type.__getattribute__()` 会用不同的方式调用 `__get__()`。前一个会传入实例，也可以包括类。后一个传入的实例为 `None`，并且总是包括类。
- 数据描述器始终会覆盖实例字典。
- 非数据描述器会被实例字典覆盖。

3.9 自动名称通知

有时，描述器想知道它分配到的具体类变量名。创建新类时，元类 `type` 将扫描新类的字典。如果有描述器，并且它们定义了 `__set_name__()`，则使用两个参数调用该方法。*owner* 是使用描述器的类，*name* 是分配给描述器的类变量名。

实现的细节在 Objects/typeobject.c 中的 `type_new()` 和 `set_names()`。

由于更新逻辑在 `type.__new__()` 中，因此通知仅在创建类时发生。之后如果将描述器添加到类中，则需要手动调用 `__set_name__()`。

3.10 ORM （对象关系映射）示例

以下代码展示了如何使用数据描述器来实现简单 object relational mapping 框架。

其核心思路是将数据存储在外部数据库中，Python 实例仅持有数据库表中对应的的键。描述器负责对值进行查找或更新：

```
class Field:

    def __set_name__(self, owner, name):
        self.fetch = f'SELECT {name} FROM {owner.table} WHERE {owner.key}=?;'
        self.store = f'UPDATE {owner.table} SET {name}=? WHERE {owner.key}=?;'

    def __get__(self, obj, objtype=None):
        return conn.execute(self.fetch, [obj.key]).fetchone()[0]

    def __set__(self, obj, value):
```

(下页继续)

(续上页)

```
        conn.execute(self.store, [value, obj.key])
        conn.commit()
```

我们可以用 `Field` 类来定义描述了数据库中每张表的模式的 models。

```
class Movie:
    table = 'Movies'                    # Table name
    key = 'title'                       # Primary key
    director = Field()
    year = Field()

    def __init__(self, key):
        self.key = key

class Song:
    table = 'Music'
    key = 'title'
    artist = Field()
    year = Field()
    genre = Field()

    def __init__(self, key):
        self.key = key
```

要使用模型，首先要连接到数据库：

```
>>> import sqlite3
>>> conn = sqlite3.connect('entertainment.db')
```

交互式会话显示了如何从数据库中检索数据及如何对其进行更新：

```
>>> Movie('Star Wars').director
'George Lucas'
>>> jaws = Movie('Jaws')
>>> f'Released in {jaws.year} by {jaws.director}'
'Released in 1975 by Steven Spielberg'

>>> Song('Country Roads').artist
'John Denver'

>>> Movie('Star Wars').director = 'J.J. Abrams'
>>> Movie('Star Wars').director
'J.J. Abrams'
```

4 纯 Python 等价实现

描述器协议很简单，但它提供了令人兴奋的可能性。有几个用例非常通用，以至于它们已预先打包到内置工具中。属性、绑定方法、静态方法、类方法和 __slots__ 均基于描述器协议。

4.1 属性

调用 `property()` 是构建数据描述器的简洁方式，该数据描述器在访问属性时触发函数调用。它的签名是：

```
property(fget=None, fset=None, fdel=None, doc=None) -> property
```

该文档显示了定义托管属性 `x` 的典型用法：

```
class C:
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

要了解 `property()` 如何根据描述器协议实现，这里是一个纯 Python 的等价实现：

```
class Property:
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

这个内置的 `property()` 每当用户访问属性时生效，随后的变化需要一个方法的参与。

例如，一个电子表格类可以通过 `Cell('b10').value` 授予对单元格值的访问权限。对程序的后续改进要求每次访问都要重新计算单元格；但是，程序员不希望影响直接访问该属性的现有客户端代码。解决方案是将对 value 属性的访问包装在属性数据描述器中：

```
class Cell:
    ...

    @property
    def value(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
```

在此示例中，内置的 `property()` 或我们实现的的 `Property()` 均适用。

4.2 函数和方法

Python 的面向对象功能是在基于函数的环境构建的。通过使用非数据描述器，这两方面完成了无缝融合。

在调用时，存储在类词典中的函数将被转换为方法。方法与常规函数的不同之处仅在于对象实例被置于其他参数之前。方法与常规函数的不同之处仅在于第一个参数是为对象实例保留的。按照惯例，实例引用称为 *self* ，但也可以称为 *this* 或任何其他变量名称。

可以使用 `types.MethodType` 手动创建方法，其行为基本等价于：

```
class MethodType:
    "Emulate PyMethod_Type in Objects/classobject.c"

    def __init__(self, func, obj):
        self.__func__ = func
        self.__self__ = obj

    def __call__(self, *args, **kwargs):
        func = self.__func__
        obj = self.__self__
        return func(obj, *args, **kwargs)
```

为了支持自动创建方法，函数包含 `__get__()` 方法以便在属性访问时绑定其为方法。这意味着函数其是非数据描述器，它在通过实例进行点查找时返回绑定方法，其运作方式如下：

```
class Function:
    ...

    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        if obj is None:
            return self
        return MethodType(self, obj)
```

在解释器中运行以下类，这显示了函数描述器的实际工作方式：

```
class D:
    def f(self, x):
         return x
```

该函数具有 qualified name 属性以支持自省：

```
>>> D.f.__qualname__
'D.f'
```

通过类字典访问函数不会调用 `__get__()`。相反，它只返回基础函数对象：

```
>>> D.__dict__['f']
<function D.f at 0x00C45070>
```

来自类的点运算符访问会调用 `__get__()`，直接返回底层的函数。

```
>>> D.f
<function D.f at 0x00C45070>
```

有趣的行为发生在从实例进行点访问期间。点运算符查找调用 `__get__()`，返回绑定的方法对象：

```
>>> d = D()
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

绑定方法在内部存储了底层函数和绑定的实例：

```
>>> d.f.__func__
<function D.f at 0x00C45070>

>>> d.f.__self__
<__main__.D object at 0x1012e1f98>
```

如果你曾好奇常规方法中的 *self* 或类方法中的 *cls* 是从什么地方来的，就是这里了！

4.3 方法的种类

非数据描述器为把函数绑定为方法的通常模式提供了一种简单的机制。

概括地说，函数对象具有 `__get__()` 方法，以便在作为属性访问时可以将其转换为方法。非数据描述器将 `obj.f(*args)` 的调用会被转换为 `f(obj, *args)` 。调用 `klass.f(*args)` 因而变成 `f(*args)` 。

下表总结了绑定及其两个最有用的变体：

转换形式	通过对象调用	通过类调用
function -- 函数	f(obj, *args)	f(*args)
静态方法	f(*args)	f(*args)
类方法	f(type(obj), *args)	f(cls, *args)

4.4 静态方法

静态方法返回底层函数，不做任何更改。调用 `c.f` 或 `C.f` 等效于通过 `object.__getattribute__(c, "f")` 或 `object.__getattribute__(C, "f")` 查找。这样该函数就可以从对象或类中进行相同的访问。

适合作为静态方法的是那些不引用 `self` 变量的方法。

例如，一个统计用的包可能包含一个实验数据的容器类。该容器类提供了用于计算数据的平均值，均值，中位数和其他描述性统计信息的常规方法。但是，可能有在概念上相关但不依赖于数据的函数。例如，`erf(x)` 是在统计中的便捷转换，但并不直接依赖于特定的数据集。可以从对象或类中调用它：`s.erf(1.5) --> .9332` 或 `Sample.erf(1.5) --> .9332`。

由于静态方法返回的底层函数没有任何变化，因此示例调用也是意料之中：

```
class E:
    @staticmethod
    def f(x):
        return x * 10
```

```
>>> E.f(3)
30
>>> E().f(3)
30
```

使用非数据描述器，纯 Python 版本的 `staticmethod()` 如下所示：

```
class StaticMethod:
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

4.5 类方法

与静态方法不同，类方法在调用函数之前将类引用放在参数列表的最前。无论调用方是对象还是类，此格式相同：

```
class F:
    @classmethod
    def f(cls, x):
        return cls.__name__, x
```

```
>>> F.f(3)
('F', 3)
>>> F().f(3)
('F', 3)
```

当方法仅需要具有类引用并且确实依赖于存储在特定实例中的数据时，此行为就很有用。类方法的一种用途是创建备用类构造函数。例如，类方法 `dict.fromkeys()` 从键列表创建一个新字典。纯 Python 的等价实现是：

```
class Dict(dict):
    @classmethod
    def fromkeys(cls, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = cls()
        for key in iterable:
            d[key] = value
        return d
```

现在可以这样构造一个新的唯一键字典：

```
>>> d = Dict.fromkeys('abracadabra')
>>> type(d) is Dict
True
```

(下页继续)

(续上页)

```
>>> d
{'a': None, 'b': None, 'r': None, 'c': None, 'd': None}
```

使用非数据描述器协议，纯 Python 版本的 `classmethod()` 如下：

```
class ClassMethod:
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, cls=None):
        if cls is None:
            cls = type(obj)
        if hasattr(type(self.f), '__get__'):
            return self.f.__get__(cls)
        return MethodType(self.f, cls)
```

`hasattr(type(self.f), '__get__')` 的代码路径在 Python 3.9 中被加入并让 `classmethod()` 可以支持链式装饰器。例如，一个类方法和特征属性可以被链接在一起：

```
class G:
    @classmethod
    @property
    def __doc__(cls):
        return f'A doc for {cls.__name__!r}'
```

```
>>> G.__doc__
"A doc for 'G'"
```

4.6 成员对象和 __slots__

当一个类定义了 `__slots__`，它会用一个固定长度的 slot 值数组来替换实例字典。从用户的视角看，效果是这样的：

1. Provides immediate detection of bugs due to misspelled attribute assignments. Only attribute names specified in `__slots__` are allowed:

```
class Vehicle:
    __slots__ = ('id_number', 'make', 'model')
```

```
>>> auto = Vehicle()
>>> auto.id_nubmer = 'VYE483814LQEX'
Traceback (most recent call last):
    ...
AttributeError: 'Vehicle' object has no attribute 'id_nubmer'
```

2. Helps create immutable objects where descriptors manage access to private attributes stored in `__slots__`:

```
class Immutable:

    __slots__ = ('_dept', '_name')          # Replace the instance dictionary

    def __init__(self, dept, name):
```

(下页继续)

(续上页)

```
        self._dept = dept                   # Store to private attribute
        self._name = name                   # Store to private attribute

    @property                               # Read-only descriptor
    def dept(self):
        return self._dept

    @property
    def name(self):                         # Read-only descriptor
        return self._name
```

```
>>> mark = Immutable('Botany', 'Mark Watney')
>>> mark.dept
'Botany'
>>> mark.dept = 'Space Pirate'
Traceback (most recent call last):
    ...
AttributeError: can't set attribute
>>> mark.location = 'Mars'
Traceback (most recent call last):
    ...
AttributeError: 'Immutable' object has no attribute 'location'
```

3. Saves memory. On a 64-bit Linux build, an instance with two attributes takes 48 bytes with `__slots__` and 152 bytes without. This flyweight design pattern likely only matters when a large number of instances are going to be created.

4. Blocks tools like `functools.cached_property()` which require an instance dictionary to function correctly:

```
from functools import cached_property

class CP:
    __slots__ = ()                          # Eliminates the instance dict

    @cached_property                        # Requires an instance dict
    def pi(self):
        return 4 * sum((-1.0)**n / (2.0*n + 1.0)
                       for n in reversed(range(100_000)))
```

```
>>> CP().pi
Traceback (most recent call last):
  ...
TypeError: No '__dict__' attribute on 'CP' instance to cache 'pi' property.
```

要创建一个一模一样的纯 Python 版的 `__slots__` 是不可能的，因为它需要直接访问 C 结构体并控制对象内存分配。但是，我们可以构建一个非常相似的模拟版，其中作为 slot 的实际 C 结构体由一个私有的 `_slotvalues` 列表来模拟。对该私有结构体的读写操作将由成员描述器来管理：

```
null = object()

class Member:

    def __init__(self, name, clsname, offset):
        'Emulate PyMemberDef in Include/structmember.h'
        # Also see descr_new() in Objects/descrobject.c
        self.name = name
        self.clsname = clsname
```

(下页继续)

(续上页)

```
        self.offset = offset

    def __get__(self, obj, objtype=None):
        'Emulate member_get() in Objects/descrobject.c'
        # Also see PyMember_GetOne() in Python/structmember.c
        value = obj._slotvalues[self.offset]
        if value is null:
            raise AttributeError(self.name)
        return value

    def __set__(self, obj, value):
        'Emulate member_set() in Objects/descrobject.c'
        obj._slotvalues[self.offset] = value

    def __delete__(self, obj):
        'Emulate member_delete() in Objects/descrobject.c'
        value = obj._slotvalues[self.offset]
        if value is null:
            raise AttributeError(self.name)
        obj._slotvalues[self.offset] = null

    def __repr__(self):
        'Emulate member_repr() in Objects/descrobject.c'
        return f'<Member {self.name!r} of {self.clsname!r}>'
```

type.__new__() 方法负责将成员对象添加到类变量：

```
class Type(type):
    'Simulate how the type metaclass adds member objects for slots'

    def __new__(mcls, clsname, bases, mapping):
        'Emuluate type_new() in Objects/typeobject.c'
        # type_new() calls PyTypeReady() which calls add_methods()
        slot_names = mapping.get('slot_names', [])
        for offset, name in enumerate(slot_names):
            mapping[name] = Member(name, clsname, offset)
        return type.__new__(mcls, clsname, bases, mapping)
```

object.__new__() 方法负责创建具有 slot 而非实例字典的实例。以下是一个纯 Python 的粗略模拟版：

```
class Object:
    'Simulate how object.__new__() allocates memory for __slots__'

    def __new__(cls, *args):
        'Emulate object_new() in Objects/typeobject.c'
        inst = super().__new__(cls)
        if hasattr(cls, 'slot_names'):
            empty_slots = [null] * len(cls.slot_names)
            object.__setattr__(inst, '_slotvalues', empty_slots)
        return inst

    def __setattr__(self, name, value):
        'Emulate _PyObject_GenericSetAttrWithDict() Objects/object.c'
        cls = type(self)
        if hasattr(cls, 'slot_names') and name not in cls.slot_names:
            raise AttributeError(
```

(下页继续)

(续上页)

```
                f'{type(self).__name__!r} object has no attribute {name!r}'
            )
        super().__setattr__(name, value)

    def __delattr__(self, name):
        'Emulate _PyObject_GenericSetAttrWithDict() Objects/object.c'
        cls = type(self)
        if hasattr(cls, 'slot_names') and name not in cls.slot_names:
            raise AttributeError(
                f'{type(self).__name__!r} object has no attribute {name!r}'
            )
        super().__delattr__(name)
```

要在真实的类中使用这个模拟版，只需从 `Object` 继承并将 metaclass 设为 `Type`：

```
class H(Object, metaclass=Type):
    'Instance variables stored in slots'

    slot_names = ['x', 'y']

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

这时，metaclass 已经为 *x* 和 *y* 加载了成员对象：

```
>>> from pprint import pp
>>> pp(dict(vars(H)))
{'__module__': '__main__',
 '__doc__': 'Instance variables stored in slots',
 'slot_names': ['x', 'y'],
 '__init__': <function H.__init__ at 0x7fb5d302f9d0>,
 'x': <Member 'x' of 'H'>,
 'y': <Member 'y' of 'H'>}
```

当实例被创建时，它们将拥有一个用于存放属性的 `slot_values` 列表：

```
>>> h = H(10, 20)
>>> vars(h)
{'_slotvalues': [10, 20]}
>>> h.x = 55
>>> vars(h)
{'_slotvalues': [55, 20]}
```

错误拼写或未赋值的属性将引发一个异常：

```
>>> h.xz
Traceback (most recent call last):
    ...
AttributeError: 'H' object has no attribute 'xz'
```