

修飾器 HowTo 指南

發  3.9.1rc1

Guido van Rossum
and the Python development team

12 月 07, 2020

Python Software Foundation
Email: docs@python.org

Contents

1 入门	3
1.1 简单示例：返回常量的描述器	3
1.2 动态查找	3
1.3 Managed attributes	4
1.4 Customized names	5
1.5 Closing thoughts	6
2 Complete Practical Example	6
2.1 Validator class	7
2.2 Custom validators	7
2.3 Practical application	8
3 Technical Tutorial	9
3.1 摘要	9
3.2 Definition and introduction	9
3.3 Descriptor protocol	9
3.4 Overview of descriptor invocation	10
3.5 Invocation from an instance	10
3.6 Invocation from a class	11
3.7 Invocation from super	11
3.8 Summary of invocation logic	11
3.9 Automatic name notification	11
3.10 ORM example	12
4 Pure Python Equivalents	13
4.1 属性	13
4.2 Functions and methods	14
4.3 Static methods	15
4.4 Class methods	16
4.5 Member objects and <code>__slots__</code>	17

作者 Raymond Hettinger

联系方式 [<python at rcn dot com>](mailto:python@rcn dot com)

目F

- 修飾器 *HowTo* 指南
 - 入门
 - * 简单示例：返回常量的描述器
 - * 动态查找
 - * *Managed attributes*
 - * *Customized names*
 - * *Closing thoughts*
 - *Complete Practical Example*
 - * *Validator class*
 - * *Custom validators*
 - * *Practical application*
 - *Technical Tutorial*
 - * 摘要
 - * *Definition and introduction*
 - * *Descriptor protocol*
 - * *Overview of descriptor invocation*
 - * *Invocation from an instance*
 - * *Invocation from a class*
 - * *Invocation from super*
 - * *Summary of invocation logic*
 - * *Automatic name notification*
 - * *ORM example*
 - *Pure Python Equivalents*
 - * 属性
 - * *Functions and methods*
 - * *Static methods*
 - * *Class methods*
 - * *Member objects and __slots__*

描述器让对象能够自定义属性查找、存储和删除的操作。

本指南分为四个主要部分：

- 1) The "primer" gives a basic overview, moving gently from simple examples, adding one feature at a time. Start here if you're new to descriptors.
- 2) 第二部分显示了完整的、实用的描述器示例。如果您已经掌握基础知识，请从此处开始。
- 3) 第三部分提供了更多技术教程，详细介绍了描述符如何工作。大多数人并不需要深入到这种程度。
- 4) The last section has pure Python equivalents for built-in descriptors that are written in C. Read this if you're curious about how functions turn into bound methods or about the implementation of common tools like `classmethod()`, `staticmethod()`, `property()`, and `__slots__`.

1 入门

在本入门中，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

The `Ten` class is a descriptor that always returns the constant 10 from its `__get__()` method:

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

To use the descriptor, it must be stored as a class variable in another class:

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

An interactive session shows the difference between normal attribute lookup and descriptor lookup:

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

In the `a.x` attribute lookup, the dot operator finds the key `x` and the value 5 in the class dictionary. In the `a.y` lookup, the dot operator finds a descriptor instance, recognized by its `__get__` method, and calls that method which returns 10.

Note that the value 10 is not stored in either the class dictionary or the instance dictionary. Instead, the value 10 is computed on demand.

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

Interesting descriptors typically run computations instead of returning constants:

```
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，更新后的答案

```

>>> s = Directory('songs')
>>> g = Directory('games')
>>> s.size                                # The songs directory has twenty files
20
>>> g.size                                # The games directory has three files
3
>>> open('games/newfile').close()         # Add a fourth file to the directory
>>> g.size                                # File count is automatically updated
4

```

Besides showing how descriptors can run computations, this example also reveals the purpose of the parameters to `__get__()`. The *self* parameter is *size*, an instance of *DirectorySize*. The *obj* parameter is either *g* or *s*, an instance of *Directory*. It is the *obj* parameter that lets the `__get__()` method learn the target directory. The *objtype* parameter is the class *Directory*.

1.3 Managed attributes

A popular use for descriptors is managing access to instance data. The descriptor is assigned to a public attribute in the class dictionary while the actual data is stored as a private attribute in the instance dictionary. The descriptor's `__get__()` and `__set__()` methods are triggered when the public attribute is accessed.

In the following example, *age* is the public attribute and *_age* is the private attribute. When the public attribute is accessed, the descriptor logs the lookup or update:

```

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

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__()

```

An interactive session shows that all access to the managed attribute *age* is logged, but that the regular attribute *name* is not logged:

```

>>> 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}

```

(下页继续)

```

>>> 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

```

One major issue with this example is that the private name `_age` is hardwired in the `LoggedAgeAccess` class. That means that each instance can only have one logged attribute and that its name is unchangeable. In the next example, we'll fix that problem.

1.4 Customized names

When a class uses descriptors, it can inform each descriptor about which variable name was used.

In this example, the `Person` class has two descriptor instances, `name` and `age`. When the `Person` class is defined, it makes a callback to `__set_name__()` in `LoggedAccess` so that the field names can be recorded, giving each descriptor its own `public_name` and `private_name`:

```

import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

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

```

An interactive session shows that the `Person` class has called `__set_name__()` so that the field names would

be recorded. Here we call `vars()` to look up the descriptor without triggering it:

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

The new class now logs access to both *name* and *age*:

```
>>> 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
```

The two *Person* instances contain only the private names:

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

1.5 Closing thoughts

A descriptor is what we call any object that defines `__get__()`, `__set__()`, or `__delete__()`.

Optionally, descriptors can have a `__set_name__()` method. This is only used in cases where a descriptor needs to know either the class where it was created or the name of class variable it was assigned to. (This method, if present, is called even if the class is not a descriptor.)

Descriptors get invoked by the dot “operator” during attribute lookup. If a descriptor is accessed indirectly with `vars(some_class)[descriptor_name]`, the descriptor instance is returned without invoking it.

Descriptors only work when used as class variables. When put in instances, they have no effect.

The main motivation for descriptors is to provide a hook allowing objects stored in class variables to control what happens during attribute lookup.

Traditionally, the calling class controls what happens during lookup. Descriptors invert that relationship and allow the data being looked-up to have a say in the matter.

Descriptors are used throughout the language. It is how functions turn into bound methods. Common tools like `classmethod()`, `staticmethod()`, `property()`, and `functools.cached_property()` are all implemented as descriptors.

2 Complete Practical Example

In this example, we create a practical and powerful tool for locating notoriously hard to find data corruption bugs.

2.1 Validator class

A validator is a descriptor for managed attribute access. Prior to storing any data, it verifies that the new value meets various type and range restrictions. If those restrictions aren't met, it raises an exception to prevent data corruption at its source.

This `Validator` class is both an abstract base class and a managed attribute descriptor:

```
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
```

Custom validators need to inherit from `Validator` and must supply a `validate()` method to test various restrictions as needed.

2.2 Custom validators

Here are three practical data validation utilities:

- 1) `OneOf` verifies that a value is one of a restricted set of options.
- 2) `Number` verifies that a value is either an `int` or `float`. Optionally, it verifies that a value is between a given minimum or maximum.
- 3) `String` verifies that a value is a `str`. Optionally, it validates a given minimum or maximum length. It can validate a user-defined `predicate` as well.

```
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}'
```

(下页继续)

```

    )
    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}'
            )

```

2.3 Practical application

Here's how the data validators can be used in a real class:

```

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

```

The descriptors prevent invalid instances from being created:

```

>>> 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 Technical Tutorial

What follows is a more technical tutorial for the mechanics and details of how descriptors work.

3.1 摘要

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

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works.

3.2 Definition and introduction

In general, a descriptor is an attribute value that has one of the methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an the attribute, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the method resolution order of `type(a)`. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined.

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 Descriptor protocol

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

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

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

以上就是全部。定义这些方法中的任何一个的对象被视为描述器，并在被作为属性时覆盖其默认行为。

If an object defines `__set__()` or `__delete__()`, it is considered a data descriptor. Descriptors that only define `__get__()` are called non-data descriptors (they are often used for methods but other uses are possible).

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 Overview of descriptor invocation

A descriptor can be called directly with `desc.__get__(obj)` or `desc.__get__(None, cls)`.

But it is more common for a descriptor to be invoked automatically from attribute access.

The expression `obj.x` looks up the attribute `x` in the chain of namespaces for `obj`. If the search finds a descriptor outside of the instance `__dict__`, its `__get__()` method is invoked according to the precedence rules listed below.

The details of invocation depend on whether `obj` is an object, class, or instance of super.

3.5 Invocation from an instance

Instance lookup scans through a chain of namespaces giving data descriptors the highest priority, followed by instance variables, then non-data descriptors, then class variables, and lastly `__getattr__()` if it is provided.

If a descriptor is found for `a.x`, then it is invoked with: `desc.__get__(a, type(a))`.

The logic for a dotted lookup is in `object.__getattribute__()`. Here is a pure Python equivalent:

```
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)
```

Interestingly, attribute lookup doesn't call `object.__getattribute__()` directly. Instead, both the dot operator and the `getattr()` function perform attribute lookup by way of a helper function:

```
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__
```

So if `__getattr__()` exists, it is called whenever `__getattribute__()` raises `AttributeError` (either directly or in one of the descriptor calls).

Also, if a user calls `object.__getattribute__()` directly, the `__getattr__()` hook is bypassed entirely.

3.6 Invocation from a class

The logic for a dotted lookup such as `A.x` is in `type.__getattribute__()`. The steps are similar to those for `object.__getattribute__()` but the instance dictionary lookup is replaced by a search through the class's method resolution order.

If a descriptor is found, it is invoked with `desc.__get__(None, A)`.

The full C implementation can be found in `type_getattro()` and `_PyType_Lookup()` in [Objects/typeobject.c](#).

3.7 Invocation from super

The logic for super's dotted lookup is in the `__getattribute__()` method for object returned by `super()`.

A dotted lookup such as `super(A, obj).m` searches `obj.__class__.__mro__` for the base class B immediately following A and then returns `B.__dict__['m'].__get__(obj, A)`. If not a descriptor, m is returned unchanged.

The full C implementation can be found in `super_getattro()` in [Objects/typeobject.c](#). A pure Python equivalent can be found in [Guido's Tutorial](#).

3.8 Summary of invocation logic

The mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`.

要记住的重要点是:

- Descriptors are invoked by the `__getattribute__()` method.
- Classes inherit this machinery from `object`, `type`, or `super()`.
- Overriding `__getattribute__()` prevents automatic descriptor calls because all the descriptor logic is in that method.
- `object.__getattribute__()` and `type.__getattribute__()` make different calls to `__get__()`. The first includes the instance and may include the class. The second puts in `None` for the instance and always includes the class.
- Data descriptors always override instance dictionaries.
- Non-data descriptors may be overridden by instance dictionaries.

3.9 Automatic name notification

Sometimes it is desirable for a descriptor to know what class variable name it was assigned to. When a new class is created, the `type` metaclass scans the dictionary of the new class. If any of the entries are descriptors and if they define `__set_name__()`, that method is called with two arguments. The *owner* is the class where the descriptor is used, and the *name* is the class variable the descriptor was assigned to.

The implementation details are in `type_new()` and `set_names()` in [Objects/typeobject.c](#).

Since the update logic is in `type.__new__()`, notifications only take place at the time of class creation. If descriptors are added to the class afterwards, `__set_name__()` will need to be called manually.

3.10 ORM example

The following code is simplified skeleton showing how data descriptors could be used to implement an [object relational mapping](#).

The essential idea is that the data is stored in an external database. The Python instances only hold keys to the database's tables. Descriptors take care of lookups or updates:

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

We can use the Field class to define [models](#) that describe the schema for each table in a database:

```
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
```

To use the models, first connect to the database:

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

An interactive session shows how data is retrieved from the database and how it can be updated:

```
>>> 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 Pure Python Equivalents

The descriptor protocol is simple and offers exciting possibilities. Several use cases are so common that they have been prepackaged into built-in tools. Properties, bound methods, static methods, class methods, and `__slots__` are all based on the descriptor protocol.

4.1 属性

Calling `property()` is a succinct way of building a data descriptor that triggers a function call upon access to an attribute. Its signature is:

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

The documentation shows a typical use to define a managed attribute `x`:

```
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.")
```

To see how `property()` is implemented in terms of the descriptor protocol, here is a pure Python equivalent:

```
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。

For instance, a spreadsheet class may grant access to a cell value through `Cell('b10').value`. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```
class Cell:
    ...

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

Either the built-in `property()` or our `Property()` equivalent would work in this example.

4.2 Functions and methods

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。

Functions stored in class dictionaries get turned into methods when invoked. Methods only differ from regular functions in that the object instance is prepended to the other arguments. By convention, the instance is called *self* but could be called *this* or any other variable name.

Methods can be created manually with `types.MethodType` which is roughly equivalent to:

```
class MethodType:
    "Emulate Py_MethodType 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)
```

To support automatic creation of methods, functions include the `__get__()` method for binding methods during attribute access. This means that functions are non-data descriptors that return bound methods during dotted lookup from an instance. Here's how it works:

```
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)
```

Running the following class in the interpreter shows how the function descriptor works in practice:

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

The function has a qualified name attribute to support introspection:

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

Accessing the function through the class dictionary does not invoke `__get__()`. Instead, it just returns the underlying function object:

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

Dotted access from a class calls `__get__()` which just returns the underlying function unchanged:

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

The interesting behavior occurs during dotted access from an instance. The dotted lookup calls `__get__()` which returns a bound method object:

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

Internally, the bound method stores the underlying function and the bound instance:

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

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

If you have ever wondered where *self* comes from in regular methods or where *cls* comes from in class methods, this is it!

4.3 Static methods

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。

To recap, functions have a `__get__()` method so that they can be converted to a method when accessed as attributes. The non-data descriptor transforms an `obj.f(*args)` call into `f(obj, *args)`. Calling `cls.f(*args)` becomes `f(*args)`.

下表总结了绑定及其两个最有用的变体：

转换形式	Called from an object	Called from a class
函数	<code>f(obj, *args)</code>	<code>f(*args)</code>
静态方法	<code>f(*args)</code>	<code>f(*args)</code>
类方法	<code>f(type(obj), *args)</code>	<code>f(cls, *args)</code>

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。

适合于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`。

Since static methods return the underlying function with no changes, the example calls are unexciting:

```
class E:
    @staticmethod
    def f(x):
        print(x)
```

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

Using the non-data descriptor protocol, a pure Python version of `staticmethod()` would look like this:

```
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
```

4.4 Class methods

Unlike static methods, class methods prepend the class reference to the argument list before calling the function. This format is the same for whether the caller is an object or a class:

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

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

This behavior is useful whenever the method only needs to have a class reference and does not rely on data stored in a specific instance. One use for class methods is to create alternate class constructors. For example, the classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```
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
```

Now a new dictionary of unique keys can be constructed like this:

```
>>> d = Dict.fromkeys('abracadabra')
>>> type(d) is Dict
True
>>> d
{'a': None, 'b': None, 'r': None, 'c': None, 'd': None}
```

Using the non-data descriptor protocol, a pure Python version of `classmethod()` would look like this:

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

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

(下页继续)


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

The code path for `hasattr(obj, '__get__')` was added in Python 3.9 and makes it possible for `classmethod()` to support chained decorators. For example, a `classmethod` and `property` could be chained together:

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

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

4.5 Member objects and `__slots__`

When a class defines `__slots__`, it replaces instance dictionaries with a fixed-length array of slot values. From a user point of view that has several effects:

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):
        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                                # Read-only descriptor
    def name(self):
        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.
```

It is not possible to create an exact drop-in pure Python version of `__slots__` because it requires direct access to C structures and control over object memory allocation. However, we can build a mostly faithful simulation where the actual C structure for slots is emulated by a private `_slotvalues` list. Reads and writes to that private structure are managed by member descriptors:

```
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
        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}>'

```

The type.__new__() method takes care of adding member objects to class variables:

```

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

    def __new__(mcls, clsname, bases, mapping):
        'Emulate 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)

```

The object.__new__() method takes care of creating instances that have slots instead of an instance dictionary. Here is a rough simulation in pure Python:

```

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(
                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)

```

To use the simulation in a real class, just inherit from Object and set the metaclass to Type:

```

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

```

(繼續上一頁)

```
slot_names = ['x', 'y']

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

At this point, the metaclass has loaded member objects for *x* and *y*:

```
>>> 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'>}
```

When instances are created, they have a `slot_values` list where the attributes are stored:

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

Misspelled or unassigned attributes will raise an exception:

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