디스크립터 사용법 안내서

출시 버전 *3.9.0*

Guido van Rossum
and the Python development team

11월 22, 2020

Contents

저자 Raymond Hettinger

연락처 <python at rcn dot com>

목차

Descriptors let objects customize attribute lookup, storage, and deletion.

This guide has four major sections:

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) The second section shows a complete, practical descriptor example. If you already know the basics, start there.
3) The third section provides a more technical tutorial that goes into the detailed mechanics of how descriptors work. Most people don't need this level of detail.
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 Primer

In this primer, we start with the most basic possible example and then we'll add new capabilities one by one.

1.1 Simple example: A descriptor that returns a constant

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.

This example shows how a simple descriptor works, but it isn't very useful. For retrieving constants, normal attribute lookup would be better.

In the next section, we'll create something more useful, a dynamic lookup.

1.2 Dynamic lookups

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
```

An interactive session shows that the lookup is dynamic — it computes different, updated answers each time:

```
>>> 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}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> 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:

```python
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.

```python
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}'
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
            )
        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 use

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)      # Allowed.
Component('Widget', 'metal', 5)      # Blocked: 'Widget' is not all uppercase
Component('WIDGET', 'metle', 5)      # Blocked: 'metle' is misspelled
Component('WIDGET', 'metal', -5)     # Blocked: -5 is negative
Component('WIDGET', 'metal', 'V')    # Blocked: 'V' isn't a number
```

3 Technical Tutorial

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

3.1 요약

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Provides an example showing how object relational mappings work.

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.

Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and `super()`. They are used throughout Python itself. Descriptors simplify the underlying C code and offer a flexible set of new tools for everyday Python programs.

3.3 Descriptor protocol

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

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

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

이것이 전부입니다. 이러한 메서드 중 하나를 정의하십시오, 그러면 객체를 디스크립터로 간주하고 어트리뷰트로 조회될 때 기본 동작을 재정의할 수 있습니다.

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).

데이터와 비 데이터 디스크립터는 인스턴스 딕셔너리의 항목과 관련하여 재정의가 계산되는 방식이 다릅니다. 인스턴스 딕셔너리에 데이터 디스크립터와 이름이 같은 항목이 있으면, 데이터 디스크립터가 우선합니다. 인스턴스의 딕셔너리에 비 데이터 디스크립터와 이름이 같은 항목이 있으면, 딕셔너리 항목이 우선합니다.

읽기 전용 데이터 디스크립터를 만들려면, `__get__()`과 `__set__()`을 모두 정의하고, `__set__()`이 호출될 때 `AttributeError`를 발생시키십시오. 데이터 디스크립터를 만들기 위해 예외를 발생시키는 자리 표시자로 `__set__()` 메서드를 정의하는 것으로 충분합니다.

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()`.

기억해야 할 중요한 사항은 다음과 같습니다:

- 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
```

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

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

>>> 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 프로퍼티

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
```

설명 (doc) 은 관리되는 어트리뷰트 x를 정의하는 일반적인 사용법을 보여줍니다:

```
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.")
```

디스크립터 프로토콜 측면에서 `property()` 가 어떻게 구현되는지 확인하려면, 여기 순수한 파이썬 동등물이 있습니다:

```
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__)
```

property() 내장은 사용자 인터페이스가 어트리뷰트 액세스를 허가한 후 후속 변경이 메서드의 개입을 요구할 때 도움을 줍니다.

예를 들어, 스프레드시트 클래스는 Cell('b10').value를 통해 셀 값에 대한 액세스를 허가할 수 있습니다. 프로그램에 대한 후속 개선은 액세스할 때마다 셀이 재계산될 것을 요구합니다; 하지만, 프로그래머는 어트리뷰트에 직접 액세스하는 기존 클라이언트 코드에 영향을 미치고 싶지 않습니다. 해결책은 프로퍼티 데이터 디스크립터로 value 어트리뷰트에 대한 액세스를 감싸는 것입니다:

```
class Cell:
    ...

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

4.2 Functions and methods

파이썬의 객체 지향 기능은 함수 기반 환경을 기반으로 합니다. 비 데이터 디스크립터를 사용하면, 두 개가 매끄럽게 병합됩니다.

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 0x1012e5ae8>

>>> 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

비 데이터 디스크립터는 함수에 메서드를 바인딩하는 일반적인 패턴을 변형하는 간단한 메커니즘을 제공합니다.

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)`.

이 표는 연결과 가장 유용한 두 가지 변형을 요약합니다:

변환	Called from an object	Called from a class
함수	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(cls, *args)

정적 메서드는 변경 없이 하부 함수를 반환합니다. `c.f` 나 `C.f` 호출은 `object.__getattribute__(c, "f")` 나 `object.__getattribute__(C, "f")` 를 직접 조회하는 것과 동등합니다. 결과적으로, 함수는 객체나 클래스에서 동일하게 액세스 할 수 있습니다.

정적 메서드에 적합한 후보는 `self` 변수를 참조하지 않는 메서드입니다.

예를 들어, 통계 패키지는 실험 데이터를 위한 컨테이너 클래스를 포함 할 수 있습니다. 이 클래스는 데이터에 의존하는 산술 평균, 평균, 중앙값 및 기타 기술 통계량을 계산하는 일반 메서드를 제공합니다. 그러나, 개념적으로 관련되어 있지만, 데이터에 의존하지 않는 유용한 함수가 있을 수 있습니다. 예를 들어, `erf(x)` 는 통계 작업에서 등장하지만, 특정 데이터 집합에 직접 의존하지 않는 편리한 변환 루틴입니다. 객체나 클래스에서 호출 할 수 있습니다: `s.erf(1.5) --> .9332` 또는 `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
```

비 데이터 디스크립터 프로토콜을 사용하면, 순수 파이썬 버전의 `staticmethod()`는 다음과 같습니다:

```
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

정적 메서드와 달리, 클래스 메서드는 함수를 호출하기 전에 클래스 참조를 인자 목록 앞에 추가합니다. 이 형식은 호출자가 객체나 클래스일 때 같습니다:

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

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

This behavior is useful whenever the method only needs to have a class reference and does 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:
    ...

    @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
```

이제 고유 키의 새로운 딕셔너리를 다음과 같이 구성 할 수 있습니다:

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

비 데이터 디스크립터 프로토콜을 사용하면, 순수 파이썬 버전의 `classmethod()`는 다음과 같습니다:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

    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}'
```

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

mark = Immutable('Botany', 'Mark Watney')   # Create an immutable instance
```

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's 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:

```
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
        return obj._slotvalues[self.offset]

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

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

```
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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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'):
            inst._slotvalues = [None] * len(cls.slot_names)
        return inst
```

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

```
class H(Object, metaclass=Type):

    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*:

```
>>> import pprint
>>> pprint.pp(dict(vars(H)))
{'__module__': '__main__',
 'slot_names': ['x', 'y'],
 '__init__': <function H.__init__ at 0x7fb5d302f9d0>,
 'x': <Member 'x' of 'H'>,
 'y': <Member 'y' of 'H'>,
 '__doc__': None}
```

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]}
```

Unlike the real `__slots__`, this simulation does have an instance dictionary just to hold the `_slotvalues` array. So, unlike the real code, this simulation doesn't block assignments to misspelled attributes:

```
>>> h.xz = 30    # For actual __slots__ this would raise an AttributeError
```