

---

# 디스크립터 사용법 안내서

출시 버전 3.8.17

Guido van Rossum  
and the Python development team

7월 06, 2023

## Contents

|   |                 |   |
|---|-----------------|---|
| 1 | 요약              | 2 |
| 2 | 정의와 소개          | 2 |
| 3 | 디스크립터 프로토콜      | 2 |
| 4 | 디스크립터 호출하기      | 3 |
| 5 | 디스크립터 예제        | 4 |
| 6 | 프로퍼티            | 4 |
| 7 | 함수와 메서드         | 6 |
| 8 | 정적 메서드와 클래스 메서드 | 7 |

---

저자 Raymond Hettinger

연락처 <python at rcn dot com>

### 목차

- 디스크립터 사용법 안내서
  - 요약
  - 정의와 소개
  - 디스크립터 프로토콜
  - 디스크립터 호출하기
  - 디스크립터 예제
  - 프로퍼티
  - 함수와 메서드
  - 정적 메서드와 클래스 메서드

## 1 요약

디스크립터를 정의하고, 프로토콜을 요약하며 디스크립터를 호출하는 방법을 보여줍니다. 사용자 정의 디스크립터와 함수, 프로퍼티, 정적 메서드 및 클래스 메서드를 포함한 몇 가지 내장 파이썬 디스크립터를 살펴봅니다. 순수 파이썬과 동등 물과 샘플 응용 프로그램을 제공하여 각각이 작동하는 방식을 보여줍니다.

디스크립터에 대한 학습은 더 큰 도구 집합에 대한 액세스를 제공할 뿐만 아니라, 파이썬의 작동 방식에 대한 심층적인 이해와 설계의 우아함에 대한 감사를 만듭니다.

## 2 정의와 소개

일반적으로 디스크립터는 “바인딩 동작”이 있는 객체 어트리뷰트로, 디스크립터 프로토콜의 메서드가 어트리뷰트 액세스를 재정의합니다. 이러한 메서드는 `__get__()`, `__set__()` 및 `__delete__()` 입니다. 이러한 메서드 중 어느 하나가 객체에 대해 정의되면, 디스크립터라고 합니다.

어트리뷰트 액세스의 기본 동작은 객체의 딕셔너리에서 어트리뷰트를 가져오거나(`get`) 설정하거나(`set`) 삭제하는(`delete`) 것입니다. 예를 들어, `a.x`는 `a.__dict__['x']`로 시작한 다음 `type(a).__dict__['x']`를 거쳐, 메타 클래스를 제외한 `type(a)`의 베이스 클래스로 계속되는 조회 체인을 갖습니다. 조회된 값이 디스크립터 메서드 중 하나를 정의하는 객체이면, 파이썬은 기본 동작을 대체하고 대신 디스크립터 메서드를 호출 할 수 있습니다. 우선순위 체인에서 이것이 어디쯤 등장하는지는 어떤 디스크립터 메서드가 정의되었는지에 따라 다릅니다.

디스크립터는 강력한 범용 프로토콜입니다. 이것들이 프로퍼티, 메서드, 정적 메서드, 클래스 메서드 및 `super()`의 뒤에 있는 메커니즘입니다. 버전 2.2에 도입된 새로운 스타일 클래스를 구현하기 위해 파이썬 자체에서 사용되었습니다. 디스크립터는 하부 C 코드를 단순화하고 일상적인 파이썬 프로그램을 위한 유연한 새 도구 집합을 제공합니다.

## 3 디스크립터 프로토콜

```
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.

객체가 `__set__()`이나 `__delete__()`를 정의하면, 데이터 디스크립터로 간주합니다. `__get__()`만 정의하는 디스크립터를 비 데이터 디스크립터라고 합니다 (보통 메서드에 사용되지만 다른 용도도 가능합니다).

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.

## 4 디스크립터 호출하기

디스크립터는 메서드 이름으로 직접 호출 할 수 있습니다. 예를 들어, `d.__get__(obj)`.

또는, 어트리뷰트 액세스 시 디스크립터가 자동으로 호출되는 것이 더 일반적입니다. 예를 들어, `obj.d`는 `obj` 딕셔너리에서 `d`를 조회합니다. `d`가 메서드 `__get__()`을 정의하면, 아래 나열된 우선순위 규칙에 따라 `d.__get__(obj)`가 호출됩니다.

호출 세부 사항은 `obj`가 객체인지 클래스인지에 따라 다릅니다.

객체의 경우, 절차는 `object.__getattribute__()`에 있으며 `b.x`를 `type(b).__dict__['x'].__get__(b, type(b))`로 변환합니다. 구현은 우선순위 체인을 통해 작동하며, 데이터 디스크립터는 인스턴스 변수보다 우선하고, 인스턴스 변수는 비 데이터 디스크립터보다 우선하고, 제공된다면 `__getattr__()`에 가장 낮은 우선순위를 지정합니다. 전체 C 구현은 [Objects/object.c](#)의 `PyObject_GenericGetAttr()`에서 찾을 수 있습니다.

클래스의 경우, 절차는 `type.__getattribute__()`에 있으며 `B.x`를 `B.__dict__['x'].__get__(None, B)`로 변환합니다. 단순한 파이썬으로 표현하면, 다음과 같습니다:

```
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

기억해야 할 중요한 사항은 다음과 같습니다:

- 디스크립터는 `__getattribute__()` 메서드에 의해 호출됩니다
- `__getattribute__()`를 재정의하면 자동 디스크립터 호출이 방지됩니다
- `object.__getattribute__()`와 `type.__getattribute__()`는 `__get__()`를 다르게 호출합니다.
- 데이터 디스크립터는 항상 인스턴스 딕셔너리를 대체합니다.
- 비 데이터 디스크립터는 인스턴스 딕셔너리로 대체될 수 있습니다.

`super()`가 반환한 객체에도 디스크립터 호출을 위한 사용자 정의 `__getattribute__()` 메서드가 있습니다. 어트리뷰트 조회 `super(B, obj).m`은 `obj.__class__.__mro__`에서 `B` 바로 다음에 오는 베이스 클래스 `A`를 검색한 다음 `A.__dict__['m'].__get__(obj, B)`를 반환합니다. 디스크립터가 아니면, `m`이 변경되지 않은 상태로 반환됩니다. 딕셔너리에 없으면, `m`은 `object.__getattribute__()`를 사용한 검색으로 되돌아갑니다.

구현 세부 사항은 [Objects/typeobject.c](#)의 `super_getattro()`에 있습니다. [Guido's Tutorial](#)에서 순수한 파이썬 동등 물을 찾을 수 있습니다.

위의 세부 사항은 디스크립터 메커니즘이 `object`, `type` 및 `super()`의 `__getattribute__()` 메서드에 내장되어 있음을 보여줍니다. 클래스는 `object`에서 파생되거나 유사한 기능을 제공하는 메타클래스가 있을 때 이 절차를 상속합니다. 마찬가지로, 클래스는 `__getattribute__()`를 재정의하여 디스크립터 호출을 끌 수 있습니다.

## 5 디스크립터 예제

다음 코드는 객체가 데이터 디스크립터인 클래스를 만들어 각 `get`이나 `set`에 대해 메시지를 인쇄합니다. `__getattr__()`를 재정의하는 것은 모든 어트리뷰트에 대해 이를 수행 할 수 있는 대안 방법입니다. 하지만, 이 디스크립터는 선택한 몇 가지 어트리뷰트만 모니터링하는 데 유용합니다:

```
class RevealAccess(object):
    """A data descriptor that sets and returns values
    normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

이 프로토콜은 간단하고 흥미로운 가능성을 제공합니다. 몇 가지 유스 케이스는 아주 흔해서 개별 함수 호출로 패키징되었습니다. 프로퍼티, 연결된 메서드, 정적 메서드 및 클래스 메서드는 모두 디스크립터 프로토콜을 기반으로 합니다.

## 6 프로퍼티

`property()` 호출은 어트리뷰트에 액세스할 때 함수 호출을 트리거 하는 데이터 디스크립터를 작성하는 간결한 방법입니다. 서명은 다음과 같습니다:

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

설명(doc)은 관리되는 어트리뷰트 `x`를 정의하는 일반적인 사용법을 보여줍니다:

```
class C(object):
    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(object):
    "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').value`를 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(object):
    . . .
    def getvalue(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
    value = property(getvalue)
```

## 7 함수와 메서드

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

클래스 디렉터리는 메서드를 함수로 저장합니다. 클래스 정의에서, 메서드는 함수 작성을 위한 일반적인 도구인 `def`나 `lambda`를 사용하여 작성됩니다. 첫 번째 인자가 객체 인스턴스에 예약되어 있다는 점에서만 메서드가 일반 함수와 다릅니다. 파이썬 규칙에 따라, 인스턴스 참조는 *self*라고 하지만 *this*나 다른 어떤 변수 이름도 될 수 있습니다.

메서드 호출을 지원하기 위해, 함수는 어트리뷰트 액세스 중에 메서드를 연결하기 위한 `__get__()` 메서드를 포함합니다. 즉, 모든 함수는 객체에서 호출될 때 연결된 메서드를 반환하는 비 데이터 디스크립터입니다. 순수한 파이썬으로 표현하면, 이것은 다음과 같이 작동합니다:

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

인터프리터를 실행하면 실제로 함수 디스크립터가 작동하는 방식을 보여줍니다:

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()

# Access through the class dictionary does not invoke __get__.
# 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 function has a __qualname__ attribute to support introspection
>>> D.f.__qualname__
'D.f'

# Dotted access from an instance calls __get__() which returns the
# function wrapped in a bound method object
>>> 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>
```

## 8 정적 메서드와 클래스 메서드

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

요약하면, 함수에는 `__get__()` 메서드가 있어서 어트리뷰트로 액세스할 때 메서드로 변환될 수 있습니다. 비 데이터 디스크립터는 `obj.f(*args)` 호출을 `f(obj, *args)`로 변환합니다. `klass.f(*args)` 호출은 `f(*args)`가 됩니다.

이 표는 연결과 가장 유용한 두 가지 변형을 요약합니다:

| 변환                        | 객체에서 호출                          | 클래스에서 호출                     |
|---------------------------|----------------------------------|------------------------------|
| 함수                        | <code>f(obj, *args)</code>       | <code>f(*args)</code>        |
| <code>staticmethod</code> | <code>f(*args)</code>            | <code>f(*args)</code>        |
| <code>classmethod</code>  | <code>f(type(obj), *args)</code> | <code>f(klass, *args)</code> |

정적 메서드는 변경 없이 하부 함수를 반환합니다. `c.f`나 `C.f` 호출은 `object.__getattr__(c, "f")`나 `object.__getattr__(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`

정적 메서드는 변경 없이 하부 함수를 반환하므로, 예제 호출은 흥미롭지 않습니다:

```
>>> class E(object):
...     def f(x):
...         print(x)
...     f = staticmethod(f)
...
>>> E.f(3)
3
>>> E().f(3)
3
```

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

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

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

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

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

```
>>> class E(object):
...     def f(klass, x):
...         return klass.__name__, x
...     f = classmethod(f)
...
>>> print(E.f(3))
('E', 3)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

이 동작은 함수가 클래스 참조만 필요하고 하부 데이터를 신경 쓰지 않을 때 유용합니다. 클래스 메서드의 한 가지 용도는 대체 클래스 생성자를 만드는 것입니다. 파이썬 2.3에서, 클래스 메서드 `dict.fromkeys()` 는 키 리스트에서 새 딕셔너리를 만듭니다. 순수한 파이썬 동등물은 다음과 같습니다:

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

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(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

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

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```