
Descriptor HowTo Guide

릴리스 3.11.13

Guido van Rossum and the Python development team

7월 07, 2025

Contents

1	입문	3
1.1	간단한 예: 상수를 반환하는 디스크립터	3
1.2	동적 조회	4
1.3	관리되는 어트리뷰트	4
1.4	사용자 정의 이름	5
1.5	마무리 생각	7
2	완전한 실용적인 예	7
2.1	유효성 검사기 클래스	7
2.2	사용자 정의 유효성 검사기	8
2.3	실용적인 응용	9
3	기술적인 자습서	10
3.1	요약	10
3.2	정의와 소개	10
3.3	디스크립터 프로토콜	10
3.4	디스크립터 호출의 개요	11
3.5	인스턴스에서 호출	11
3.6	클래스에서 호출	12
3.7	super에서 호출	12
3.8	호출 로직 요약	12
3.9	자동 이름 알림	13
3.10	ORM 예제	13
4	순수한 파이썬 등가물	14
4.1	프로퍼티	14
4.2	함수와 메서드	16
4.3	메서드의 종류	17
4.4	정적 메서드	17
4.5	클래스 메서드	18
4.6	멤버 객체와 __slots__	19

저자

Raymond Hettinger

목차

- *Descriptor HowTo Guide*

- 입문

- * 간단한 예: 상수를 반환하는 디스크립터
 - * 동적 조회
 - * 관리되는 어트리뷰트
 - * 사용자 정의 이름
 - * 마무리 생각

- 완전한 실용적인 예

- * 유효성 검사기 클래스
 - * 사용자 정의 유효성 검사기
 - * 실용적인 응용

- 기술적인 자습서

- * 요약
 - * 정의와 소개
 - * 디스크립터 프로토콜
 - * 디스크립터 호출의 개요
 - * 인스턴스에서 호출
 - * 클래스에서 호출
 - * *super*에서 호출
 - * 호출 로직 요약
 - * 자동 이름 알림
 - * *ORM* 예제

- 순수한 파이썬 등가물

- * 프로퍼티
 - * 함수와 메서드
 - * 메서드의 종류
 - * 정적 메서드
 - * 클래스 메서드
 - * 멤버 객체와 `__slots__`

디스크립터는 객체가 어트리뷰트 조회, 저장 및 삭제를 사용자 정의 할 수 있도록 합니다.

이 지침서는 네 개의 주요 섹션으로 구성됩니다:

- 1) “입문”은 간단한 예제에서 부드럽게 이동하여 한 번에 하나의 기능을 추가하는 기본 개요를 제공합니다.

디스크립터를 처음 사용하면 여기에서 시작하세요.

- 2) 두 번째 섹션은 완전하고 실용적인 디스크립터 예제를 보여줍니다. 이미 기본 사항을 알고 있다면, 여기에서 시작하십시오.
- 3) 세 번째 섹션에서는 디스크립터가 작동하는 방식에 대한 자세한 메커니즘에 관해 설명하는 더 기술적인 자습서를 제공합니다. 대부분의 사람은 이러한 수준의 세부 정보가 필요하지 않습니다.
- 4) 마지막 섹션에는 C로 작성된 내장 디스크립터에 대한 순수한 파이썬 등가물이 있습니다. 함수가 연결된 메서드로 바뀌는 방법이나 `classmethod()`, `staticmethod()`, `property()` 및 `__slots__`와 같은 일반적인 도구의 구현에 대해 궁금하면 이 문서를 읽으십시오.

1 입문

이 입문서에서는, 가능한 가장 기본적인 예제로 시작한 다음 새로운 기능을 하나씩 추가할 것입니다.

1.1 간단한 예: 상수를 반환하는 디스크립터

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

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

디스크립터를 사용하려면, 다른 클래스에 클래스 변수로 저장해야 합니다:

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

대화 형 세션은 일반 어트리뷰트 조회와 디스크립터 조회의 차이점을 보여줍니다:

```
>>> 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 `'x': 5` in the class dictionary. In the `a.y` lookup, the dot operator finds a descriptor instance, recognized by its `__get__` method. Calling that method returns 10.

10 값이 클래스 딕셔너리나 인스턴스 딕셔너리에 저장되지 않음에 유의하십시오. 대신, 10 값은 요청 시 계산됩니다.

이 예는 간단한 디스크립터가 어떻게 작동하는지 보여 주지만, 그다지 유용하지는 않습니다. 상수를 꺼내려면, 일반 어트리뷰트 조회가 더 좋습니다.

다음 섹션에서는, 좀 더 유용한 동적 조회를 만들 것입니다.

1.2 동적 조회

흥미로운 디스크립터는 보통 상수를 반환하는 대신 계산을 실행합니다:

```
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
```

대화 형 세션은 조회가 동적임을 보여줍니다 - 매번 다른 갱신된 답변을 계산합니다:

```
>>> 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
```

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 관리되는 어트리뷰트

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.

다음 예에서, *age*는 공용 어트리뷰트이고 *_age*는 개인 어트리뷰트입니다. 공용 어트리뷰트에 액세스하면, 디스크립터는 조회나 갱신을 로그 합니다:

```
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):
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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__()
```

대화 형 세션은 관리되는 어트리뷰트 *age*에 대한 모든 액세스가 로그 되지만, 일반 어트리뷰트 *name*은 로그 되지 않음을 보여줍니다:

```
>>> 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
```

이 예제의 한 가지 주요 문제는 개인 이름 *_age*가 *LoggedAgeAccess* 클래스에 고정되어 있다는 것입니다. 즉, 각 인스턴스는 하나의 로그 되는 어트리뷰트 만 가질 수 있으며 해당 이름을 변경할 수 없습니다. 다음 예에서는, 이 문제를 수정합니다.

1.4 사용자 정의 이름

클래스가 디스크립터를 사용할 때, 어떤 변수 이름이 사용되었는지 각 디스크립터에 알릴 수 있습니다.

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)
```

(다음 페이지에 계속)

```

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'}

```

이제 새 클래스는 `name`과 `age` 모두에 대한 액세스를 로그 합니다:

```

>>> 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

```

두 개의 `Person` 인스턴스에는 개인 이름만 포함됩니다:

```

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

```

1.5 마무리 생각

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.

디스크립터는 클래스 변수로 사용될 때만 작동합니다. 인스턴스에 넣으면, 효과가 없습니다.

디스크립터의 주요 동기는 클래스 변수에 저장된 객체가 어트리뷰트 조회 중에 발생하는 일을 제어 할 수 있도록 하는 혹은 제공하는 것입니다.

전통적으로, 호출하는 클래스가 조회 중에 어떤 일이 일어날지 제어합니다. 디스크립터는 그 관계를 역전시키고 조회 중인 데이터가 발언권을 갖도록 합니다.

디스크립터는 언어 전체에서 사용됩니다. 함수가 연결된 메서드로 바뀌는 방법입니다. `classmethod()`, `staticmethod()`, `property()` 및 `functools.cached_property()` 와 같은 일반적인 도구는 모두 디스크립터로 구현됩니다.

2 완전한 실용적인 예

이 예에서는, 찾기가 매우 어려운 것으로 악명 높은 데이터 손상 버그의 위치를 찾기 위한 실용적이고 강력한 도구를 만듭니다.

2.1 유효성 검사기 클래스

유효성 검사기는 관리되는 어트리뷰트 액세스를 위한 디스크립터입니다. 데이터를 저장하기 전에, 새 값이 다양한 형과 범위 제한을 충족하는지 확인합니다. 이러한 제한 사항이 충족되지 않으면, 데이터 손상을 방지하기 위해 원천에서 예외가 발생합니다.

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 사용자 정의 유효성 검사기

다음은 세 가지 실용적인 데이터 유효성 검사 유틸리티입니다:

- 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}'
            )
        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(
```

(다음 페이지에 계속)


```
f'Expected {self.predicate} to be true for {value!r}'
)
```

2.3 실용적인 응용

실제 클래스에서 데이터 유효성 검사기를 사용하는 방법은 다음과 같습니다:

```
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
```

디스크립터는 잘못된 인스턴스가 만들어지는 것을 방지합니다:

```
>>> 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 기술적인 자습서

다음은 디스크립터의 작동 원리와 세부 사항에 대한 더 기술적인 자습서입니다.

3.1 요약

디스크립터를 정의하고, 프로토콜을 요약하며 디스크립터를 호출하는 방법을 보여줍니다. 객체 관계형 매핑(object relational mappings)이 작동하는 방식을 보여주는 예를 제공합니다.

디스크립터에 대한 학습은 더 큰 도구 집합에 대한 액세스를 제공할 뿐만 아니라, 파이썬의 작동 방식에 대한 심층적인 이해를 만듭니다.

3.2 정의와 소개

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 attribute, it is said to be a descriptor.

어트리뷰트 액세스의 기본 동작은 객체의 디렉터리에서 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, `type(a)`의 메서드 결정 순서로 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.

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 디스크립터 프로토콜

```
descr.__get__(self, obj, type=None)
descr.__set__(self, obj, value)
descr.__delete__(self, obj)
```

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 디렉터리의 항목과 관련하여 재정의가 계산되는 방식이 다릅니다. 인스턴스 디렉터리에 데이터 디스크립터와 이름이 같은 항목이 있으면, 데이터 디스크립터가 우선합니다. 인스턴스의 디렉터리에 비 데이터 디스크립터와 이름이 같은 항목이 있으면, 디렉터리 항목이 우선합니다.

To make a read-only data descriptor, define both `__get__()` and `__set__()` with the `__set__()` raising an `AttributeError` when called. Defining the `__set__()` method with an exception raising placeholder is enough to make it a data descriptor.

3.4 디스크립터 호출의 개요

디스크립터는 `desc.__get__(obj)` 나 `desc.__get__(None, cls)` 로 직접 호출 할 수 있습니다.

하지만 어트리뷰트 액세스 시 디스크립터가 자동으로 호출되는 것이 더 일반적입니다.

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.

호출 세부 사항은 `obj` 가 객체, 클래스 혹은 `super`의 인스턴스인지에 따라 다릅니다.

3.5 인스턴스에서 호출

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.

`a.x`에 대한 디스크립터가 발견되면, `desc.__get__(a, type(a))` 로 호출됩니다.

점 조회의 로직은 `object.__getattribute__()` 에 있습니다. 다음은 순수한 파이썬 등가물입니다:

```
def find_name_in_mro(cls, name, default):
    "Emulate _PyType_Lookup() in Objects/typeobject.c"
    for base in cls.__mro__:
        if name in vars(base):
            return vars(base)[name]
    return default

def object_getattribute(obj, name):
    "Emulate PyObject_GenericGetAttr() in Objects/object.c"
    null = object()
    objtype = type(obj)
    cls_var = find_name_in_mro(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)
```

Note, there is no `__getattr__()` hook in the `__getattribute__()` code. That is why calling `__getattribute__()` directly or with `super().__getattribute__` will bypass `__getattr__()` entirely.

Instead, it is the dot operator and the `getattr()` function that are responsible for invoking `__getattr__()` whenever `__getattribute__()` raises an `AttributeError`. Their logic is encapsulated in 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__'):
```

(다음 페이지에 계속)

```

        raise
    return type(obj).__getattr__(obj, name) # __getattr__

```

3.6 클래스에서 호출

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.

디스크립터가 발견되면, `desc.__get__(None, A)` 로 호출됩니다.

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

3.7 super에서 호출

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

점 조회 `super(A, obj).m`은 `obj.__class__.__mro__`에서 A 바로 다음에 오는 베이스 클래스 B를 검색한 다음 `B.__dict__['m'].__get__(obj, A)`를 반환합니다. 디스크립터가 아니면, m이 변경되지 않은 상태로 반환됩니다.

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 호출 로직 요약

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.
- 클래스는 `object`, `type` 또는 `super()`로부터 이 절차를 상속합니다.
- 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.
- 데이터 디스크립터는 항상 인스턴스 디렉터리를 대체합니다.
- 비 데이터 디스크립터는 인스턴스 디렉터리로 대체될 수 있습니다.

3.9 자동 이름 알림

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 예제

The following code is a simplified skeleton showing how data descriptors could be used to implement an **object relational mapping**.

핵심 아이디어는 데이터가 외부 데이터베이스에 저장된다는 것입니다. 파이썬 인스턴스는 데이터베이스 테이블에 대한 키만 보유하고 있습니다. 디스크립터가 조회나 갱신을 처리합니다:

```
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
```

모델을 사용하려면, 먼저 데이터베이스에 연결하십시오:

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

대화 형 세션은 데이터베이스에서 데이터를 꺼내는 방법과 데이터를 갱신하는 방법을 보여줍니다:

```
>>> 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 순수한 파이썬 등가물

디스크립터 프로토콜은 간단하고 흥미로운 가능성을 제공합니다. 몇 가지 유스 케이스는 아주 흔해서 내장 도구에 미리 패키징되었습니다. 프로퍼티, 연결된 메서드, 정적 메서드, 클래스 메서드 및 `__slots__`는 모두 디스크립터 프로토콜을 기반으로 합니다.

4.1 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
```

설명(doc)은 관리되는 어트리뷰트 `x`를 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.")
```

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
        self.__name__ = ''
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

def __get__(self, obj, objtype=None):
    if obj is None:
        return self
    if self.fget is None:
        raise AttributeError(f"property '{self._name}' has no getter")
    return self.fget(obj)

def __set__(self, obj, value):
    if self.fset is None:
        raise AttributeError(f"property '{self._name}' has no setter")
    self.fset(obj, value)

def __delete__(self, obj):
    if self.fdel is None:
        raise AttributeError(f"property '{self._name}' has no deleter")
    self.fdel(obj)

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

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

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

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
```

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

4.2 함수와 메서드

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

클래스 디렉터리에 저장된 함수는 호출될 때 메서드로 바뀝니다. 객체 인스턴스가 다른 인자들 앞에 추가된다는 점에서만 메서드가 일반 함수와 다릅니다. 관례에 따라, 이 인스턴스는 *self*라고 하지만 *this*나 다른 어떤 변수 이름도 될 수 있습니다.

대략 다음과 동등한 `types.MethodType`을 사용하여 메서드를 수동으로 만들 수 있습니다:

```
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)
```

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)
```

인터프리터에서 다음 클래스를 실행하면 실제로 함수 디스크립터가 작동하는 방식을 보여줍니다:

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

함수에는 내부 검사를 지원하는 정규화된 이름 어트리뷰트가 있습니다:

```
>>> 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>>
```

내부적으로, 연결된 메서드는 하부 함수와 연결된 인스턴스를 저장합니다.

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

>>> d.f.__self__
<__main__.D object at 0x00B18C90>
```

일반 메서드에서 *self*가 어디에서 오는지 또는 클래스 메서드에서 *cls*가 어디에서 오는지 궁금한 적이 있다면, 바로 이겁니다!

4.3 메서드의 종류

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	객체에서 호출	클래스에서 호출
함수	<code>f(obj, *args)</code>	<code>f(*args)</code>
staticmethod	<code>f(*args)</code>	<code>f(*args)</code>
classmethod	<code>f(type(obj), *args)</code>	<code>f(cls, *args)</code>

4.4 정적 메서드

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.

For instance, a statistics package may include a container class for experimental data. The class provides normal methods for computing the average, mean, median, and other descriptive statistics that depend on the data. However, there may be useful functions which are conceptually related but do not depend on the data. For instance, `erf(x)` is handy conversion routine that comes up in statistical work but does not directly depend on a particular dataset. It can be called either from an object or the class: `s.erf(1.5) --> .9332` or `Sample.erf(1.5) --> .9332`.

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

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

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

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

```
import functools

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

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

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

    def __call__(self, *args, **kwargs):
        return self.f(*args, **kwargs)
```

The `functools.update_wrapper()` call adds a `__wrapped__` attribute that refers to the underlying function. Also it carries forward the attributes necessary to make the wrapper look like the wrapped function: `__name__`, `__qualname__`, `__doc__`, and `__annotations__`.

4.5 클래스 메서드

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
```

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

이 동작은 함수가 클래스 참조 만 필요하고 특정 인스턴스에 저장된 데이터에 의존하지 않을 때 유용합니다. 클래스 메서드의 한 가지 용도는 대체 클래스 생성자를 만드는 것입니다. 예를 들어, 클래스 메서드 `dict.fromkeys()` 는 키 리스트에서 새 딕셔너리를 만듭니다. 순수한 파이썬 동등 물은 다음과 같습니다:

```
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
```

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

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

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

```
import functools

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

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

    def __get__(self, obj, cls=None):
        if cls is None:
            cls = type(obj)
        if hasattr(type(self.f), '__get__'):
            # This code path was added in Python 3.9
            # and was deprecated in Python 3.11.
            return self.f.__get__(cls, cls)
        return MethodType(self.f, cls)
```

The code path for `hasattr(type(self.f), '__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. In Python 3.11, this functionality was deprecated.

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

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

The `functools.update_wrapper()` call in `ClassMethod` adds a `__wrapped__` attribute that refers to the underlying function. Also it carries forward the attributes necessary to make the wrapper look like the wrapped function: `__name__`, `__qualname__`, `__doc__`, and `__annotations__`.

4.6 멤버 객체와 `__slots__`

클래스가 `__slots__` 를 정의하면, 인스턴스 디렉터리를 슬롯값의 고정 길이 배열로 바꿉니다. 사용자 관점에서 여러 가지 효과가 있습니다:

1- 철자가 잘못된 어트리뷰트 대입으로 인한 버그를 즉시 감지합니다. `__slots__` 에 지정된 어트리뷰트 이름만 허용됩니다:

```
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 - 디스크립터가 `__slots__`에 저장된 개인 어트리뷰트에 대한 액세스를 관리하는 불변 객체를 만드는 데 도움이 됩니다:

```
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: property 'dept' of 'Immutable' object has no setter
>>> mark.location = 'Mars'
Traceback (most recent call last):
...
AttributeError: 'Immutable' object has no attribute 'location'
```

3 - 메모리를 절약합니다. 64비트 리눅스 빌드에서 두 개의 어트리뷰트가 있는 인스턴스는 `__slots__`가 있으면 48바이트, 없으면 152바이트를 사용합니다. 이 플라이웨이트(flyweight) 디자인 패턴은 많은 수의 인스턴스가 만들어질 때만 중요합니다.

4. Improves speed. Reading instance variables is 35% faster with `__slots__` (as measured with Python 3.10 on an Apple M1 processor).

5. 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.
```

`__slots__`의 정확한 순수 파이썬 드롭인 버전을 만드는 것은 불가능합니다. C 구조체에 직접 액세스하고 객체 메모리 할당을 제어해야 하기 때문입니다. 그러나, 슬롯에 대한 실제 C 구조체가 개인 `_slotvalues` 리스트에 의해 묘사되는 가장 충실한 시뮬레이션을 구축 할 수 있습니다. 해당 개인 구조체에 대한 읽기와 쓰기는 멤버 디스크립터에 의해 관리됩니다:

```
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
        if obj is None:
            return self
        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, **kwargs):
        '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)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
return type.__new__(mcls, clsname, bases, mapping, **kwargs)
```

`object.__new__()` 메서드는 인스턴스 디렉터리 대신 슬롯이 있는 인스턴스를 만드는 것을 권장합니다. 다음은 순수 파이썬의 대략적인 시뮬레이션입니다:

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

    def __new__(cls, *args, **kwargs):
        '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'{cls.__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'{cls.__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'

    slot_names = ['x', 'y']

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

이 시점에서, 메타 클래스는 `x`와 `y`에 대한 멤버 객체를 로드했습니다:

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

인스턴스가 만들어질 때, 어트리뷰트가 저장되는 `slot_values` 리스트를 갖습니다:

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

철자가 틀리거나 지정되지 않은 어트리뷰트는 예외를 발생시킵니다:

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