
Descriptor Guide

Versión 3.14.0rc3

Guido van Rossum and the Python development team

octubre 01, 2025

Python Software Foundation
Email: docs@python.org

Índice general

1	Guía introductoria	3
1.1	Ejemplo simple: un descriptor que retorna una constante	3
1.2	Búsquedas dinámicas	3
1.3	Atributos gestionados	4
1.4	Nombres personalizados	5
1.5	Pensamientos finales	6
2	Ejemplo completo práctico	7
2.1	Clase validadora	7
2.2	Validadores personalizados	7
2.3	Aplicación práctica	8
3	Tutorial técnico	9
3.1	Resumen	9
3.2	Definición e introducción	9
3.3	Protocolo de descriptors	9
3.4	Visión general de invocación de descriptors	10
3.5	Invocación desde una instancia	10
3.6	Invocación desde una clase	11
3.7	Invocación desde super	11
3.8	Resumen de la lógica de invocación	11
3.9	Notificación automática de nombre	11
3.10	Ejemplo de mapeos objeto-relacional (<i>ORM</i>)	12
4	Equivalentes en Python puro	13
4.1	Propiedades	13
4.2	Funciones y métodos	14
4.3	Tipos de métodos	16
4.4	Métodos estáticos	16
4.5	Métodos de clase	17
4.6	Objetos miembros y <code>__slots__</code>	18

Autor

Raymond Hettinger

Contacto

<python at rcn dot com>

Contenido

- *Descriptor Guide*
 - *Guía introductoria*
 - * *Ejemplo simple: un descriptor que retorna una constante*
 - * *Búsquedas dinámicas*
 - * *Atributos gestionados*
 - * *Nombres personalizados*
 - * *Pensamientos finales*
 - *Ejemplo completo práctico*
 - * *Clase validadora*
 - * *Validadores personalizados*
 - * *Aplicación práctica*
 - *Tutorial técnico*
 - * *Resumen*
 - * *Definición e introducción*
 - * *Protocolo de descriptores*
 - * *Visión general de invocación de descriptores*
 - * *Invocación desde una instancia*
 - * *Invocación desde una clase*
 - * *Invocación desde super*
 - * *Resumen de la lógica de invocación*
 - * *Notificación automática de nombre*
 - * *Ejemplo de mapeos objeto-relacional (ORM)*
 - *Equivalentes en Python puro*
 - * *Propiedades*
 - * *Funciones y métodos*
 - * *Tipos de métodos*
 - * *Métodos estáticos*
 - * *Métodos de clase*
 - * *Objetos miembros y __slots__*

Los descriptores permiten a objetos personalizar la búsqueda, almacenamiento y eliminación de atributos.

Esta guía tiene cuatro secciones principales:

- 1) La guía introductoria da una visión general básica, moviéndose gentilmente por ejemplos simples, añadiendo una funcionalidad a la vez. Comienza acá si eres nuevo con los descriptores.
- 2) La segunda sección muestra un ejemplo completo y práctico de un descriptor. Si ya sabes lo básico comienza acá.
- 3) La tercera sección provee un tutorial más técnico que adentra en la mecánica detallada de cómo funcionan los descriptores. La mayoría de la gente no necesita este nivel de detalle.

- 4) La última sección tiene equivalentes en Python puro para descriptores incorporados que están escritos en C. Lee esta sección si tienes curiosidad de cómo las funciones se convierten en métodos vinculados, o sobre la implementación de herramientas comunes como `classmethod()`, `staticmethod()`, `property()`, y `__slots__`.

1 Guía introductoria

En esta guía introductoria comenzamos con el ejemplo más básico posible y luego vamos añadiendo nuevas funcionalidades una a una.

1.1 Ejemplo simple: un descriptor que retorna una constante

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

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

Para usar el descriptor, éste se debe almacenar como una variable de clase en otra clase:

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

Una sesión interactiva muestra la diferencia entre una búsqueda de atributo normal y la búsqueda a través del descriptor:

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

En la búsqueda de atributo `a.x`, el operador punto encuentra 'x': 5 en el diccionario de la clase. En la búsqueda `a.y`, el operador punto encuentra una instancia de un descriptor, reconocible por su método `__get__`. Llamar a ese método retorna 10.

Nota que el valor 10 no es almacenado ni en el diccionario de la clase ni en el diccionario de la instancia. En cambio, el valor 10 es calculado bajo demanda.

Este ejemplo muestra cómo funciona un descriptor simple, pero no es muy útil. Para recuperar constantes una búsqueda de atributos normal sería mejor.

En la próxima sección crearemos algo más útil, una búsqueda dinámica.

1.2 Búsquedas dinámicas

Descriptores interesantes típicamente ejecutan cálculos en vez de retornar constantes:

```
import os

class DirectorySize:

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

class Directory:

    size = DirectorySize()           # Descriptor instance
```

(continúe en la próxima página)

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

Una sesión interactiva muestra que la búsqueda es dinámica — calcula respuestas diferentes y actualizadas en cada ocasión:

```
>>> 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 Atributos gestionados

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.

En el siguiente ejemplo, *age* es el atributo público y *_age* es el atributo privado. Cuando el atributo público es accedido, el descriptor registra la búsqueda o actualización:

```
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__()
```

Una sesión interactiva muestra que todos los accesos al atributo gestionado *age* son registrados, pero que el atributo

normal *name* no es registrado:

```
>>> 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
```

Un gran problema con este ejemplo es que el nombre privado *_age* está fijado en la clase *LoggedAgeAccess*. Esto significa que cada instancia puede sólo puede registrar un atributo, y que su nombre no se puede cambiar. En el siguiente ejemplo solucionaremos ese problema.

1.4 Nombres personalizados

Cuando una clase usa descriptores, puede informar a cada descriptor el nombre se usó para la variable.

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:
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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'}
```

La nueva clase ahora registrar accesos tanto a *name* como a *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
```

Las dos instancias de *Person* contienen sólo los nombres privados:

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

1.5 Pensamientos finales

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.)

Los descriptores son invocados por el operador punto durante la búsqueda de atributos. Si un descriptor es accedido indirectamente con `vars(una_clase)[nombre_del_descriptor]`, la instancia del descriptor es retornada sin ser invocada.

Los descriptores sólo funcionan cuando se usan como variables de clase. Cuando son puestos en una instancia no tienen efecto.

La mayor motivación detrás de los descriptores es el proveer un gancho que permita a los objetos guardados en variables de clase controlar lo que ocurre al buscar un atributo.

Tradicionalmente, la clase que llama controla qué ocurre durante la búsqueda. Los descriptores invierten esta relación y permiten que los datos que están siendo buscados tengan algo que decir al respecto.

Los descriptores se usan a través de todo el lenguaje. Es cómo funciones se convierten en métodos vinculados. Herramientas comunes como `classmethod()`, `staticmethod()`, `property()`, y `functools.cached_property()` se implementan todas como descriptores.

2 Ejemplo completo práctico

En este ejemplo creamos una herramienta práctica y poderosa para encontrar errores de corrupción de datos que son notoriamente difíciles de encontrar.

2.1 Clase validadora

Un validador es un descriptor que da acceso a un atributo gestionado. Antes de almacenar cualquier dato, verifica que el nuevo valor cumple con varias restricciones de tipo y rango. Si esas restricciones no se cumplen, lanza una excepción para así prevenir corrupción de datos en su origen.

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 Validadores personalizados

Acá hay tres utilidades de validación de datos prácticas:

- 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):
```

(continúe en la próxima página)

```

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 a 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 Aplicación práctica

Acá se muestra cómo se puede usar los validadores de datos en una clase real:

```

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

```

Los descriptores previenen que se creen instancias inválidas:

```
>>> Component('Widget', 'metal', 5)      # Blocked: 'Widget' is not all uppercase
```

(continúe en la próxima página)


```

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 Tutorial técnico

Lo que sigue es un tutorial más práctico sobre las mecánicas y detalles de cómo funcionan los descriptores.

3.1 Resumen

Define los descriptores, resume el protocolo, y muestra cómo los descriptores son llamados. Provee ejemplos mostrando cómo funcionan los mapeos objeto-relacional (*ORM*).

Aprender acerca de los descriptores no sólo brinda acceso a un conjunto de herramientas mayor, sino que genera una comprensión más profunda de cómo funciona Python.

3.2 Definición e introducción

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.

El comportamiento predeterminado para el acceso a los atributos es obtener, establecer o eliminar el atributo del diccionario de un objeto. Por ejemplo, `a.x` tiene una cadena de búsqueda que comienza con `a.__dict__['x']`, luego `type(a).__dict__['x']` y continúa a través del orden de resolución de métodos de `type(a)`. Si el valor buscado es un objeto que define uno de los métodos de descriptores, entonces Python puede anular el comportamiento predeterminado e invocar el método del descriptor en su lugar. El lugar donde esto ocurre en la cadena de precedencia depende de qué métodos de descriptores fueron definidos.

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 Protocolo de descriptores

```

descr.__get__(self, obj, type=None)

descr.__set__(self, obj, value)

descr.__delete__(self, obj)

```

Eso es todo lo que hay que hacer. Si se define cualquiera de estos métodos, el objeto se considera un descriptor y puede anular el comportamiento predeterminado al ser buscado como un atributo.

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).

Los descriptores de datos y de no-datos difieren en cómo se calculan las anulaciones con respecto a las entradas en el diccionario de una instancia. Si el diccionario de una instancia tiene una entrada con el mismo nombre que un descriptor de datos, el descriptor de datos tiene prioridad. Si el diccionario de una instancia tiene una entrada con el mismo nombre que un descriptor de no-datos, la entrada del diccionario tiene prioridad.

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 Visión general de invocación de descriptores

Un descriptor puede ser llamado directamente con `desc.__get__(obj)` o `desc.__get__(None, cls)`.

Pero es más común que un descriptor sea invocado automáticamente por la búsqueda de atributos.

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.

Los detalles de la invocación dependen de si `obj` es un objeto una clase, o una instancia de super.

3.5 Invocación desde una instancia

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.

Si se encuentra un descriptor para `a.x` entonces se invoca con `desc.__get__(a, type(a))`.

La lógica para una búsqueda con puntos se encuentra en `object.__getattribute__()`. Acá hay un equivalente en Python puro:

```
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__'):
            raise
        return type(obj).__getattr__(obj, name)           # __getattr__
```

3.6 Invocación desde una clase

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.

Si se encuentra un descriptor, se invoca con `desc.__get__(None, A)`.

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

3.7 Invocación desde super

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

Una búsqueda con puntos tal como `super(A, obj).m` busca `obj.__class__.__mro__` para la clase base B que sigue inmediatamente a A y luego retorna `B.__dict__['m'].__get__(obj, A)`. Si no es un descriptor, m se retorna sin cambiar.

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 Resumen de la lógica de invocación

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

Los puntos importantes a recordar son:

- Descriptors are invoked by the `__getattribute__()` method.
- Las clases heredan esta maquinaria desde `object`, `type`, o `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.
- Los descriptors de datos siempre anulan los diccionarios de instancia.
- Los descriptors de no-datos pueden ser reemplazados por los diccionarios de instancia.

3.9 Notificación automática de nombre

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 Ejemplo de mapeos objeto-relacional (ORM)

El siguiente código es un esqueleto simplificado que muestra cómo descriptores de datos pueden ser usados para implementar un mapeo objeto-relacional.

La idea esencial es que los datos se almacenan en una base de datos externa. Las instancias de Python sólo mantienen llaves a las tablas de la base de datos. Los descriptores se hacen cargo de las búsquedas o actualizaciones:

```
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
```

Para usar los modelos, primera conéctate a la base de datos:

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

Una sesión interactiva muestra cómo los datos son obtenidos desde la base de datos y cómo se pueden actualizar:

```
>>> 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
```

(continúe en la próxima página)

```
'John Denver'

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

4 Equivalentes en Python puro

El protocolo de descriptores es simple y ofrece posibilidades estimulantes. Varios casos de uso son tan comunes que han sido pre-empaquetados en herramientas incorporadas. Propiedades, métodos vinculados, métodos estáticos, métodos de clase y `__slots__` están todos basados en el protocolo de descriptores.

4.1 Propiedades

Llamar a `property()` es una forma sucinta de construir un descriptor de datos que desencadena llamadas a funciones al acceder a un atributo. Su firma es:

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

La documentación muestra un uso típico para definir un atributo gestionado `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 that implements most of the core functionality:

```
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 __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
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError
        self.fset(obj, value)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
def __delete__(self, obj):
    if self.fdel is None:
        raise AttributeError
    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__)
```

La función incorporada `property()` es de ayuda cuando una interfaz de usuario ha otorgado acceso a atributos y luego los cambios posteriores requieren la intervención de un método.

Por ejemplo, una clase de hoja de cálculo puede otorgar acceso al valor de una celda a través de `Cell('b10').value`. Las mejoras posteriores del programa requieren que la celda se vuelva a calcular en cada acceso; sin embargo, la programadora no quiere afectar al código de cliente existente que accede al atributo directamente. La solución es envolver el acceso al valor del atributo en un descriptor de datos propiedad:

```
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 Funciones y métodos

Las características orientadas a objetos de Python se basan en un entorno basado en funciones. Usando descriptores de no-datos, ambas se combinan perfectamente.

Las funciones almacenadas en diccionarios de clase son convertidas en métodos cuando son invocadas. Los métodos sólo difieren de funciones regulares en que la instancia del objeto es antepuesta a los otros argumentos. Por convención, la instancia se llama *self*, pero podría ser llamada *this* o cualquier otro nombre de variable.

Los métodos se pueden crear manualmente con `types.MethodType`, lo que es aproximadamente equivalente a:

```
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)

    def __getattr__(self, name):
        "Emulate method_getset() in Objects/classobject.c"
```

(continúe en la próxima página)

(proviene de la página anterior)

```
if name == '__doc__':
    return self.__func__.__doc__
return object.__getattribute__(self, name)

def __getattr__(self, name):
    "Emulate method_getattro() in Objects/classobject.c"
    return getattr(self.__func__, name)

def __get__(self, obj, objtype=None):
    "Emulate method_descr_get() in Objects/classobject.c"
    return self
```

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)
```

Ejecutar la siguiente clase en el intérprete muestra cómo funciona el descriptor de función en la práctica:

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

class D2:
    pass
```

La función tiene un atributo de nombre calificado para soportar introspección:

```
>>> 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>>
```

Internamente, el método vinculado guarda la función subyacente y la instancia vinculada:

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

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

Si alguna vez te preguntaste de dónde viene *self* en métodos regulares, o de dónde viene *cls* en métodos de clase, ¡es acá!

4.3 Tipos de métodos

Los descriptores de no-datos proporcionan un mecanismo simple para variaciones de los patrones habituales para vincular funciones en métodos.

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)`.

Este cuadro resume el enlace (*binding*) y sus dos variantes más útiles:

Transformación	Llamado desde un objeto	Llamado desde una clase
función	<code>f(obj, *args)</code>	<code>f(*args)</code>
método estático	<code>f(*args)</code>	<code>f(*args)</code>
método de clase	<code>f(type(obj), *args)</code>	<code>f(cls, *args)</code>

4.4 Métodos estáticos

Los métodos estáticos retornan la función subyacente sin cambios. Llamar a `c.f` o `C.f` es equivalente a una búsqueda directa en `object.__getattr__`(`c`, "f") o en `object.__getattr__`(`C`, "f"). Como resultado, la función se vuelve idénticamente accesible desde un objeto o una clase.

Buenos candidatos para ser métodos estáticos son los métodos que no hacen referencia a la variable `self`.

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) --> 0.9332` or `Sample.erf(1.5) --> 0.9332`.

Dado que los métodos estáticos retornan la función subyacente sin cambios, las llamadas de ejemplo carecen de interés:

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

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

Usando el protocolo de descriptores de no-datos, una versión pura de Python de `staticmethod()` se vería así:

```
import functools

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

(continúe en la próxima página)


```

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)

@property
def __annotations__(self):
    return self.f.__annotations__

```

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, including `__name__`, `__qualname__`, and `__doc__`.

4.5 Métodos de clase

A diferencia de los métodos estáticos, los métodos de clase anteponen la referencia de clase a la lista de argumentos antes de llamar a la función. Este formato es el mismo si quien llama es un objeto o una clase:

```

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

```

```

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

```

Este comportamiento es útil siempre que la función solo necesite tener una referencia de clase y no necesita contar con los datos almacenados en una instancia específica. Un uso de los métodos de clase es crear constructores de clase alternativos. Por ejemplo, el método de clase `dict.fromkeys()` crea un nuevo diccionario a partir de una lista de claves. El equivalente puro de Python es:

```

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

```

Ahora se puede construir un nuevo diccionario de claves únicas así:

```

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

```

Usando el protocolo de descriptores de no-datos, una implementación pura en Python de `classmethod()` se vería así:

```
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)
        return MethodType(self.f, cls)
```

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 Objetos miembros y `__slots__`

Cuando una clase define `__slots__`, reemplaza los diccionarios de instancia por un arreglo de valores de ranura de largo fijo. Desde el punto de vista del usuario esto tiene varios efectos:

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'
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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. 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. 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.
```

No es posible crear una versión exacta de `__slots__` en Python puro porque requiere acceso directo a estructuras en C y control sobre asignación de memoria de objetos. Sin embargo podemos construir una simulación casi totalmente fiel donde la estructura real en C para las ranuras es emulada con una lista privada `_slotvalues`. Las lecturas y escrituras de esta estructura privada se manejan con descriptores miembros:

```
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
```

(continúe en la próxima página)

```

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

```

El método `object.__new__()` se hace cargo de crear instancias que tienen ranuras en vez un diccionario de instancia. Acá hay una simulación aproximada en Python puro:

```

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(

```

(continúe en la próxima página)

(proviene de la página anterior)

```
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
```

En este punto, la metaclasses ha cargado los objetos miembros para *x* e *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'>}
```

Cuando se crean instancias, éstas tienen una lista `slot_values` donde se almacenan los atributos:

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

Atributos mal deletreados o no asignados lanzarán una excepción:

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