

---

# Guía práctica de uso de los descriptores

Versión 3.11.13

Guido van Rossum and the Python development team

julio 07, 2025

Python Software Foundation  
Email: docs@python.org

## Índice general

<b>1</b>	<b>Guía introductoria</b>	<b>3</b>
1.1	Ejemplo simple: un descriptor que retorna una constante	3
1.2	Búsquedas dinámicas	4
1.3	Atributos gestionados	4
1.4	Nombres personalizados	5
1.5	Pensamientos finales	6
<b>2</b>	<b>Ejemplo completo práctico</b>	<b>7</b>
2.1	Clase validadora	7
2.2	Validadores personalizados	7
2.3	Aplicación práctica	8
<b>3</b>	<b>Tutorial técnico</b>	<b>9</b>
3.1	Resumen	9
3.2	Definición e introducción	9
3.3	Protocolo de descriptores	10
3.4	Visión general de invocación de descriptores	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	12
3.9	Notificación automática de nombre	12
3.10	Ejemplo de mapeos objeto-relacional ( <i>ORM</i> )	12
<b>4</b>	<b>Equivalentes en Python puro</b>	<b>13</b>
4.1	Propiedades	13
4.2	Funciones y métodos	15
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

- *Guía práctica de uso de los descriptores*
  - *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 descriptors. La mayoría de la gente no necesita este nivel de detalle.
- 4) La última sección tiene equivalentes en Python puro para descriptors 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

La clase `Ten` es un descriptor cuyo método `__get__()` siempre retorna la constante `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

    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
```

Además de mostrar cómo los descriptores puede ejecutar cálculos, este ejemplo también revela el propósito de los parámetros de `__get__()`. El parámetro `self` es `size`, una instancia de `DirectorySize`. El parámetro `obj` es `g` o `s`, una instancia de `Directory`. Es el parámetro `obj` el que permite que al método `__get__()` saber el directorio objetivo. El parámetro `objtype` es una clase `Directory`.

## 1.3 Atributos gestionados

Un uso popular de descriptores es la gestión de acceso a datos de una instancia. El descriptor se asigna a un atributo público en el diccionario de clase, mientras que los datos reales se guardan en atributos privados en el diccionario de instancia. Los métodos `__get__()` and `__set__()` del descriptor se activan cuando se accede al atributo público.

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
```

(continúe en la próxima página)

```

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.

En este ejemplo, la clase *Person* tiene dos instancias de descriptores, *name* y *age*. Cuando la clase *Person* se define, hace una retrollamada a *\_\_set\_name\_\_()* en *LoggedAccess* para que se pueda registrar los nombres de los campos, dándole a cada descriptor su propio *public\_name* y *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

```

(continúe en la próxima página)

(proviene de la página anterior)

```
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
```

Una sesión interactiva muestra que la clase `Person` ha llamado a `__set_name__()` para que los nombres de los campos sean registrados. Aquí llamamos a `vars()` para ver el descriptor sin activarlos:

```
>>> 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

Llamamos un descriptor a cualquier objeto que define `__get__()`, `__set__()` o `__delete__()`.

Opcionalmente, los descriptors pueden tener un método `__set_name__()`. Éste sólo se usa en los casos en los que el descriptor necesita saber ya sea la clase donde fue creado, o el nombre de la variable de clase a la que fue asignado. (Este método, si está presente, es llamada incluso si la clase no es un descriptor.)

Los descriptors 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 descriptors 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.

Esta clase `Validator` es una tanto una clase base abstracta como un descriptor de un atributo gestionado:

```
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
```

Validadores personalizados necesitan heredar de `Validator` y deben proveer un método `validate()` method para probar las restricciones que sean necesarias.

### 2.2 Validadores personalizados

Acá hay tres utilidades de validación de datos prácticas:

- 1) `OneOf` verifica que un valor está dentro de un grupo restringido de opciones.
- 2) `Number` verifica que un valor es `int` o `float`. Opcionalmente verifica que un valor está entre un mínimo y un máximo.
- 3) `String` verifica que un valor es un `str`. Opcionalmente valida que tenga un largo mínimo o máximo. Puede también validar un `predicado` definido por el usuario.

```
class OneOf(Validator):

    def __init__(self, *options):
```

(continúe en la próxima página)

```

        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(
                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

```

(continúe en la próxima página)



```
self.kind = kind
self.quantity = quantity
```

Los descriptores previenen que se creen instancias inválidas:

```
>>> 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 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

En general, un descriptor es un valor atributo que tiene uno de los métodos del protocolo de descriptores. Estos métodos son `__get__()`, `__set__()`, y `__delete__()`. Si cualquiera de esos métodos se definen en un atributo, se dice que éste es un 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.

Los descriptores son un protocolo poderoso y de propósito general. Son el mecanismo detrás de propiedades, métodos, métodos estáticos y `super()`. Se usan a través de Python mismo. Los descriptores simplifican el código C subyacente y ofrecen un grupo flexible de nuevas herramientas para programas habituales de Python.

### 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.

Si un objeto define `__set__()` o `__delete__()`, se considera un descriptor de datos. Los descriptors que solo definen `__get__()` se denominan descriptors de no-datos (normalmente se utilizan para métodos, pero son posibles otros usos).

Los descriptors 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.

Para crear un descriptor de datos de sólo lectura, define tanto `__get__()` como `__set__()` donde `__set__()` lanza un `AttributeError` cuando es llamado. Definir el método `__set__()` de forma que lance una excepción genérica es suficiente para convertirlo en un descriptor de datos.

### 3.4 Visión general de invocación de descriptors

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

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

La expresión `obj.x` busca el atributo `x` en la cadena de nombres de espacio de `obj`. Si la búsqueda encuentra un descriptor fuera del `__dict__` de la instancia, su método `__get__()` es invocado de acuerdo a la lista de reglas de precedencia mostradas debajo.

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

La búsqueda en instancias escanea a través de una cadena de nombres de espacio dando la más alta prioridad a descriptors de datos, seguidos por variables de instancia, luego descriptors de no-datos, luego variables de clase, y finalmente a `__getattr__()` si se provee.

Si se encuentra un descriptor para `a.x` entonces se invoca con `descr.__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__'))
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    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)
```

Nota, no hay un gancho `__getattr__()` en el código de `__getattribute__()`. Es por eso que llamar a `__getattribute__()` directamente o con `super().__getattribute__` evitará completamente a `__getattr__()`.

En cambio, es el operador punto y la función `getattr()` los que son responsables de invocar `__getattr__()` cada vez que `__getattribute__()` lanza un `AttributeError`. Su lógica está encapsulada en una función auxiliar:

```
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

La lógica para una búsqueda con puntos tal como `A.x` se encuentra en `type.__getattribute__()`. Los pasos son similares a los de `object.__getattribute__()`, pero la búsqueda en el diccionario de instancia se reemplaza por una búsqueda a través del orden de resolución de métodos de la clase.

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

La lógica de la búsqueda con puntos para `super` está en el método `__getattribute__()` para el objeto retornado por `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

El mecanismo de descriptores está embebido en los métodos `__getattribute__()` de `object`, `type`, y `super()`.

Los puntos importantes a recordar son:

- Los descriptores son invocados por el método `__getattribute__()`.
- Las clases heredan esta maquinaria desde `object`, `type`, o `super()`.
- Redefinir `__getattribute__()` previene las llamadas automáticas a descriptores porque toda la lógica de descriptores está en ese método.
- `object.__getattribute__()` y `type.__getattribute__()` realizan diferentes llamadas a `__get__()`. El primero incluye la instancia y puede incluir la clase. El segundo establece `None` como instancia, y siempre incluye la clase.
- Los descriptores de datos siempre anulan los diccionarios de instancia.
- Los descriptores de no-datos pueden ser reemplazados por los diccionarios de instancia.

## 3.9 Notificación automática de nombre

A veces es deseable que un descriptor sepa qué nombre fue asignado a una variable de clase. Cuando una nueva clase es creada, la metaclass `type` escanea el diccionario de la nueva clase. Si alguna de las entradas es un descriptor, y si define `__set_name__()`, ese método se llama con dos argumentos. El argumento *owner* es la clase donde se usa el descriptor, y *name* es la variable de clase a la cual el descriptor se asigna.

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

Dado que la lógica de actualización está en `type.__new__()`, las notificaciones ocurren sólo al momento de crear la clase. Si se añade descriptores a la clase más tarde, `__set_name__()` tendrá que ser llamado manualmente.

## 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()
```

Podemos usar la clase `Field` para definir modelos que describen el esquema para cada tabla en la base de datos:

```
class Movie:
    table = 'Movies'           # Table name
    key = 'title'              # Primary key
    director = Field()
    year = Field()
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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
'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.")
```

Para ver cómo se implementa `property()` en términos del protocolo de descriptores, aquí hay un equivalente puro de Python:

```

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 = ''

    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

```

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()

```

(continúe en la próxima página)

```
return self._value
```

Tanto la función incorporada `property()` como nuestra equivalente `Property()` funcionarían en este ejemplo.

## 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)
```

Para soportar la creación automática de métodos, las funciones incluyen un método `__get__()` para vincular métodos durante el acceso a atributos. Esto significa que las funciones son descriptores de no-datos que retornan métodos vinculados durante la búsqueda con puntos desde una instancia. Así es como funciona:

```
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, x):
        return x
```

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

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

Accediendo a la función a través del diccionario de clase no invoca `__get__()`. En cambio, retorna el objeto función subyacente:

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

Acceso con puntos desde una clase llama a `__get__()`, lo cual sólo retorna la función subyacente sin cambiar:

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

El comportamiento interesante ocurre durante el acceso con puntos desde una instancia. Las búsquedas con punto llaman a `__get__()`, el cual retorna un objeto de método vinculado:

```
>>> 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.

Para recapitular, las funciones tienen un método `__get__()` para que se puedan convertir en un método cuando se accede a ellas como atributos. El descriptor de no-datos transforma una llamada a `obj.f(*args)` en `f(obj, *args)`. Llamar a `cls.f(*args)` se convierte en `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`.

Por ejemplo, un paquete de estadística puede incluir una clase contenedora para datos experimentales. La clase proporciona métodos normales para calcular el promedio, la media, la mediana y otras estadísticas descriptivas que dependen de los datos. Sin embargo, puede haber funciones útiles que están relacionadas conceptualmente pero que no dependen de los datos. Por ejemplo, `erf(x)` es una práctica rutinaria de conversión que surge en el trabajo estadístico pero que no depende directamente de un conjunto de datos en particular. Se puede llamar desde un objeto o la clase: `s.erf(1.5) --> .9332` o `Sample.erf(1.5) --> .9332`.

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

```
class E:
    @staticmethod
```

(continúe en la próxima página)



(proviene de la página anterior)

```
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"  
  
    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, **kwds):  
        return self.f(*args, **kwds)
```

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 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)
        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)
```

La ruta de código para `hasattr(obj, '__get__')` fue añadida en Python 3.9, y hace posible que `classmethod()` soporte decoradores encadenados. Por ejemplo, un `classmethod` y un `property` se puede encadenar. En Python 3.11, esta funcionalidad fue marcada como obsoleta.

```
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 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'
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()
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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}>'
```

El método `type.__new__()` se hace cargo de añadir objetos miembros a variables de clase:

```
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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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)
```

Para usar la simulación en una clase real, sólo hereda de `Object` y establece metaclass a `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 metaclass ha cargado los objetos miembros para `x` y `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'
```