# Descriptor Guide

***Version 3.12.9***

**Guido van Rossum and the Python development team**

**mars 10, 2025**

**Python Software Foundation**
**Email : docs@python.org**

## Table des matières



**Auteur**
Raymond Hettinger

**Contact**
<python at rcn dot com>

**Sommaire**



Les descripteurs permettent de personnaliser la recherche, le stockage et la suppression des attributs des objets.

Ce guide comporte quatre parties principales :

1) l'« introduction » donne un premier aperçu, en partant d'exemples simples, puis en ajoutant une fonctionnalité à la fois. Commencez par là si vous débutez avec les descripteurs ;
2) la deuxième partie montre un exemple de descripteur complet et pratique. Si vous connaissez déjà les bases, commencez par là ;
3) la troisième partie fournit un didacticiel plus technique qui décrit de manière détaillée comment fonctionnent les descripteurs. La plupart des gens n'ont pas besoin de ce niveau de détail ;
4) la dernière partie contient des équivalents en pur Python des descripteurs natifs écrits en C. Lisez ceci si vous êtes curieux de savoir comment les fonctions se transforment en méthodes liées ou si vous voulez connaître l'implémentation d'outils courants comme `classmethod()`, `staticmethod()`, `property()` et __slots__.

# 1 Introduction

Dans cette introduction, nous commençons par l'exemple le plus simple possible, puis nous ajoutons de nouvelles fonctionnalités une par une.

## 1.1 Un exemple simple : un descripteur qui renvoie une 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
```

Pour utiliser le descripteur, il doit être stocké en tant que variable de classe dans une autre classe :

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

Une session interactive montre la différence entre la recherche d'attribut normale et la recherche *via* un descripteur :

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

Dans la recherche d'attribut `a.x`, l'opérateur « point » trouve `'x': 5` dans le dictionnaire de classe. Dans la recherche `a.y`, l'opérateur « point » trouve une instance de descripteur, reconnue par sa méthode `__get__`. L'appel de cette méthode renvoie `10`.

Notez que la valeur `10` n'est stockée ni dans le dictionnaire de classe ni dans le dictionnaire d'instance. Non, la valeur `10` est calculée à la demande.

Cet exemple montre comment fonctionne un descripteur simple, mais il n'est pas très utile. Pour récupérer des constantes, une recherche d'attribut normale est préférable.

Dans la section suivante, nous allons créer quelque chose de plus utile, une recherche dynamique.

## 1.2 Recherches dynamiques

Les descripteurs intéressants exécutent généralement des calculs au lieu de renvoyer des 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
```

Une session interactive montre que la recherche est dynamique — elle calcule des réponses différentes, mises à jour à chaque fois :

```
>>> s = Directory('songs')
>>> g = Directory('games')
>>> s.size                             # The songs directory has twenty files
20
>>> g.size                             # The games directory has three files
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 Attributs gérés

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.

Dans l'exemple qui suit, *age* est l'attribut public et *_age* est l'attribut privé. Lors de l'accès à l'attribut public, le descripteur journalise la recherche ou la mise à jour :

```
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__()
```

Une session interactive montre que tous les accès à l'attribut géré *age* sont consignés, mais que rien n'est journalisé pour l'attribut normal *name* :

```
>>> 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}
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> 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 problème majeur avec cet exemple est que le nom privé *_age* est écrit en dur dans la classe *LoggedAgeAccess*. Cela signifie que chaque instance ne peut avoir qu'un seul attribut journalisé et que son nom est immuable. Dans l'exemple suivant, nous allons résoudre ce problème.

## 1.4 Noms personnalisés

Lorsqu'une classe utilise des descripteurs, elle peut informer chaque descripteur du nom de variable utilisé.

In this example, the `Person` class has two descriptor instances, *name* and *age*. When the `Person` class is defined, it makes a callback to `__set_name__()` in *LoggedAccess* so that the field names can be recorded, giving each descriptor its own *public_name* and *private_name* :

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r', self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', self.public_name, value)
        setattr(obj, self.private_name, value)

class Person:

    name = LoggedAccess()                # First descriptor instance
    age = LoggedAccess()                 # Second descriptor instance

    def __init__(self, name, age):
        self.name = name                 # Calls the first descriptor
        self.age = age                   # Calls the second descriptor

    def birthday(self):
        self.age += 1
```

An interactive session shows that the `Person` class has called `__set_name__()` so that the field names would be

recorded. Here we call `vars()` to look up the descriptor without triggering it :

```
>>> vars(vars(Person)['name'])
{'public_name': 'name', 'private_name': '_name'}
>>> vars(vars(Person)['age'])
{'public_name': 'age', 'private_name': '_age'}
```

La nouvelle classe enregistre désormais l'accès à la fois à *name* et *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
```

Les deux instances de *Person* ne contiennent que les noms privés :

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

### 1.5 Réflexions 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.)

Les descripteurs sont invoqués par l'opérateur « point » lors de la recherche d'attribut. Si on accède indirectement au descripteur avec `vars(some_class)[descriptor_name]`, l'instance du descripteur est renvoyée sans l'invoquer.

Les descripteurs ne fonctionnent que lorsqu'ils sont utilisés comme variables de classe. Lorsqu'ils sont placés dans des instances, ils n'ont aucun effet.

La principale raison d'être des descripteurs est de fournir un point d'entrée permettant aux objets stockés dans des variables de classe de contrôler ce qui se passe lors de la recherche d'attributs.

Traditionnellement, la classe appelante contrôle ce qui se passe pendant la recherche. Les descripteurs inversent cette relation et permettent aux données recherchées d'avoir leur mot à dire.

Les descripteurs sont utilisés partout dans le langage. C'est ainsi que les fonctions se transforment en méthodes liées. Les outils courants tels que `classmethod()`, `staticmethod()`, `property()` et `functools.cached_property()` sont tous implémentés en tant que descripteurs.

## 2 Exemple complet pratique

Dans cet exemple, nous créons un outil pratique et puissant pour localiser les bogues de corruption de données notoirement difficiles à trouver.

### 2.1 Classe « validateur »

Un validateur est un descripteur pour l'accès aux attributs gérés. Avant de stocker des données, il vérifie que la nouvelle valeur respecte différentes restrictions de type et de plage. Si ces restrictions ne sont pas respectées, il lève une exception pour empêcher la corruption des données à la source.

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 Validateurs personnalisés

Voici trois utilitaires concrets de validation de données :

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}'
            )
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 Application pratique

Voici comment les validateurs de données peuvent être utilisés par une classe réelle :

```
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
```

Les descripteurs empêchent la création d'instances non valides :

```
>>> 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 Tutoriel technique

Ce qui suit est un tutoriel plus technique relatif aux mécanismes et détails de fonctionnement des descripteurs.

## 3.1 Résumé

Ce tutoriel définit des descripteurs, résume le protocole et montre comment les descripteurs sont appelés. Il fournit un exemple montrant comment fonctionnent les correspondances relationnelles entre objets.

L'apprentissage des descripteurs permet non seulement d'accéder à un ensemble d'outils plus vaste, mais aussi de mieux comprendre le fonctionnement de Python.

## 3.2 Définition et introduction

In general, a descriptor is an attribute value that has one of the methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an attribute, it is said to be a descriptor.

Le comportement par défaut pour l'accès aux attributs consiste à obtenir, définir ou supprimer l'attribut dans le dictionnaire d'un objet. Par exemple, pour chercher `a.x` Python commence par chercher `a.__dict__['x']`, puis `type(a).__dict__['x']`, et continue la recherche en utilisant la MRO (l'ordre de résolution des méthodes) de `type(a)`. Si la valeur recherchée est un objet définissant l'une des méthodes de descripteur, Python remplace le comportement par défaut par un appel à la méthode du descripteur. Le moment où cela se produit dans la chaîne de recherche dépend des méthodes définies par le descripteur.

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 Protocole descripteur

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

```
descr.__set__(self, obj, value)
```

```
descr.__delete__(self, obj)
```

C'est tout ce qu'il y a à faire. Définissez n'importe laquelle de ces méthodes et un objet est considéré comme un descripteur ; il peut alors remplacer le comportement par défaut lorsqu'il est recherché en tant qu'attribut.

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).

Les descripteurs de données et les descripteurs hors-données diffèrent dans la façon dont les changements de comportement sont calculés en ce qui concerne les entrées du dictionnaire d'une instance. Si le dictionnaire d'une instance comporte une entrée portant le même nom qu'un descripteur de données, le descripteur de données est prioritaire. Si le dictionnaire d'une instance comporte une entrée portant le même nom qu'un descripteur hors-données, l'entrée du dictionnaire a la priorité.

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 Présentation de l'appel de descripteur

Un descripteur peut être appelé directement par `desc.__get__(obj)` ou `desc.__get__(None, cls)`.

Mais il est plus courant qu'un descripteur soit invoqué automatiquement à partir d'un accès à un attribut.

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.

Les détails de l'appel varient selon que `obj` est un objet, une classe ou une instance de *super*.

## 3.5 Appel depuis une instance

Instance lookup scans through a chain of namespaces giving data descriptors the highest priority, followed by instance variables, then non-data descriptors, then class variables, and lastly `__getattr__()` if it is provided.

Si un descripteur est trouvé pour `a.x`, alors il est appelé par `desc.__get__(a, type(a))`.

La logique d'une recherche « après un point » se trouve dans `object.__getattribute__()`. Voici un équivalent en Python pur :

```
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 Appel depuis une classe

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 un descripteur est trouvé, il est appelé par `desc.__get__(None, A)`.

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

## 3.7 Appel depuis super

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

La recherche d'attribut `super(A, obj).m` recherche dans `obj.__class__.__mro__` la classe `B` qui suit immédiatement A, et renvoie `B.__dict__['m'].__get__(obj, A)`. Si ce n'est pas un descripteur, `m` est renvoyé inchangé.

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 Résumé de la logique d'appel

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

Les points importants à retenir sont :

- Descriptors are invoked by the `__getattribute__()` method.
- les classes héritent ce mécanisme de `object`, `type` ou `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.
- les descripteurs de données sont toujours prioritaires sur les dictionnaires d'instances.
- les descripteurs hors-données peuvent céder la priorité aux dictionnaires d'instance.

## 3.9 Notification automatique des noms

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 Exemple d'ORM

Le code suivant est une ossature simplifiée montrant comment les descripteurs de données peuvent être utilisés pour implémenter une correspondance objet-relationnel.

L'idée essentielle est que les données sont stockées dans une base de données externe. Les instances Python ne contiennent que les clés des tables de la base de données. Les descripteurs s'occupent des recherches et des mises à jour :

```
class Field:

    def __set_name__(self, owner, name):
        self.fetch = f'SELECT {name} FROM {owner.table} WHERE {owner.key}=?;'
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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
```

Pour utiliser les modèles, connectons-nous d'abord à la base de données :

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

Une session interactive montre comment les données sont extraites de la base de données et comment elles peuvent être mises à jour :

```
>>> 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 Équivalents en Python pur

Le protocole descripteur est simple et offre des possibilités très intéressantes. Plusieurs cas d'utilisation sont si courants qu'ils ont été regroupés dans des outils intégrés. Les propriétés, les méthodes liées, les méthodes statiques et les méthodes de classe sont toutes basées sur le protocole descripteur.

## 4.1 Propriétés

Appeler `property()` construit de façon succincte un descripteur de données qui déclenche un appel de fonction lors de l'accès à un attribut. Sa signature est :

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

La documentation montre une utilisation caractéristique pour définir un attribut géré `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.")
```

Pour voir comment `property()` est implémentée dans le protocole du descripteur, voici un équivalent en Python pur :

```
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!r} of {type(obj).__name__!r} object has no␣
↪getter'
             )
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError(
                f'property {self._name!r} of {type(obj).__name__!r} object has no␣
↪setter'
             )
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError(
                f'property {self._name!r} of {type(obj).__name__!r} object has no␣
↪deleter'
             )
        self.fdel(obj)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 fonction native `property()` aide chaque fois qu'une interface utilisateur a accordé l'accès à un attribut et que des modifications ultérieures nécessitent l'intervention d'une méthode.

Par exemple, une classe de tableur peut donner accès à une valeur de cellule via `Cell('b10').value`. Les améliorations ultérieures du programme exigent que la cellule soit recalculée à chaque accès ; cependant, le programmeur ne veut pas impacter le code client existant accédant directement à l'attribut. La solution consiste à envelopper l'accès à l'attribut *value* dans un descripteur de données :

```
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 Fonctions et méthodes

Les fonctionnalités orientées objet de Python sont construites sur un environnement basé sur des fonctions. À l'aide de descripteurs hors-données, les deux sont fusionnés de façon transparente.

Les fonctions placées dans les dictionnaires des classes sont transformées en méthodes au moment de l'appel. Les méthodes ne diffèrent des fonctions ordinaires que par le fait que le premier argument est réservé à l'instance de l'objet. Par convention Python, la référence de l'instance est appelée *self*, bien qu'il soit possible de l'appeler *this* ou tout autre nom de variable.

Les méthodes peuvent être créées manuellement avec `types.MethodType`, qui équivaut à peu près à :

```
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)
```

L'exécution de la classe suivante dans l'interpréteur montre comment le descripteur de fonction se comporte en pratique :

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

La fonction possède un attribut `__qualname__` (nom qualifié) pour prendre en charge l'introspection :

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

Accessing the function through the class dictionary does not invoke `__get__()`. Instead, it just returns the underlying function object :

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

Dotted access from a class calls `__get__()` which just returns the underlying function unchanged :

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

The interesting behavior occurs during dotted access from an instance. The dotted lookup calls `__get__()` which returns a bound method object :

```
>>> d = D()
>>> d.f
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

En interne, la méthode liée stocke la fonction sous-jacente et l'instance liée :

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

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

Si vous vous êtes déjà demandé d'où vient *self* dans les méthodes ordinaires ou d'où vient *cls* dans les méthodes de classe, c'est ça !

## 4.3 Types de méthodes

Les descripteurs hors-données constituent un moyen simple pour modifier le modèle usuel de transformation des fonctions en méthodes.

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)`.

Ce tableau résume le lien classique (*binding*) et ses deux variantes les plus utiles :

| Transformation | Appelée depuis un objet | Appelée depuis une classe |
|---|---|---|
| fonction | f(obj, *args) | f(*args) |
| méthode statique | f(*args) | f(*args) |
| méthode de classe | f(type(obj), *args) | f(cls, *args) |

## 4.4 Méthodes statiques

Les méthodes statiques renvoient la fonction sous-jacente sans modification. Appeler `c.f` ou `C.f` est l'équivalent d'une recherche directe dans `objet.__getattribute__(c, "f")` ou `objet.__getattribute__(C, "f")`. Par conséquent, l'accès à la fonction devient identique que ce soit à partir d'un objet ou d'une classe.

Les bonnes candidates pour être méthode statique sont des méthodes qui ne font pas référence à la variable `self`.

Par exemple, un paquet traitant de statistiques peut inclure une classe qui est un conteneur pour des données expérimentales. La classe fournit les méthodes normales pour calculer la moyenne, la médiane et d'autres statistiques descriptives qui dépendent des données. Cependant, il peut y avoir des fonctions utiles qui sont conceptuellement liées mais qui ne dépendent pas des données. Par exemple, la fonction d'erreur `erf(x)` est souvent utile lorsqu'on travaille en statistique mais elle ne dépend pas directement d'un ensemble de données particulier. Elle peut être appelée à partir d'un objet ou de la classe : `s.erf(1.5) --> .9332` ou `Sample.erf(1.5) --> .9332`.

Puisque les méthodes statiques renvoient la fonction sous-jacente sans changement, les exemples d'appels sont d'une grande banalité :

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

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

En utilisant le protocole de descripteur hors-données, une version Python pure de `staticmethod()` ressemblerait à ceci :

```
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 Méthodes de classe

Contrairement aux méthodes statiques, les méthodes de classe ajoutent la référence de classe en tête de la liste d'arguments, avant d'appeler la fonction. C'est le même format que l'appelant soit un objet ou une classe :

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

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

Ce comportement est utile lorsque la fonction n'a besoin que d'une référence de classe et ne se soucie pas des données propres à une instance particulière. Une des utilisations des méthodes de classe est de créer des constructeurs de classe personnalisés. Par exemple, la méthode de classe `dict.fromkeys()` crée un nouveau dictionnaire à partir d'une liste de clés. L'équivalent Python pur est :

```
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
```

Maintenant un nouveau dictionnaire de clés uniques peut être construit comme ceci :

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

En utilisant le protocole de descripteur hors-données, une version Python pure de `classmethod()` ressemblerait à ceci :

```
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.
```

(suite sur la page suivante)

(suite de la page précédente)

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

La portion de code pour `hasattr(type(self.f), '__get__')` a été ajoutée dans Python 3.9 et permet à `classmethod()` de prendre en charge les décorateurs chaînés. Par exemple, un décorateur « méthode de classe » peut être chaîné à un décorateur « propriété ». Dans Python 3.11, cette fonctionnalité est devenue obsolète.

```
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 Objets membres et *__slots__*

Lorsqu'une classe définit `__slots__`, Python remplace le dictionnaire d'instance par un tableau de longueur fixe de créneaux prédéfinis. D'un point de vue utilisateur, cela :

1/ permet une détection immédiate des bogues dus à des affectations d'attributs mal orthographiés. Seuls les noms d'attribut spécifiés dans `__slots__` sont autorisés :

```
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/ aide à créer des objets immuables où les descripteurs gèrent l'accès aux attributs privés stockés dans `__slots__` :

```
class Immutable:

    __slots__ = ('_dept', '_name')          # Replace the instance dictionary

    def __init__(self, dept, name):
        self._dept = dept                   # Store to private attribute
        self._name = name                   # Store to private attribute

    @property                               # Read-only descriptor
    def dept(self):
        return self._dept

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

```
>>> mark = Immutable('Botany', 'Mark Watney')
>>> mark.dept
```

(suite sur la page suivante)

(suite de la page précédente)

```
'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/ économise de la mémoire. Sur une version Linux 64 bits, une instance avec deux attributs prend 48 octets avec `__slots__` et 152 octets sans. Ce patron de conception poids mouche n'a probablement d'importance que si un grand nombre d'instances doivent être créées ;

4/ améliore la vitesse. La lecture des variables d'instance est 35 % plus rapide avec `__slots__` (mesure effectuée avec Python 3.10 sur un processeur Apple M1) ;

5/ bloque les outils comme `functools.cached_property()` qui nécessitent un dictionnaire d'instance pour fonctionner correctement :

```
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.
```

Il n'est pas possible de créer une version Python pure exacte de `__slots__` car il faut un accès direct aux structures C et un contrôle sur l'allocation de la mémoire des objets. Cependant, nous pouvons construire une simulation presque fidèle où la structure C réelle pour les *slots* est émulée par une liste privée `_slotvalues`. Les lectures et écritures dans cette structure privée sont gérées par des descripteurs de membres :

```
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]
```

(suite sur la page suivante)

(suite de la page précédente)

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

La méthode `object.__new__()` s'occupe de créer des instances qui ont des *slots* au lieu d'un dictionnaire d'instances. Voici une simulation approximative en Python pur :

```
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'
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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
```

À ce stade, la métaclasse a chargé des objets membres pour *x* et *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'>}
```

Lorsque les instances sont créées, elles ont une liste `slot_values` où les attributs sont stockés :

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

Les attributs mal orthographiés ou non attribués lèvent une exception :

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