Guide pour l'utilisation des descripteurs

Version 3.10.16

Guido van Rossum
and the Python development team

décembre 07, 2024

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

La classe `Ten` est un descripteur dont la méthode `__get__()` renvoie toujours la constante `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
3
>>> os.remove('games/chess')                # Delete a game
>>> g.size                                  # File count is automatically updated
2
```

En plus de montrer comment les descripteurs peuvent exécuter des calculs, cet exemple révèle également le but des paramètres de `__get__()`. Le paramètre *self* est *size*, une instance de *DirectorySize*. Le paramètre *obj* est soit *g* soit *s*, une instance de *Directory*. C'est le paramètre *obj* qui permet à la méthode `__get__()` de connaître le répertoire cible. Le paramètre *objtype* est la classe *Directory*.

1.3 Attributs gérés

Une utilisation courante des descripteurs est la gestion de l'accès aux données d'instances. Le descripteur est affecté à un attribut public dans le dictionnaire de classe tandis que les données réelles sont stockées en tant qu'attribut privé dans le dictionnaire d'instance. Les méthodes `__get__()` et `__set__()` du descripteur sont déclenchées lors de l'accès à l'attribut public.

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
{'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 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.

Dans cet exemple, la classe `Person` a deux instances de descripteurs, *name* et *age*. Lorsque la classe `Person` est définie, `__set_name__()` est appelée automatiquement dans *LoggedAccess* afin que les noms de champs puissent être enregistrés, en donnant à chaque descripteur ses propres *public_name* et *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
```

Une session interactive montre que la classe `Person` a appelé `__set_name__()` pour que les noms des champs soient enregistrés. Ici, nous appelons `vars()` pour rechercher le descripteur sans le déclencher :

```
>>> 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

Nous appelons descripteur tout objet qui définit `__get__()`, `__set__()` ou `__delete__()`.

Facultativement, les descripteurs peuvent avoir une méthode `__set_name__()`. Elle n'est utile que dans les cas où un descripteur doit connaître soit la classe dans laquelle il a été créé, soit le nom de la variable de classe à laquelle il a été affecté (cette méthode, si elle est présente, est appelée même si la classe n'est pas un descripteur).

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.

Cette classe `Validator` est à la fois une classe mère abstraite et un descripteur d'attributs gérés :

```
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
```

Les validateurs personnalisés doivent hériter de `Validator` et doivent fournir une méthode `validate()` pour tester diverses restrictions adaptées aux besoins.

2.2 Validateurs personnalisés

Voici trois utilitaires concrets de validation de données :

1) `OneOf` vérifie qu'une valeur fait partie d'un ensemble limité de valeurs ;
2) `Number` vérifie qu'une valeur est soit un `int` soit un `float`. Facultativement, il vérifie qu'une valeur se situe entre un minimum ou un maximum donnés ;
3) `String` vérifie qu'une valeur est une `chaîne de caractères`. Éventuellement, il valide les longueurs minimale ou maximale données. Il peut également valider un prédicat défini par l'utilisateur.

```
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}'
            )
```

(suite sur la page suivante)

(suite de la page précédente)

```
        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 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> 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 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 Definition and introduction

En général, un descripteur est la valeur d'un attribut qui possède une des méthodes définies dans le « protocole descripteur ». Ces méthodes sont : `__get__()`, `__set__()` et `__delete__()`. Si l'une de ces méthodes est définie pour un attribut, il s'agit d'un descripteur.

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.

Les descripteurs sont un protocole puissant et à usage général. Ils constituent le mécanisme qui met en œuvre les propriétés, les méthodes, les méthodes statiques, les méthodes de classes et `super()`. Ils sont utilisés dans tout Python lui-même. Les descripteurs simplifient le code C sous-jacent et offrent un ensemble flexible de nouveaux outils pour les programmes Python quotidiens.

3.3 Descriptor protocol

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

descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None
```

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 et peut remplacer le comportement par défaut lorsqu'il est recherché comme un attribut.

Si un objet définit `__set__()` ou `__delete__()`, il est considéré comme un descripteur de données. Les descripteurs qui ne définissent que `__get__()` sont appelés descripteurs hors-données (ils sont généralement utilisés pour des méthodes mais d'autres utilisations sont possibles).

Les descripteurs de données et les descripteurs *non-data* diffèrent dans la façon dont les dérogations sont calculées 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 *non-data*, l'entrée du dictionnaire a la priorité.

Pour faire un descripteur de données en lecture seule, définissez à la fois `__get__()` et `__set__()` avec `__set__()` levant une erreur `AttributeError` quand il est appelé. Définir la méthode `__set__set__()` avec une exception élevant le caractère générique est suffisant pour en faire un descripteur de données.

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.

L'expression `obj.x` recherche l'attribut `x` dans les espaces de noms pour `obj`. Si la recherche trouve un descripteur en dehors de l'instance `__dict__`, sa méthode `__get__()` est appelée selon les règles de priorité listées ci-dessous.

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

La recherche d'instance consiste à parcourir la liste d'espaces de noms en donnant aux descripteurs de données la priorité la plus élevée, suivis des variables d'instance, puis des descripteurs hors-données, puis des variables de classe, et enfin `__getattr__()` s'il est fourni.

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)
```

Notez qu'il n'y a pas d'appel vers `__getattr__()` dans le code de `__getattribute__()`. C'est pourquoi appeler `__getattribute__()` directement ou avec `super().__getattribute__` contourne entièrement `__getattr__()`.

Au lieu, c'est l'opérateur « point » et la fonction `getattr()` qui sont responsables de l'appel de `__getattr__()` chaque fois que `__getattribute__()` déclenche une `AttributeError`. Cette logique est présentée encapsulée dans une fonction utilitaire :

```
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

La logique pour une recherche « après un point » telle que `A.x` se trouve dans `type.__getattribute__()`. Les étapes sont similaires à celles de `object.__getattribute__()` mais la recherche dans le dictionnaire d'instance est remplacée par une recherche suivant l'ordre de résolution des méthodes de la classe.

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

L'implémentation C complète peut être trouvée dans `type_getattro()` et `_PyType_Lookup()` dans Objects/typeobject.c.

3.7 Appel depuis super

La logique de la recherche « après un point » de super se trouve dans la méthode `__getattribute__()` de l'objet renvoyé par `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.

L'implémentation C complète est dans `super_getattro()` dans Objects/typeobject.c. Un équivalent Python pur peut être trouvé dans Guido's Tutorial (page en anglais).

3.8 Résumé de la logique d'appel

Le fonctionnement des descripteurs se trouve dans les méthodes `__getattribute__()` de `object`, `type` et `super()`.

Les points importants à retenir sont :

- les descripteurs sont appelés par la méthode `__getattribute__()` ;
- les classes héritent ce mécanisme de `object`, `type` ou `super()` ;
- redéfinir `__getattribute__()` empêche les appels automatiques de descripteur car toute la logique des descripteurs est dans cette méthode ;
- `objet.__getattribute__()` et `type.__getattribute__()` font différents appels à `__get__()`. La première inclut l'instance et peut inclure la classe. La seconde met `None` pour l'instance et inclut toujours la classe ;
- 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

Il est parfois souhaitable qu'un descripteur sache à quel nom de variable de classe il a été affecté. Lorsqu'une nouvelle classe est créée, la métaclasse `type` parcourt le dictionnaire de la nouvelle classe. Si l'une des entrées est un descripteur et si elle définit `__set_name__()`, cette méthode est appelée avec deux arguments : *owner* (propriétaire) est la classe où le descripteur est utilisé, et *name* est la variable de classe à laquelle le descripteur a été assigné.

Les détails d'implémentation sont dans `type_new()` et `set_names()` dans Objects/typeobject.c.

Comme la logique de mise à jour est dans `type.__new__()`, les notifications n'ont lieu qu'au moment de la création de la classe. Si des descripteurs sont ajoutés à la classe par la suite, `__set_name__()` doit être appelée manuellement.

3.10 Exemple d'ORM

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

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}=?;'
        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()
```

Nous pouvons utiliser la classe `Field` pour définir des modèles qui décrivent le schéma de chaque table d'une base de données :

```
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if self.fget is None:
            raise AttributeError(f'unreadable attribute {self._name}')
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError(f"can't set attribute {self._name}")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError(f"can't delete attribute {self._name}")
        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 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
```

Soit la `property()` native, soit notre équivalent `Property()` fonctionnent dans cet exemple.

4.2 Functions and methods

Les fonctionnalités orientées objet de Python sont construites sur un environnement basé sur des fonctions. À l'aide de descripteurs *non-data*, 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)
```

Pour prendre en charge la création automatique des méthodes, les fonctions incluent la méthode `__get__()` pour lier les méthodes pendant l'accès aux attributs. Cela signifie que toutes les fonctions sont des descripteurs hors-données qui renvoient des méthodes liées au cours d'une recherche d'attribut d'une instance. Cela fonctionne ainsi :

```
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'
```

L'accès à la fonction *via* le dictionnaire de classe n'invoque pas `__get__()`. À la place, il renvoie simplement l'objet de fonction sous-jacent :

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

La recherche d'attribut depuis une classe appelle `__get__()`, qui renvoie simplement la fonction sous-jacente inchangée :

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

Le comportement intéressant se produit lors d'une recherche d'attribut à partir d'une instance. La recherche d'attribut appelle `__get__()` qui renvoie un objet « méthode liée » :

```
>>> 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>
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> d.f.__self__
<__main__.D object at 0x1012e1f98>
```

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 *non-data* fournissent un mécanisme simple pour les variations des patrons habituels des fonctions de liaison dans les méthodes.

Pour résumer, les fonctions ont une méthode `__get__()` pour qu'elles puissent être converties en méthodes lorsqu'on y accède comme attributs. Le descripteur hors-données transforme un appel `obj.f(*args)` en `f(obj, *args)`. L'appel `cls.f(*args)` devient `f(*args)`.

Ce tableau résume le lien (*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 modifications. 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, la fonction devient accessible de manière identique 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 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, `erf(x)` est une routine de conversion pratique qui apparaît dans le travail statistique mais qui 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 :

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

(suite sur la page suivante)

(suite de la page précédente)

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

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

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

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 :

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

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

    def __get__(self, obj, cls=None):
        if cls is None:
```

(suite sur la page suivante)

(suite de la page précédente)

```
            cls = type(obj)
        if hasattr(type(self.f), '__get__'):
            return self.f.__get__(cls, cls)
        return MethodType(self.f, cls)
```

The code path for `hasattr(type(self.f), '__get__')` was added in Python 3.9 and makes it possible for `classmethod()` to support chained decorators. For example, a classmethod and property could be chained together :

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

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

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
'Botany'
>>> mark.dept = 'Space Pirate'
Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    ...
AttributeError: can't set attribute
>>> 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]
        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
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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}>'
```

La méthode `type.__new__()` s'occupe d'ajouter des objets membres aux variables de classe :

```
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'
        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)
```

Pour utiliser la simulation dans une classe réelle, héritez simplement de `Object` et définissez la métaclasse à `Type` :

```
class H(Object, metaclass=Type):
    'Instance variables stored in slots'
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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'
```