Guide pour l'utilisation des descripteurs

Version 2.7.18

Guido van Rossum
and the Python development team

mai 20, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

Author Raymond Hettinger

Contact <python at rcn dot com>

Sommaire

1 Résumé

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

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 et d'apprécier l'élégance de sa conception.

2 Définition et introduction

En général, un descripteur est un attribut objet avec un « comportement contraignant », dont l'accès à l'attribut a été remplacé par des méthodes dans le protocole du descripteur. Ces méthodes sont : `__get__()`, `__set__()`, et `__delete__()`. Si l'une de ces méthodes est définie pour un objet, il s'agit d'un descripteur.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined. Note that descriptors are only invoked for new style objects or classes (a class is new style if it inherits from `object` or `type`).

Les descripteurs sont un protocole puissant et à usage général. Ils sont le mécanisme derrière les propriétés, les méthodes, les méthodes statiques, les méthodes de classes et `super()`. Ils sont utilisés dans tout Python lui-même pour implémenter les nouvelles classes de style introduites dans la version 2.2. Les descripteurs simplifient le code C sous-jacent et offrent un ensemble flexible de nouveaux outils pour les programmes Python quotidiens.

3 Protocole descripteur

```
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 à la fois `__get__()` et `__set__()`, il est considéré comme un descripteur de données. Les descripteurs qui ne définissent que `__get__()` sont appelés descripteurs *non-data* (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.

4 Invocation des descripteurs

Un descripteur peut être appelé directement par son nom de méthode. Par exemple, `d.__get__(obj)`.

Alternativement, il est plus courant qu'un descripteur soit invoqué automatiquement lors de l'accès aux attributs. Par exemple, `obj.d` recherche `d` dans le dictionnaire de `obj.d`. Si `d` définit la méthode `__get__()`, alors `d.__get__(obj)` est invoqué selon les règles de priorité énumérées ci-dessous.

The details of invocation depend on whether `obj` is an object or a class. Either way, descriptors only work for new style objects and classes. A class is new style if it is a subclass of `object`.

Pour les objets, la machinerie est en `object.__getattribute__()` qui transforme `b.x` en `type(b).__dict__['x'].__get__(b, type(b))`. L'implémentation fonctionne à travers une chaîne de priorité qui donne la priorité aux descripteurs de données sur les variables d'instance, la priorité aux variables d'instance sur les descripteurs *non-data*, et attribue la priorité la plus faible à `__getattr__()` si fourni. L'implémentation complète de C peut être trouvée dans `PyObject_GenericGetAttr()` dans Objects/object.c.

Pour les classes, la machinerie est dans `type.__getattribute__()` qui transforme `B.x` en `B.__dict__['x'].__get__(None, B)`. En Python pur, il ressemble à :

```
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

Les points importants à retenir sont :

- les descripteurs sont appelés par la méthode `__getattribute__()`
- redéfinition `__getattribute____()` empêche les appels automatiques de descripteurs
- `__getattribute__()` is only available with new style classes and objects
- `objet.__getattribute__()` et `type.__getattribute__()` font différents appels à `__get__()`.
- les descripteurs de données remplacent toujours les dictionnaires d'instances.
- les descripteurs *non-data* peuvent être remplacés par des dictionnaires d'instance.

L'objet retourné par `super()` a aussi une méthode personnalisée `__getattribute__()` pour appeler les descripteurs. L'appel `super(B, obj).m()` recherche `obj.__class__.__mro__` pour la classe de base `A` immédiatement après `B` et renvoie ensuite `A.__dict__['m'].__get__(obj, B)`. Si ce n'est pas un descripteur, `m` est retourné inchangé. Si ce n'est pas dans le dictionnaire, `m` renvoie à une recherche avec `object.__getattribute__()`.

Note, in Python 2.2, `super(B, obj).m()` would only invoke `__get__()` if `m` was a data descriptor. In Python 2.3, non-data descriptors also get invoked unless an old-style class is involved. The implementation details are in `super_getattro()` in Objects/typeobject.c.

Les détails ci-dessus montrent que le mécanisme des descripteurs est intégré dans les méthodes `__getattribute__()` pour `object`, `type` et `super()`. Les classes héritent de cette machinerie lorsqu'elles dérivent de `object` ou si elles ont une méta-classe fournissant des fonctionnalités similaires. De même, les classes peuvent désactiver l'appel de descripteurs en remplaçant `__getattribute__()`.

5 Exemple de descripteur

Le code suivant crée une classe dont les objets sont des descripteurs de données qui impriment un message pour chaque lecture ou écriture. Redefinir `__getattribute__()` est une approche alternative qui pourrait le faire pour chaque attribut. Cependant, ce descripteur n'est utile que pour le suivi de quelques attributs choisis :

```
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print 'Retrieving', self.name
        return self.val

    def __set__(self, obj, val):
        print 'Updating', self.name
        self.val = val

>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

The protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound and unbound methods, static methods, and class methods are all based on the descriptor protocol.

6 Propriétés

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

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

La documentation montre une utilisation typique pour définir un attribut géré `x` :

```
class C(object):
    def getx(self): return self.__x
```

(suite sur la page suivante)

(suite de la page précédente)

```
    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 implémenté dans le protocole du descripteur, voici un un équivalent Python pur :

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

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

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

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

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

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

La fonction `property()` intégrée 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 affecter le code client existant accédant directement à l'attribut. La solution consiste à envelopper l'accès à l'attribut de valeur dans un descripteur de données de propriété :

```
class Cell(object):
    . . .
    def getvalue(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
```

(suite sur la page suivante)

(suite de la page précédente)

```
    value = property(getvalue)
```

7 Fonctions et méthodes

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.

Class dictionaries store methods as functions. In a class definition, methods are written using `def` and `lambda`, the usual tools for creating functions. The only difference from regular functions is that the first argument is reserved for the object instance. By Python convention, the instance reference is called *self* but may be called *this* or any other variable name.

To support method calls, functions include the `__get__()` method for binding methods during attribute access. This means that all functions are non-data descriptors which return bound or unbound methods depending whether they are invoked from an object or a class. In pure python, it works like this :

```
class Function(object):
    . . .
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj, objtype)
```

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

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']  # Stored internally as a function
<function f at 0x00C45070>
>>> D.f               # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f               # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

The output suggests that bound and unbound methods are two different types. While they could have been implemented that way, the actual C implementation of `PyMethod_Type` in Objects/classobject.c is a single object with two different representations depending on whether the `im_self` field is set or is *NULL* (the C equivalent of `None`).

Likewise, the effects of calling a method object depend on the `im_self` field. If set (meaning bound), the original function (stored in the `im_func` field) is called as expected with the first argument set to the instance. If unbound, all of the arguments are passed unchanged to the original function. The actual C implementation of `instancemethod_call()` is only slightly more complex in that it includes some type checking.

8 Méthodes statiques et méthodes de classe

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 méthode lorsqu'on y accède comme attributs. Le descripteur *non-data* transforme un appel `obj.f(*args)``en ``f(obj, *args)`. Appeler `klass.f(*args)` devient `f(*args)`.

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

Transformation	Appelé depuis un Objet	Appelé depuis un Classe
fonction	f(obj, *args)	f(*args)
méthode statique	f(*args)	f(*args)
méthode de classe	f(type(obj), *args)	f(klass, *args)

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`.

Depuis que les méthodes statiques renvoient la fonction sous-jacente sans changement, les exemples d'appels ne sont pas excitants :

```
>>> class E(object):
...     def f(x):
...         print x
...     f = staticmethod(f)
...
>>> print E.f(3)
3
>>> print E().f(3)
3
```

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

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

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

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

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

```
>>> class E(object):
...     def f(klass, x):
...          return klass.__name__, x
...     f = classmethod(f)
...
>>> print E.f(3)
('E', 3)
>>> print E().f(3)
('E', 3)
```

Ce comportement est utile lorsque la fonction n'a besoin que d'une référence de classe et ne se soucie pas des données sous-jacentes. Une des utilisations des méthodes de classe est de créer d'autres constructeurs de classe. En Python 2.3, la méthode de classe `dict.fromkeys()` crée un nouveau dictionnaire à partir d'une liste de clés. L'équivalent Python pur est :

```
class Dict(object):
    . . .
    def fromkeys(klass, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys = classmethod(fromkeys)
```

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

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

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

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

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

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```