“Les descriptors dans Python.”*

MOUSSA DIALLO DATA ANALYST, MASTER I-AI

Juin, 2022

Table des matières

*THANKS TO all DIT STAFF AND PARTICULARY TO PATRICK NSUKAMI

Introduction

Python est un logiciel de programmation très avancée. Il offre de nombreux outils pratiques et qui nous permettent de programmer intuitivement. Parmi les multiples facilités il y'a les méthodes magiques telque `__get__`, `__add__`, les décorateurs `@staticmethod`, `@property` etc. La maitrise de ces outils rend la programmation orientée objet dans python beaucoup plus fun. Cependant, un developeur python qui utilise ses outils se rendra un jour compte qu'il lui faut plus. Autrement dit, il va falloir remuer ces outils pour voir ce qui se cache réellement derrière. C'est à ce moment seulement qu'il pourra les adapter pour une utilisation plus approfondi et plus personnelle. Par exemple la manipulation des attributs d'un objet d'une classe à travers la méthode @property se trouve dans certains cas inadaptée. Il faudra faire appelle à une méthode appelé "Descriptor" pour plus de souplesse et de réutilisabilité. Dans ce présent article, il sera question d'explorer les quelques spécificité de cette méthode. Il s'agira entre autre : - de présenter la relation entre classe, objet et attribut - présenter la méthode @property - de definir les descriptors - de présenter ses champs d'applicabilté - de voir dans quel cas l'utilisation des « descriptors » est recommandée

I- Classe, objet et attribut

Nous savons tous la liaison entre une classe et un objet

```
class Cercle():  # definition de ma classe Cercle

    PI = 3.14
    def __init__(self,rayon) :
        self.rayon = rayon

mon_cercle=Cercle(2) # j'instancie un objet de la classe Cercle
print(mon_cercle.rayon) # Affiche le rayon du cercle
print(mon_cercle.PI) # Affiche la constante PI=3,14
```

```
2
3.14
```

Quand est-il de la liasion entre l'attribues d'un objet et celui d'une classe

Comment acceder à l'attribut d'un OBJET

Quand on voulait afficher le rayon de l'objet `mon_cercle` on a juste écrit `mon_cercle.rayon`. ceci ne fait que renvoyer une valeur stocké dans un dictionnaire de l'objet.

Comme on peut le vérifier à traver le code suivant

```
print(mon_cercle.__dict__)
```

```
{'rayon': 2}
```

Accéder aux attributs d'une classe

Cependant, de la même manière on pouvait accéder à l'attribut directement au niveau de la class. Cet attribut est enregistré dans un dictionnaire cet fois ci de la classe.

```
print(Cercle.PI)
print(Cercle.__dict__)
print(mon_cercle.PI)
```

```
3.14
```

```
{'__module__': '__main__', 'PI': 3.14, '__init__': <function Cercle.__init__ at 0x0000027A5931FBE0>, '__dict
3.14
```

Parfois les attributs ne sont pas suffisants. Nous avons besoin de procédés plus puissant. Regardons ensemble une des limites des attributs.

```
class Circle():
    PI = 3.14
    def __init__(self,radius):
        self.radius = radius
        self.circumference = 2*radius*self.PI

mycircle = Circle(2)
# Affichons la circonférence du cercle
print(mycircle.circumference)
# Changeons le rayon à 3 au lieu de 2

mycircle.radius = 3
# Affichons encore la circonférence du cercle
print(mycircle.circumference) # Oops la circonférence ne change pas
```

```
12.56
12.56
```

Heureusement la magie des @property peut nous sauver.

II- La méthode @property

A l'aide de la magie des décorateurs on sait comment contourner le problème,n'est ce pas !.

```
class Circle():
    PI = 3.14
    def __init__(self,radius):
        self.radius = radius
 #Super @property nous sauve la vie
    @property
    def circumference(self):
        return 2*self.radius*self.PI

mycircle = Circle(2)
print(mycircle.circumference)
mycircle.radius = 3
print(mycircle.circumference) # Fixed!
```

```
12.56
18.84
```

Ainsi, On peut ajouter des getters et des setters dans notre classe pour garder nos attributs aussi simple que possibles tout en intégrant des propriétés super puissantes. Cependant, savez vous comment ça fonctionne réellement? Aussi, est-ce toujours suffisant pour faire le travail proprement ?

III- C'est quoi un "descriptor"

Les descripteurs sont des objets Python qui implémentent au moins une méthode du `descriptor protocol` (`__get__`, `__set__` ou `__delete__`), ce qui vous donne la possibilité de créer des objets qui ont un comportement spécial lorsqu'ils sont accédés en tant qu'attributs d'autres objets.

On appelle `__data-descriptor__` un descriptor qui implémente à la fois la méthode `__get__` et `__set__`. Un descriptor qui implémentes seulement la méthode `__get__` est un `non-data-descriptor`. Pour créer un descripteur de données en lecture seule, définissez à la fois `__get__()` et `__set__()` avec le `__set__()` générant une AttributeError lorsqu'il est appelé. Définir la méthode `__set__()` avec une exception suffit à en faire un descripteur de données.

Comment ça fonctionne ?

Ce qu'il faut retenir est quand vous appelez un attribut `foo` de votre objet `obj` à travers la méthode `obj.foo`, python suit un protocole bien defini et bien hierarchisé pour retrouvé l'attribut en question. En effet, il commence par :

```
1. chercher Le résultat de la propriété du même nom si elle est définie
2. ou voir si la valeur correspondante existe dans obj.__dict__
3. ou bien il va remonter dans la hierarchie pour chercher dans le type(obj).__dict__
4. répéter ces étapes pour chaque type dans le mro (methode resolution order : montre la chaine d'héritage )
si votre class a héritée d'autre classes jusqu'à ce qu'il trouve une correspondance
5. Si c'est une affectation, ça crée toujours une entrée dans obj.__dict__
6. Sauf s'il y avait une propriété setter auquel cas vous appelez une fonction.
```

Pour résumé ici lorsque nous accédons aux attributs dans de cette façon, ce qui se passe, c'est que python recherche les valeurs de ceux-ci dans le dictionnaire d'instances, en fait, nous pouvons jeter un œil au dictionnaire d'instanciation en tapant `obj.__dict__`".

Comment écrire un descriptors

De manière simple le "descriptor protocol" s'écrit comme suit :

```
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 l'une de ces méthodes et un objet est considéré comme un descripteur et peut remplacer le comportement par défaut lorsqu'il est recherché en tant qu'attribut.

Ecrivons notre premier descriptor :

```
class MyDescriptor():
    def __get__(self,obj,type):
        print(self, obj, type)
    def __set__(self,obj,val):
        print("Got %s" %val)

class Myclass():
    x = MyDescriptor() # On vient d'instancier notre premier descriptor ): ,
```

Le desriptor est très pratique ça nous permet d'interagir avec notre attribut à l'aide de fonctions pratiques.

```
obj= Myclass()
```

```
print(obj.x) # un appel de fonction se cache ici

print(Myclass.x) # et ici!

obj.x=4  # ici aussi
```

```
<__main__.MyDescriptor object at 0x0000027A48536DD0> <__main__.Myclass object at 0x0000027A4857CF10> <class '
None
<__main__.MyDescriptor object at 0x0000027A48536DD0> None <class '__main__.Myclass'>
None
Got 4
```

Plus de détails sur la signatures des descriptors :

- self est l'instance du descriptor
- obj est l'instance de l'objet pour qui le descripteur est attaché
- type est le class avec qui le descriptor est attaché
- **get** peut être appelé à travers la class ou l'objet, **set** peut être appelé seulement à travers l'objet

IV- Utilisation des descriptors

Pourquoi l'utilise t-on ?

> Essayons de rendre notre descriptor plus utile.

Avec, les descripteur on peut stocker la valeur des attributs à l'intérieur. Cependant, regardons ensemble ce code et essayons de trouver ce qui ne va pas!

```
class MyDescriptor(object):
    def __get__(self,obj,type):
        return self.data
    def __set__(self,obj,val):
        self.data=val

class Myclass(object):
    val=MyDescriptor()

obj1=Myclass()
# Ici on  definit une valeur sur obj1 (derrière c'est la méthode set qui s'en charge)
obj1.val=10
# Essayons d'instancier un nouveau attribut obj2
obj2=Myclass()
# Voyons ce qui est dans obj2
# Oops c'est la même val que obj 1 , il a just re-implémenté
print(obj2.val)
```

```
10
```

En procédant ainsi, on ré-implémente la même valeur encore et encore sans l'adapter à l'attribut qui l'utilise Le problème c'est que telque définit le descriptor ne connait pas le nom de l'attribut dans lequel le descriptor est instancié.

En faisant

```
val = Mydescriptor()
```

le descripteur ne connait pas à priori que c'est `val` qui l'appelle. On peut contourner le problème en se répétant un peu : on donne à chaque appelle de notre descriptor le nom de l'attribut qui l'appelle.

Comme suit :

```
class Myclass():
val = MyDescriptor("val")
```

C'est pas pratique de taper le nom manuellement à chaque instanciation, n'est ce pas !

Ainsi il faudra adapter notre descriptor

Avec cette méthode on exige le nom de l'attribut à chaque appelle de la classe.

```
class MyDescriptor():
    def __init__(self,field=""):
        self.field = field
    def __get__(self,obj,type):
        print("Called __get__")
        return obj.__dict__.get(self.field)
    def __set__(self,obj,val):
        print("Called __set__")
        obj.__dict__[self.field] = val
```

Pour résumé, a chaque fois que `obj.x` est exécuté il va interpeler le descriptor qui à son tour lui renvoi sa valeur. Cette valeur est en fait cacher dans un dictionnaire accessible à l'aide de clé x : `obj.__dict__['x']`.

Utiliser cette méthode nous conduit à se répéter un peu. Si seulement on maitriser les métaclass

Si seulement on maitrisait les métaclasse !

```
def named_descriptors(kclass):
    for name, attr in kclass.__dict__.items():
        if isinstance(attr,MyDescriptor):
            attr.field = name
        return kclass

@named_descriptors
class Myclass(object):
    x = MyDescriptor()

#Which works


obj = Myclass()
obj.x=10

print(obj.x)
```

```
Called __set__
Called __get__
10
```

> Tous ça c'est compliqué, et si on continuait à utiliser les @property

La magie `@property` est génial et facile à implémenter de même que `@staticmethod` et de `@classmethod`, pourquoi se casser la tête avec les decriptor. Les `@property` font l'essentiel avec un simple interface pour API complexe. Serais-je dans une situation qui m'obligerait à utiliser un descriptor ?

> La response est oui !

V- Descriptors vs @property

Malheurement, l'utilisation de la méthode `@property` n'est pas recommandé dans tous les cas où vous devez intercepter l'accès aux attributs. Imaginons une classe qui doit stocker divers montants en dollars dans des attributs. Puisque les montants sont en décimal, on nous demande de les stocker avec seulement un ou deux chiffres après la virgule.

Essayons de ressoudre le problème avec la méthode `@property`

Avec un peu de copier coller !

```
from decimal import Decimal, ROUND_UP
from locale import currency
class BankTransaction(object):
    _cent = Decimal('0.01')
    def __init__(self,account,before,after,min,max):
        self.account = account
        self._before = before
        self._after = after
        self._min = min
        self.max= max
    @property
    def before(self):
        return Decimal(self._before).quantize(self._cent,ROUND_UP)
    @before.setter
    def before(self,val):
        self.before = str(val)
    # Copier coller !
    @property
    def after(self):
        return Decimal(self._after).quantize(self._cent,ROUND_UP)
    @after.setter
    def after(self,val):
        self.after = str(val)
```

Ainsi, on fera du copier coller de getters et de setter encore et encore pour chaque variable. Ce qui en programmation est très déconseillé puisque ça allourdi les codes pour rien. A

> répétez les getters et les setters passe-partout encore et encore

NON, NON... Je pensais que @property était censé me sauver du code passe-partout !

Avec le descriptor c'est beaucoup plus simple, avec un code réutilisable pour chaque variable.

```
class CurrencyField():
     #_cent = Decimal(self.pos)
    def __init__(self,pos):
```

```
        self.pos=pos

    def __get__(self,obj,type):
            return Decimal(self.data).quantize(Decimal(self.pos),ROUND_UP)
    def __set__(self,obj,val):
        self.data=str(val)


def named_descriptors(kclass):
    for name, attr in kclass.__dict__.items():
        if isinstance(attr,CurrencyField):
            attr.field = name
        return kclass

@named_descriptors
class BankTransaction(object):
    before= CurrencyField('0.01')
    after = CurrencyField('0.1')
    def __init__(self,account,before,after):
        self.account = account
        self.before = before
        self.after = after

dev_cfa=BankTransaction("dollar", 123.456, 789.123)
print("APPELLE DE LA VALEUR AVANT")
print(dev_cfa.before)
print("APPELLE DE LA VALEUR APRES")
print(dev_cfa.after)
print("CHANGEONS LA VALEUR DE APRES")
dev_cfa.after=90.12354
print("REGARDONS LA NOUVELLE VALEUR DE APRES")
print(dev_cfa.after)
```

```
APPELLE DE LA VALEUR AVANT
123.46
APPELLE DE LA VALEUR APRES
789.2
CHANGEONS LA VALEUR DE APRES
REGARDONS LA NOUVELLE VALEUR DE APRES
90.2
```

CAS PRATIQUES

définition des variables

Regardons un autre cas pratique où l'utilisation d'un descriptor est très recommandé : la définition des variables d'une base de donnée.

Chaque variable est bien définit avec des règle derrières. Ces règles sont ré-utilisables dans d'autres class de noms différents.

```
class Person(object):
    id = PrimaryKeyField()
```

```
    name = VarCharField(max_lenght=255)


class NickName(objectj):
    id = PrimaryKeyField()
    person_id = ForeignKey(Person)
    name = VarCharField(max_lenght=255)
```

Ceci nous est vaquement familier n'est ce pas !

Lazy Properties

Regardons un autre exemple où l'utilisation des "descriptors" est très recommandé. Il s'agit des propriétés paresseuses. Ce sont des propriétés dont les valeurs initiales ne sont pas chargées jusqu'à ce qu'elles soient accédées pour la première fois. Ensuite, ils chargent leur valeur initiale et conservent cette valeur en cache pour une réutilisation ultérieure.

Prenons l'exemple suivant. Vous avez une classe PenseeProfond qui contient une méthode essence_de_la_vie() qui renvoie une valeur après beaucoup de temps passé en forte concentration :

```
# slow_properties.py
import time

class PenseeProfond:
    def essence_de_la_vie(self):
        time.sleep(3)
        return 42

ma_PenseeProfond_instance = PenseeProfond()
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie())
end=time.time()
print("durée de la concentration",end-start)
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie())
end=time.time()
print("durée de la concentration",end-start)
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie())
end=time.time()
print("durée de la concentration",end-start)
```

```
42
durée de la concentration 3.011887550354004

42
durée de la concentration 3.0033371448516846

42
durée de la concentration 3.004000425338745
```

Si vous exécutez ce code et essayez d'accéder à la méthode trois fois, vous obtenez une réponse toutes les trois secondes, ce qui correspond à la durée du temps de veille à l'intérieur de la méthode.

Désormais, une propriété paresseuse peut à la place évaluer cette méthode une seule fois lors de sa première exécution. Ensuite, il mettra en cache la valeur résultante afin que, si vous en avez à nouveau besoin, vous puissiez l'obtenir en un rien de temps. Vous pouvez y parvenir en utilisant des descripteurs Python :

```
# lazy_properties.py
import time

class LazyProperty:
    def __init__(self, function):
        self.function = function
        self.name = function.__name__

    def __get__(self, obj, type=None) -> object:
        obj.__dict__[self.name] = self.function(obj)
        return obj.__dict__[self.name]

class PenseeProfond:
    @LazyProperty
    def essence_de_la_vie(self):
        time.sleep(3)
        return 42

ma_PenseeProfond_instance = PenseeProfond()
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie)
end=time.time()
print("durée de la concentration",end-start)
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie)
end=time.time()
print("durée de la concentration",end-start)
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie)
end=time.time()
print("durée de la concentration",end-start)
```

```
42
durée de la concentration 3.0082767009735107
42
durée de la concentration 0.0010025501251220703
42
durée de la concentration 0.0009963512420654297
```

Regardons ensemble la puissance des descripteurs. Dans cet exemple, lorsque le descripteur @LazyProperty est utilisé, on instancie un descripteur en lui transmettant .essence_de_la_vie(). Ce descripteur stocke à la fois la méthode et son nom en tant que variables d'instance.

Puisqu'il s'agit d'un `non-data descriptor`, lorsque vous accédez pour la première fois à la valeur de l'attribut essence_de_la_vie, .__get__() est automatiquement appelé et exécute .essence_de_la_vie() sur l'objet ma_PenseeProfond_instance. La valeur résultante est stockée dans l'attribut **dict** de l'objet lui-même. Lorsque vous accédez à nouveau à l'attribut essence_de_la_vie, Python utilisera la chaîne de recherche pour trouver une valeur pour cet attribut dans l'attribut **dict**, et cette valeur sera renvoyée immédiatement.

Il faudra noter que si l'astuce a marché c'est parce que dans cet exemple nous avons implémenté que la mé-

thode .__get__() du protocole de descripteur et donc c'est un `non-data descripteur`. Si à la place on avait implémenté `data descripteur`, l'astuce n'aurait pas fonctionné. Après la chaîne de recherche, elle aurait eu priorité sur la valeur stockée dans **dict**. Pour tester cela, exécutez le code suivant :

```
# wrong_lazy_properties.py
import time

class LazyProperty:
    def __init__(self, function):
        self.function = function
        self.name = function.__name__

    def __get__(self, obj, type=None) -> object:
        obj.__dict__[self.name] = self.function(obj)
        return obj.__dict__[self.name]

    def __set__(self, obj, value):
        pass

class PenseeProfond:
    @LazyProperty
    def essence_de_la_vie(self):
        time.sleep(3)
        return 42

ma_PenseeProfond_instance = PenseeProfond()
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie)
end=time.time()
print("durée de la concentration",end-start)
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie)
end=time.time()
print("durée de la concentration",end-start)
start=time.time()
print(ma_PenseeProfond_instance.essence_de_la_vie)
end=time.time()
print("durée de la concentration",end-start)
```

```
42
durée de la concentration 3.0016744136810303

42
durée de la concentration 3.002692461013794

42
durée de la concentration 3.0031869411468506
```

Dans cet exemple, vous pouvez voir que le simple fait d'implémenter .__set__(), même s'il ne fait rien du tout, crée un descripteur de données. Maintenant, l'astuce de la propriété paresseuse cesse de fonctionner.

En résumé

Dans cet articel, il a été question de faire une briève présentation des descriptors qui semble complexe vu de loin mais ô combien puissant pour rendre notre code succinct et réutilisable. Cependant, nous n'avons pas pu couvrit tous les champs d'applicabilité des descriptors. c'est pourquoi je vous invite à creuser davantage sur la documentation officielle que je mettrais sur la bibliographie.

BIBLIOGRAPHY

- official documentation on descritpors
- Real PYTHON_descriptors.
- simeonfranklin talk about descriptors