Python 3 : la notion de descripteur 2

Python 3 : la notion de descripteur

La notion de descripteur est abordée dans la documentation officielle de **Python 3** (**Descriptor HowTo Guide**). La présente note s'inspire largement de ce *HowTo ;* on a abandonné quelques considérations sur l'implémentation de C-Python et adopté un point de vue plus proche du programmeur.

1. l'accès à un attribut

1.1. rappels sur la notion d'attribut

Considérons l'exemple suivant :

```
class Test_Classe (object) :
    """ la classe qui va être testée    """
    leTitre = "******** classe de test *************"
    def __init__ (self, parNom, parCode) :
        """ initialisation de la classe Test_Classe """
        self.nom = parNom
        self.code = parCode
        print ("création d'une instance")

    def display (self) :
        """ cette méthode display est documentée """
        print ("@@@ nom : "+self.nom, " @@@  code : ", self.code)
```

*** `Test_Classe`** est un objet classe : ses attributs déclarés dans le script sont **`leTitre`**, **`__init__`**, **`display`**

*** `self`** représente une instance de la classe ; le mot n'a pas d'importance. Chaque instance possède les attributs définis pour **`self`**, soit ici **`nom`** et **`code`** ; la valeur de chaque attribut est spécifique de l'instance.

* Un attribut est un **objet** quelconque, fonction, entier, chaîne, objet utilisateur etc. Un attribut a trois caractéristiques : son identificateur, sa chaîne de recherche, sa valeur : **`display`**, **`"display"`**, ***`la fonction`***. En général, quand il y a identificateur et chaîne de recherche, l'identificateur est calqué sur la chaîne de recherche.

* l'attribut relève de trois utilisations : sa valeur peut intervenir dans une expression, sa valeur peut changer par une affectation (instruction d'affectation) ou il peut être supprimé (**`delete`**). Dans chaque cas, **il est nécessaire d'accéder à l'attribut** ; on le recherche à partir de sa chaîne de recherche, ou **clef**.

* L'affectation d'un attribut non existant entraîne la création de cet attribut.

* Tout objet possède des attributs qui lui sont concédés (de façon transparente) lors de sa création. En particulier les attributs de **`object`**, la classe au sommet de l'arbre hiérarchique des objets. Mais un objet a **toujours** en propre un **dictionnaire** qui répertorie **les attributs qui sont définis sur l'objet**. Le dictionnaire est lui-même un attribut qui s'identifie par **`__dict__`** sur son propriétaire.

* Il y a deux façons de manipuler un attribut : par qualification ou par dictionnaire :

`Test_Classe.display` est équivalent à **`Test_Classe.__dict__["display"]`**. Pour les objets qui en disposent, l'attribut **`__name__`** restitue la clef de l'attribut :

`Test_Classe.display.__name__ == "display"` et

`eval("Test_Classe.display") == Test_Classe.display`

```
for clef, valeur in Test_Classe.__dict__.items() :
    print (">>>",clef, "===>>>", valeur)

print ()
instance = Test_Classe ("Caroline", 1973) # création d'instance
for clef in instance.__dict__.keys() : # autre affichage possible :
    print (">>>",clef, "===>>>", instance.__dict__[clef])

print()
Test_Classe.display(instance)
instance.display()
```

```
>>> __module__ ===>>> __main__
>>> __init__ ===>>> <function Test_Classe.__init__ at 0x7f22c31bcb90>
```

```
>>> leTitre ===>>> ******** classe de test *************
>>> display ===>>> <function Test_Classe.display at 0x7f22c31bce60>
>>> __weakref__ ===>>> <attribute '__weakref__' of 'Test_Classe' objects>
>>> __doc__ ===>>>  la classe qui va être testée
>>> __dict__ ===>>> <attribute '__dict__' of 'Test_Classe' objects>

création d'une instance
>>> nom ===>>> Caroline
>>> code ===>>> 1973

@@@ nom : Caroline  @@@  code :  1973
@@@ nom : Caroline  @@@  code :  1973
```

* remarques : si pour l'instance il n'y a que les attributs effectivement créés dans le dictionnaire, la classe est un peu plus généreuse, avec des attributs obligatoires : **`module`**, **`doc`**, **`weakref`** et ... **`__dict__`**.

1.2. accès à un attribut

Accéder à un attribut, c'est retrouver son dictionnaire d'implémentation, où sa clef apparaît comme une entrée. On peut alors accéder à sa valeur, la modifier ou supprimer l'attribut. En effet, il faut se rappeler que l'ensemble des attributs disponibles pour un objet (utiliser la fonction prédéfinie **`dir()`** pour obtenir la liste des chaînes de recherches valides) comporte deux familles :

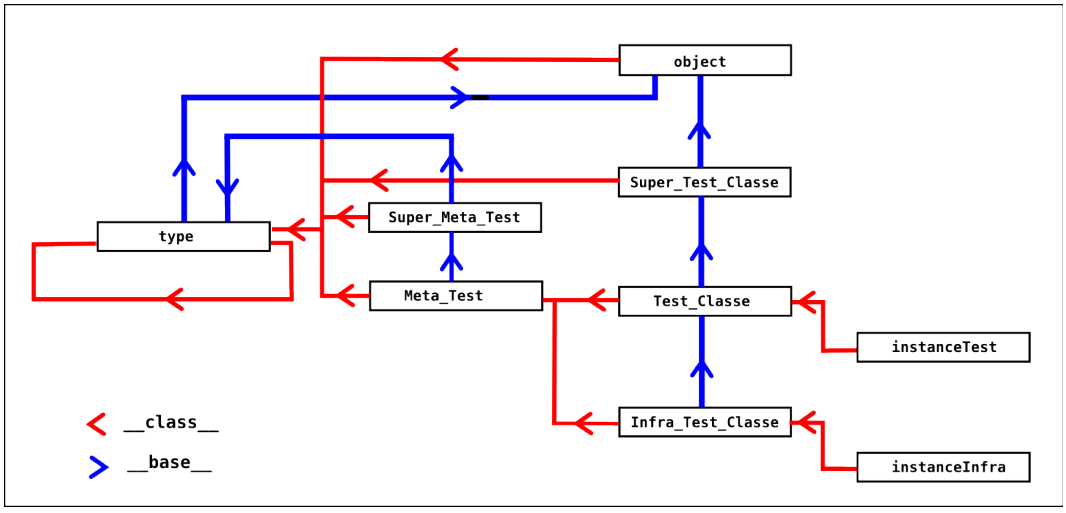
- les attributs disponibles dans le dictionnaire de l'objet en tant qu'instance ;
- les attributs disponibles dans le dictionnaire de la classe de l'objet et dans les dictionnaires de toutes les classes de l'ascendance.

La recherche d'un attribut se fait dans l'ordre suivants :

- le dictionnaire de l'objet en tant qu'instance. On rappelle qu'en Python, tout objet est une instance !
- le dictionnaire de la classe de l'objet ;
- le dictionnaire de la classe située hiérarchiquement avant le classe de l'objet et ainsi de suite jusque la rencontre avec le dictionnaire de **`object`** ; on n'abordera pas ici la question de l'héritage multiple.

La recherche s'arrête sur la première occurrence trouvée, occultant les occurrences éventuelles qui son hiérarchiquement antérieures.

Si l'instance est une classe, il faut bien se rappeler que la classe de la classe, ou métaclasse, est donc la seconde partie de la recherche ; comme toutes les métaclasses ont une ascendance qui comporte «`type`», cette classe «type» avec son dictionnaire fait nécessairement partie de la hiérarchie.



1.3. la machinerie

le mécanisme

La machinerie de recherche en vue de **récupérer la valeur d'un attribut** est une méthode **`__getattribut__()`** qui, par défaut, retourne **la valeur de l'attribut !** Mais si l'attribut possède une méthode

__get__(), alors la méthode **__getattribut__()** retourne l'appel de cette méthode sur l'attribut reconnu. Comme on peut définir la méthode **__get__()** sur un objet quelconque, on peut manipuler à loisir la valeur retournée lorsque l'attribut est évoqué en vue de récupérer sa valeur . C'est ce type de manipulation qui est l'objet de la présente note. Pour l'instant, il suffit de savoir que les fonctions (et méthodes) ont par défaut cette méthode **__get__()**.

Dans l'exemple donné dans le script :

```
Test_Classe.display(instance)
instance.display()
```

on a les invocation suivantes :

```
Test_Classe.__getattribute__("display")  et :
instance.__getattribute__("display")
```

Pour l'utilisation en affectation de valeur, on dispose de même de la méthode **__setattribute__()** et pour la suppressions **__delattribute__().** Pour l'instant on va se contenter de voir le mécanisme mis en œuvre par **__getattribute__()**

implémentation de __getattribute__()

L'important est de savoir où se trouve cette méthode **__getattribut__()**. Elle est définie pour **object()** et est donc accessible par la hiérarchie des classes pour tous les objets ; seulement elle est **wrappée** dans son passage par **type()**. Reprenons les deux cas d'accès évoqués ci-dessus :

Test_Classe.__getattribute__("display") : **__getattribute__** est recherché dans **Test_Classe** ; il n'y est pas !. On passe donc à la classe de **Test_Classe**, qui est une métaclasse, et en remontant on passe par **type** : l'appel réel est l'appel du **__getattribute__** de **type**.

instance.__getattribute__("display") : **__getattribute__** est recherché dans **instance**. Il n'y est évidemment pas ! On passe donc à la classe **Test_Classe** et on remonte la hiérarchie des **bases** (ascendants). On remonte ainsi à **object**.

Pour suivre sur le schéma du paragraphe 1.2., on rappelle que la classe d'un objet est représenté par **une flèche rouge**, et que les ascendants d'une classe se retrouvent **en suivant les flèches bleues**.

Si l'attribut invoqué possède la méthode **__get__()**, alors selon la méthode **__getattribute__()** utilisée, celle de **type** ou celle de **object**, les arguments passés à **__get__()** peuvent être spécifiques, suivant que le propriétaire est une classe ou une instance. C'est là le principe qui fonde l'utilisation des attributs (méthodes incluses), lorsque qu'on les préfixe par une classe ou une instance.

arguments de __get__()

En résumé, si on veut la valeur de l'attribut **x** de l'objet **objapp**, c'est-à-dire ce que le programme a noté **objapp.x**, il y a trois cas possibles :

- l'attribut **x** ne possède pas l'attribut **__get__**: alors la valeur retournée est la valeur brute à laquelle **__getattr__** a accédé dans le dictionnaire ad hoc par le procédé usuel d'identification.

- l'attribut **x** possède l'attribut **__get__** et **objapp** a été identifié comme une classe : alors la valeur retournée par **__getattr__** est celle que retournera la méthode **__get__** dans l'exécution de **objapp.x.__get__(arguments pour la classe app)**.

- l'attribut **x** possède l'attribut **__get__** et **objapp** a été identifié comme une instance : alors la valeur retournée par **__getattr__** est celle que retournera la méthode **__get__** dans l'exécution de **objapp.x.__get__(arguments pour l'instance app)**.

Il y a deux arguments passés à la méthode **__get__** lorsqu'elle est appelée et qui sont donc **discriminants** pour l'établissement du résultat :

- **objapp** est une classe : le premier argument est **None** et le second est la classe **objapp ;**

- **objapp** est une instance, le premier argument est **app** et le second est la classe **objapp.__class__** qui est identique à **type(app).**

1.4. exemple

Examinons quatre cas d'appel valides pour l'exemple du début :

* **Test_Classe.leTitre** : la méthode appelée est le .**__getattribute__** de **type**. Il n'y a pas de méthode **__get__** pour **leTitre** ; la valeur retournée est celle du dictionnaire de **Test_Classe**.

* **instance.leTitre** : la méthode appelée est le .**__getattribute__** de **object**. Il n'y a pas de

méthode **__get__** pour **leTitre** ; la valeur est recherchée dans instance, et comme elle n'y est pas, c'est celle du dictionnaire de **Test_Classe** qui est trouvée et retournée.

* **Test_Classe.display (instance)** : cette fois, il y a une méthode **__get__** pour **display**, comme pour toutes les fonctions ; la valeur retournée pour **Test_Classe.display** résulte de l'appel de **display.__get__(None, Test_Classe)**, c'est-à-dire la **fonction display**, qui est appelée dans un second temps avec l'argument **instance**.

* **instance.display()** : il y a une méthode **__get__** pour **display**, mais cette fois la valeur retournée est trouvée non dans le dictionnaire de instance, mais dans celle de sa classe : Il a donc appel de **display.__get__(instance, Test_Classe)**. La programmation par défaut du **__get__** est adaptée aux arguments et la fonction **display**, est appelée avec l'argument **instance**.

C'est le code de la méthode **__get__** qui permet, selon les argument qu'elle reçoit, de s'adapter à l'objet d'appel.

2. Le cas de l'affectation

2.1. l'instruction d'affectation

Le schéma grammatical de l'instruction d'affectation d'un attribut est :

objapp.x = *expression valeur*

objapp est l'objet d'appel, **x** est l'attribut, l'expression valeur est supposée évaluée ; on posera donc :

objapp.x = valeur

C'est cette expression qui est traitée lors de l'interprétation. Elle comporte deux étapes :

- accéder à l'attribut ;

- changer, dans le dictionnaire ad hoc la valeur associée à l'attribut en fonction de la valeur calculée dans l'évaluation de l'expression valeur.

Le traitement de l'expression relève de la méthode **__setattribute__** qui est parallèle à **__getattribute__**. La valeur associée à l'identificateur **x** doit changer. **__setattribute__** a donc deux arguments, la clef et la valeur.

Mais attention car le comportement peut être déroutant et cela est lié comportement de l'affectation en Python. En particulier, au fait dans le cas où l'attribut n'est pas trouvé, il est créé, en prenant la valeur d'affectation. On a donc à examiner les cas suivants :

* **une_instance.un_attribut = valeur** : la méthode **__setattribute__** recherche l'attribut selon la méthode vue avec **__getattribute__** mais accorde une précédence à l'attribut doté de la méthode **__set__**. Si l'attribut est trouvé avec la méthode **__set__** cette méthode est appelée et fait le travail défini dans la méthode.

* Si l'attribut n'est pas trouvé disposant d'une méthode **__set__**, alors soit il existe dans **une_instance**, et on a alors l'affectation tout à fait classique, soit il n'existe pas dans une_instance, auquel cas il est créé comme attribut de **une_instance**. C'est bien ce que l'on fait habituellement pour enrichir un objet ; mais attention cela ne fonctionne correctement que si l'attribut avec une méthode **__set__** n'est pas accessible ; il est tout à fait valide que le même attribut soit un attribut de classe ; cela n'a aucune incidence sur le processus.

* **une_classe.un_attribut = valeur**: l'affectation est directe, sans passer par la méthode **__set__**. C'est évidemment assez déroutant, mais dans ce cas, la valeur initiale de l'attribut est changée pour la nouvelle valeur, sans passer par **__set__.**

2.2. diverses utilisations

```
Test_Classe.leTitre = "------------- classe de test ------------"
print (Test_Classe.leTitre)
instance.leTitre = "################ classe test ###############"
print (Test_Classe.leTitre)
print (instance.leTitre)
instance.code = 2010
instance.display()
Test_Classe.code = 1001
```

On prendra garde cependant au fait que si une tentative de lecture d'un attribut échoue, il y a erreur, alors que dans une affectation, si la tentative d'accès échoue, l'affectation se transforme en la création d'un nouvel attribut. C'est même la façon commune de créer une attribut !

2.3. la méthode __set__

La méthode **__set__** est appelée avec deux attributs : le premier est l'objet propriétaire de l'attribut (une instance), quel que soit l'objet d'appel. Le second est la **valeur**.

3. le cas de la suppression d'un attribut

L'instruction de suppression est : **`del un_objet.un_attribut`**

Si **`un_objet`** est le propriétaire de l'attribut, alors **`un_objet.un_ attribut`** est considéré comme un objet habituel, et l'opérateur **`del`** est appliqué. On rappelle que **`del`** supprime une référence sur l'objet et donc ne le met à disposition du ramasse miettes que s'il n'y a pas d'autre référence sur l'objet ; dans ce cas, un événement est produit sur l'objet ; si celui-ci possède la méthode **__del__**, cette méthode est appelée. Mais attention : la méthode permet uniquement de placer un avertissement ou un traitement préalable à la suppression ; la suppression est sans regret.

Si **`un_objet`** est une instance, la machinerie habituelle, qui ici met enjeu la méthode de **`object()`** **__delattrib__**, permet de remonter à l'attribut et la méthode **__delete__** de l'attribut, si elle existe est appelée avec comme paramètre **`un_objet`**. Sinon, il y a erreur ! Il revient donc au programmeur de programmer la mise hors-jeu de **`un_attribut`** ; si on essaie de supprimer **`un_attribut`** depuis la méthode **__delete__** on se trouve devant une impossibilité : supprimer un objet à partir d'une de ses instances. La seule possibilité est de trouver comment mettre l'attribut hors d'atteinte. Le procédé pratique souvent rencontré consiste à effacer la valeur retournée par la méthode **__get__** (si elle existe).

4. descripteurs

4.1. définitions

* Un attribut qui possède au moins une des trois méthodes **__get__**, **__set__** ou **__del__** est appelée un descripteur.

* Un descripteur qui possède la méthode **__get__** mais pas la méthode **__set__** est un descripteur sans données. L'exemple type de descripteur sans données est la fonction ; c'est le fait d'être un tel descripteur par défaut qui permet aux méthodes d'être appelées à la fois depuis une instance et la classe.

* un descripteur qui possède la méthode **__set__** est appelé un descripteur de données ; sauf cas d'espèce, les descripteurs de données possèdent également la méthode **__get__**.

* un attribut descripteur est nécessairement **un attribut de classe**, jamais un attribut d'instance.

4.2. exemple type

thème

on définit une classe **`Ma_Classe`** réduite à deux champs de classe : l'un **`x`** est un descripteur et l'autre **`y`** n'est pas un descripteur. Le descripteur est une instance d'une classe **`Mon_Constructeur`**, ce qui permet d'associer clairement les méthodes descripteur à l'attribut.

le script

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
# desc-montrer.py

class Mon_Descripteur (object) :
    """ Le descripteur comporte cinq méthodes dont les trois méthodes
        descripteur. On a donné les moyens de tracer les méthodes.
    """
    def __init__ (self, initValeur) :
        print ("\n>>> création du descripteur")
        print ("-"*10,"self : ", self)
        print ("-"*10,"initValeur : ", initValeur)
        self.__valeur = initValeur

    def __get__ (self, objet, objetType) :
        print ("\n>>> récupération de la valeur par __get__")
```

```
        print ("-"*10,"self : ", self)
        print ("-"*10,"objet :", objet)
        print ("-"*10,"objetType :", objetType)
        return self.__valeur

    def __set__ (self, objet, valeur):
        print ("\n>>> changement de la valeur par __set__")
        print ("-"*10,"self : ", self)
        print ("-"*10,"objet :", objet)
        print ("-"*10,"valeur :", valeur)
        self.__valeur = valeur

    def ___delete__ (self, objet) :
        print ('\neffacement par __delete__')
        print ("objet", objet)
        del self.__valeur

    def __del__(self) :
        print ('\neffacement par OPÉRATEUR del')
        print ("l'effacement est sans regret !")

class Ma_Classe (object):
    x = Mon_Descripteur (10000)
    y = "je suis l'attribut normal y"


monInstance = Ma_Classe()

print ("\n","*"*34,"\n********** tests sur x ***********","\n","*"*34)
print ("+"*15,"monInstance.x :", monInstance.x)
print ("+"*15,"Ma_Classe.x :", Ma_Classe.x)

print ("\n","*"*34,"\n********** tests sur y ***********","\n","*"*34,"\n")
print ("+"*15,"monInstance.y :", monInstance.y)
print ("+"*15,"Ma_Classe.y :", Ma_Classe.y)
```

```
>>> création du descripteur
---------- self :  <__main__.Mon_Constructeur object at 0x7ff3d56a0910>
---------- initValeur :  10000

 **********************************
********** tests sur x ***********
 **********************************

>>> récupération de la valeur par __get__
---------- self :  <__main__.Mon_Constructeur object at 0x7ff3d56a0910>
---------- objet : <__main__.Ma_Classe object at 0x7ff3d56a0990>
---------- objetType : <class '__main__.Ma_Classe'>
+++++++++++++++ monInstance.x : 10000

>>> récupération de la valeur par __get__
---------- self :  <__main__.Mon_Constructeur object at 0x7ff3d56a0910>
---------- objet : None
---------- objetType : <class '__main__.Ma_Classe'>
+++++++++++++++ Ma_Classe.x : 10000

 **********************************
********** tests sur y ***********
 **********************************

+++++++++++++++ monInstance.y : je suis l'attribut normal y
+++++++++++++++ Ma_Classe.y : je suis l'attribut normal y
```

compléments pour l'affectation

```
print ("\n","*"*39,"\n***** affectation par monInstance.x *****","\n","*"*39)
monInstance.x = 999999
print ("+"*15,"monInstance.x = 999999 :")
print ("+"*15,"monInstance.x :", monInstance.x)
print ("+"*15,"Ma_Classe.x :", Ma_Classe.x)

print ("\n","*"*39,"\n****** affectation par Ma_Classe.x ******","\n","*"*39)
Ma_Classe.x = 101010
```

```
print ("+"*15,"Ma_Classe.x = 101010 :")
print ("+"*15,"monInstance.x :", monInstance.x) # instance -> classe
print ("+"*15,"Ma_Classe.x :", Ma_Classe.x)
```

```
 ************************************
***** affectation par monInstance.x *****
 ************************************

>>> changement de la valeur par __set__
---------- self :  <__main__.Mon_Constructeur object at 0x7f5fb88e7bd0>
---------- objet : <__main__.Ma_Classe object at 0x7f5fb88e7c50>
---------- valeur : 999999
+++++++++++++++ monInstance.x = 999999 :

>>> récupération de la valeur par __get__
---------- self :  <__main__.Mon_Constructeur object at 0x7f5fb88e7bd0>
---------- objet : <__main__.Ma_Classe object at 0x7f5fb88e7c50>
---------- objetType : <class '__main__.Ma_Classe'>
+++++++++++++++ monInstance.x : 999999

>>> récupération de la valeur par __get__
---------- self :  <__main__.Mon_Constructeur object at 0x7f5fb88e7bd0>
---------- objet : None
---------- objetType : <class '__main__.Ma_Classe'>
+++++++++++++++ Ma_Classe.x : 999999

 ************************************
****** affectation par Ma_Classe.x ******
 ************************************
effacement par OPÉRATEUR del
l'effacement est sans regret !
+++++++++++++++ Ma_Classe.x = 101010 :
+++++++++++++++ monInstance.x : 101010
+++++++++++++++ Ma_Classe.x : 101010
```

Le fait que l'attribut soit un attribut de la classe entraîne un remplacement direct lors de l'affectation **`Ma_Classe.x = 101010`**. L'attribut descripteur **`x`** disparaît : cela explique le message émis par la méthode **`__del__`** qui est alors invoquée.

ajout dynamique d'un descripteur

```
print ("\n","*"*46,"\n****** création dynamique de constructeur ******","\n","*"*46)
monInstance.z = 3.111598
Ma_Classe.z = Mon_Descripteur (44444444)
print ("+"*15,"monInstance.z :", monInstance.z)
print ("+"*15,"Ma_Classe.z :", Ma_Classe.z)
print ("\ndico de monInstance : ", str(monInstance.__dict__))
```

```
 ******************************************
****** création dynamique de constructeur ******
 ******************************************

>>> création du descripteur
---------- self :  <__main__.Mon_Descripteur object at 0x7ff2a9b52d10>
---------- initValeur :  44444444

>>> récupération de la valeur par __get__
---------- self :  <__main__.Mon_Descripteur object at 0x7ff2a9b52d10>
---------- objet : <__main__.Ma_Classe object at 0x7ff2a9b52d90>
---------- objetType : <class '__main__.Ma_Classe'>
+++++++++++++++ monInstance.z : 44444444

>>> récupération de la valeur par __get__
---------- self :  <__main__.Mon_Descripteur object at 0x7ff2a9b52d10>
---------- objet : None
---------- objetType : <class '__main__.Ma_Classe'>
+++++++++++++++ Ma_Classe.z : 44444444

dico de monInstance :  {'z': 3.141598}
```

* il y a occultation de l'attribut **`z`** dans le dictionnaire de l'instance.

effacement d'un descripteur

```
print ("\n","*"*41,"\n****** suppression d'un descripteur ******","\n","*"*41)
print ("del Ma_classe.z")
del Ma_Classe.z
try :
    print ("+"*15,"monInstance.z :", monInstance.z)
    print ("+"*15,"Ma_Classe.z :", Ma_Classe.z)
except Exception as e  :
    print (e)
    print ("la suppression du descripteur est effective")
```

```
 ****************************************
****** suppression d'un descripteur ******
 ****************************************
del Ma_classe.z

effacement par OPÉRATEUR del
l'effacement est sans regret !
+++++++++++++++ monInstance.z : 3.141598
type object 'Ma_Classe' has no attribute 'z'
la suppression du descripteur est complète
```

occultation d'un descripteur

Afin de mieux expliciter l'occultation, on a légèrement modifié les méthodes du descripteur, de façon à lecer directement une exception en cas d'utilisation erronée de **__get__** et **__set__ :**

```
    def __get__ (self, objet, objetType) :
        if not hasattr (self,"_Mon_Descripteur__valeur") :
            raise Exception("Erreur -- l'attribut a été supprimé") # lève une erreur
        print ("\n>>> récupération de la valeur par __get__")
        print ("-"*10,"self : ", self)
        print ("-"*10,"objet :", objet)
        print ("-"*10,"objetType :", objetType)
        return self.__valeur

    def __set__ (self, objet, valeur):
        if not hasattr (self,"_Mon_Descripteur__valeur") :
            raise Exception("Erreur -- l'attribut a été supprimé") # lève une erreur
        print ("\n>>> changement de la valeur par __set__")
        print ("-"*10,"self : ", self)
        print ("-"*10,"objet :", objet)
        print ("-"*10,"valeur :", valeur)
        self.__valeur = valeur
```

le code :

```
Ma_Classe.u = Mon_Descripteur (77777777)
print ("+"*30,"\ndel monInstance.u")
del monInstance.u

try :
    print ("\nMa_Classe.u :")
    print (Ma_Classe.u)
except Exception as e  :
    print (e)
    print ("::::::::::: l'attribut est illisible ")

try :
    print ("\nmonInstance.u = 123456789")
    monInstance.u = 123456789
except Exception as e :
    print (e)
    print ("::::::::: l'attribut est inutilisable")
```

```
++++++++++++++++++++++++++++++
del monInstance.u
```

```
effacement par __delete__
objet <__main__.Ma_Classe object at 0x7f28a2ac8e90>

Ma_Classe.u :
Erreur -- l'attribut a été supprimé
:::::::::: l'attribut est illisible

monInstance.u = 123456789
Erreur -- l'attribut a été supprimé
::::::::: l'attribut est inutilisable
```

5. la fonction prédéfinie `property()`

5.1. la fonction

il s'agit d'une méthode globale prédéfinie. Sa définition est :

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

Elle retourne un attribut descripteur.

5.2. `property()` en pur Python (première partie).

La meilleure façon de procéder et de donner une simulation en pur Python de ce descripteur.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# desc-property.py

class property (object) :
    """ simulation en pur Python de la fonction globale property() """
    def __init__ (descself, fget=None, fset=None, fdel=None, doc=None) :
        descself.fget = fget
        descself.fget = fget
        descself.fdel = fdel
        if doc is None and fget is not None :
            doc = fget.__doc__  # emprunt de la doc de fget
        descself.doc = doc
    # ****************************
    # les méthodes de descripteur *
    # ****************************
    def __get__ (descself, objet, typeobjet= None) :
        if objet is None : # l'objet d'appel est la classe
            return self
        if descself.fget is None :
            raise AttributeError ("l'attribut n'est pas lisible" )
        return descself.fget (objet) # l'objet d'appel est une instance

    def __set__ (descself, objet, valeur) :
        if descself.fset is None :
            raise AttributeError ("l'attribut n'est pas accessible en écriture" )
        descself.fset (objet, valeur)

    def __delete__(descself, objet) :
        if descself.fset is None :
            raise AttributeError ("l'attribut ne peut être supprimé par delete" )
        descself.fdel (objet, valeur)
```

Cela implique des contraintes pour **`fget, fset, fdel`** qui apparaissent comme paramètres puis comme attributs du descripteur (en lecture seule dans l'implémentation par défaut) : **`fget`** doit retourner une valeur, **`fset`** prend en paramètre la valeur associée au descripteur. On a appelé **`descself`** le paramètre usuellement dénommé **`self`** pour rappeler que l'instance de la classe est un descripteur.

L'archétype de définition de **`fget, fset, fdelete`** est donné par le schéma de classe Python dont le descripteur **`x`** est l'attribut :

```
class C:
    def __init__(self):
        self._x = None
    def getx(self):
```

```
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx, setx, delx)
```

5.3. `property()` en pur Python (deuxième partie).

La classe **`property`** a trois méthodes en plus, qui lui permettent d'enrichir ou de modifier un un objet **`propriety`** qui existe déjà.

```
    # ***********************************
    # les méthodes d'ajout / substitution *
    # ***********************************
    def getter (descself, fget) :
        """ appelée depuis un descripteur, la méthode ajoute une méthode __get__
            ou remplace celle qui existe
        """
        return type(descself)(fget, descself.fset, descself.fdelete, descself.__doc__)

    def setter (descself, fset) :
        """ appelée depuis un descripteur, la méthode ajoute une méthode __set__
            ou remplace celle qui existe
        """
        return type(descself)(descself.fget, fset, descself.fdelete, descself.__doc__)

    def deleter (descself, fdel) :
        """ appelée depuis un descripteur, la méthode ajoute une méthode __delete__
            ou remplace celle qui existe
        """
        return type(descself)(descself.fget, descself.fset, fdelete, descself.__doc__)
```

6. méthodes de classe et méthodes statiques

6.1. rappel

Une méthode statique est une méthode qui utilise les paramètres de la même façon, que l'appel se fasse à partir d'une classe ou d'une instance :

Soit la méthode de la classe **`Ma_Classe`** et de signature : **`methStatique (p1, p2, p3).`** Si elle est statique les deux appels suivants, où **`monInstance`** est une instance de **`Ma_Classe`**, son équivalents :

```
Ma_Classe.methStatique (val1, val2, val3)
monInstance.methStatique (val1, val2, val3)
```

Une méthode statique n'a pas nécessairement de paramètre dans la signature.

Une méthode de classe est une méthode dont le premier paramètre est remplacé non par une instance, mais par la classe à laquelle elle appartient. Ce premier paramètre est donc nécessaire dans la signature. Par exemple si la signature est **`methKlasse (klasse, p1, p2)`** l'appel se fait sous la forme suivante :

```
Ma_Classe.methKlasse (val1, val2)
monInstance.methKlasse (val1, val2)
```

et le paramètre **`klasse`** désigne la classe dans la méthode.

Il existe deux **fonctions prédéfinies** qui permettent de transformer une fonction en méthode statique ou méthode de classe : **`staticmethod()`** et **`classmethod()`** qui prennent comme argument une fonction et retournent la méthode statique ou de classe correspondante.

L'objectif de la présente note est de monter comment l'utilisation de descripteur permet une implémentation en pur Python de ces fonctions prédéfinies.

6.2. enveloppement

principe

Le principe est de créer un descripteur sans données **Mon_Getter** dont la méthode **__get__** retourne une fonction. Cette fonction enveloppe une fonction passée au constructeur du descripteur, en lui ajoutant quelques fonctionnalités. Pour faciliter l'analyse, on a tracé les paramètres passés par **__setattribute__**.

On a utilisé **_display** et **display** ; il n'y a pas d'inconvénient à utiliser le même mot, et même à prendre une fonction globale pour la première fonction.

Le script montre comment on fait un appel de fonction selon que la fonction descripteur sans donnée (**display**) a été appelées depuis une instance ou depuis la classe. La méthode **__get__()** utilisée ici est en pur Python celle des méthodes.

le script

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# desc-fonction-get.py

class Mon_Getter (object) :
    def __init__(selfdesc, fonction) :
        selfdesc.fonction = fonction

    def __get__ (selfdesc, obj, objtype) :
        print ('\n====>>> Récupération par getter')
        print ("    self ", selfdesc)
        print ("    obj", obj)
        print ("    objType", objtype)
        def nouvelleFonction (*arg) :
            if obj : #instance
                return selfdesc.fonction (obj,*arg)
            else :
                return selfdesc.fonction (*arg)
        print ("="*20)
        return nouvelleFonction


class Test_Classe (object) :
    """ la classe qui va être testée    """
    leTitre = "******** classe de test *************"
    def __init__ (self, parNom, parCode) :
        """ initialisation de la classe Test_Classe  """
        self.nom = parNom
        self.code = parCode
        print ("création d'une instance")

    def _display (self, info) :
        """ cette méthode display est documentée """
        print (">>> info : ", info)
        print (">>> nom : "+self.nom, " //  code : ", self.code)
    display = Mon_Getter (_display)

def titre (self) :
    print (self.leTitre)
Test_Classe.titre = titre

instance = Test_Classe ("Caroline", 1973)
Test_Classe.display(instance,"appel par :: Test_Classe.display")
instance.display("appel par :: instance.display")
```

```
création d'une instance

====>>> Récupération par getter
    self  <__main__.Mon_Getter object at 0x7f89bb8f6590>
    obj None
    objType <class '__main__.Test_Classe'>
====================
>>> info :  appel par :: Test_Classe.display
>>> nom : Caroline  //  code :  1973

====>>> Récupération par getter
    self  <__main__.Mon_Getter object at 0x7f89bb8f6590>
    obj <__main__.Test_Classe object at 0x7f89bb8f65d0>
    objType <class '__main__.Test_Classe'>
====================
>>> info :  appel par :: instance.display
```

```
>>> nom : Caroline  //  code :  1973
```

6.3. enveloppe pour méthode statique

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# desc-fonction-static.py

class Mon_Getter_Static (object) :
    def __init__(selfdesc, fonction) :
        selfdesc.fonction = fonction

    def __get__ (selfdesc, obj, objtype) :
        def nouvelleFonction (*arg) :
            return selfdesc.fonction (*arg)
        nouvelleFonction.__doc__ = selfdesc.fonction.__doc__
        nouvelleFonction.__dict__ = selfdesc.fonction.__dict__
        nouvelleFonction.__name__ = selfdesc.fonction.__name__
        return nouvelleFonction

class Test_Classe (object) :
    """ la classe qui va être testée    """
    leTitre = "******** classe de test *************"
    def __init__ (self, parNom, parCode) :
        """ initialisation de la classe Test_Classe """
        self.nom = parNom
        self.code = parCode

    def display (self, info) :
        """ cette méthode display est documentée """
        print (">>> info : ", info)
        print (">>> nom : "+self.nom, " //  code : ", self.code)

    def affInfo (info) :
        print (info)
    affInfo = Mon_Getter_Static (affInfo)

instance = Test_Classe ("Caroline", 1973)
instance.display("appel par :: instance.display")

Test_Classe.affInfo ("appel depuis :: Test_Classe.affInfo")
instance.affInfo ("appel depuis :: instance.affInfo")
print ("\n","!"*5,instance.affInfo.__name__,instance.affInfo.__doc__)
```

```
>>> info :  appel par :: instance.display
>>> nom : Caroline  //  code :  1973
appel depuis :: Test_Classe.affInfo
appel depuis :: instance.affInfo

 !!!!! affInfo    cette méthode affiche son argument
```

6.4. enveloppe pour méthode de classe

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# desc-fonction-class.py

class Mon_Getter_Class (object) :
    def __init__(selfdesc, fonction) :
        selfdesc.fonction = fonction

    def __get__ (selfdesc, obj, objtype) :
        def nouvelleFonction (*arg) :
            return selfdesc.fonction (objtype, *arg)
        nouvelleFonction.__doc__ = selfdesc.fonction.__doc__
        nouvelleFonction.__dict__ = selfdesc.fonction.__dict__
        nouvelleFonction.__name__ = selfdesc.fonction.__name__
        return nouvelleFonction

class Test_Classe (object) :
    """ la classe qui va être testée    """
```

```
    leTitre = "******** classe de test *********"
    def __init__ (self, parNom, parCode) :
        """ initialisation de la classe Test_Classe """
        self.nom = parNom
        self.code = parCode

    def display (self, info) :
        """ cette méthode display est documentée """
        print (">>> info : ", info)
        print (">>> nom : "+self.nom, " //  code : ", self.code)

    def affInfo (klass, info) :
        """ méthode de classe obtenue par descripteur """
        print ("\n~~~", info)
        print ("~~~ la classe s'appelle ",klass.__name__)
        print ("~~~ voici son dictionnaire :")
        for clef in klass.__dict__.keys() :
            print ((" "*4+clef+" "*20)[:20],klass.__dict__[clef])
    affInfo = Mon_Getter_Class (affInfo)

instance = Test_Classe ("Caroline", 1973)
instance.display("\nappel de :: instance.display")


Test_Classe.affInfo ("premier appel, depuis :: Test_Classe.affInfo")
instance.affInfo ("second appel, depuis :: instance.affInfo")

print ("\n","!"*5,instance.affInfo.__name__,instance.affInfo.__doc__)
```

```
>>> info :
appel de :: instance.display
>>> nom : Caroline  //  code :  1973

~~~ premier appel, depuis :: Test_Classe.affInfo
~~~ la classe s'appelle  Test_Classe
~~~ voici son dictionnaire :
    affInfo          <__main__.Mon_Getter_Class object at 0x7f911413e0d0>
    __doc__           la classe qui va être testée
    __init__         <function Test_Classe.__init__ at 0x7f91141463b0>
    __weakref__      <attribute '__weakref__' of 'Test_Classe' objects>
    __dict__         <attribute '__dict__' of 'Test_Classe' objects>
    leTitre          ******** classe de test *********
    display          <function Test_Classe.display at 0x7f9114146440>
    __module__       __main__

~~~ second appel, depuis :: instance.affInfo
~~~ la classe s'appelle  Test_Classe
~~~ voici son dictionnaire :
    affInfo          <__main__.Mon_Getter_Class object at 0x7f911413e0d0>
    __doc__           la classe qui va être testée
    __init__         <function Test_Classe.__init__ at 0x7f91141463b0>
    __weakref__      <attribute '__weakref__' of 'Test_Classe' objects>
    __dict__         <attribute '__dict__' of 'Test_Classe' objects>
    leTitre          ******** classe de test *********
    display          <function Test_Classe.display at 0x7f9114146440>
    __module__       __main__

 !!!!! affInfo  méthode de classe obtenue par descripteur
```

7. rappel sur les décorateurs

7.1. les wrapper

On rappelle que l'on nomme décorateur une fonction (out tout callable) qui prend en argument une fonction (en général une méthode) et retourne une fonction (ou méthode ou un objet quelconque) qui est un prolongement de la fonction passée en argument. La fonction argument est exécutée lorsque l'on appelle la fonction retournée, et quelques fonctionnalité lui sont ajoutés. C'est le cas des **callables** que sont **`Mon_Getter_Static`** et **`MonGetter_Class`**.

La fonction retournée, quant elle redéfinit la fonction argument (même identificateur, même nom, dictionnaire, documentation) est dit une enveloppe ou **wrapper** de la fonction argument.

Or, il existe un procédé pratique pour définir un wrapper : la décoration. Au lieu d'appeler la fonction transformante **`fnTrans`** en utilisant le même identificateur **`fnArg`** pour l'argument et le retour, il suffit de faire précéder la ligne ou se trouve le **`def fnArg (arguments)`** de **`@fnTrans`**.

Le décorateur **`@fnTrans`** doit occupe une ligne à lui seul. Il peut y avoir plusieurs décorateurs pour une fonction, exécutés du plus profond au plus extérieur.

```
    @Mon_Getter_Static
    def affInfo (info) :
        """   cette méthode affiche son argument """
        print (info)
    #affInfo = Mon_Getter_Static (affInfo)
```

7.2. `property` et décorateur

Si une fonction décorateur retourne en général une méthode, rien n'y oblige : elle peut retourner un descripteur tout aussi bien. Et une fonction décorateur peut de même engendrer un décorateur. Comme la fonction **`property()`** retourne un descripteur, rien ne s'oppose non plus à ce **`@property`** soit un décorateur sur une fonction qui retourne une valeur ; comme un décorateur retourne un objet de même nom que la fonction qui lui est soumise (en l'occultant par ailleurs) le descripteur et la fonction soumise au décorateur ont le même nom.

Voici un exemple typique :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# desc-deco.py

class Test_Classe (object) :
    """ la classe qui va être testée    """
    leTitre = "******** classe de test *********"
    def __init__ (self, parNom, parCode) :
        """ initialisation de la classe Test_Classe """
        self.nom = parNom.upper()
        self.code = parCode

    def display (self, info) :
        """ cette méthode display est documentée """
        print (">>> info : ", info)
        print (">>> nom : "+self.nom, " //  code : ", self.code)

    @property
    def donnerTitre (self) :
        return self.leTitre

    @donnerTitre.setter
    def donnerTitre (self, valeur="") : # le même nom
        self.leTitre = valeur
    #donnerTitre = donnerTitre.setter (donnertitre)

monInstance = Test_Classe ("Jean-François", 2013)
monInstance.display ("essais de descripteurs")
print (monInstance.donnerTitre)

monInstance.donnerTitre = "++++++++++ nouveau titre ++++++++++"
print (monInstance.donnerTitre)
```

```
>>> info :  essais de descripteurs
>>> nom : JEAN-FRANÇOIS  //  code :  2013
******** classe de test *********
++++++++++ nouveau titre ++++++++++
```