1 Méthodes d'instances

Nous avons déjà parlé des méthodes aux chapitres 1 et 5. Nous savons que toute fonction écrite dans le corps d'une classe devient une méthode des instances. Voici un exemple :

```
class A:
    def méthode(self):
        print(id(self))
```

Nous venons d'implémenter une fonction nommée `méthode` dans le corps de la classe `A`. Pour Python, `A.méthode` est bien une fonction :

```
>>> type(A.méthode)
<class 'function'>
```

En revanche, si `obj` est une instance de `A`, alors `obj.méthode` est une méthode :

```
>>> obj = A()
>>> type(obj.méthode)
<class 'method'>
```

On dit que `obj.méthode` est une *méthode d'instance*. Il s'agit d'un objet appelable et lorsqu'on l'appelle, Python appelle la fonction correspondante en faisant passer l'instance d'appel en premier argument. Concrètement ici, l'appel `obj.méthode()` équivaut à l'appel `A.méthode(obj)`. La preuve :

```
>>> id(obj)
4339289952
>>> obj.méthode()
4339289952
>>> A.méthode(obj)
4339289952
```

De manière plus générale, si la méthode a pour signature `(a,b,c,d)`, comme ci-dessous, l'appel `obj.méthode(x,y,z)` est équivalent à `A.méthode(obj,x,y,z)` :

```
class A:
    def méthode(a,b,c,d): pass
```

On retiendra ceci : le premier paramètre d'une fonction écrite dans le corps d'une classe reçoit l'instance d'appel en premier argument lorsque l'appel a lieu depuis une instance. On recommande de nommer `self` le premier paramètre (afin de faciliter la lecture du script). Le bon style pour la méthode précédente est donc :

```
class A:
    def méthode(self,b,c,d): pass
```

Note. Le mot `self` n'est pas réservé.

On peut imaginer, pour simplifier, que `obj.méthode` est un attribut de `obj` (tout se passe comme si tel était le cas), mais en vérité, les choses sont un peu plus compliquées : chaque fois que Python lit `obj.méthode`, il crée une méthode liée à `obj` à partir de la fonction `A.méthode`.

```
>>> a = A()
>>> a.méthode is a.méthode
False
```

2 Méthodes de classes

La fonction `classmethod` transforme une fonction en une méthode de classe. Commençons par une petite expérience écrite dans un style inhabituel mais pédagogiquement intéressant :

```
def f(x):
    print(id(x))
class A:
    mc = classmethod(f)
```

Nous avons muni la classe `A` d'un attribut de type `classmethod`.

La règle suivie par Python sur cet exemple précis est la suivante : l'appel `A.mc()` équivaut à l'appel `f(A)`. La preuve :

```
>>> A.mc()
4540427432
>>> f(A)
4540427432
>>> id(A)
4540427432
```

Le style recommandé pour implémenter ce qui précède est celui-ci :

```
class A:
    @classmethod
    def mc(cls):
```

```
        print(id(cls))
```

On recommande de nommer `cls` le premier paramètre d'une méthode de classe (puisque lors d'un appel, c'est la classe qui est passée en premier argument).

Note. Le mot `cls` n'est pas réservé. Il s'agit d'une simple convention.

Chaque fois que Python lit « `A.mc` » il crée une méthode liée à `A` à partir de l'objet `mc`. L'objet `mc` est de type `classmethod`, mais l'objet `A.mc` est de type *method* :

```
>>> A.mc is A.mc
False
>>> type(A.mc)
<class 'method'>
```

Note. Si `obj` est une instance de `A`, chaque fois que Python lit « `obj.mc` » il crée une méthode liée à `A` (et non `obj`) à partir de l'objet `mc`. Ainsi, l'appel `obj.mc()` équivaut à `A.mc()`. On peut donc appeler une méthode de classe depuis une instance, même si cela n'apporte rien (Python fera passer la classe en premier argument).

3 Méthodes statiques

La fonction `staticmethod` transforme une fonction en une méthode statique. Commençons par une expérience écrite dans un style inhabituel mais pratique pour comprendre :

```
def f():
    print("hey")
class A:
    ms = staticmethod(f)
```

Nous avons muni la classe `A` d'un attribut de type `staticmethod`.

La règle suivie par Python sur cet exemple précis est la suivante : l'appel `A.ms()` équivaut à l'appel `f()`. La preuve :

```
>>> A.ms()
hey
>>> f()
hey
```

De même, l'appel `a.ms()` sur une instance `a` de `A` équivaut à `f()` :

```
>>> a = A()
```

```
>>> a.ms()
hey
```

On peut croire qu'il revient au même d'écrire `ms` comme fonction dans le corps de `A` puisque dans ce cas, l'appel `A.ms()` produit le même résultat. Faisons l'expérience avec une autre classe :

```
class B:
    def ms(): print("hey")
```

Cela produit bien le même résultat quand on appelle `ms` sur `B` :

```
>>> B.ms()
hey
```

mais les choses se gâtent quand on appelle depuis une instance :

```
>>> b = B()
>>> b.ms()
TypeError: ms() takes 0 positional arguments but 1 was given
```

C'est bien normal : `b.ms` n'est pas la fonction `B.ms`, mais la méthode liée à l'objet `b`. Par conséquent l'appel à `b.ms` fait passer l'instance `b` en premier argument, selon le mécanisme habituel.

On mesure ici la différence entre l'écriture d'une `staticmethod` et l'écriture d'une fonction dans le corps d'une classe.

Le style recommandé pour implémenter une méthode statique est le suivant :

```
class A:
    @staticmethod
    def ms():
        print("hey")
```

4 Cas d'utilisation

4.1 Généralités

Soit `obj` une instance de `A` et `action` un attribut-fonction de `A`.

– Si `obj.action` a pour but de modifier l'état de `obj`, ou bien si elle a besoin de connaître l'état de `obj` pour produire son effet, il n'y a aucun doute à avoir : `action` doit être implémentée comme méthode de l'instance `obj`.

– Si `obj.action` n'a besoin ni de `obj`, ni de `A` pour produire son effet, il n'y a aucun doute à avoir : `action` doit être implémentée comme méthode statique de `A`.

– Si `obj.action` n'a pas besoin de `obj` mais a besoin de `A` pour produire son effet, il n'y a aucun doute à avoir : `action` doit être implémentée comme méthode de la classe `A`.

On retiendra également ceci :

1. Les méthodes de classe typiques sont les *factory methods* (voir sous-section suivante).
2. Il est recommandé de regrouper toutes les fonctions possédant un lien avec une classe `A` dans le corps de `A` en les décorant avec `@staticmethod` (au passage, cela empêche de surcharger l'espace des noms globaux).

4.2 Exemples internes

4.2.1 `float.fromhex`

On rappelle que si `ch` est une chaîne contenant l'écriture d'un nombre en base 16 (chaîne commençant par `"0x"`), alors `float.fromhex(ch)` retourne la conversion de ce nombre en base 10 :

```
>>> float.fromhex("0xaa120c")
11145740.0
>>> float.fromhex("0x1.aaaa")
1.666656494140625
```

Cette méthode retourne un `float`, c'est une *factory method*. Pas étonnant qu'il s'agisse d'une méthode de classe. La preuve : si on dérive `float`

```
class Longueur(float):
    def __new__(cls,x,u="m"):
        instance = super().__new__(cls,x)
        instance.unité = u
        return instance
```

et qu'on appelle `fromhex` depuis la dérivée, on obtient ceci :

```
>>> L = Longueur.fromhex("0xaaa")
>>> type(L)
<class '__main__.Longueur'>
```

On constate que `fromhex` retourne non pas un `float`, mais une `Longueur` : la méthode a donc reçu la classe en argument.

Note. On aurait pu dériver `float` en écrivant tout simplement ceci :

```
class A(float):
    pass
```

4.2.2 `int.from_bytes`

On rappelle que si `b` est une chaîne d'octets (ie un objet de type `bytes` ou `bytearray`), il peut être interprété comme l'écriture en base 16 d'un entier naturel (chaque byte correspond à 2 chiffres) et `int.from_bytes(b,byteorder="big")` retourne la conversion en base 10 de cet entier :

```
>>> hex(8000)
0x1f40
>>> int.from_bytes(b"\x1f\x40",byteorder="big")
8000
```

Si on prend « `byteorder = "little"` », la chaîne est lue à l'envers :

```
>>> int.from_bytes(b"\x40\x1f",byteorder="little")
8000
```

La fonction `from_bytes` retourne un `int`, c'est une *factory method*. Pas étonnant qu'il s'agisse d'une méthode de classe. On le démontre comme on a fait à la sous-section précédente :

```
class Z(int): pass
```

On appelle `from_bytes` depuis la dérivée :

```
>>> n = Z.from_bytes(b"abc",byteorder="big")
>>> type(n)
<class '__main__.Z'>
```

4.2.3 `str.maketrans`

On rappelle que la méthode `str.maketrans` fabrique le dictionnaire qui sert de table à la méthode `translate` pour crypter une chaîne. Supposons par exemple que l'on souhaite crypter une chaîne en remplaçant chaque `"R"` par `"M"` et chaque `"e"` par `"o"`. La table serait :

$$\begin{array}{rcl} R & \longmapsto & M \\ e & \longmapsto & o \end{array}$$

On implémente cette table comme ceci :

```
>>> table = str.maketrans("Re","Mo")
```

```
>>> table
{82: 77, 101: 111}
```

Le dictionnaire ainsi obtenu ne contient pas les caractères mais les numéros unicode. Pour crypter la chaîne `"Reeds"` avec cette table, on fait comme ceci :

```
>>> "Reeds".translate(table)
'Moods'
```

La méthode `str.maketrans` est statique :

```
>>> help(str.maketrans)
Help on built-in function maketrans:
maketrans(...)
  str.maketrans(x[, y[, z]]) -> dict (static method)
```

On l'aurait deviné : d'une part, cette méthode n'est pas une *factory method*. D'autre part, elle reçoit deux chaînes à partir desquelles elle construit le dictionnaire qu'elle retourne. Elle n'a pas besoin de savoir dans quelle classe elle a été écrite (`str` n'est pas un argument pour elle).

4.2.4 `dict.fromkeys`

On rappelle que si `L` est un itérable et `v` un objet quelconque, `dict.fromkeys(L,v)` retourne le dictionnaire qui à chaque item de `L` associe la valeur `v`. Pour que cette opération puisse avoir lieu, les items de `L` doivent être hachables :

```
>>> dict.fromkeys("aeiou",0)
{'a': 0, 'e': 0, 'i': 0, 'o': 0, 'u': 0}
```

Cette méthode retourne un `dict`, c'est une *factory method*. Pas étonnant qu'il s'agisse d'une méthode de classe. Même démonstration qu'à la sous-sous-section 6.4.2.1 :

```
class Dico(dict): pass
```

On regarde le type d'un dictionnaire produit avec `Dico.fromkey` :

```
>>> d = Dico.fromkeys([1,2,3])
>>> type(d)
<class '__main__.Dico'>
```

Attention. L'affichage de `d` peut faire croire que son type est `dict` :

```
>>> d
{1: None, 2: None, 3: None}
```

En effet, nous n'avons pas surchargé la méthode *repr* chez `Dico`, par conséquent, `d` possède la représentation formelle héritée de `dict`.

4.2.5 `bytearray.fromhex`

On rappelle que si `ch` est une chaîne de caractères contenant des octets écrits en hexadécimal, `bytearray.fromhex(ch)` retourne la chaîne d'octets correspondante, sous la forme d'un `bytearray` :

```
>>> bytearray.fromhex("1f40")
bytearray(b'\x1f@')
```

La documentation de Python 3.3.2 affirmait que `bytearray.fromhex` était une méthode statique :

```
>>> help(bytearray.fromhex)
Help on built-in function fromhex:
fromhex(...)
bytearray.fromhex(string) -> bytearray (static method)
```

or nous avons affaire à une *factory method*, ce qui nous pousse à remettre en cause cette affirmation. Pour en avoir le cœur net, nous procédons comme d'habitude, en dérivant `bytearray` :

```
class A(bytearray): pass
```

et en testant le type d'une chaîne d'octets obtenue avec `A.fromhex` :

```
>>> b = A.fromhex("1f40")
>>> type(b)
<class '__main__.A'>
```

Force est de constater que `A.fromhex` s'est servi de `A` pour fonctionner, il s'agit donc d'une méthode de classe.

Attention. L'affichage de `b` peut faire croire que son type est `bytearray` :

```
>>> b
bytearray(b'\x1f@')
```

(Nous avons connu cela avec la classe `Dico` à la section précédente.)

4.2.6 `bytes.fromhex`

La méthode `bytes.fromhex` fonctionne comme `bytearray.fromhex`. Le docstring ne précise pas s'il s'agit d'une méthode statique ou de classe, mais il s'agit d'une *factory method* et le test ci-dessous montre que c'est bien une méthode de classe :

```
class A(bytes): pass
```

On teste une chaîne d'octets obtenue avec `A.fromhex` :

```
>>> b = A.fromhex("1f40")
>>> type(b)
<class '__main__.A'>
```

4.3 Cas des fonctions d'un module

Si on importe un module comme ceci :

```
>>> import math
```

les fonctions offertes par ce dernier apparaissent comme des attributs du module :

```
>>> math.cos
<built-in function cos>
```

(on rappelle que chez Python, un attribut est un nom écrit juste après un point, voir définition 5.3.) Nous savons que `cos` est une instance de *function* représentant la fonction mathématique cosinus, mais son comportement ici est le même que celui d'une méthode statique :

```
>>>> math.cos(3.141592653589793)
-1.0
```

De manière générale, on recommande de déclarer comme méthodes statiques toutes les fonctions utilitaires utilisées dans une classe : cela évite de polluer l'espace des noms global.

4.4 Exemples avec des matrices

Nous supposons ici que le lecteur connaît les matrices de l'algèbre linéaire. Le code ci-dessous implémente la classe `Matrice` :

```
class Matrice:
    def __init__(self,L=None):
        if L:
            m = len(L)
            n = len(L[0])
            self.dim = m, n
            data = {}
            for i in range(1,m+1):
```

```
                for j in range(1,n+1):
                    data[i,j] = L[i-1][j-1]
            self.data = data
        else: self.data = {}
    def __getitem__(self,ij):
        return self.data[ij]
    def __setitem__(self,ij,v):
        self.data[ij] = v
    def longueur_max(self):
        maxi = 0
        for ij in self.data:
            maxi = max(maxi,len(str(self[ij])))
        return maxi
    def __repr__(self):
        m, n = self.dim
        k = self.longueur_max() + 2
        br = "\n"
        ch = ""
        for i in range(1,m+1):
            for j in range(1,n+1):
                ch = ch + str(self[i,j]).ljust(k)
            ch = ch + br
        return ch
```

On a surchargé la méthode spéciale *repr*. Cette méthode fabrique la chaîne de caractères retournée par la fonction `repr` lorsqu'on lui fait passer une instance de `Matrice`. On a également surchargé les méthodes *getitem* et *setitem* pour les accès aux coefficients de la matrice (voir chapitre 15).

On rappelle que si `x` est une chaîne, `x.ljust(k)` retourne la chaîne obtenue en ajoutant des espaces à `x` par la droite de sorte que la longueur totale soit égale à `k`. Le résultat est une chaîne dans laquelle le contenu de `x` se trouve aligné à gauche. Par exemple `"Reeds".ljust(10)` retourne

"	R	e	e	d	s						"

Pour définir la `Matrice` A ci-dessous :

$$\begin{pmatrix} 1 & 3 & 5 \\ 8 & 4 & 2 \end{pmatrix}$$

on écrit :

```
>>> a = Matrice([[1,3,5],[8,4,2]])
>>> a
```

```
1  3  5
8  4  2
```

Pour accéder a_{ij}, on écrit `a[i,j]` :

```
>>> a[2,1]
8
```

Les dimensions de A sont 2×3 :

```
>>> a.dim
(2, 3)
```

On peut instancier une matrice « vide » `m = Matrice()`. Cela nous sert ci-après.

On pourrait écrire une fonction pour instancier une matrice nulle de dimensions quelconques comme ceci :

```
def nulle(m,n=None):
    if n==None: n = m
    N = Matrice()
    N.dim = m, n
    for i in range(1,m+1):
        for j in range(1,n+1):
            N[i,j] = 0
    return N
```

Ceci fonctionnerait très bien, mais il est préférable, pour des raisons de style et de clarté, d'écrire cette fonction à l'intérieur du corps de la classe `Matrice`, et d'en faire une méthode de classe (puisque c'est une *factory method*) :

```
class Matrice:
    (...)
    @classmethod
    def nulle(cls,m,n=None):
        if n is None: n = m
        N = cls()
        N.dim = m, n
        for i in range(1,m+1):
            for j in range(1,n+1):
                N[i,j] = 0
        return N
```

On pourrait également l'implémenter comme une méthode statique :

```
@staticmethod
def nulle(m,n=None):
    if n is None: n = m
    N = Matrice()
    N.dim = m, n
    for i in range(1,m+1):
        for j in range(1,n+1):
            N[i,j] = 0
    return N
```

Si on teste l'une ou l'autre, on obtient ceci :

```
>>> B = Matrice.nulle(3)
>>> B
0  0  0
0  0  0
0  0  0
```

Opter pour `classmethod` possède deux avantages :

- si un jour on décide de remplacer le nom `Matrice` par `Mat`, nous n'avons pas besoin de faire des changements dans le corps de `nulle` (nous y avons écrit `cls` et non `Matrice`) ;
- si on dérive une classe `MatriceCarrée` de la classe `Matrice`, l'instruction :

```
>>> A = MatriceCarrée.nulle(4)
```

  crée bien une instance de `MatriceCarrée` (alors qu'avec une méthode statique, on obtiendrait une `Matrice`).

Un autre exemple : on rappelle que la trace d'une matrice carrée est la somme des coefficients de la diagonale principale. Si on voulait implémenter une fonction `trace`, on l'écrirait comme une méthode statique de la classe `Matrice` :

```
@staticmethod
def trace(M):
    m, n = M.dim
    if m != n: raise TypeError
    s = 0
    for i in range(1,m+1):
        s = s + M[i,i]
```

```
    return s
```

On laisse le lecteur ajouter des fonctions comme `identité` (pour retourner des matrices identités), `aléatoire` (pour générer une matrice avec des coefficients aléatoires), `colonne` (pour générer une matrice $n \times 1$), `ligne` (pour générer une matrice $1 \times n$), `diagonale` (pour générer une matrice diagonale), `saisir` (pour saisir à la main une matrice en utilisant `input`), etc.

4.5 Exemples avec des dates

Nous implémentons ici une classe `Date` pour encoder des dates comme `18-04-1971`. Cette dernière aurait pour attributs : `jour=18` ; `mois=4` et `année=1971`. On munit `Date` d'une méthode `to_Date` permettant de convertir une chaîne du genre `"16-05-2000"` ou `"16/05/2000"` en date. Nous voulons que `to_Date` retourne une instance de `Date`, même si on fait l'appel depuis une sous-classe. Il est donc raisonnable de déclarer `to_Date` comme méthode statique. On implémente également une méthode `conversion` faisant le même travail que `to_Date` à une différence près : si on appelle cette méthode à partir d'une sous-classe `A`, nous souhaitons que le résultat retourné soit de type `A`. Il est donc raisonnable de déclarer `conversion` comme méthode de classe :

```
class Date:
    def __init__(self,j,m,a):
        self.jour = j
        self.mois = m
        self.année = a
    @staticmethod
    def to_Date(chaîne):
        if "-" in chaîne: c = "-"
        else: c = "/"
        L = chaîne.split(c)
        j, m, a = int(L[0]),int(L[1]),int(L[2])
        return Date(j,m,a)
    @classmethod
    def conversion(cls,chaîne):
        if "-" in chaîne: c = "-"
        else: c = "/"
        L = chaîne.split(c)
        j, m, a = int(L[0]),int(L[1]),int(L[2])
        return cls(j,m,a)
    def __repr__(self):
        p = "{}-{}-{}"
```

```
        return p.format(self.jour, self.mois, self.année)
```

On a surchargé *repr* pour un meilleur affichage. Testons :

```
>>> ysé = Date(16,5,2000)
>>> ysé
16-5-2000
>>> yanis = Date.to_Date("27/01/2000")
>>> yanis
27-1-2000
>>> richard = Date.conversion("18/04/1971")
>>> richard
18-4-1971
```

Dérivons de `Date` la classe `DateHeure`. Une instance de cette dernière possèdera 3 attributs supplémentaires qui sont `heure`, `minute` et `seconde` :

```
class DateHeure(Date):
    def __init__(self,j,m,a,h=None,min=None,s=None):
        Date.__init__(self,j,m,a)
        self.heure = h
        self.minute = min
        self.seconde = s
    def __repr__(self):
        ch = Date.__repr__(self)
        p = ch + " {}:{}:{}"
        return p.format(self.heure, self.minute, self.seconde)
```

Testons :

```
>>> richard = DateHeure(18,4,1971,2,25,0)
>>> richard
18-04-1971 2:25:0
>>> david = DateHeure.to_Date("1/12/1973")
>>> david
1-12-1973
>>> david = DateHeure.conversion("1/12/1973")
1-12-1973 None:None:None
```

On constate bien que notre *factory method* `conversion` ne produit pas le même type d'objets selon qu'on l'appelle depuis `Date` ou `DateHeure`, tout le contraire de la statique et bien nommée `to_Date`, qui elle ne produit que des instances de `Date`.

5 Introspection

5.1 Explorer *method*

On crée une classe `A` munie d'une méthode `action` :

```
class A:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __repr__(self):
        p = "A({},{})"
        return p.format(self.x, self.y)
    def action(self):
        "Fonction pour agir"
        print("action")
```

Nous avons surchargé les méthodes spéciales *init* et *repr*. Voici le résultat :

```
>>> a = A(1,5)
>>> a
A(1,5)
```

Puisque `a` est une instance de `A`, alors `a.action` est une méthode liée à `a`. Concrètement, `a.action` est de type *method* :

```
>>> type(a.action)
<class 'method'>
```

L'affichage de `dir(a.action)` montre 5 attributs propres aux méthodes :

- `__self__` : contient l'instance à laquelle est liée la méthode ;
- `__func__` : contient la fonction d'où provient la méthode ;
- `__doc__` : contient le docstring de la dite fonction ;
- `__name__` : contient le nom de la dite fonction sous forme de chaîne ;
- `__module__` : contient le module dans lequel est définie la méthode.

Observons ces attributs :

```
>>> a.action.__self__
A(1,5)
>>> a.action.__func__
<function A.action at 0x00F94660>
>>> a.action.__doc__
```

```
'Fonction pour agir'
>>> a.action.__name__
'action'
>>> a.action.__module__
'__main__'
```

Les objets de type *method* sont appelables parce qu'ils possèdent une méthode spéciale *call*. Concrètement, lorsqu'on appelle `a.action()`, Python lance `a.action.__call__()` et le lecteur aura deviné que l'appel `a.action.__call__()` retourne `a.action.__func__(__self__)`. De manière plus générale, l'appel `a.action(x,y,z)` retourne `a.action.__call__(x,y,z)`, c'est-à-dire `a.action.__func__(__self__,x,y,z)`.

On peut se demander pourquoi `a.action` ne pointe pas vers l'attribut `action` de `A`. Ceci est dû au fait que `action` possède, comme toute fonction, une méthode spéciale *get*. Cette dernière fait que `a.action` retourne `action.__get__(a,A)`. La méthode *get* est écrite pour que `action.__get__(a,A)` retourne un objet de type *method* lié à `a` grâce par le biais de ses attributs `__self__`, `__func__`, etc. :

```
>>> action = A.__dict__["action"]
>>> action.__get__(a,A)
<bound method A.action of A(1,5)>
>>> a.action
<bound method A.action of A(1,5)>
```

Nous émulons le type *method* au chapitre 30.

5.2 Explorer `classmethod`

On parle de `classmethod` comme s'il s'agissait d'une fonction mais en réalité, il s'agit d'un type :

```
>>> classmethod
<class 'classmethod'>
>>> type(classmethod)
<class 'type'>
```

La « fonction » `classmethod` convertit toute fonction en objet de type `classmethod` :

```
>>> def f(): return
>>> F = classmethod(f)
>>> F
<classmethod object at 0x102c43c18>
>>> type(F)
<class 'classmethod'>
```

Tout objet de type `classmethod` possède un attribut spécial `__func__` :

```
>>> F.__func__
<function f at 0x112dd37b8>
```

Cet attribut contient naturellement la fonction ayant servi à construire la méthode de classe :

```
>>> F.__func__ is f
True
```

Tout objet de type `classmethod` possède une méthode spéciale *get*. Cette méthode est fondamentale. Montrons pourquoi sur un exemple :

```
class A:
    @classmethod
    def mc(cls): return cls
```

Lorsque Python lit « `A.mc` », il remplace par `<mc>.__get__(None,A)`. Autrement dit : c'est la méthode *get* qui explique à Python ce qu'il doit retourner lorsqu'on demande `A.mc`. Nous avons vu à la section 6.2 que `A.mc` ne retourne pas l'attribut `mc` de `A` :

```
>>> A.__dict__["mc"] is A.mc
False
```

L'attribut `mc` de `A` est l'objet de type `classmethod` construit à partir de la fonction `mc`, alors que `A.mc` retourne une méthode liée à `A` (construite à partir de l'objet de type `classmethod`) :

```
>>> A.__dict__["mc"]
<classmethod object at 0x102c515c0>
>>> A.mc
<bound method A.mc of <class '__main__.A'>>
```

Si on recommence tout pour garder une trace de la fonction initiale `mc`, on constate sans surprise que l'attribut `__func__` de la méthode `A.mc` pointe vers la fonction `mc` :

```
def f(cls): return cls
class A:
    mc = classmethod(f)
```

On utilise le dictionnaire de `A` pour accéder à l'attribut `mc` :

```
>>> mc = A.__dict__["mc"]
```

Nous avons maintenant 3 objets : `f` de type *function*, `mc` de type `classmethod` et `A.mc` de type *method*. Les attributs `__func__` de `mc` et `A.mc` pointent vers `f` :

```
>>> mc.__func__ is A.mc.__func__ is f
True
```

L'objet `A.mc` de type *method* possède un attribut `__self__` pointant vers `A` et une méthode `__call__` pour être appelable. La sous-section précédente nous a montré comment fonctionnait l'appel. L'appel de `A.mc` retourne `f(A)`, et nous savons très exactement pourquoi.

Le lecteur sait maintenant pourquoi Python retourne `False` ci-dessous :

```
>>> A.mc is A.mc
False
```

Nous émulons le type `classmethod` au chapitre 30.

5.3 Explorer `staticmethod`

On parle de `staticmethod` comme s'il s'agissait d'une fonction mais en réalité, il s'agit d'un type :

```
>>> staticmethod
<class 'staticmethod'>
>>> type(staticmethod)
<class 'type'>
```

La « fonction » `staticmethod` convertit toute fonction en objet de type `staticmethod` :

```
>>> def f(): return
>>> F = staticmethod(f)
>>> F
<staticmethod object at 0x112cbda58>
>>> type(F)
<class 'staticmethod'>
```

Tout objet de type `classmethod` possède un attribut spécial `__func__` :

```
>>> F.__func__
<function f at 0x112e917b8>
```

Cet attribut contient naturellement la fonction ayant servi à construire la méthode statique :

```
>>> F.__func__ is f
True
```

Tout objet de type `staticmethod` possède une méthode spéciale *get*. Cette méthode est fondamentale. Montrons pourquoi sur un exemple :

```
class A:
    @staticmethod
    def ms(x): return x
```

Lorsque Python lit « `A.ms` », il remplace par `<ms>.__get__(None,A)`. Autrement dit : c'est la méthode *get* qui explique à Python ce qu'il doit retourner lorsqu'on demande `A.ms`. Concrètement, lorsqu'on demande `A.ms`, on n'obtient pas l'attribut `ms` de `A`, mais la fonction ayant servi à construire la méthode statique :

```
>>> A.__dict__["ms"] is A.ms
False
>>> A.__dict__["ms"]
<staticmethod object at 0x10345c0b8>
>>> A.ms
<function A.ms at 0x112e81bf8>
```

Recommençons notre programme pour garder une trace de la fonction initiale `ms` :

```
def f(x): return x
class A:
    ms = staticmethod(f)
```

On utilise le dictionnaire de `A` pour accéder à l'attribut `ms` :

```
>>> ms = A.__dict__["ms"]
```

Si on a bien suivi : `f` est de type *function* et `ms` de type `staticmethod`. L'attribut `__func__` de `ms` pointe vers `f`. Si on appelle `A.ms`, on obtient l'attribut `__fonc__` de `ms`, c'est-à-dire la fonction `f` :

```
>>> A.ms is ms.__func__ is f
True
```

D'une certaine manière, le *get* de `staticmethod` est là pour défaire tout ce que le *get* de *function* fait : si `action` est de type *function*, quand on demande `a.action`, on n'obtient pas l'attribut `action` de `a` mais un enrobage de `action`. Le type `staticmethod` contourne cet enrobage et nous permet d'atteindre la fonction (comme si on avait affaire à un attribut-donnée quelconque). Le lecteur comprendra mieux tout ce mécanisme grâce au chapitre 30.

En attendant, il sait pourquoi Python retourne `True` ci-dessous :

```
>>> A.ms is A.ms
True
```

Nous émulons le type `staticmethod` au chapitre 30.

6 Bilan

Soient `A` une classe, `f` un attribut de `A` et `a` une instance de `A`. Nous retiendrons ceci :

- si `f` est de type *function*, alors `a.f` est une méthode de l'instance `a`. Un appel de `a.f` déclenche un appel de `f` en faisant passer `a` en premier argument ;
- si `f` est de type `classmethod`, alors `A.f` est une méthode de la classe `A`. Un appel de `A.f` ou de `a.f` déclenche un appel de `f` en faisant passer `A` en premier argument ;
- si `f` est de type `staticmethod`, alors `A.f` est une méthode statique. Un appel de `A.f` ou de `a.f` déclenche un appel de `f`.

Lorsque `f` est une `classmethod` ou une `staticmethod`, tout se passe comme si `a.f` et `A.f` pointaient vers le même objet : il est donc possible d'appeler les méthodes de classes et les méthodes statiques depuis une instance (même si cela ne sert à rien).

On mesure ici la simplicité de Python : ce dernier fonctionne sur un petit nombre de règles de base à partir desquelles on déduit tout le reste.

Note. Guido Van Rossum écrit dans [17] qu'il a créé les méthodes de classes pour la méthode spéciale *new*. Finalement, *new* n'a jamais été une méthode de classe, mais une méthode statique. Python a néanmoins conservé les méthodes de classes. On pourra consulter [18] (*The inside story on new-style classes*).