W5-S6-C1-Complement-surcharge-operateurs-1

December 15, 2014

1 Surcharge d'opérateurs (1)

1.1 Complément - niveau intermédiaire

Ce complément vise à illustrer certaines des possibilités de surcharge d'opérateurs, ou plus généralement les mécanismes disponibles pour étendre le langage et donner un sens à des fragments de code comme : * `objet1 + objet2` * `item in objet` * `objet[key]` * `objet.key` * `for i in objet:` * `if objet:` * `objet(arg1, arg2)` (et non pas `classe(arg1,arg2)`) * etc..

que jusqu'ici on n'a expliqué que pour des objets de type prédéfini - sauf pour la boucle `for`, rappelez-vous de la méthode `__iter__()` que l'on a vue à propos des itérables en fin de Semaine 3.

Le mécanisme général pour cela consiste à définir des **méthodes spéciales**, avec un nom en `__nom__`. Comme annoncé dans la vidéo, il existe un total de près de 80 méthodes dans ce système de surcharges, aussi il n'est pas question ici d'être exhaustif. Vous trouverez dans ce document une liste complète de ces possibilités.

Il nous faut également signaler que les mécanismes mis en jeu ici sont **de difficultés assez variables**. Dans le cas le plus simple il suffit de définir une méthode sur la classe pour obtenir le résultat (par exemple, définir `__call__` pour rendre un objet callable). Mais parfois on parle d'un ensemble de méthodes qui doivent être cohérentes, voyez par exemple les descriptors qui mettent en jeu les méthodes `__get__`, `__set__` et `__delete__`, et qui sont particulièrement cryptiques.

Nous vous conseillons de commencer par des choses simples, et surtout de n'utiliser ces techniques que lorsqu'elles apportent vraiment quelque chose. Le constructeur et l'affichage sont pratiquement toujours définis, mais pour tout le reste il convient d'utiliser ces traits avec le plus grand discernement. Dans tous les cas écrivez votre code avec la documentation sous les yeux, c'est plus prudent :)

Nous avons essayé de présenter cette sélection par difficulté croissante. Par ailleurs, et pour alléger la présentation, cet exposé a été coupé en deux notebooks différents.

1.1.1 Rappels

Pour rappel, on a vu dans la vidéo : * la méthode `__init__` pour définir un **constructeur**, * et la méthode `__str__` pour définir comment une instance s'imprime avec `print`.

1.1.2 Affichage : `__repr__` et `__str__`

Nous commençons par signaler la méthode `__repr__` qui est assez voisine de `__str__`, et qui donc doit retourner un objet de type chaîne de caractères, sauf que : * `__str__` est utilisée par `print` (affichage orienté utilisateur du programme; priorité au confort visuel), * alors que `__repr__` est utilisée par la fonction `repr()` (affichage orienté programmeur, aussi peu ambigu que possible); * il faut savoir que `__repr__` est utilisée **aussi** par `print` si `__str__` n'est pas définie.

Pour cette seconde raison, on trouve dans la nature `__repr__` plutôt plus souvent que `__str__`; voyez ce lien pour davantage de détails.

Quand est utilisée `repr()` `repr()` est utilisée massivement dans les informations de debugging comme les traces de pile lorsqu'une exception est levée. Elle est aussi utilisée lorsque vous affichez un objet sans passer par `print`, c'est-à-dire par exemple :

```
In []: class Foo: pass
       foo = Foo()
       # lorsque vous affichez un objet comme ceci
       foo
       # en fait vous utilisez repr()
```

Deux exemples Voici deux exemples simples de classes; dans le premier on n'a défini que `__repr__`, dans le second on a redéfini les deux méthodes :

```
In []: # une classe qui ne définit que __repr__
       class Point (object):
           "première version de Point - on ne définit que __repr__"
           def __init__(self, x, y):
               self.x = x
               self.y = y
           def __repr__(self):
               return "Point({x},{y})".format(**vars(self))

       point = Point (0,100)

       print "avec print", point

       # si vous affichez un objet sans passer par print
       # vous utilisez repr()
       point
```

```
In []: # la même chose mais où on redéfinit __str__ et __repr__
       class Point2 (object):
           "seconde version de Point - on définit __repr__ et __str__"
           def __init__(self, x, y):
               self.x = x
               self.y = y
           def __repr__(self):
               return "Point2({x},{y})".format(**vars(self))
           def __str__(self):
               return "({x},{y})".format(**vars(self))

       point2 = Point2 (0,100)

       print "avec print", point2

       # format utilise aussi __str__
       print "avec format {}".format(point2)

       # si vous affichez un objet sans passer par print
       # vous utilisez repr()
       point2
```

1.1.3 `__nonzero__`

Vous vous souvenez que la condition d'un test dans un `if` peut ne pas retourner un booléen (nous avons vu cela en Semaine 2, Séquence "Introduction aux tests if/else"). Nous avions noté que pour les types prédéfinis, sont considérés comme *faux* `None`, la liste vide, un tuple vide, etc...

Avec `__nonzero__` on peut redéfinir le comportement des objets d'une classe vis-à-vis des conditions - ou si l'on préfère, quel doit être le résultat de `bool(instance)`.

Attention pour éviter les comportements imprévus, comme on est en train de redéfinir le comportement des conditions, il **faut** renvoyer un **booléen** (ou à la rigueur 0 ou 1), on ne peut pas dans ce contexte retourner d'autres types d'objet.

Nous allons **illustrer** cette méthode dans un petit moment avec une nouvelle implémentation de la classe `Matrix2`.

Remarquez enfin qu'en l'absence de méthode `__nonzero__`, on cherche aussi la méthode `__len__` pour déterminer le résultat du test; une instance de longueur nulle est alors considéré comme `False`, en cohérence avec ce qui se passe avec les types *builtin* `list`, `dict`, `tuple`, etc.

Ce genre de *protocole*, qui cherche d'abord une méthode, puis une autre en cas d'absence de la première, est relativement fréquent dans la mécanique de surcharge des opérateurs; c'est entre autres pourquoi la documentation est indispensable lorsqu'on surcharge les opérateurs.

1.1.4 `__add__` et apparentés (`__mul__`, `__sub__`, `__div__`, `__and__`, etc..)

On peut également redéfinir les opérateurs arithmétiques et logiques. Dans l'exemple qui suit nous allons l'illustrer sur l'addition de matrices. On rappelle pour mémoire que :

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} \\ a_{21}+b_{21} & a_{22}+b_{22} \end{pmatrix}$$

1.1.5 Une nouvelle version de la classe `Matrix2`

Voici (encore) une nouvelle implémentation de la classe de matrices 2x2, qui illustre cette fois : * la possibilité d'ajouter deux matrices; * la possibilité de faire un test sur une matrice - le test sera faux si la matrice a tous ses coefficients nuls; * et, bien que ce ne soit pas le sujet immédiat, cette implémentation illustre aussi la possibilité de construire la matrice à partir : * soit des 4 coefficients, comme par exemple : `Matrix2(a, b, c, d)` * soit d'une séquence , comme par exemple : `Matrix2(range(4))`

Cette dernière possibilité va nous permettre de simplifier le code de l'addition, comme on va le voir.

```
In []: # notre classe Matrix2 avec encore une autre implémentation
       class Matrix2 (object):

           def __init__(self, *args):
               """
               le constructeur accepte
               (*) soit les 4 coefficients individuellement
               (*) soit une liste - ou + generalement une sequence - des memes
               """
               # on veut pouvoir créer l'objet à partir des 4 coefficients
               if len(args) == 4:
                   self.coefs = tuple(args)
               # ou bien d'une séquence de 4 coefficients
               elif len(args) == 1:
                   self.coefs = tuple(*args)

           def __repr__(self):
               "l'affichage"
               return "[" + ", ".join(["{}".format(c) for c in self.coefs]) + "]"
```

```
    def __add__(self, other):
        """
        l'addition de deux matrices retourne un nouvel objet
        la possibilite de creer une matrice a partir d'une liste rend ce code
        beaucoup plus facile a ecrire
        """
        return Matrix2 ( [ a+b for a,b in zip(self.coefs, other.coefs) ] )

    def __nonzero__(self):
        """
        on considère que la matrice est non nulle
        si un au moins de ses coefficients est non nul
        """
        # ATTENTION le retour doit être un booléen
        # ou à la rigueur 0 ou 1
        for c in self.coefs:
            if c:
                return True
        return False
```

On peut à présent créer deux objets, les ajouter, et vérifier que la matrice nulle se comporte bien comme attendu :

```
In []: zero     = Matrix2 ([0,0,0,0])

    matrice1 = Matrix2 (1,2,3,4)
    matrice2 = Matrix2 ( range(10,50,10) )

    print 'avant matrice1', matrice1
    print 'avant matrice2', matrice2

    print 'somme', matrice1 + matrice2

    print 'après matrice1', matrice1
    print 'après matrice2', matrice2

    if matrice1:
        print matrice1,"n'est pas nulle"
    if not zero:
        print zero,"est nulle"
```

Voici en vrac quelques commentaires sur cet exemple.

Utiliser un tuple Avant de parler de la surcharge des opérateurs *per se*, vous remarquerez que l'on range les coefficients dans un **tuple**, de façon à ce que notre objet `Matrix2` soit indépendant de l'objet qu'on a utilisé pour le créer (et qui peut être ensuite modifié par l'appelant).

Créer un nouvel objet Vous remarquez que l'addition `__add__` renvoie un **nouvel objet**, au lieu de modifier `self` en place. C'est la bonne façon de procéder tout simplement parce que lorsqu'on écrit

```
print 'somme', matrice1 + matrice2
```

on ne s'attend pas du tout à ce que `matrice1` soit modifié après cet appel.

Du code qui ne dépend que des 4 opérations Le fait d'avoir défini l'addition nous permet par exemple de bénéficier de la fonction *builtin* `sum`. En effet le code de `sum` fait lui-même des additions, il n'y a donc aucune raison de ne pas pouvoir l'exécuter avec en entrée un liste de matrices puisque maintenant on sait les additionner, (mais on a dû toutefois passer à `sum` comme élément neutre `zero`) :

```
In []: # il se trouve que dans l'environnement de FUN
       # le symbole sum correspond à celui de numpy
       # je préfère vous montrer la fonction sum builtin,
       # et donc je vais utiliser l'astuce qu'on a vue
       # en semaine 4 sequence 6
       from __builtin__ import sum

       sum ( [matrice1, matrice2, matrice1] , zero)
```

C'est un effet de bord du typage dynamique. On ne vérifie pas *a priori* que tous les arguments passés à `sum` savent faire une addition; *a contrario*, si ils savent s'additionner on peut exécuter le code de `sum`.

De manière plus générale, si vous écrivez par exemple un morceau de code qui travaille sur les éléments d'un anneau (au sens anneau des entiers $\mathbb{Z}$) - imaginez un code qui factorise des polynômes - vous pouvez espérer utiliser ce code avec n'importe quel anneau, c'est à dire avec une classe qui implémente les 4 opérations (pourvu bien sûr que cet ensemble soit effectivement un anneau).

On peut aussi redéfinir un ordre La place nous manque pour illustrer la possibilité, avec les opérateurs `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, et `__ge__`, de redéfinir un ordre sur les instances d'une classe.

Signalons à cet égard qu'il existe un mécanisme "intelligent" qui permet de définir un ordre à partir d'un sous-ensemble seulement de ces méthodes, l'idée étant que si vous savez faire $>$ et `=`, vous savez sûrement faire tout le reste. Ce mécanisme est documenté ici; il repose sur **un décorateur** (`@total_ordering`), un mécanisme que nous étudierons en semaine 7, mais que vous pouvez utiliser dès à présent.

De manière analogue à `sum` qui fonctionne sur une liste de matrices, si on avait défini un ordre sur les matrices, on aurait pu alors utiliser les *builtin* `min` et `max` pour calculer une borne supérieure ou inférieure dans une séquence de matrices.

1.2 Complément - niveau avancé

Le produit avec un scalaire On implémenterait la multiplication de deux matrices d'une façon identique (quoique plus fastidieuse naturellement).

La multiplication d'une matrice par un scalaire (un réel ou complexe pour fixer les idées), comme ici :

```
matrice2 = reel * matrice1
```

peut être également réalisée par surcharge de l'opérateur `__rmul__`.

Il s'agit d'une astuce, destinée précisément à ce genre de situations, où on veut étendre la classe de l'opérande de **droite**, sachant que dans ce cas précis l'opérande de gauche est un type de base, qu'on ne peut pas étendre (les classes *builtin* sont non mutables, pour garantir la stabilité de l'interpréteur).

Voici donc comment on s'y prendrait. Pour éviter de reproduire tout le code de la classe, on va l'étendre à la volée.

```
In []: # remarquez que les opérandes sont apparemment inversés
       # dans le sens où pour evaluer
       #     reel * matrice
       # on écrit une méthode qui prend en argument
       #   la matrice, puis le réel
       # mais n'oubliez pas qu'on est en fait en train
       # d'écrire une méthode sur la classe 'Matrix2'
       def multiplication_scalaire(self, alpha):
           return Matrix2 ([ alpha*coef for coef in self.coefs ])
```

```
    # on ajoute la méthode spéciale __rmul__
    Matrix2.__rmul__ = multiplication_scalaire
```

In []:
```
matrice1
```

In []:
```
12*matrice1
```