Chapitre 14 : notions sur les classes en PYTHON. Illustration sur les matrices

Table des matières

Introduction

Nous connaissons déjà bien la notion de *type* de variables. Ainsi en PYTHON, `a=1` va fabriquer une variable `a` de type *integer* alors que `a="1"` va fabriquer une variable de type *string* et `a=[1]` une liste. Pour chacun de ces types de variables, on a une opération `+` mais qui ne fait pas la même chose.

On dit que la fonction `+` est « polymorphe » : elle s'applique à des types de variables différentes.

1 Classes et objets

1.1 Essai de définition d'une classe

a) **Première idée sur les classes (très vague) :** « L'ensemble » de toutes les variables de type `integer` va s'appeler la *classe* des `integer`. Un entier particulier par exemple `a=2` sera un *objet* ou une *instance* de la classe.

b) **La limite de la comparaison avec les ensembles :** *ce ne sont pas les objets qui définissent la classe mais la forme et les propriétés communes.* Quand on va programmer, la définition de la classe précède celle des objets, un peu comme la confection d'un moule précède la confection d'un objet.

Autrement dit :

Définir une classe, c'est plutôt définir une forme et des propriétés qu'auront les objets de la classe

c) Les *opérations* qu'on peut faire avec les objets *font partie* de la définition de la classe.

Par exemple, on va définir plus loin une classe `Matrice` dont chaque objet sera une matrice avec un nombre de lignes, de colonnes, et avec des opérations possibles sur ces matrices. Bien sûr, tout ce qu'on va faire va un peu faire double emploi avec par exemple les `np.array` de `numpy`, mais le but est ici de montrer comment faire notre classe à nous, en utilisant seulement le type liste qui est un type de base de Python.

1.2 La notion de classe est plus générale que celle de type

Prenons l'exemple d'une variable `L` en PYTHON intervenant dans le code suivant :

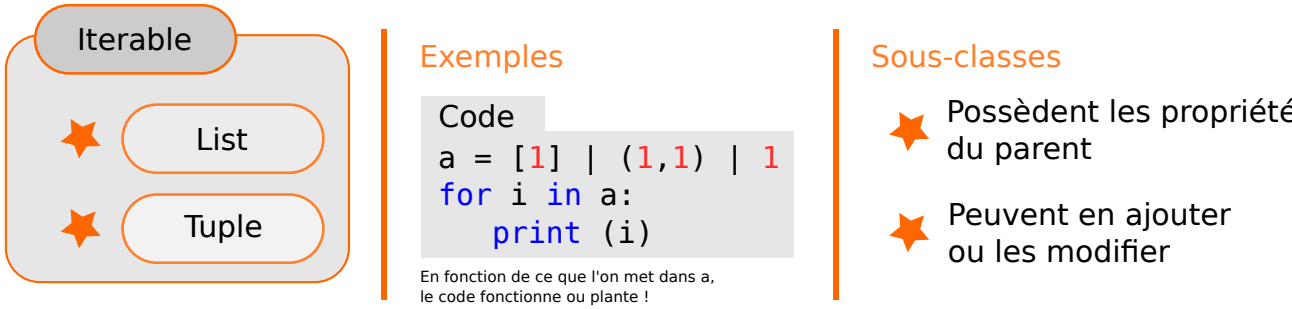
```
for i in L:
    print(i)
```

Pour que le code précédent soit valide, il est nécessaire que l'objet `L` soit *iterable* i.e. supporte une boucle `for`, et que les éléments de `L` soient *affichables* (pour le `print`).

En PYTHON, il existe une classe `Iterable` qui permet de dire que les objets de la classe ont cette propriété, avec une méthode `__iter__` qui retourne un objet qu'on peut parcourir [1].

1.3 Les classes font des enfants

Considérons le schéma suivant :



La classe `Iterable` admet (notamment) les classes `List` et `Tuple` comme classes filles. Le code du milieu fonctionnera pour `a=[1]` ou `a=(1,1)` mais pas pour `a=1`. Le type `Integer` ne possède pas de méthode `__iter__`

Cette possibilité de faire « hériter » une sous-classe de prop. donne beaucoup d'efficacité !

Exemple : pour un jeu vidéo on peut considérer une classe `éléments` avec deux classes filles `personnage` et `décor` et pour la classe `personnage` deux classes filles `personnage géré par l'ordi.` ou `personnage géré par le joueur`.

Toutefois dans ce qui suit nous ne considérerons qu'une classe... par manque de temps !

1. Nous revenons plus loin sur le sens du *double-underscore* autour de `iter`

1.4 Les objets : leurs attributs et leurs valeurs

Pour définir une classe, on doit d'abord définir les *attributs* de ses *objets*

Par exemple, on peut penser à une classe `Taupin` avec *l'objet* `Paul` a un *attribut* `moyenneMath` avec une *valeur* `14`.

L'idée : tous les objets de la classe auront l'attribut `moyenneMath` mais avec des *valeurs* différentes.

1.5 Les fonctions attachées à une classe s'appellent des *méthodes* :

Pour définir une classe, on doit ensuite définir les *méthodes* qui opèrent sur ses *objets*

On connaît déjà bien la syntaxe des *méthodes* pour les listes par exemples : `l.append(1)`.

D'une manière générale, quand on va *définir une classe*, on va, à la suite des premières instructions qui vont modeler les objets de la classe, définir un certain nombres de fonctions qui s'appliqueront aux objets de la classe, et seulement à ces objets [2]. Ce sont les *méthodes* de la classe.

Par exemple avec `help(list)` on voit toutes les méthodes définies pour la classe liste et les *docstrings* qu'on aura écrites comme aide des fonctions apparaîtront dans cette aide.

1.6 Le *self* : une autre façon d'appeler les méthodes

On a rappelé au paragraphe précédent la syntaxe d'appel des méthodes d'une classe. En réalité, il en existe une autre : par exemple pour rajouter à une liste `L` une entrée `12`, on peut au lieu de faire `L.append(12)` faire :

```
list.append(L,12)
```

Il s'agit de la *même fonction* appelée avec les arguments `self, 12` : le premier argument est la liste sur laquelle la méthode va opérer. On met `list` devant le `append` pour dire à Python où la chercher alors qu'avec `L.append` ce n'est pas nécessaire puisque en voyant `L` il sait que c'est une liste.
D'une manière générale : `objet.méthode(arguments)` et `nomclasse.méthode(objet,arguments)` sont des appels équivalents de la fonction `méthode`. Sous la deuxième forme, ce premier argument est appelé `self`.

2 Une construction concrète : l'exemple de la classe `Matrice`

Convention : (*pas forcément toujours respectée*) les noms de classe en Python commencent par une majuscule.

2.1 Pour pouvoir définir des objets : la méthode `__init__`

Voici le début du code de **déclaration** de notre classe `Matrice` avec la première fonction `__init__` expliquée ci-après.

```
class Matrice:
    def __init__ (self, m, n):
        self.lignes = m
        self.colonnes = n
        self.tableau = []  # Initialise notre représentation interne d'une matrice
        # comme un tableau
        for i in range (self.lignes):
            l = []
            for j in range (self.colonnes):
                l.append (None)
            self.tableau.append (l)
```

2. ou aux objets d'une classe fille.

Le `def` est une déclaration de *fonction*. Comme, il est à l'intérieur de la classe, cette fonction est une *méthode* de la classe. Le mot `__init__` est un nom *réservé* de PYTHON. Avant de détailler ce qu'il y a à l'intérieur de la déclaration de notre fonction `__init__`, voyons déjà :

A quoi sert notre méthode `__init__` ? A initialiser i.e. à donner une forme à des objets

Utilisation concrète :

```
a=Matrice(3,3)
type(a)
# <class '__main__.Matrice'>  ## youpi on  a fabriqué un objet de la classe Matrice,
qui fait partie des classes directement accessibles (dans __main__).
```

La méthode `__init__` est « magique » : on ne l'appelle pas par son nom, mais par le nom de la classe. [3]

Retour sur la déclaration `def __init__` : les arguments `self,m,n`

Dans le code de déclaration `def __init__` prend trois arguments `self,m,n`.

Dans l'utilisation, on n'a rentré que deux arguments, qui correspondent à `m,n`. En fait, l'écriture *magique* `a=Matrice(3,3)` est équivalente à `Matrice.__init__(a,3,3)` dans laquelle `self` est l'objet `a` surlequel agit la méthode (ici en l'initialisant) ou encore à `__init__(a,3,3)`.

Davantage sur les attributs : La suite du code

```
self.lignes = m
self.colonnes = n
self.tableau = []
```

définit *trois* attributs pour l'objet `self` créé par cette méthode. L'attribut `lignes`, l'attribut `colonnes` et l'attribut (moins parlant à ce stade) `t`.

La *valeur* donnée à ce attributs est pour les deux premiers respectivement `m` et `n`. Ainsi :

Lors de l'appel de `a=Matrice(3,3)` équivalent à `a=__init__(3,3)` la méthode fabrique *l'objet* `a` avec comme *valeurs* `3` et `3` pour les deux *attributs* `a.lignes` et `a.colonnes`

Suite du code : la déf. de l'attribut `tableau` ici : La ligne `self.tableau=[]` n'était que l'initialisation de la valeur de l'attribut `self.tableau`. Le reste du code donne la vraie nature de l'attribut `self.tableau`. A la fin de ce code `self.tableau` est une liste (de longueur `m`) de listes de longueurs `n` dont toutes les entrées contiennent la variable `None`.

Ici `self.tableau` donne donc le formatage par défaut de la matrice.

Tous les attributs contiennent des valeurs associées à notre matrice, directement accessibles :

```
>>>A=Matrice(3,2)
>>>A.lignes
3
>>> A.tableau
[[None, None], [None, None], [None, None]]
```

3. En fait c'est plus compliqué que cela : l'appel de `Matrice` provoque l'appel d'une méthode qui crée un objet, puis de la méthode `__init__`. En revanche, une fois qu'un objet est créé, `__init__` pourra le modifier

2.2 Excursion : regardons une autre classe et d'autres attributs

```
>>>help(complex)
Help on class complex in module builtins:

class complex(object)
 |  complex(real[, imag]) -> complex number
 |
 |  Create a complex number from a real part and an optional imaginary part.
  |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
.......
------------
 |  Data descriptors defined here:
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  real
 |      the real part of a complex number
```

J'ai sauté la liste des méthodes, mais les `data descriptors` sont les attributs, et donc la syntaxe pour extraire partie réelle et partie imaginaire est :

```
z=complex(15,7) # crée le complexe 15+7i
z.real # oui sans parenthèse après car attributs et pas fonctions....
z.imag
```

2.3 Une deuxième méthode que nous allons mettre dans notre classe

Pour remplir agréablement les matrices

Toujours à la suite (et donc indenté comme le premier `def __init__`) déclarons dans notre classe `Matrice` : (d'où l'indentation !)

```
   def remplir(self, f):
    """ f est une fonction de deux variables, remplir   fabrique la matrice M
    telle que pour tout (i,j), M(i,j) = f(i,j)"""
        for i in range (self.lignes):
            for j in range (self.colonnes):
                self.tableau[i][j] = f (i,j)
```

On pourra utiliser cela pour définir notre matrice de Hilbert du T.P. sur les matrices comme suit :

```
def f(i,j):
    return 1/(i+j+1)
A=Matrice(3,3)
A.remplir(f) # Noter la syntaxe, qui est celle des méthodes.
A.tableau
```

2.4 Des méthodes magiques

On appelle *méthodes magiques* des méthodes qui s'utilisent très agréablement à l'aide par exemple d'un opérateur comme `+,*` ou une autre commande, plutôt que par la syntaxe usuelle d'appel de méthodes. Ces méthodes ont un *nom réservé* en PYTHON, encadré par des __. Ce *nom de méthode* peut être utilisé pour des *classes* très différentes : la méthode sous-jacente faisant des résultats très différents. On va le voir en reprenant l'exemple de l'addition de l'introduction.

Nous avons déjà vu que `__init__` était magique. En voici une autre, avant l'addition :

2.4.1 Méthode magique `__call__` : transforme en fonction

```
def __call__ (self, i, j):
      """ permet d'obtenir l'entrée (i,j) de la matrice A en tapant A(i,j), autrement dit
      informatiquement, en  considérant  la matrice comme fonction de i,j"""
    return self.tableau[i][j]
```

Alors on pourra voir l'entrée `A(2,1)` d'une matrice `A` simplement avec la syntaxe :

```
>>A(2,1)
```

Remarque : ceci doit vous faire comprendre les messages d'erreurs que vous avez souvent rencontrés : par exemple si on fait

```
>>>a=2
>>>a(2)
TypeError: 'int' object is not callable
```

Le message d'erreur `not callable` dit que vous essayer d'appliquer une syntaxe de fonction i.e. une parenthèse, à un objet qui n'a pas de méthode `call` comme ici un entier

2.4.2 Méthode magique `__add__` : permet d'utiliser le +

```
def __add__ (self, autre):
    """ renvoie la somme de la matrice self et de la matrice autre"""
    if isinstance (autre, Matrice):
        if self.lignes == autre.lignes and self.colonnes == autre.colonnes:
            m = Matrice (self.lignes, self.colonnes)
            m.remplir (lambda i,j: self(i,j) + autre(i,j)) # lambda définit une fonction
            return m
        else:
            raise ValueError ("Dimension des matrices incompatibles")
    else:
        raise ValueError("L'objet que vous voulez ajouter n'est pas une matrice")
```

Cette méthode est *magique* car ensuite, si `A` et `B` sont deux objets de notre classe matrice, elle sera appelée si on tape `A+B`.

Détail sur ce qui se passe lorsqu'on tape `A+B` pour PYTHON.
En fait PYTHON exécute `A.__add__(B)`. L'interpréteur PYTHON lit de gauche à droite :

• Il rencontre d'abord l'objet `A` et regarde à quelle classe, il appartient. Pour nous c'est une matrice.

• Il rencontre ensuite la méthode `__add__` : il va chercher si la classe à laquelle appartient l'objet `A` a bien une méthode `__add__`. Pour nous c'est le cas, on vient de définir cette méthode sur les matrices.

• Enfin il lit l'argument `B` qui est la variable `autre` de notre programme ci-dessus.

Dans le programme, il y a deux cas : celui ou `B` est une matrice, et l'autre cas, où le message renvoie une erreur. Même pour une matrice il renvoie une erreur si la taille de `B` n'est pas adaptée.

Exercice : Ecrire de même le code de la méthode magique `__mul__` pour la multiplication qu'on peut appeler ensuite avec le symbole $*$.

2.4.3 Méthode magique `__iter__`

On a évoquée cette méthode au § 1.2 : on a envie de pouvoir faire une boucle `for` en parcourant la matrice, comme on le fait pour une liste en PYTHON, autrement dit, on veut donner du sens à `for a in A` pour `A` une matrice et `a` parcourant les entrées de `a`. Cela se fait ainsi :

```
def __iter__ (self):
    """ Permet de faire des boucles
    for qui parcourent les entrées de la matrice """
    for ligne in self.tableau:
         for entree in ligne:
             yield entree
```

N.B. Ici, il y a une subtilité, le `return` a été remplacé par un `yield` : grossièrement un `yield` renvoie une valeur, mais ne sort pas de la fonction. Plus précisément, le `yield` fabrique un objet appelé *générateur* qui justement est ce qu'on peut utiliser dans une boucle `for`. (Par exemple un `range(1,n)` est un générateur aussi).

2.4.4 Méthode magique `__str__`

La méthode `__str__` est ce qui permet d'utiliser la commande `print`. En ligne de commande, jusqu'à maintenant si on rentre

```
>>>A=Matrice(3,3)
>>>A
<__main__.Matrice object at 0x1712950>
```

on nous dit juste que c'est un objet de la classe Matrice dans le module `__main__`. Et `print(A)` donne la même chose.

On aimerait que le `print` nous donne un joli affichage de `A.tableau`.

Pour cela, on doit, comme annoncé, munir notre classe `Matrice` d'une méthode `__str__`

```
def __str__ (self):
    """ représentation textuelle de la matrice """
    sortie = ""
    # détermination de la longueur de la plus grande entrée de la matrice
    m = 0
    for entree  in self: # utilise la méthode iter pour parcourir la matrice
        s = str (entree) # convertit en chaîne de caract.
        m = max (m, len(s))
   # m contient en fn de boucle  cette longueur maximale
    for Ligne in self.tableau:
        for entree in Ligne :   #
            s = str(entree)
            espaces = " " * (m - len (s)) # va combler la différence d'espaces
            sortie += espaces + s + "  " #
        sortie += "\n" # saut de ligne
    return sortie
```

Et voilà, avec cela, on pourra avoir un joli `print` d'une matrice. Si on trouve que l'affichage compte trop de décimales, on peut remplacer la ligne :

```
s = str(entree) # convertit en chaîne de caract.
```

par

```
s="{:.3f}".format(entree) # convertit en chaîne de caract. et affiche les flottants avec 3 chif
```

éventuellement en testant d'abord si l'entrée est bien un flottant.

2.4.5 Méthode magique __repr__

Si on sait maintenant faire un joli affichage avec `print`, si `A` est une matrice et qu'on tape `A` dans le shell, on aura toujours un truc du genre

```
<__main__.Matrice object at 0x10a03b4a8>
```

En revanche si on crée une méthode `__repr__` on peut choisir ce qui s'affiche dans le shell quand on tape `A` dans le shell pour une matrice `A`. On peut, pour ne pas s'embêter choisir de renvoyer `self.__str__()`. Mais ceci explique pourquoi pour certains objets vous avez des résultats différents entre `A` et `print(A)` (par exemple pour une chaîne de caractères).

2.5 Et les opérations du chapitre sur les matrices alors ?

2.5.1 Voici la réécriture des premières

```
    def echange (self, i, j):
        """ Échange les ligne i et j  : L(i) <-> L(j) attention au pb. de la copie non autonome
        A=self.tableau
        Li=[]
        for p in range (len(A[i])): #
            Li+=[A[i][p]] # on copie entrée par entrée.
    # fin de boucle Li est une copie autonome de A[i] car les entrées sont des nombres sj
        for p in range (len(A[j])):
            A[i][p]=A[j][p]
        for p in range(len(A[i])):
            A[j][p]=Li[p]
```

Bien sûr il serait mieux de gérer directement les problèmes de copies avec une fonction `copy` comme dans `np.copy` ou `deepcopy`

```
    def transvection (self, i, j, mu):
        """ Effectue une transvection :L(i) <- L(i) + mu * L(j)"""
        for k in range (self.colonnes):
            self.tableau[i][k] = self.tableau[i][k] + mu * self.tableau[j][k]
```

2.5.2 Vous pourrez jouer à faire les autres

2.6 Remarque finale sur le polymorphisme illustrée sur les problèmes des copies

Ce qui suit est une incitation à se méfier des copies avec `[:]` : utilisez deepcopy par prudence

Une situation connue

```
L=[1,2,3]  # je crée une liste
C=L[:]     # je fais une copie de la liste (shallow but sufficient)
L[1]=2000 # je parie que C n'est pas modifié
print(C)    # gagné
```

Une vilaine surprise

```
T=np.array([1,2,3]) # je crée un tableau numpy
R=T[:]  # je pense faire une copie du tableau ....
T[1]=2000 # je parie que R n'est pas modifié
print(R)  # bah non, je me suis planté...
```

Horrible non ?

La raison : toujours le polymorphisme. Pour chaque classe, on peut redéfinir comme on veut la valeur d'une fonction ici celle appelée par `[:]`.

La méthode correspondante porte le nom significatif de `__getitem__`. Il se trouve que la méthode a été définie différemment pour les deux classes (comme le `+` etc.)

Une morale pour survivre :

Moralité : faire des `copy.deepcopy` par sécurité.

Un exemple radical pour comprendre

On va définir une classe stupide :

```
class A:
    def __getitem__(self,_):
        return 17

a = A() # l'objet a est crée même sans init : il n' a pas de forme..
print(a[:]) #
print(a[1]) #
```

Un autre exemple pour le même `getitem` avec les matrices

Le code de Maud pour le calcul de Gram-Schmidt au DM 20 repose sur la définition, pratique dans ce contexte, du `getitem` pour les matrices, différentes de celles des `np.array`.

En effet :

```
import numpy as np
A=np.matrix([[1,2,3],[4,5,6]])
L=A[0]
print(L)
```

affiche :

```
[[1 2 3]]
```

3 Le cas des polynômes formels

3.1 Une classe toute faite : dans la doc. de l'oral de Centrale : cf. feuille jointe

3.2 Comment faire notre propre classe polynôme : TP

Instancier les objets polynômes avec comme attribut une liste de coefficients. Ecrire des foncions (méthodes) qui renvoie, le degré, calcule la somme, le produit, la division euclidienne (quotient et reste), le pgcd avec l'algorithme d'Euclide.

C'est votre mission pour le TP 17.

4 Bonus (hors-cours) exemple classe mère/classe fille, attributs privés, méthodes privées

4.1 Classe mère : les points algébriques

4.1.1 Première version de la méthode `__init__`

On va définir un point algébrique par l'entrée d'un couple de coordonnées cartésiennes d'où la méthode `__init__` ci-dessous :

```
class PointAlg:
    def __init__(self, xy=[0,0]):
        self.x = float(xy[0])
        self.y = float(xy[1])
        self.r = None  # r,a seront module et argument à compléter après.
        self.a = None
```

Question : Pourquoi ne pas directement définir plutôt :

```
self.r=m.sqrt(self.x**2 + self.y**2) # le m. pour le module math
```

Essayons : en regardant ce qui se passe si on modifie un attribut `P.x` d'un point `P` ?
On comprend la nécessité de fonctions qui vont mettre à jour les attributs.

4.1.2 Seconde version du `__init__` avec attributs privés

On va empêcher l'utilisateur d'avoir directement accès aux attributs, en les déclarant comme *attributs privés*

```
class PointAlg:
    def __init__(self, xy=[0,0]):
        self.__x = float(xy[0])  # avec un double underscore devant l'utilisateur n'a pas accès
        self.__y = float(xy[1])
        self.__r = None  # r,a seront module et argument à compléter après.
        self.__a = None
```

Cette fois l'utilisateur n'a pas accès aux attributs : essayez !
A quoi bon avoir des attributs si on ne peut pas y accéder ? On définit des méthodes d'accès en lecture :

```
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y
    def getXY(self):
        return self.__x,self.__y
    def getRho(self):
        return self.__r
    def getTheta(self):
         return m.degrees(self.__a)
```

et des méthodes d'accès en écriture :

```
    def setX(self, x):
        self.__x = x
        self.__majRA()
    def setY(self, y):
        self.__y = y
        self.__majRA()
    def setTheta(self, angleDegre):
        self.__a = m.radians(angleDegre)
            # Attributs liés
        self.__majXY()
    def setRho(self, rho_mm):
```

```
        self.__r = rho_mm
        # Attributs liés
        self.__majXY()
```

Quelle différences notez vous ?
Il faut bien sûr définir les méthodes (privés i.e. l'utilisateur n'y a pas accès) `__majRA` et `__majXY`.

```
 def __majXY(self):
     self.__x=self.__r*m.cos(self.__a)
     self.__y=self.__r*m.sin(self.__a)
 def __majRA(self):
    self.__r = m.sqrt(self.__x**2 + self.__y**2)
    self.__a = m.atan2(self.__y,self.__x) # en rad
```

Que manque-t-il encore dans la définition de la méthode `__init__` ?

4.2 Classe fille : les points géométriques

Dans le script qui suit, on définit une classe `PointGraphique(PointAlg)` ce qui signifie une classe `PointGraphique` fille de la classe `PointAlg`. On la munit d'une méthode `__init__` qui lui est propre : ce n'est pas forcément nécessaire : parfois la classe fille peut simplement hériter de la méthode `__init__` de la classe mère et être simplement pourvue d'autre méthode.

Ici donc avec ce nouveau `__init__`, les objets `PointGraphique` auront tous les attributs des objets `PointAlg` (grâce à la ligne `PointAlg.__init__(self,coord)`), mais auront deux autres attributs.

```
import matplotlib.pyplot as plt
class PointGraphique(PointAlg):
    def __init__(self,coord,couleur,forme):
        PointAlg.__init__(self,coord)# création d'un point alg
        self.couleur=couleur
        self.forme=forme
     # le point graphique hérite des attributs et des méthodes
     # du point algébrique avec  des attributs supplémentaires..
    def affiche(self):
        x=self.getX()
        y=self.getY()
        plt.plot(x,y,color=self.couleur,marker=self.forme)
        plt.show()
```