Cours_Python_SRI_2_objets

September 25, 2020

1 Programmation objet en python

1.1 Abstraction & Encapsulation

On a vu l'utilité des structures de données "natives": listes, dictionnaires, ensembles, et qu'elles peuvent servir à définir des types de données différents, spécifiques, comme dans l'exemple des polynômes et des vecteurs creux.

Pour ces derniers on a défini un ensemble de fonctions et procédures, certaines basiques, d'autres plus élaborées et utilisant les basiques.

D'une façon générale, un type de données peut toujours être ramené à un ensemble d'opérations de base, ou "primitives", sur lesquelles on peut développer l'ensemble du code utile.

Prenons l'exemple des polynômes, sur lesquels on peut définir :

- la valeur en un point x_0
- l'addition, la multiplication
- la dérivation

Un polynôme peut être défini par la liste des coefficients de ses puissances, de 0 jusqu'à l'exposant non nul maximum

$1 + x^2 + 3x^5 \rightarrow$ [1,0,1,0,0,3]

On a vu aussi avec des vecteurs creux qu'on aurait tout aussi bien pu définir ce type avec un dictionnaire répertoriant les coefficients non nuls, ici {0:1, 2:1, 5:3}

Sur le long terme, on ne veut pas que le programme développé sur ces bases soit trop dépendants des choix d'implémentation de départ, tout ce que l'on veut est d'utiliser les opérations qui ont du sens pour les données en question. C'est le sens de la programmation objet: "encapsuler" au maximum le code.

Que faut-il pour définir un type suivant ces principes ?

- pouvoir initialiser une variable de ce type
- avoir accès à son contenu (coefficients), pouvoir les modifier au besoin
- un ensemble d'opérations de base

Reprenons maintenant l'exemple des vecteurs creux, il faut

- initialiser un vecteur avec un ensemble de valeurs
- accéder aux valeurs, les modifier

- opérations vectorielles: produit par un scalaire, addition

Première version:

[2]:
```
# initialisation à partir d'un vecteur "complet"
def faire_creux(plein):
    v = {}
    for (i,x) in enumerate(plein):
        if x!=0.:
            v[i]=x
    return v
```

[3]:
```
l=[0,10,0,0,0,0,0,-2.5,0,0.,0.]
v = faire_creux(l)
print(v)
```

```
{1: 10, 7: -2.5}
```

[4]:
```
# modifier des valeurs
v[1] = 2
v[2] = -6.3
```

[5]:
```
# accéder aux valeurs
print(v[2])
print(v[5])
```

```
-6.3

---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-5-49fbca277312> in <module>()
  1 # accéder aux valeurs
  2 print(v[2])
----> 3 print(v[5])


KeyError: 5
```

[6]:
```
v.get(5,0)
```

[6]: 0

Pas très pratique avec un dictionnaire : on aimerait bien ne pas avoir à s'occuper de tester les valeurs à chaque fois qu'on veut accéder à une coordonnée.

Continuons avec les autres opérations : addition, multiplication terme à terme, produit vectoriel, ou par un scalaire.

[7]:
```
def add(v1,v2):
    v = {}
    for a in (set(v1.keys())| set(v2.keys())):
        v[a] = v1.get(a,0) + v2.get(a,0)
        if v[a]==0:
            del v[a]
    return v
```

Encore une fois on doit s'occuper de gérer l'absence possible du coefficient.

[8]:
```
def mult(v1,v2):
    v = {}
    for a in (set(v1.keys()) & set(v2.keys())):
        v[a] = v1[a]*v2[a]
    return v

def prod_scalaire(v1,v2):
    res = 0.
    for a in (set(v1.keys()) & set(v2.keys())):
        res = res + v1[a]*v2[a]
    return res

def scalaire(k,v):
    res = {}
    if k!=0.:
        for x in v:
            res[x] = v[x]*k
    return res
```

On passe à : soustraction, norme, que l'on peut définir à partir des opérations précédentes:

[9]:
```
from math import sqrt

def soustraction(v1,v2):
    return add(v1,scalaire(-1.,v2))

def norme(v):
    return sqrt(prod_scalaire(v,v))
```

[10]:
```
norme(v)
```

[10]:
```
7.066823897621901
```

[11]:
```
soustraction(v,v)
```

[11]:
```
{}
```

Encore quelques fonctions: distance euclidienne entre 2 vecteurs, distance de hamming (nombre de coordonnées différentes)

[12]:
```
def dist_euclide(v1,v2):
    return norme(soustraction(v1,v2))

def hamming(v1,v2):
    v = soustraction(v1,v2)
    return len(v.keys())
```

Dommage de devoir savoir l'accès aux coefficients non nuls ... ajoutons une fonction

[13]:
```
def non_zeros(v):
    return (v.keys())

def hamming(v1,v2):
    v = soustraction(v1,v2)
    return len(non_zeros(v))
```

[14]:
```
hamming(v,v)
```

[14]:
```
0
```

[15]:
```
hamming(v,scalaire(2,v))
```

[15]:
```
3
```

[16]:
```
non_zeros(v)
```

[16]:
```
3
```

Basique	Non basique
faire_creux	norme
non_zeros	soustraction
get	distance euclidienne
addition	distance de hamming
multiplication	..
scalaire	..
prod_vect	..

Conclusion: si on veut changer quelque chose, on ne touche qu'à la première colonne.

- plus facile de mettre à jour le code, voir de changer complètement les choix de base de l'implémentation
- plus facile d'ajouter de nouvelles fonctions: on n'utilise que les fonctions de base, censées

être plus intuitives
- plus élégant et facile à lire

Peut-on faire mieux ?

- get dépend du fait qu'on a un dictionnaire
- addition et multiplication sont déjà des opérations complexes

On ajoute:

- une fonction pour récupérer la valeur d'un coeff, nul ou non, indépendant du type dictionnaire
- non_zeros peut servir à récupérer les coeffs non nuls

Attention quand même: dans certains cas, les fonctions peuvent être plus efficaces en ayant accès explicitement à la structure, il y a donc un compromis à trouver (pas seulement le nombre de fonctions de base donc).

[17]:
```
def coeff(v,i):
    return v.get(i,0.)

def non_zeros(v):
    return v.keys()

def nb_nonzeros(v):
    return len(non_zeros(v))

def add(v1,v2):
    v = {}
    for a in (set(non_zeros(v1)) | set(non_zeros(v2))):
        v[a] = coeff(v1,a) + coeff(v2,a)
        if v[a]==0:
            del v[a]
    return v
```

Toujours dépendant du dictionnaire … à cause de

- test de non nullité
- modif d'un coeff du résultat
- initialisation du vecteur résultat

[18]:
```
def change(vect,indice,valeur):
    if valeur==0:
        if vect.has_key(indice):
            del vect[indice]
    else:
        vect[indice]=valeur
```

Et on a bien maintenant une addition "abstraite" définie sur les primitives :

[19]:
```
def addition(v1,v2):
    v = faire_creux([])
    for a in (set(non_zeros(v1)) | set(non_zeros(v2))):
        change(v,a,(coeff(v1,a) + coeff(v2,a)))
    return v
```

Basique	Non basique
faire_creux	norme
non_zeros	soustraction
change	distance euclidienne
coeff	distance de hamming
	addition

Peut-on faire de même avec ? (exercice)

- scalaire

- multiplication
- prod_vect

On s'est rapproché de l'approche objet, avec essentiellement des types de méthodes suivantes:

- des constructeurs pour créer de nouvelles données (ici faire_creux)
- des accesseurs pour obtenir les valeurs des données (coeff, non_zeros)
- des manipulateurs pour changer les données (change)

Parfois on ajoute aussi la notion de **destructeur** pour "faire le ménage" (par nécessaire en python)

Dans tous les cas, en remontant dans l'abstraction on facilite l'extension et l'utilisation du code.

Reprenons l'exemple des compteurs d'éléments d'une liste : comment structurer de façon plus abstraite ? Par exemple on peut vouloir faire des comptes séparés que l'on fusionne a posteriori selon différentes options.

[21]:
```
def creer(l):
    res = {}
    for x in l:
        res[x] = res.get(x,0)+1
    return res

l = [1,3,4,2,1,2,5,6,9,4,1]
c =creer(l)

def get(c,v):
    return c.get(v,0)

def tolist(c):
    l= []
```

```
    for x in c:
        l.extend([x]*c[x])
    return l

print(tolist(c))


def top(c,n=1):
    l = []
    items = [(nb,cle) for (cle,nb) in c.items()]
    items.sort()
    items = items[-n:]
    return [(cle,nb) for (nb,cle) in items]

print(top(c,n=3))

def add0(c1,c2):
    return creer(tolist(c1)+tolist(c2))

def add(c1,c2):
    res = {}
    for x in c1:
        res[x] = c1[x] + c2.get(x,0)
    for x in c2:
        if x not in c1:
            res[x] = c2[x]
    return res

def add_bis(c1,c2):
    res = {}
    for x in set(c1.keys()) | set(c2.keys()):
        res[x] = c1.get(x,0) + c2.get(x,0)
    return res


def sub(c1,c2):
    res = {}
    for x in c1:
        res[x] = max(0,c1[x] - c2.get(x,0))
    return res


def inter(c1,c2):
    res = {}
    for x in set(c1.keys()) | set(c2.keys()):
        res[x] = min(c1.get(x,0), c2.get(x,0))
    return res
```

```python
def union(c1,c2):
    res = {}
    for x in set(c1.keys()) | set(c2.keys()):
        res[x] = max(c1.get(x,0),c2.get(x,0))
    return res



l = ["a","b","a","c","b","a","d","e","f","e","a"]
c =creer(l)
c2 = add(c,c)
print(c2)
c['z'] = 3
print(c)
print(sub(c2,c))
```

```
[1, 1, 1, 2, 2, 3, 4, 4, 5, 6, 9]
[(2, 2), (4, 2), (1, 3)]
{'b': 4, 'c': 2, 'e': 4, 'f': 2, 'd': 2, 'a': 8}
{'b': 2, 'c': 1, 'e': 2, 'f': 1, 'd': 1, 'z': 3, 'a': 4}
{'b': 2, 'c': 1, 'e': 2, 'f': 1, 'd': 1, 'a': 4}
```

1.2 Types de données hétérogènes

Les objets servent aussi à manipuler des données qui rassemblent des éléments de nature différentes, ce pour quoi on pourrait utiliser des tuples, comme par exemple des dates:

```
d=(3,12,2014)
```

Si on a beaucoup de variables de ce type, on se retrouve souvent à accéder aux éléments avec des indices, et il faut alors se souvenir où est quoi, en l'absence de conventions évidentes. Vaut-il mieux ordonner en jour,mois,année, ou en année, mois, jour, pour pouvoir faire des comparaisons ?

```
d[0] # le jour ? le mois ?
```

Une solution simple est d'utiliser un dictionnaire pour cela:

```
date = {"jour":3, "mois":12, "année":2014}
```

C'est pratique, mais on ne contrôle pas beaucoup ce que peut contenir une variable date, en particulier pour les champs jour et mois. On peut aussi vouloir permettre plusieurs façons de rentrer les données du mois ("décembre" / 12), et fournir des conversions implicites.

1.3 La notion d'objet

Pour aller vers plus de contrôle, la **programmation orientée objet** pousse les notions précédentes jusqu'à définir explicitement des données sur lesquelles sont définies des méthodes qui leur sont

propres, et qui protègent l'accès au contenu de ces données.

Un problème manifeste avec les fonctions créés sur les données vues plus haut, par exemple les vecteurs creux, est que l'on arrive vite à retomber sur les mêmes noms pour des opérations qui portent sur des types différents (par exemple add). On pourrait ajouter systématiquement au nom des fonctions le nom du type, mais c'est loin d'être pratique. Les méthodes objets prennent également ceci en compte.

Reprenons l'exemple des dates, en python on définit des **classes** d'objets comme suit:

[3]:
```
class Date:
    pass
    # etc ...
```

1.3.1 Constructeur et attributs

Que faut-il pour définir une classe ? On a vu qu'on se reposait souvent sur des constructeurs, accesseurs et manipulateurs.

En python, seul le constructeur est obligatoire, et se définit avec le nom réservé __init__

Le contenu des données est initialisé par ce constructeur dans ce qu'on appelle des attributs.

Il n'y a pas besoin de destructeurs explicites, les objets sont libérés en l'absence de référence active pointant sur eux. ("garbage collection")

[4]:
```
class Date:

    def __init__(self,jour,mois,annee):
        if jour>=1 and jour<=31:
            self.jour = jour
        else:
            self.jour = None
        if mois>=1 and mois<=12:
            self.mois = mois
        else:
            self.mois = None
        if annee!=0:
            self.annee = annee
        else:
            self.annee = None
```

[25]:
```
#utilisation du constructeur
d = Date(4,4,2014)
print(d)

#utilisation des attributs
print(d.annee)
```

```
print(d.jour)

d2 = Date(56,1,2014)
print(d2.annee)
print(d2.jour)
```

```
<__main__.Date object at 0x7f6ed55c1240>
2014
4
2014
None
```

A noter:

- les opérations sur les objets sont des **méthodes**, on les appelle avec la syntaxe classique objet.methode(parametres)
- pour définir une méthode on la définit à l'intérieur d'une classe, et on doit toujours ajouter comme paramètre l'objet qui l'appelle (par convention on le note souvent "self")
- __init__ est une méthode particulière: elle renvoie l'objet créé de la classe, et donc ne doit pas être appelé sur un objet existant. Par ailleurs, à l'appel, elle prend le nom de la classe.
- à part cela, les méthodes sont définies comme des fonctions et peuvent donc avoir des arguments optionnels. Par exemple on aurait pu définir le constructeur comme suit: def **init**(self,jour=None,mois=None,annee=None): etc . . .

  et l'appeler : d= Date(jour=1,mois=4)

On peut affecter et changer les attributs ouvertement, mais si on veut "protéger" l'information pour éviter les valeurs incorrectes, il vaut mieux définir explicitement des accesseurs et manipulateurs (on verra plus tard une meilleure façon de faire la même chose en définissant des attributs publics, appelés "property") :

[5]:
```
class Date:

    def __init__(self,jour,mois,annee):
        if jour>=1 and jour<=31:
            self.jour = jour
        else:
            self.jour = None
        if mois>=1 and mois<=12:
            self.mois = mois
        else:
            self.mois = None
        if annee!=0:
            self.annee = annee
        else:
            self.annee = None

    def set_jour(self,new):
```

```
        if new>=1 and new<=31:
            self.jour = new
            return True
        else:
            self.jour = None
            return False

    def get_jour(self):
        return self.jour

#######

d = Date(1,1,2014)
d.set_jour(3)
print(d.get_jour())
d.set_jour(-6)
print(d.get_jour())
print(d)
```

```
3
None
<__main__.Date object at 0x1046cae10>
```

Exercice: définir un type durée, que l'on affiche en heures/minutes/secondes et sur lequel on peut faire des additions et soustractions.

[4]:
```
a = Duree(3000)
print a.convertir()
```

```
(0, 50, 0)
```

1.3.2 Autres méthodes courantes

De même que __init__ est une méthode spéciale, on peut redéfinir certaines méthodes suivantes pour avoir des effets particuliers :

- __repr__(self) définit la chaine qui s'affiche quand on fait un 'print'
- __getitem__ peut définir des accès avec des indices si besoin est, en lecture (... = x[0])
- __setitem__ peut définir des accès avec des indices si besoin est, en écriture (x[0]= ...)
- __contains__ peut redéfinir l'opérateur in

Les opérateurs mathématiques peuvent être **surchargés** pour prendre comme arguments des objets de la classe considérée, en redéfinissant les méthodes suivantes:

- __add__(self, other) : pour utiliser +
- __sub__(self, other) : -
- __mul__(self, other) : *

Enfin il faut noter que l'on documente les classes comme des fonctions.

1.3.3 Exercices:

- Reprendre les vecteurs creux pour en faire un bel objet.
- définir la classe, les attributs et le constructeur
- redéfinir l'addition, la multiplication etc en surchargeant des opérateurs mathématiques +, *, etc
- Définir la classe des nombres complexes
- Définir la classe des nombres rationnels

[1]:
```
class Complexe:

    def __init__(self,r,i):
        self.r = r
        self.i = i

    def __add__(self,other):
        r = self.r + other.r
        i = self.i + other.i
        return Complexe(r,i)

    def __mul__(self,other):
        return Complexe(self.r*other.r-self.i*other.i,
                        self.i*other.r + self.r*other.i)

    def __sub__(self,other):
        return self + other*Complexe(-1,0)

    def __repr__(self):
        return "%s + (%s)i"%(self.r,self.i)
```

[18]:
```
c1 = Complexe(2.5,-3)
c2 = Complexe(3,0)
print(c1 + c2)
print(c1*c2)
print(c2-c1)
```

```
5.5 + (-3)i
7.5 + (-9.0)i
0.5 + (3.0)i
```

[5]:
```
(1+2j)+(-2-2j)
```

[5]:
```
(-1+0j)
```

1.4 Héritage

Une caractéristique essentielle de la notion d'objet est de pouvoir définir des objets de plus en plus spécifiques, qui reprennent ce qui leur est commun avec des objets plus généraux, en spécifiant juste leur différence.

Prenons l'exemple classique de la représentation d'un rectangle, que l'on peut spécifier en carré. Le lien d'héritage en python se définit de façon classique, par surcharge des méthodes.

[29]:
```
class Rectangle:
    def __init__(self,largeur,longueur):
        self.large = largeur
        self.long = longueur

    def aire(self):
        return self.large*self.long

    def get_largeur(self):
        return self.large

    def get_longueur(self):
        return self.long

    def __repr__(self):
        return str(self.long)+"x"+str(self.large)
```

[31]:
```
class Carre(Rectangle):
    def __init__(self,cote):
        Rectangle.__init__(self,cote,cote)
        self.blabla = 0

    def get_cote(self):
        return self.get_longueur()

r = Rectangle(10,30)
c = Carre(10)
print(r)
print(c)
print(c.get_cote())
```

```
30x10
10x10
10
```

Le type Carre **hérite** de rectangle, et donc toutes les méthodes de Rectangle sont aussi des méthodes de Carre, sauf celles que l'on redéfinit explicitement. On peut quand même appeler les méthodes de la super-classe dans la sous-classe (cf init dans l'exemple)

1.5 Résumé

les concepts à retenir :

- classe
- objet, instance
- attribut
- méthode
- héritage
- composition
- surcharge

1.6 Retour sur la gestion de la mémoire et des références

On a déjà mentionné que les variables de type structurées sont des références (pointeurs). Il faut donc être très prudent avec les affectations de ces variables, et cela concerne bien évidemment les objets et les attributs des objets.

[20]:
```
l = [1,2,3]
a = l
b = l
a[0] = -1
print(a, b, l)
```

```
[-1, 2, 3] [-1, 2, 3] [-1, 2, 3]
```

[21]:
```
a is b
```

[21]:
```
True
```

[7]:
```
l = [1,2]

class Test:
    def __init__(self,data):
        self.data = data

a = Test(l)
b = Test(l)
print(a is b)
print(a.data is b.data)
a == b
```

```
False
True
```

[7]:
```
False
```

[25]:
```
b = Test(3)
c = b
c is b
```

[25]:
```
True
```

Comment faire pour avoir des objets distincts ? Il faut faire des “copies” :

[38]:
```
import copy
c = copy.copy(b)
c is b
```

[38]:
```
False
```

Mais cette copie est “superficielle”, elle ne fait que recopier les attributs. Si l’attribut est lui-même un pointeur sur un objet structuré, il n’est pas distinct :

[39]:
```
c = copy.copy(a)
c is a, c.data is a.data
```

[39]:
```
(False, True)
```

Il faut alors utiliser une “copie profonde” :

[40]:
```
c = copy.deepcopy(a)
c is a, c.data is a.data
```

[40]:
```
(False, False)
```

Inconvénients :

- plus lent
- évidemment, duplique la place en mémoire

Comment sont gérées ces références, et la mémoire occupée ?

- allocation dynamique de la mémoire nécessaire
- comptage des pointeurs actifs sur une zone mémoire allouée
- “libération” quand le comptage revient à 0
- le “garbage collector”, comme en Java, fait le ménage périodiquement.

Pièges à éviter:

- références circulaires
- l’allocation de chaines peut facilement manger beaucoup de mémoire

[41]:
```
# exemple, avec accumulation de sous-chaines dans une chaine résultat
a = "abc" # 1 allocation
a = a + "def" # + 2 allocations
a = a + "ghi" # + 3 allocations ...
#... o(n^2) allocations
```

```
#mieux : o(n) allocations
a = "".join(["abc","def","ghi"])
```

1.7 Exercices

- retour sur le TD sur les compteurs: comment aurait-on pu définir cette classe ? (définir les méthodes suivantes: addition de compteurs, mise à jour d'un compteur par un autre)
- définir une classe polygone, représenté comme une liste de points en 2D. définir la méthode périmètre et définir la sous-classe rectangle
- faire un type tas : arbre binaire par couches, représenté dans une liste.