# Informatique en CPGE (2015-2016)
# TD 10 : notion de classe

**Exercice 1**

Nous allons définir une **classe** d'**objets**, qui permet de regrouper dans une même entité des données (les **attributs**) et des fonctions (les **méthodes**). On parle d'encapsulation. Un objet est un élément de la classe, ou une **instance** de la classe.

Nous commençons par un exemple simple en définissant une classe **Vecteur()**. Dans le plan rapporté à un repère orthonormé un vecteur est défini par ses deux coordonnées $x$ et $y$.

```
class Vecteur:
    "Définition d'un vecteur dans le plan"
```

La chaîne de caractère est un commentaire sur la classe qui s'affiche par exemple avec l'instruction `help(Vecteur)` ou l'instruction `print(v1.__doc__)` si v1 est une instance de la classe.

Nous créons maintenant un objet de cette classe :

```
v1=Vecteur()
```

L'instruction `type(v1)` renvoie `<class '__main__.Vecteur'>`.

Nous pouvons compléter le code définissant la classe avec des méthodes :

```
class Vecteur:
    "Définition d'un vecteur dans le plan"
    def __init__(self):
        self.x=0
        self.y=0
    def norme(self):
        return (self.x**2+self.y**2)**0.5
    def affiche(self):
        print("("+str(self.x)+";"+str(self.y)+")")
```

La méthode __init__ (avec deux tirets de soulignement de chaque côté de init) est un **constructeur**. Ici, un vecteur aura par défaut les coordonnées (0 ; 0). Cette méthode est exécutée automatiquement lorsqu'on crée un nouvel objet de la classe.

```
v1=Vecteur()
v1.x,v1.y=3,4

v2=Vecteur()
v2.x,v2.y=5,9

v3=Vecteur()
v3.x,v3.y=3,4

# v1 et v3 sont deux objets différents
# v1==v3 affiche False

v3.affiche()
# affiche (3;4)
```

Nous améliorons un peu le constructeur et compétons avec quelques méthodes :

```
class Vecteur:
    "Définition d'un vecteur dans le plan"
    def __init__(self,abscisse=0,ordonnee=0):
        self.x=abscisse
        self.y=ordonnee
    def norme(self):
        return (self.x**2+self.y**2)**0.5
    def affiche(self):
        print("("+str(self.x)+";"+str(self.y)+")")
    def ajoute(self,v2):
        return Vecteur(self.x+v2.x,self.y+v2.y)
    def __add__(self,v):
        # surcharge de l'opérateur +
        return Vecteur(self.x+v.x,self.y+v.y)
    def __sub__(self,v):
        # surcharge de l'opérateur -
        return Vecteur(self.x-v.x,self.y-v.y)

v1=Vecteur(3,4)

v2=Vecteur(5,8)

# instructions à tester (comparer les résultats)
v3=v1.ajoute(v2)
v3=Vecteur.ajoute(v1,v2)
v3=v1+v2
v3=v1-v2
```

**Exercice 2**

L'objectif est de définir une classe **Pile** constituée par :

**les attributs :**

- la taille : un entier (nombre d'éléments entrés)
- la pile : un tableau de longueur taille (de type list)

**les méthodes :**

- procédure "init() " : le constructeur (crée une liste vide et met la taille à 0) ;
- procédure "empiler(element)" : entre un élément sur la pile et incrémente la taille ;
- fonction "depiler()" : retourne le sommet de la pile, le supprime de la pile et décrémente la taille ;
- fonction "pile_vide()" : retourne un booléen "la pile est-elle vide ?" ;
- fonction "sommet()" retourne l'élément au sommet de la pile sans dépiler ;
- des méthodes d'affichage.

Recopier et compléter le code suivant :

```
class Pile:
    def __init__(self):
        ...
        ...

    def empiler(self,x):
        ...
        ...
```

```
    def depiler(self):
        ...
        ...
        ...

    def pile_vide(self):
        ...
        ...
        ...
        ...

    def sommet(self):
        ...
        ...

    def affiche(self):
        ...

    def __str__(self):
        return str(self.vals)
```

Tester les commandes :

```
p=Pile()
p.empiler(3)
p.empiler(7)
p.empiler(4)
p.affiche()
print(p.taille)
p.depiler()
print(str(p))
print(p.taille)
print(type(p))
```

**Exercice 3**

Nous reprenons le problème des "tours de Hanoï" (exercice 4 du TD 1).

Si A, B et C sont les trois tours et $x_n$ le nombre de déplacements de disques nécessaires au déplacement d'une tour complète de A vers C, alors pour déplacer une tour de $n$ disques de A vers C, on effectue ces trois étapes :

- déplacer la tour des $n-1$ premiers disques de A vers B (soit $x_{n-1}$ déplacements) ;
- déplacer le plus grand disque de A vers C (un déplacement) ;
- déplacer la tour des $n-1$ premiers disques de B vers C ($x_{n-1}$ déplacements).

Au total cela fait $2x_{n-1}+1$ déplacements.

La relation de récurrence :

$$x_0 = 0 \text{ et } x_n = 2x_{n-1}+1 \text{ si } n \geq 1$$

donne après calculs $x_n = 2^n - 1$.

Il est impossible de faire mieux car, pour déplacer la tour de n disques de A vers C, on doit forcément déplacer le plus grand disque de A vers C, et pour ce faire, on doit avoir empilé les n $-1$ premiers disques en B.

Avec 32 disques, le minimum de déplacements est $2^{32} - 1 = 4294967295$. S'il faut une seconde pour déplacer un disque (à la main), il faudra environ 136 années pour finir le jeu ; et avec 60 disques, il faudrait environ trois fois l'âge de l'univers (13,7 milliards d'années).

Pour 500 000 déplacements à la seconde sur un ordinateur il faudra plus de deux heures avec 32 disques et plus de 73000 années avec 60 disques.

1. Ecrire une procédure récursive **hanoi** qui utilise la méthode décrite ci-dessus.

   La procédure **hanoi** prendra quatre paramètres : **n** le nombre de disques utilisés, **d** l'emplacement de départ, **a** l'emplacement d'arrivée, **i** l'emplacement intermédiaire.

   Les trois piles seront des instances de la classe Pile présentée plus haut.
2. Ecrire une procédure **joueHanoi** qui prend en paramètre un entier n, (le nombre de disques), crée trois piles, p1 la pile initiale contenant les n disques, p2 et p3, et affiche les trois piles finales en utilisant la procédure **hanoi**.
3. Compléter le programme avec un compteur afin d'afficher le nombre de déplacements effectués.