# TD N°2: Programmation Orientée Objet

Classe, Instances, Atributs, Méthodes, Encapsulation
Héritage, Composition, Polymorphisme
Object, Méthodes spéciales

## 1 Classe, Objets, instanciation, Attributs et Méthodes

**Exercice 1**

1. Créer une *classe* Python nommée **Point**, définie par deux attributs d'instance, $x$ et $y$, représentant les coordonnées d'un point.
2. Créez deux *instances* **p1** et **p2** de la classe **Point** avec les mêmes coordonnées, $x = 1$ et $y = 2$.
3. Essayez les opérations suivantes :
   - `print(isinstance(p1, Point)); print(type(p2) == Point)` : Vérifiez si **p1** et **p2** sont des instances de la classe **Point**.
   - Testez l'égalité des objets **p1** et **p2**.
   - `p1 = p2; p1.x = 7.0; print(p2.x)`
4. Créer une classe Python nommée **Rectangle**, définie par :
   - **coinSupG** : un objet de type **Point** représentant la position du coin supérieur gauche du rectangle.
   - **largeur** : la largeur du rectangle.
   - **hauteur** : la hauteur du rectangle.
5. Créer une instance nommée **boite** de la classe **Rectangle**.
6. Définir une fonction Python **trouver_centre** qui prend en argument un objet de type **Rectangle** et renvoie un objet de type **Point** représentant les coordonnées du centre du rectangle.
7. Appelez cette fonction en utilisant comme argument l'objet **boite** défini précédemment.
8. Affichez les coordonnées du centre trouvé.

**Exercice 2**

Créer et manipuler une classe en Python pour gérer une liste de tâches :

1. **Définir une classe** `Tache` avec les attributs suivants :
   - `titre` : le titre de la tâche.
   - `description` : une brève description de la tâche.
   - `complete` : un booléen indiquant si la tâche est complétée ou non.
2. **Méthodes de la classe** :
   - `__init__` : Méthode d'initialisation qui permet de définir le `titre`, la `description` et l'état `complete` lors de la création de l'objet.
   - `marquer_complete` : Méthode qui marque la tâche comme complétée.
   - `marquer_incomplete` : Méthode qui marque la tâche comme non complétée.
   - `afficher_details` : Méthode qui affiche les détails de la tâche (titre, description, état).
3. **Définir une classe** `ListeDeTaches` qui contiendra une liste d'objets `Tache`. Cette classe doit avoir les méthodes suivantes :

- `ajouter_tache` : Méthode pour ajouter une nouvelle tâche à la liste.
- `supprimer_tache` : Méthode pour supprimer une tâche par son titre.
- `afficher_taches` : Méthode pour afficher toutes les tâches de la liste.

4. **Tester les classes** en créant des instances de `Tache` et de `ListeDeTaches`, et en utilisant les méthodes définies pour gérer la liste des tâches.

## 2 Héritage et polymorphisme

**Exercice 3**

Vous allez créer une hiérarchie de classes pour gérer une bibliothèque en utilisant les concepts d'héritage et de polymorphisme.

1. Créez une classe de base appelée `Livre` avec les attributs suivants :

   - `titre` : le titre du livre
   - `auteur` : l'auteur du livre
   - `année_de_publication` : l'année de publication du livre

   Ajoutez également un attribut de classe pour suivre le nombre total de livres :

   - `nombre_total_livres` : nombre total de livres dans la bibliothèque

   **Méthodes à inclure :**

   - `__init__(self, titre, auteur, année_de_publication)` : constructeur pour initialiser les attributs d'instance.
   - `description(self)` : méthode pour retourner une description générique du livre.
   - `nombre_total_livres(cls)` : méthode de classe pour retourner le nombre total de livres.

2. Créez deux sous-classes :

   - **LivrePapier** (hérite de `Livre`) : Ajoutez un attribut supplémentaire pour le nombre de pages :
     - `nombre_de_pages` : nombre total de pages du livre papier.

     **Méthodes supplémentaires :**
     - `description(self)` : redéfinissez cette méthode pour fournir une description spécifique des livres papier.
   - **LivreNumerique** (hérite de `Livre`) : Ajoutez un attribut supplémentaire pour la taille du fichier :
     - `taille_fichier_mo` : taille du fichier en mégaoctets pour les livres numériques.

     **Méthodes supplémentaires :**
     - `description(self)` : redéfinissez cette méthode pour fournir une description spécifique des livres numériques.

3. Créez une classe `Bibliotheque` pour gérer une collection de livres :

   - **Attribut :**
     - `livres` : une liste pour stocker les objets `Livre`, `LivrePapier`, et `LivreNumerique`.
   - **Méthodes :**
     - `ajouter_livre(self, livre)` : ajoute un livre à la bibliothèque.
     - `supprimer_livre(self, titre)` : supprime un livre de la bibliothèque en fonction de son titre.

- `afficher_livres(self)` : affiche la liste de tous les livres dans la bibliothèque en utilisant la méthode `description()`.

4. Créez quelques instances de livres (à la fois `LivrePapier` et `LivreNumerique`) avec des titres, auteurs, années de publication, et les attributs spécifiques aux sous-classes.
5. Ajoutez ces livres à une instance de la classe `Bibliotheque`.
6. Affichez la liste des livres disponibles dans la bibliothèque.

## 3 Classe Object et surcharge des méthodes spéciales

**Exercice 4**

L'objectif de cet exercice est la manipulation des polynômes creux à une seule variable. Un polynôme creux est un polynôme dont certains coefficients sont nuls. Un polynôme est construit à partir de monômes. Un monôme est une expression de la forme $ax^n$ où $a$ $(a \neq 0)$ est le coefficient du monôme et $n(n > 0)$ son degré.
Un monôme est représenté par un dictionnaire à un élément dont la clé est le degré $n$ et la valeur est le coefficient $a$.
Exemple : Le monôme $8x^2$ est représenté par le dictionnaire $\{2 : 8\}$.
Un polynôme creux est alors défini comme une association de monômes de degrés différents.
Exemples :
Le polynôme $-x^4 + 8x^2 - 5x$ est représenté par le dictionnaire $\{2 : 8, 1 : -5, 4 : -1\}$.
Le dictionnaire $\{0 : 1, 5 : 1, 8 : 1\}$ représente le polynôme $x^8 + x^5 + 1$.
On se propose de construire la classe PolynomeCreux à coefficients réels dont le squelette (à compléter) est défini par :

```
class PlynomeCreux :
    """Manipulation des polynômes creux à une seule variable """
    def __init__(self):
        self.data = {} #initialisation à un polynôme nul

    def ajout_monome(self, monome ={}):
        """
        Cette méthode ajoute un monôme saisi au clavier si le paramètre
        monome est nul ou ajoute le monôme nommé sinon
        """
        if len(monome) == 0 :
            #réponse à la question 1
        else : #si monome est non vide
            degre = list(monome.keys())[0] # extraction du degré
            coeff = list(monome.values())[0] # extraction du coefficient
            try :
                assert degre >= 0
                assert type(degre) == int
                assert type(coeff) == int or type(coeff) == float
                assert len(monome) == 1
                self.data.update(monome) # self.data[degre] = coeff
            except :
                print ("Erreur d'ajout monome")

    def degree(self):          #réponse à la question 2
    def __call__(self, x0):    #réponse à la question 3
    def __add__(self, other):  #other est un polynôme creux
```

```
#réponse à la question 4
def __mul__(self, other):  #other est un polynôme creux
#réponse à la question 5
def __str__(self):     #réponse à la question 6
def primitive(self):   #réponse à la question 7
```

**Travail demandé :**

1. Compléter le script de la méthode **ajout_monome**. On rappelle que cette méthode ajoute un monôme saisi au clavier (en faisant les contrôles nécessaires) si le paramètre monome est nul ou ajoute le monôme nommé *monome* sinon.
2. Écrire le script de la méthode, nommée **degree**, qui retourne le degré du polynôme.
3. Écrire le script de la méthode, nommée **__call__** qui retourne la valeur du polynôme pour un réel $x0$ donné.
4. Écrire le script de la méthode, nommée **__add__** , qui retourne le polynôme somme de deux polynômes.
   Remarque : aucun monôme nul ne doit apparaître dans le polynôme résultat.
5. Écrire le script de la méthode, nommée **__mul__** , qui retourne le polynôme produit de deux polynômes.
   Remarque : aucun monôme nul ne doit apparaître dans le polynôme résultat.
6. Écrire le script de la méthode, nommée **__str__**, qui retourne la chaîne représentant l'expression du polynôme ordonné par ordre décroissant.
   Pour le polynôme représenté par $\{4 : 4, 0 : 4, 12 : 6, 9 : 1, 7 : -1\}$, la chaîne retournée est : "6*x**12 + x**9 - x**7 + 4*x**4 + 4"
7. Écrire le script de la méthode, nommée **primitive**, qui retourne le polynôme représentant la primitive. On suppose que la constante d'intégration est nulle.
8. On définit, l'intégrale d'un polynôme creux $P$ en $x$ entre les bornes $a$ et $b$, par: $S = \int_a^b P dx$.
   Écrire le script de la fonction, nommée **integrale**, permettant de retourner la valeur de $S$ à partir d'un polynôme $P$, de type PolynomeCreux, et des bornes d'intégration $a$ et $b$ réels.

## Exercice 5: Gestion des Fonctions Mathématiques

Créez un système en Python pour représenter et manipuler des fonctions mathématiques. Vous devrez définir plusieurs classes pour représenter les fonctions et effectuer des opérations sur celles-ci.

1. **Classe** `Fonction` : Cette classe représente une fonction mathématique de base. Elle doit inclure les attributs suivants :
   - `nom` : Le nom de la fonction.
   - `fonction` : Une fonction lambda ou une fonction définie qui représente l'expression mathématique.

   La classe doit inclure :
   - Un constructeur pour initialiser les attributs.
   - Une méthode `__str__()` pour retourner une représentation en chaîne de caractères de la fonction.
   - Une méthode `evaluer(x)` qui retourne la valeur de la fonction pour un argument donné `x`.
2. **Classe** `FonctionComposee` : Cette classe représente la composition de deux fonctions. Elle doit hériter de la classe `Fonction` et inclure :

- Les attributs `fonction1` et `fonction2` : Deux instances de la classe `Fonction`.

La classe doit inclure :

- Un constructeur pour initialiser les attributs et créer une fonction composée.
- Une méthode `__str__()` pour retourner une représentation en chaîne de caractères de la fonction composée.
- Une méthode `evaluer(x)` qui évalue la fonction composée pour un argument donné `x`.

3. **Classe** `FonctionInverse` : Cette classe représente l'inverse d'une fonction. Elle doit hériter de la classe `Fonction` et inclure :

- Une méthode `calculer_inverse(x)` pour calculer l'inverse de la fonction pour un argument `x` donné .
- Un constructeur pour initialiser les attributs et créer une fonction inverse.
- Une méthode `__str__()` pour retourner une représentation en chaîne de caractères de la fonction inverse.
- Une méthode `evaluer(x)` qui évalue la fonction inverse pour un argument donné `x`.

**Exemple d'utilisation :**

```
# Définir des fonctions
f1 = Fonction("f(x) = x^2", lambda x: x**2)
f2 = Fonction("g(x) = x + 1", lambda x: x + 1)

# Composer des fonctions
f_composée = FonctionComposee(f1, f2)

# Calculer et afficher des valeurs
print(f_composée)  # Affiche la composition de f et g
print(f_composée.evaluer(2))  # Affiche la valeur de (x^2 + 1) pour x = 2

# Calculer l'inverse d'une fonction
f_inverse = FonctionInverse(f1, lambda y: y**0.5)
print(f_inverse)  # Affiche l'inverse de f
print(f_inverse.evaluer(4))  # Affiche la valeur inverse de x^2 pour x = 4
```

## 4 Encapsulation

**Exercice 6**

Concevoir une hiérarchie de classes en Python pour modéliser une réaction chimique, en utilisant les concepts de réactifs et de produits.

1. **Définir une classe** `Reactif` avec les attributs suivants :

- `nom` : le nom du réactif (par exemple, "$H_2O$").
- `quantite` : la quantité du réactif (exprimée en moles).

2. **Définir une classe** `Produit` héritant de la classe `Reactif` avec un attribut supplémentaire :

- `rendement` : le rendement du produit (exprimé en pourcentage, 0.05 pour 5%), indiquant l'efficacité de la conversion des réactifs en produits.

3. **Définir une classe** `ReactionChimique` avec les attributs suivants :

- `reactifs` : une liste d'objets `Reactif` impliqués dans la réaction.
- `produits` : une liste d'objets `Produit` résultants de la réaction.

4. **Encapsulation des attributs** :
   - Les attributs `reactifs` et `produits` doivent être privés. Ils ne doivent pas être accessibles directement depuis l'extérieur de la classe. Fournir des méthodes publiques pour obtenir et modifier ces attributs.
   - Les modifications des listes de réactifs et de produits doivent être validées pour garantir qu'elles contiennent des objets valides de type `Reactif` ou `Produit`.
5. **Méthodes de la classe** `ReactionChimique` :
   - `get_reactifs` : Retourne la liste des réactifs.
   - `set_reactifs` : Permet de définir la liste des réactifs, uniquement si les objets sont valides.
   - `get_produits` : Retourne la liste des produits.
   - `set_produits` : Permet de définir la liste des produits, uniquement si les objets sont valides.
   - `verifier_equilibre` : Vérifie si le nombre total de moles des réactifs est égal au nombre total de moles des produits.
   - `__str__` : Retourne une représentation textuelle de la réaction chimique pour faciliter l'affichage des informations. Cette représentation doit inclure :
     - La liste des réactifs avec leur nom et quantité.
     - La liste des produits avec leur nom et quantité.
     - L'état de l'équilibrage de la réaction (équilibrée ou non équilibrée).
6. **Tester les classes** :
   - Créez des instances des classes `Reactif` et `Produit`.
   - Créez une instance de la classe `ReactionChimique` en utilisant les instances des réactifs et des produits.
   - Utilisez les méthodes de la classe `ReactionChimique` pour vérifier leur fonctionnalité.

**Exercice 7**

Soit les classes A et B suivantes. Quelle sera la sortie du programme suivant ?

```
class A:
   i = 1
   k = 3
   def __init__(self):
     self.i = 2
   def f(self):
     return self.g()
   def g(self):
     return 'A'

class B(A):
   def g(self):
     return 'B'
```

```
a = A(); b = B() ;
type(a) ; help(a) ;
print(a.g(), b.g()) # affiche : ?
print(a.f(), b.f()) # affiche : ?
print(A.i) # affiche ?
print(A().i) # affiche ?
a.j = 0 #Ajouter un attributobjet a
print(a.j) # affiche ?
print(b.j) # affiche ?
b = a ; print(b.j) ;
b.j = -1 ; print(a.j) # affiche ?
```