# Terminale - Spécialité NSI

# TP - Programmation orientée objet

## Introduction à la programmation orientée objet

### Exercice 1 : Création d'une classe JeuVideo *

Un jeu vidéo est défini par :

- Un **titre** (chaîne de caractères == string)
- Une ou plusieurs **plateformes** (liste de strings), par exemple PC, PS4 et XBOX One
- Le fait qu'il soit jouable ou non en **multijoueur** (booléeen)

**Complétez et documentez** le code de la classe ci-dessous modélisant un jeu-vidéo.

```
class JeuVideo:
    def __init__(self, titre, plateformes, multijoueur):
        self.adefinir = ...
        self.adefinir = ...
        self.adefinir = ...

    def addPlateforme(self, ...):
        if not(... in self.adefinir):
            self.adefinir.append(...)

    def removePlateforme(self, ...):
        if ... in self.adefinir:
            self.adefinir.remove(...)

    def isOnPlateforme(self, ...):
        return ... in self.adefinir

    def isMultijoueur(self):
        return self.adefinir
```

### Test de la classe

```
>>> mario64 = JeuVideo("Mario 64", ["N64", "DS", "Switch"], False)
>>> mario64.titre
"Mario 64"
>>> mario64.isMultijoueur()
False
>>> mario64.addPlateforme("PS4")
>>> mario64.plateformes
["N64", "DS", "Switch", "PS4"]
>>> mario64.isOnPlateforme("Switch")
True
>>> mario64.removePlateforme("PS4")
>>> mario64.isOnPlateforme("PS4")
False
```

### Affichage

Tentez d'exécuter le code suivant :

```
mario64 = JeuVideo("Mario 64", ["N64", "DS", "Switch"], False)
print(mario64)
```

### Qu'observez vous ? Pourquoi ?

Maintenant, rajoutez la méthode suivante dans votre classe **JeuVideo** :

```
def __str__(self):
    texteAffichage = self.titre + "\n"
    texteAffichage += "Multijoueur : " + str(self.isMultijoueur()) + "\n"
    texteAffichage += "Plateformes : " + str(self.plateformes) + "\n"
    return texteAffichage
```

**Retentez l'exécution** du code précédent, qu'observez vous dorénavant ?

La méthode **__str__()** est une **"Méthode magique"** de python, pour les classes. Sa **signature** est équivalente pour n'importe quelle classe. Tentez d'expliquer ce qu'elle permet de faire.

## Exercice 2 : Fractions *

Une fraction désigne la représentation d'un nombre relationnel sous la forme d'un quotient de deux entiers. La fraction $\frac{a}{b}$ désigne le quotient de "a" par "b" où **"a"** est nommé **numérateur** et **"b" dénominateur**.
On se sert également des fractions pour représenter des **valeurs exactes** (par exemple $\frac{1}{3}$ qui ne peut qu'être qu'approché si on l'écrit en tant que nombre à virgule.

Le but de cet exercice va être de créer une classe **Fraction** sous python et de la manipuler

afin de représenter des fractions et de les multiplier.
Pour cela :

1. Définissez la classe **Fraction**
2. Implémentez le **constructeur** (deux attributs de l'objet : Le numérateur et le dénominateur)
3. Implémentez la méthode **valeur(self)** qui donne la valeur de la division du numérateur par le dénominateur de la fraction.
4. Implémentez la méthode **multiplier(self, fraction_bis)** qui va **renvoyer une nouvelle fraction** issue du **produit** de la fraction **self** (sur laquelle la méthode est appelée) et la fraction **fraction_bis**. Pour rappel, pour multiplier deux fraction, on multiplie simplement les numérateurs ensembles et les dénominateurs ensembles.
5. Implémentez la **méthode magique __str__(self)** afin d'avoir une représentation de la fraction ainsi : (a / b)

**Test de la classe**

```
>>> f1 = Fraction(1,3)
>>> print(f1)
(1 / 3)
>>> print(f1.valeur())
0.3333333333333
>>> f2 = Fraction(3,2)
>>> print(f2.valeur())
1.5
>>> f3 = f1.multiplier(f2)
>>> print(f3)
(3 / 6)
>>> print(f3.valeur())
0.5
```

## Exercice 3 : Dates *

Dans cet exercice, vous allez implémenter une classe permettant de représenter et des comparer des objets de type **"Date"** définis à partir d'un jour, d'un mois et d'une année.
Pour cela :

1. Définissez la classe **Date**
2. Implémentez le **constructeur** (trois attributs de l'objet : Le jour, le mois et l'année, des entiers)
3. Implémentez la méthode **nomMois(self)** qui donne le nom du mois de la date (par exemple pour une date 20/09/2020, cela donne "Septembre")
4. Implémentez la **méthode magique __str__(self)** afin d'avoir une représentation de la date sous la forme : Jour Nom du Mois Année. Par exemple pour le 04/03/2021 : 4 Mars 2021.

On aimerait aussi pouvoir comparer deux dates entres elles de manière simple, par exemple, avec les opérateurs logiques $<$, $>$, $==$, $!=$, $<=$ et $>=$
**Quel problème(s) va-t-on rencontrer** si on essaye de comparer en l'état deux objets "Date" ?

Pour ça aussi, **il existe des méthodes "magiques"** permettant d'implémenter un tel comportement.
Implémentez la méthode **__lt__(self, dateComparee)** signifiant "plus petit que" afin de gérer la comparaison entre deux dates avec "<". Cette méthode doit renvoyer "True" si la date représentée par l'objet courant (self) est plus petite que l'objet comparé (dateComparee) et "False" sinon.

**Test de la classe**

```
>>> d1 = Date(3,2,2024)
>>> print(d1)
3 fevrier 2024
>>> d2 = Date(8,9,2024)
print(d1 < d2)
>>> True
print(d2 < d1)
>>> False
```

L'exercice se termine ici, mais si vous souhaitez aller plus loin, vous pouvez aller chercher sur internet le nom des méthodes magiques permettant de gérer les autres opérateurs logiques que nous avons cités !

# Piles et Files, le retour !

Lors d'un précédent TP vous avez implémentés les structures de piles et de files à l'aide des listes en python. Maintenant que vous connaissez la programmation orientée objet, le but des prochains exercices va être de réécrire ces structures à l'aide d'objet **sans utiliser les listes**, à l'aide d'un **système chaîné** (où l'objet contient un lien vers l'objet le suivant).

## Exercice 1 : La classe Cellule *

La classe **"Cellule"** est définie comme un objet possédant deux attributs :

- Une valeur numérique (**valeur**)
- Un objet de type Cellule (**suivant**) définissant le "maillon" suivant de la structure chaînée
- Pas de méthodes particulières (hors constructeur)

**Définissez, documentez et testez** la classe **Cellule** telle que définie au-dessus.

**Test de la classe**

```
>>> c3 = Cellule(5, None)
>>> c2 = Cellule(7, c3)
>>> c1 = Cellule(19, c2)
>>> c1.valeur
19
>>> c1.suivant.valeur
7
>>> c1.suivant.suivant.valeur
5
>>> c1.suivant.suivant.suivant
```

*"None" représente un objet non-défini, c'est à dire qu'il modélise l'absence de valeur*

## Exercice 2 : La classe Pile **

La classe **Pile** est définie comme suit :

- Un attribut **sommet** qui est un objet de type **Cellule** qui représente le sommet de la pile (initialisé à **None** dans le constructeur, car pas de sommet lors de la création de la pile)
- Un attribut **taille**, initialisée à 0, représentant le nombre d'éléments dans la pile
- Une méthode **estVide(self)** qui renvoie "True" si la pile est vide et "False" sinon
- Une méthode **valeurSommet(self)** qui renvoie la **valeur** du sommet de la pile
- Une méthode **empiler(self, x)** qui "empile" l'élément x dans la pile (et met à jour l'attribut **taille**). Pour faire cela, on crée une nouvelle instance de **Cellulle** qui deviendra le nouveau sommet de la pile (à vous de déterminer avec quelles valeurs initialiser cet instance)
- Une méthode **depiler(self)** qui "dépile" le sommet de la pile, met à jour l'attribut **taille** et renvoie la valeur de l'élément dépilé. Pour faire cela, le sommet de la pile devient l'élément qui suit le sommet courant.
- Une méthode **__len__(self)** qui renvoie la taille de la pile (il s'agit en fait d'une implémentation d'une méthode magique qui permettra d'appeler la méthode **len** directement sur la pile)

**Définissez, documentez et testez** la classe **Pile** telle que définie ci-dessus.

**Test de la classe**

```
>>> pile_test = Pile()
>>> len(pile_test)
0
>>> pile_test.estVide()
True
>>> pile_test.empiler('A')
```

```
>>> pile_test.empiler('B')
>>> pile_test.empiler('C')
>>> pile_test.empiler('D')
>>> len(pile_test)
4
>>> pile_test.estVide()
False
>> pile_test.valeurSommet()
'D'
>>> pile_test.depiler()
'D'
>> pile_test.valeurSommet()
'C'
>>> len(pile_test)
3
```

## Exercice 3 : La classe File ***

La classe **File** est définie comme suit :

- Un attribut **tete** qui est un objet de type **Cellule** qui représente la tête de la file (initialisée à **None** dans le constructeur, car pas de tête lors de la création de la file)
- Un attribut **queue** qui est un objet de type **Cellule** qui représente le bout de la file, c'est à dire le dernier élément (initialisée à **None** dans le constructeur, car pas de dernier élément lors de la création de la file)
- Un attribut **taille**, initialisée à 0, représentant le nombre d'éléments dans la file
- Une méthode **estVide(self)** qui renvoie "True" si la file est vide et "False" sinon
- Une méthode **valeurTete(self)** qui renvoie la **valeur** da la tête de la file
- Une méthode **ajouterFile(self, x)** qui rajoute l'élément x au bout de la file (et met à jour l'attribut **taille**). Pour faire cela, on crée une nouvelle instance de **Cellulle** qui ne sera suivi par rien (None). On observe ensuite deux cas. Soit la file est vide et la tête devient donc cette cellule, ou bien la file n'est pas vide et on fait donc en sorte que cette cellule soit la "suivante" de la cellule "queue" actuelle. Enfin, dans tous les cas, l'attribut "queue" est affecté à la cellule nouvellement crée.
- Une méthode **retirerFile(self)** qui retirer la tête de file, met à jour l'attribut **taille** et renvoie la valeur de l'élément. Pour faire cela, il suffit d'affecter la tête avec l'élément suivant de la tête actuelle.
- Une méthode **__len__(self)** qui renvoie la taille de la file (il s'agit en fait d'une implémentation d'une méthode magique qui permettra d'appeler la méthode **len** directement sur la file)

**Définissez, documentez et testez** la classe **File** telle que définie ci-dessus.

**Test de la classe**

```
>>> file_test = File()
>>> len(file_test)
0
>>> file_test.estVide()
True
>>> file_test.ajouterFile('A')
>>> file_test.ajouterFile('B')
>>> file_test.ajouterFile('C')
>>> file_test.ajouterFile('D')
>>> len(file_test)
4
>>> file_test.estVide()
False
>>> file_test.valeurTete()
'A'
>>> file_test.retirerFile()
'A'
>>> file_test.valeurTete()
'B'
>>> len(file_test)
3
```

# Gestion des classes d'un lycée

Dans cette dernière section, nous nous proposons de modéliser le fonctionnement de classes (d'élèves) dans un lycée.

## Exercice 1 : Classe Eleve *

Tout d'abord, vous allez devoir modéliser la classe **"Eleve"**
Un élève est défini par :

- Un attribut **nom** (string)
- Un attribut **prenom** (string)
- Un attribut **notes** (liste de nombres, initialisée vide)
- Une méthode **ajouterNote(self, x)** qui ajoute une note à la liste des notes de l'élève.
- Une méthode **moyenne(self)** renvoyant la moyenne de l'élève à partir de ses notes
- Une méthode **noteLaPlusBasse(self)** renvoyant la note la plus basse de l'élève
- Une méthode **noteLaPlusHaute(self)** renvoyant la note la plus haute de l'élève
- Une méthode **__str__(self)** renvoyant une chaîne de caractères présentant l'élève avec son nom, son prénom et se moyenne.

**Définissez, documentez et testez** la classe **Eleve** telle que définie ci-dessus.

### Test de la classe

```
>>> eleve1 = Eleve("John", "Smith")
>>> eleve1.moyenne()
0
>>> eleve1.ajouterNote(11)
>>> eleve1.ajouterNote(13)
>>> eleve1.ajouterNote(8)
>>> eleve1.ajouterNote(15)
>>> eleve1.moyenne()
11.75
>>> eleve1.noteLaPlusHaute()
15
>>> eleve1.noteLaPlusBasse()
8
>>> print(eleve1)
John Smith - Moyenne : 11.75
```

## Exercice 2 : Amélioration des notes **

Maintenant que vous avez défini l'objet représentant un élève, il pourrait être intéressant d'avoir un objet décrivant les notes plutôt que de simples nombres.

### La classe Note

Une note est définie par :

- Un attribut **valeur** (nombre représentant la note)
- Un attribut **coefficient** (nombre représentant le coefficient de la note)
- Un attribut **matiere** (string représentant le nom de la matière)
- Pas de méthodes particulières (hors constructeur)

**Définissez, documentez** la classe **Note** telle que définie ci-dessus.

### Retour sur Eleve

Maintenant, il va falloir adapter la classe **Eleve** avec la classe **Note**
Vous devez donc modifier :

- La méthode **moyenne(self)**
- La méthode **noteLaPlusBasse(self)**
- La méthode **noteLaPlusHaute(self)**

**Remarque :** On pourrait éviter de (trop) modifier ces méthodes directement et agir sur la classe **Note** en implémentant certaines méthodes magiques, comme dans l'exercice d'introduction avec la classe **Date**...Si le temps de permet, cherchez lesquelles et implémentez les dans **Note**.

Enfin, maintenant que la note donne également des informations sur **matière**, **rajoutez, documentez et testez** les méthodes suivantes (dans **Eleve**) :

- Une méthode **moyenneMatiere(self, matiere)** renvoyant la moyenne de l'élève pour une matière donnée à partir de ses notes.
- Une méthode **noteLaPlusBasseMatiere(self, matiere)** renvoyant la note la plus basse de l'élève pour une matière (string) donnée
- Une méthode **noteLaPlusHauteMatiere(self, matiere)** renvoyant la note la plus haute de l'élève pour une matière (string) donnée

### Test de la classe

```
>>> eleve1 = Eleve("John", "Smith")
>>> eleve1.ajouterNote(Note(11, 2, "NSI"))
>>> eleve1.ajouterNote(Note(13, 5, "SVT"))
>>> eleve1.ajouterNote(Note(8, 1, "NSI"))
>>> eleve1.ajouterNote(Note(15, 3, "Math"))
>>> eleve1.moyenne()
12.72
>>> eleve1.moyenneMatiere("NSI")
10
>>> eleve1.moyenneMatiere("Math")
15
>>> eleve1.noteLaPlusHauteMatiere("NSI")
11
>>> eleve1.noteLaPlusBasseMatiere("SVT")
13
```

## Exercice 3 : La classe ClasseEleve ***

Pour ce dernier exercice, nous allons donc créer un objet représentant une classe d'élèves. (au sens du lycée)
Une classe d'élèves est défini par :

- Un attribut **label** (nom de la classe, string, par exemple : "TG03")
- Un attribut **profPrincipal** (string)
- Un attribut **listeEleves** (liste contenant des objets de type **Eleve**, initialisée vide)
- Une méthode **ajouterEleve(self, eleve)** qui ajoute un élève (de type **Eleve**) à la liste des élèves de la classe
- Une méthode **supprimer(self, nom, prenom)** qui supprime un élève (de la liste des élèves de la classe en se basant sur nom et son prénom (on considère qu'il n'y a pas d'homonyme)
- Une méthode **moyenneClasse(self)** renvoyant la moyenne de la classe.
- Une méthode **moyenneClasseMatiere(self, matiere)** renvoyant la moyenne de la classe pour une matière (string) donnée.

— Une méthode **meilleurEleve(self)** renvoyant le meilleur élève (objet) de la classe (en se basant sur sa moyenne générale)

— Une méthode **meilleurEleveMatiere(self, matiere)** renvoyant le meilleur élève (objet) de la classe pour une matière (string) donnée (en se basant sur sa moyenne générale dans cette matière).

— Une méthode **dernierEleve(self)** renvoyant le moins bon élève (objet) de la classe (en se basant sur sa moyenne générale)

— Une méthode **dernierEleveMatiere(self, matiere)** renvoyant le moins bon élève (objet) de la classe pour une matière (string) donnée (en se basant sur sa moyenne générale dans cette matière).

— Une méthode **__str__(self)** renvoyant une chaîne de caractères présentant la classe avec : Le label de la classe sur une ligne, le professeur principal sur une autre et enfin une ligne pour chaque élève (avec sa moyenne générale).

**Définissez, documentez et testez** la classe **ClasseEleve** telle que définie ci-dessus.

## Évolutions possibles ****

Voici quelques pistes afin d'améliorer et d'étendre le mini-projet que vous venez de réaliser avec la gestion des données d'une classe d'élèves (si vous avez fini les exercices précédent et que le temps le permet).

— Faire en sorte que les élèves soient présentés par ordre alphabétique (avec une autre méthode ou dans __str__(self)

— Créer une méthode permettant de classer les élèves (tri) selon leur moyenne générale, puis une autre méthode pour faire la même chose par matières.

— Placez autant d'objets que possible et étendez le projet à votre guise ! Exemples : Matiere, Lycee, SalleDeCours, Professeur...