Chapitre I : Structure de Données

I) Programmation Orientée Objet : Classes, attributs, méthodes
II) Structures linéaires : Tableau, Pile, File, Liste Chainée

I) Programmation Orientée Objet

La programmation Orientée Objet (POO) permet de représenter les objets du réel ou des objets abstraits avec un code en général plus stable et plus facile à maintenir que dans d'autres styles.

a) Classes et Attributs

Exemple : Fiche de personne dans un lycée.
On va représenter des personnes du lycée (élèves, professeurs, administratifs…) ainsi que des informations diverses (Nom, Prénom, Fonction…)

On va voir plusieurs façons d'implémenter une fiche de personne.

- Méthode basique : La Liste

On crée une liste qui contient nom, prénom, statut et date de naissance.

```
# Liste :
Perso1 = ["Kasparian","Yves","Professeur","01/01/1900"]
```

Avantage : Très simple, facile d'accès
Inconvénient : Se souvenir que le premier élément de la liste est le nom, le second le prénom…

- Méthode plus évoluée : Le Dictionnaire

```
# Dictionnaire :
Perso2 = {"Nom": "Kasparian", "Prénom":"Yves", "Statut":"Professeur",
          "Date_Naissance": "01/01/1900"}
```

Avantage : On a accès aux informations de façon plus simple
Inconvénient : plus difficile à écrire

- Méthode POO : On définit une classe Personnage.

Chaque objet de la classe sera un personnage. Voici comment faire :

```
# Classe
class Perso:
    def __init__(self, n, p, s, d_n):
        self.nom = n
        self.prenom = p
        self.statut = s
        self.date_naissance = d_n
```

Il faut comprendre cette écriture :

Perso : Est une classe d'objets. Tous les objets de la classe *Perso* auront donc les mêmes propriétés.

__init__ : Est ce que l'on appelle **un constructeur**. C'est ce qui va définir exactement les objets de la classe *Perso*.

Ici les objets de type *Perso* ont 4 **attributs** : *nom*, *prenom*, *statut* et *date_naissance*
Un objet de la classe est appelé **instance**.

Dans le programme (ou dans le *shell*) pour définir une instance de la classe *Perso* et accéder aux attributs de cette instance, voici la syntaxe :

```
Perso3 = Perso("Kasparian","Yves","Professeur","01/01/1900")
```

Perso3 est une instance de la classe *Perso*

```
>>> Perso3.date_naissance
'01/01/1900'
```

On accède à l'attribut *date_naissance* de Perso3 avec le point

Remarque : On peut définir des **attributs par défaut**. Par exemple la nationalité. Comme elle est souvent la même indiquer par défaut que la réponse est « Française ». Ainsi si on ne complète rien alors la nationalité sera celle par défaut, sinon on complète.
Faites des tests avec l'exemple ci-dessous :

```python
# Classe avec Attributs par défaut
class Perso:
    def __init__(self,n, p, s, d_n,nat = "Française"):
        self.nom = n
        self.prenom = p
        self.statut = s
        self.date_naissance = d_n
        self.nationalite = nat


Perso4 = Perso("Kasparian","Yves","Professeur","01/01/1900")
Perso5 = Perso("Kasparian","Yves","Professeur","01/01/1900","Belge")
```

_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._

Exercice : Imaginons qu'on veuille créer un second attribut par défaut (le sexe par exemple qui sera fixé à « M »).
Modifier la classe *Perso* pour intégrer cet attribut par défaut et créer un objet qui serait de nationalité française mais de sexe « F ».

_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._

b) Méthodes

Une méthode est une fonction que l'on va appliquer sur un objet de la classe.

Pour reprendre la classe *Perso* créée ci-dessus, on peut vouloir avoir une fonction qui va afficher toutes les informations de la personne.

Voici par exemple ce que l'on peut faire :

```python
class Perso:
    def __init__(self,n, p, s, d_n,nat = "Française"):
        self.nom = n
        self.prenom = p
        self.statut = s
        self.date_naissance = d_n
        self.nationalite = nat

    def info(self):
        print("Voici toutes les informations disponibles :")
        print("Nom :",self.nom)
        print("Prénom :",self.prenom)
        print("Statut :",self.statut)
        print("Date de Naissance :",self.date_naissance)
        print("Nationalité :",self.nationalite)


Perso4 = Perso("Kasparian","Yves","Professeur","01/01/1900")
```

Dans le *shell* il ne reste plus qu'à appliquer la méthode comme ceci :

```
>>> Perso4.info()
Voici toutes les informations disponibles :
Nom : Kasparian
Prénom : Yves
Statut : Professeur
Date de Naissance : 01/01/1900
Nationalité : Française
```

Remarque : Notez bien que le premier paramètre d'une méthode sera toujours *self*, en effet une méthode s'applique à l'objet que l'on considère (ici le personnage).

Exercice 1 : Ecrire une méthode qui renvoie l'âge de la personne.

Voici quelques aides/rappels sur les fonctions, les chaines de caractères et les entiers :

```
def age(self):
    ...
    ...
    ...
    return ...
```

```
>>> texte = "bonjour"
>>> texte[1:3]
'on'
```

```
>>> int("25")
25
```

Exercice 2 : Ecrire une méthode nommée *majeur* qui renvoie un booléen : *True* si la personne est majeure, *False* sinon. (On se servira de la méthode *age*)

Exercice 3 : Nous allons ajouter un attribut *classe* aux objets de la classe *Perso*. Mais attention il y a trois statuts différents et donc cet attribut aura différentes formes :

- Si le personnage est un élève alors l'attribut *classe* est une chaine de caractère qui représente sa classe (208, 104, T06…)
- Si le personnage est un professeur alors l'attribut *classe* est une liste composée des chaines de caractères qui représentent toutes les classes qu'il a.
- Si le personnage est un administratif alors il n'a pas l'attribut *classe*.

Question a : Modifier le constructeur de la classe *Perso* pour ajouter l'attribut *classe*.
Question b : Modifier la méthode *info* en conséquence.
Question c : Ecrire une méthode *est_eleve_de* qui prend un personnage en paramètre et qui renvoie le booléen correspondant à l'exemple ci-dessous :

```
>>> Perso_e = Perso("Ab","Cd","Elève","01/01/2005","104")
>>> Perso_p = Perso("Pro","Ef","Professeur","01/01/1980",["201","104","T03"])
>>> Perso_e.est_eleve_de(Perso_p)
True
```

Attention : Ne pas confondre attribut et méthode. La notation est presque identique (juste les parenthèses pour la méthode) mais on parle de deux choses bien distinctes. C'est le même type de différence qu'avec les mots variables et fonctions.

Remarque : Les notions d'héritage et de surcharge d'opérateurs ne sont pas au programme de T NSI, dommage…

II) Structures Linéaires

Une structure linéaire sur un ensemble *E* est une suite d'éléments de *E*, c'est-à-dire une liste ordonnée d'éléments de *E*. Chaque élément a une place bien précise.

Vocabulaire :

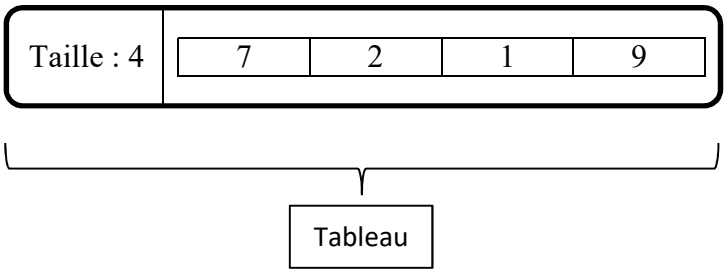
- Structure **Linéaire** ou non linéaire : Lorsque les éléments sont ordonnés (liste) on parle de structure linéaire. Un dictionnaire est une structure non linéaire.

- Structure **homogène** ou non : Les éléments sont tous du même type ou non. En Python on peut mettre n'importe quel type dans des listes, donc la structure sera non homogène.

- Structure **Statique** ou **Dynamique** : La taille est fixée et ne peut être modifiée en fonction des besoins. Une liste est une structure dynamique.

a) Les Tableaux

En informatique, un tableau (*array* en anglais) est la donnée de deux choses : Sa taille et la liste de ses valeurs.



LA grande différence entre un tableau et une liste c'est qu'un tableau n'est pas dynamique : sa taille est fixée au départ et ne peut pas être modifiée.

Il n'y a pas réellement de tableaux en Python, nous allons donc les implémenter et créer une classe.

Exercice 4 :
a) Créer une classe nommée *Tab* qui va modéliser le comportement d'un tableau.
Une instance de *Tab* aura deux attributs : *taille* (de type *int*) et *valeurs* (de type *list*).

b) Même chose mais cette fois ci on va réfléchir à l'attribut *valeurs*. Il va falloir le gérer de façon à ce que l'utilisateur puisse ne pas le saisir, le saisir ou même n'en saisir qu'une partie s'il en a envie.

Voici des exemples :

L'utilisateur ne saisit rien, le tableau est composé de 5 éléments *None* :

```
>>> T1=Tab(5)
>>> T1.valeurs
[None, None, None, None, None]
```

L'utilisateur ne saisit que 3 valeurs, le reste du tableau est complété de *None*.

```
>>> T2=Tab(5,[10,20,30])
>>> T2.valeurs
[10, 20, 30, None, None]
```

L'utilisateur saisit exactement ce qu'il faut :

```
>>> T3=Tab(5,[10,20,30,40,50])
>>> T3.valeurs
[10, 20, 30, 40, 50]
```

L'utilisateur saisit trop de valeurs, le tableau ne prend que les 5 premières valeurs.

```
>>> T4=Tab(5,[10,20,30,40,50,60,70])
>>> T4.valeurs
[10, 20, 30, 40, 50]
```

Exercice 5 : Tableaux : Facile d'accès, difficile à modifier…
L'idée est de comprendre que cette structure est simple lorsque l'on veut accéder à des données mais pas évidente si on les modifier.
a) Ecrire une méthode *insert* : elle aura deux paramètres, la valeur à insérer dans le tableau ainsi que sa place.
Aide :

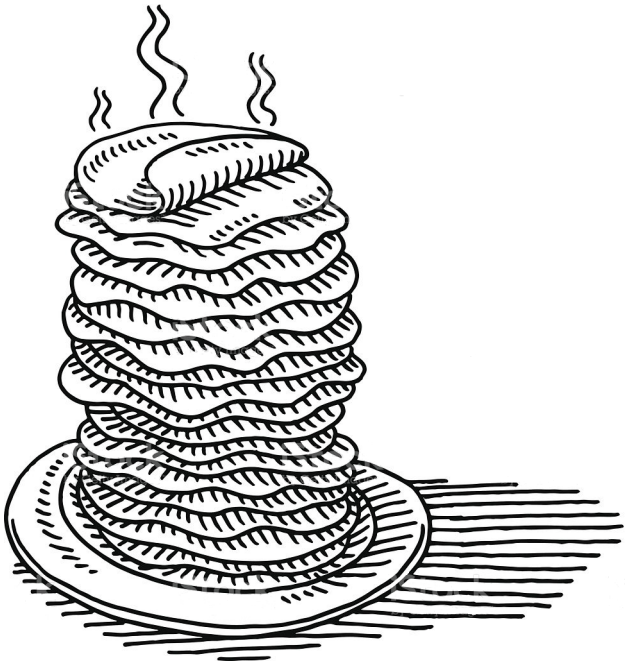
```
for i in range(5,2,-1):
    print(i)
```

b) Ecrire la méthode supprime qui doit supprimer l'élément à la place indiquée (on supprimera de façon à ne pas laisser de « trou » dans le tableau…)

b) Les Piles

Une structure de pile (*stack* en anglais) est considérée comme une liste mais avec des conditions d'utilisation extrêmement restreintes : les seules opérations possibles sont l'insertion ou la suppression d'un élément **mais uniquement à la fin**.

On utilise l'acronyme **LIFO** : « Last In First Out ». (Dernier entré premier sorti)



On peut imaginer le principe d'une pile de crêpes :

Si on veut en prendre une on doit prendre celle du dessus, c'est-à-dire la dernière posée.

Nous allons utiliser la programmation orientée objet pour implémenter une pile.
En plus de la construction de la classe non aurons besoin de deux méthodes essentielles sur les piles :

- Une qui ajoute un élément sur la pile, en français *empile*, en anglais *push*
- Une qui retire l'élément qui est sur le sommet de la pile et qui le renvoie. En français *depile*, en anglais *pop*.

Dans la pratique, il peut être pratique de savoir si la pile est vide ou quel est son sommet (la méthode sommet est appelée *peek* en anglais).

Exercice 6 :
a) Créer une classe nommée *Pile* qui va implémenter les piles. Une instance de cette classe aura deux attributs :

- *taille* : indique la taille de la pile (de type *int*)
- *valeurs* : la liste des valeurs de la pile (de type *list*)

L'attribut valeurs sera une liste vide par défaut.

b) Ecrire les méthodes *empile* et *depile*.

c) Ecrire les méthodes *pile_vide* et *sommet*.

Remarque :

- noter bien la différence entre *sommet* (*peek*) et *depile* (*pop*).

Pile à capacité bornée

En pratique une pile a une capacité limitée. Il peut y avoir un débordement lorsque la capacité maximale est dépassée. Concrètement, empiler un élément sur une pile pleine ne provoquera pas d'erreur, on se contentera de ne rien faire.
Dans le cas d'une pile à capacité bornée il peut être utile d'avoir une méthode qui nous indique si la pile est pleine.

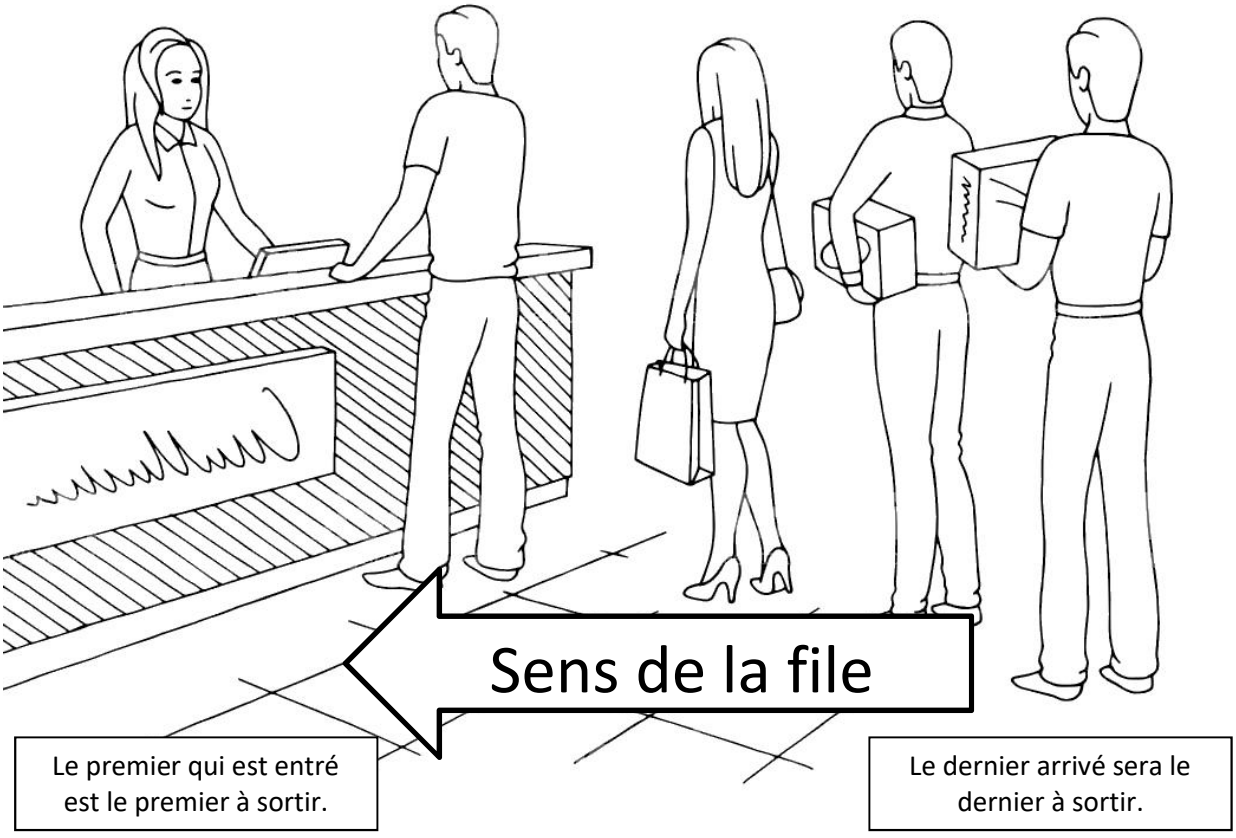
Exercice 7 :
a) Créer une classe nommée *Pile_capacite_bornee* qui va implémenter les piles à capacité bornée. Un attribut supplémentaire sera *capacite*, il sera fixé par défaut à 10.

b) Ecrire les méthodes *empile* et *depile*. Puis *pile_vide*, *sommet* et *pile_pleine*.

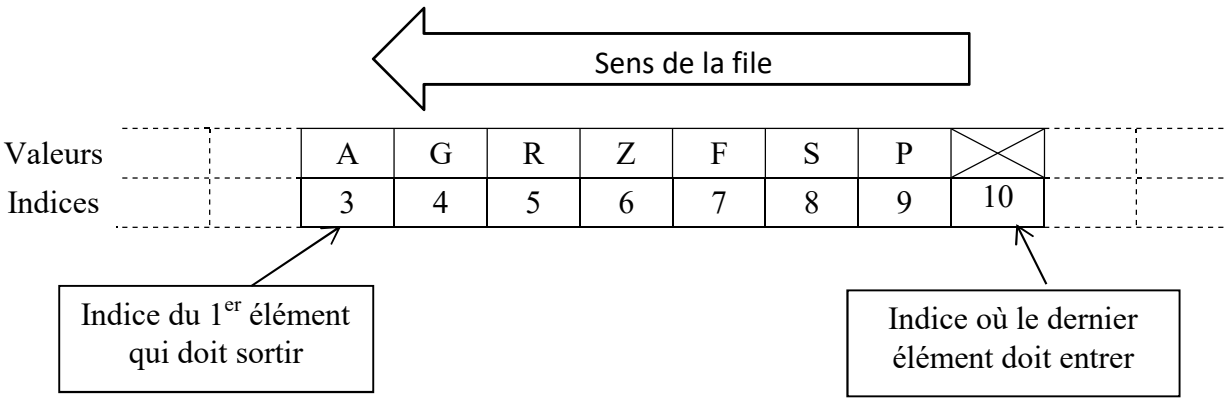
c) Les Files

Une structure de file (*queue* en anglais) est une liste où l'on autorise l'insertion d'un côté et la suppression d'un l'autre.
Pour décrire une file on utilise l'acronyme **FIFO** : « First In First Out ».

C'est le principe d'une file d'attente :



Pour implémenter une file on va utiliser une liste suivant ce modèle :



Tout comme pour les piles : il faudra implémenter les méthodes *enfile* et *defile*. (Méthodes qui effectueront la même chose que *empile* et *depile*).

En anglais :

- enfile : enqueue
- defile : dequeue

Exercice 8 :

a) Créer une classe nommée *File* qui va implémenter les files. Une instance de cette classe aura trois attributs :

- *valeurs* : la liste des valeurs de la file (de type *list*)
- *sortie* : indice du premier élément qui doit sortir (de type *int*)
- *entree* : indice où le dernier élément doit entrer (de type *int*)

Voici les valeurs si l'utilisateur ne spécifie rien de particulier :

valeurs : une liste composée de 100 éléments *None*
sortie : 0
entree : 0

b) Ecrire les méthodes *enfile* et *défile*.

c) Ecrire les méthodes *file_vide* et *taille* (de la file).

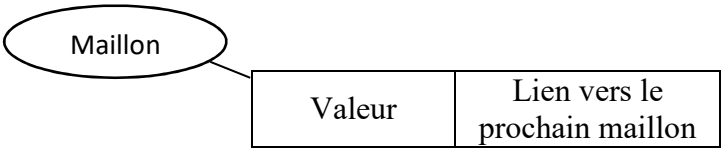
d) Ecrire une méthode affiche : elle devra afficher les éléments de la file (et non pas la liste des *None*…).

Voici un exemple d'affichage :

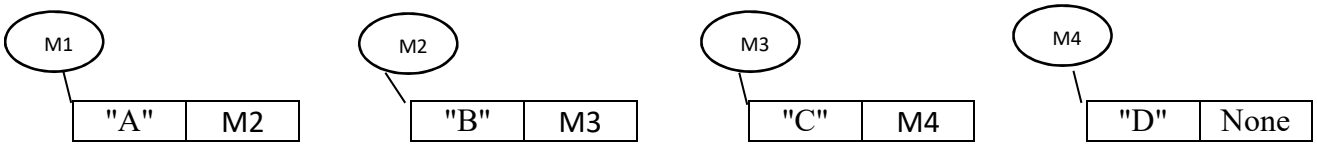
```
>>> f=File()
>>> f.affiche()
  Votre file est vide
>>> f.enfile(12)
>>> f.affiche()
  Sortie <- 12 <-  Entrée
>>> f.enfile(20)
>>> f.affiche()
  Sortie <- 12 <- 20 <-  Entrée
>>> f.enfile(31)
>>> f.affiche()
  Sortie <- 12 <- 20 <- 31 <-  Entrée
>>> f.defile()
12
>>> f.affiche()
  Sortie <- 20 <- 31 <-  Entrée
```

d) Les Listes Chainées

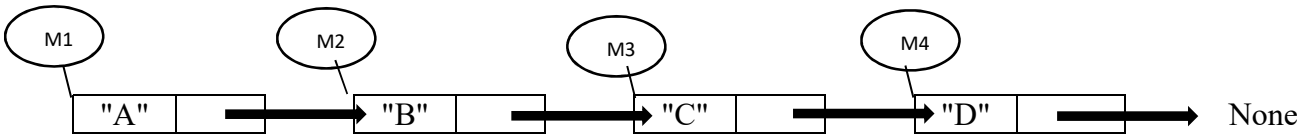
La dernière structure linéaire à étudier cette année est **la liste chainée**.
Une liste chainée est un ensemble de maillons, chaque maillon est de la forme suivante :



Ainsi une liste chainée est de ce type :



On représente cela aussi de cette façon :



Chaque maillon indique qui est le maillon suivant, le dernier maillon indique logiquement None. **Ainsi une liste chainée est uniquement la donnée du premier maillon.**

Pour implémenter une liste chainée, nous allons devoir créer deux classes : une première classe pour les maillons, une seconde pour la liste en elle-même.

Exercice 9 :
a) Créer une classe nommée *Maillon* qui va implémenter les maillons. Une instance de cette classe aura deux attributs :

- *valeur* : la valeur du maillon
- *suivant* : le maillon suivant

Ces deux attributs seront fixés par défaut à None.

b) Créer une classe nommée *Liste_chainee* qui va implémenter une liste chainée. Une instance de cette classe n'aura qu'un seul attribut : le maillon de tête. Cet attribut sera par défaut fixé à None.

c) Dans la classe *Liste_chainee*, écrire une méthode *affiche* : cette méthode devra afficher les éléments de la liste. Si la liste est vide alors elle devra afficher : Votre liste est vide.

Voici un exemple :
A écrire dans le programme :

```
liste_vide = liste_chainee()
m4 = Maillon("D",None)
m3 = Maillon("C",m4)
m2 = Maillon("B",m3)
m1 = Maillon("A",m2)
Ma_liste = liste_chainee(m1)
```

Ensuite dans la console cela donne :

```
>>> liste_vide.affiche()
  La liste est vide
```

```
>>> Ma_liste.affiche()
  A => B => C => D
```

Exercice 10 : *Difficile*
En dehors des classes *Maillon* et *Liste_chainee*, écrire une fonction *creation_lc* (pour création d'une liste chainée) qui prendra en paramètre une liste (au sens Python du terme) et qui renvoie la liste chainée créée à partir de cette liste. Voici un exemple :

```
>>> l = creation_lc([1, 10, 100, 1000])
>>> l.affiche()
  1 => 10 => 100 => 1000
```

Aide : Si vous observez bien la création « à la main » de la liste chainée de l'exercice 9, on commence par le dernier maillon pour remonter petit à petit. Ainsi il y aura une boucle for qui doit partir du dernier élément de la liste et revenir au premier…

La fonction *creation_lc* pourra aussi créer une liste chainée vide si on le souhaite :

```
>>> l = creation_lc()
>>> l.affiche()
  La liste est vide.
```