Semaine 3
Structures de données linéaires

Lors des deux premières semaines, nous avons abordé différents paradigmes de programmation, chacun présentant des avantages et des inconvénients. Cependant, lors de la création d'un programme, le choix du paradigme n'est pas la seule question à laquelle nous devons réfléchir. En effet, le choix d'une structure de données adaptée peut grandement influencer la qualité de notre programme informatique. Comme pour les paradigmes de programmation, chaque structure de données possède ses avantages et ses inconvénients, les connaitre et les maîtriser offre un avantage indéniable lors de la conception d'un programme informatique.

Au cours de l'année précédente nous avons abordé la notion de **séquence** en Python à travers les **listes** et les **tuples**. Ces structures de données appartiennent à la catégorie des structures de données linéaires. Bien que similaires, celles-ci possèdent des différences qui rendent leur utilisation plus ou moins optimale selon la problématique que nous souhaitons résoudre.

1. Structures de données linéaires : Définition et rappels

1.1. Définition

Une structure de données linéaires est une organisation de données dans un ensemble fini et ordonné d'éléments.

Cette définition inclut les *listes* (ou *Array*) et les *tuples* vu en Python mais également les *String* (Séquence ordonnée de caractères), elle exclut cependant les *dictionnaires* car ces derniers sont des ensembles finis et non ordonnés d'éléments.

1.2. Rappels sur les listes et les tuples

Pour rappel, en Python, une *liste* est une séquence ordonnée de valeurs. Il est possible de modifier une liste en ajoutant, en retirant ou en modifiant une valeur. Chaque élément est accessible à l'aide de sa position dans la liste (autrement appelé *index*).

Note 3.1

Indexation des listes

En Python, les éléments d'une liste de longueur *N* sont indexés de 0 à *N*-1. C'est le cas dans de nombreux langages de programmation. Il existe cependant des langages pour

lesquels l'indexation des listes commencent à 1. C'est le cas notamment du langage FORTRAN ou encore des langages R et MATLAB.

Les tuples fonctionnent de la même manière que les listes à l'exception que les tuples sont **immuable,** c'est-à-dire qu'ils ne peuvent plus être modifiés une fois qu'ils ont été créés en mémoire. Par conséquent, les tuples ne disposent pas des méthodes *.append()*, *.clear()*, *.remove(),* etc.. qui sont propres aux listes. Pour consulter l'ensemble des méthodes propres à un type, vous pouvez utiliser la fonction dir([object]) :

```
list_num = [1,2,3,4,5,6]
print(dir(list_num))
tuple_num = (1,2,3,4,5,6)
print(dir(tuple_num))
```

Les tuples ont cependant l'avantage d'être moins gourmands en mémoire que les listes et peuvent être utilisés comme clés dans des dictionnaires. Vous pouvez mesurer la taille en mémoire d'une variable à l'aide de la méthode __sizeof__() :

```
print("Taille de la liste en memoire : ", list_num.__sizeof__()) #Taille de la liste en memoire :  88

print("Taille du tuple en memoire : ", tuple_num.__sizeof__()) #Taille du tuple en memoire :  72
```

Les tuples présentent une alternative aux listes en offrant une structure immuable et des performances accrues. Cependant, il ne s'agit pas de la seule structure de données linéaires pouvant remplacer une liste.

1.3 Interface et implémentation : définitions

Comme nous venons de le voir, les listes et les tuples sont des structures de données linéaires différentes, elles répondent chacune à des critères spécifiques. L'ensemble de ces critères est appelé ***Interface***.

L'interface d'une structure de données définit l'ensemble des opérations qui permettent de manipuler cette même structure de données.

L'interface est indépendante du langage de programmation utilisé, les *listes* en python et les *arrays* en javascript disposent de méthodes similaires (insérer ou retirer un élément,

déterminer si la liste ou l'array est vide, déterminer la longueur de la listes …). Ces deux ***Implémentations*** reposent sur la même interface.

L'implémentation d'une structure de données est la mise en œuvre pratique dans un langage de programmation.

Ces deux notions sont fondamentales, et seront approfondies dans la suite du cours.

2. Les files

2.1 Définition

Les *files* (ou *queue* en anglais) sont une autre structure de données linéaires. Il s'agit d'une liste d'éléments qui ne peut être manipulée qu'en ajoutant un élément à la fin de la file ou en retirant un élément au début de la file. On parle alors d'un fonctionnement *FIFO* (pour *First In – First Out*).

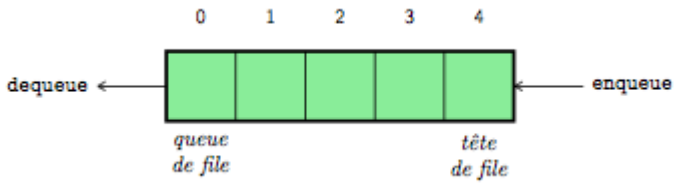

Figure 3.1 – Illustration d'une file.

Les files comme leur nom l'indique peuvent être vues comme des files d'attente, la première personne arrivée dans la file est la première à sortir de la file. Une personne peut s'ajouter à la fin de la file et personne ne peut s'insérer au milieu de la file.

2.2 Interface d'une file

La définition de la file nous donne un aperçu de son interface. En effet nous pouvons préciser un ensemble de méthodes propres aux files à partir de la définition :

- Une méthode qui nous indique si la file est vide,
- Une méthode qui nous renvoie le nombre d'éléments dans la file,
- Une méthode qui affiche les éléments de la file de la queue vers la tête,
- Une méthode qui nous permet d'ajouter un élément en tête de file,
- Une méthode qui permet de retirer l'élément en queue de file.

2.3 Détails d'implémentation

En python, il existe une implémentation des files disponibles via le package *collections*.

La documentation officielle de python nous fournit l'exemple suivant :

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now
leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now
leaves
'John'
>>> queue                           # Remaining queue in order of
arrival
deque(['Michael', 'Terry', 'Graham'])
```

Cependant, si nous souhaitons approfondir le sujet et réaliser nous même une implémentation d'une file, nous pouvons définir une classe File :

```
class File:
    def __init__(self):
        """Initialise la file comme une file vide."""
        self.elements = []
```

Nous pouvons ensuite ajouter à cette classe une méthode pour déterminer si la file est vide, une méthode pour déterminer le nombre d'éléments dans la file et une méthode pour afficher les éléments de la file.

```
    def est_vide(self):
        """Renvoie True si la liste est vide, False sinon."""
        if(len(self.elements) == 0):
            return True
        else:
            return False

    def nb_elements(self):
        """Renvoie le nombre d'éléments de la file. """
        return len(self.elements)

    def afficher(self):
```

```
        """Affiche de gauche à droite les éléments de la file, de la queue
        de la file vers la tête. La tête est alors l'élément
        affiché le plus à droite. Les éléments sont séparés par une
        virgule. Si la file est vide la méthode affiche « file
        vide »."""
        if(self.est_vide()):
            print("file vide")
        else:
            affichage = ", ".join(self.elements)
            print(affichage)
```

Enfin, nous pouvons ajouter deux méthodes pour manipuler la file, une méthode *enfiler* qui permet d'ajouter un élément à la file et une méthode *défiler* qui permet de retirer un élément de la file :

```
    def enfiler(self, e):
        """Ajoute l'élément e sur le sommet de la file,
        ne renvoie rien."""
        self.elements.append(e)

    def defiler(self):
        """Retire l'élément au sommet de la file et le renvoie."""
        return self.elements.pop(0)
```

Nous pouvons ensuite tester notre implémentation de la file :

```
file = File()
file.est_vide() # True
file.enfiler("3")
file.est_vide() # False
file.enfiler("4")
file.enfiler("7")
file.nb_elements() # 3
file.afficher() # 3, 4, 7
file.defiler() # 3
file.afficher()# 4, 7
```

3. Les piles

3.1 Définition

Les *piles* (ou *stack* en anglais) constituent elles aussi une autre structure de données linéaires. Il s'agit d'une liste d'éléments qui ne peut être manipulée qu'en ajoutant ou en retirant un élément au sommet de la pile. On parle alors d'un fonctionnement *LIFO* (pour *Lirst In – First Out*).

Figure 3.2 – Illustration d'une pile.

Comme pour les files, les piles ont un nom assez évocateur puisqu'elles peuvent être vues comme des piles d'assiettes ou des piles de t-shirt par exemple.

3.2 Interface d'une pile

La définition de la pile nous donne un aperçu de son interface. En effet, nous pouvons préciser un ensemble de méthodes propres aux piles à partir de la définition :

- Une méthode qui nous indique si la pile est vide,
- Une méthode qui nous renvoie le nombre d'éléments dans la pile,
- Une méthode qui affiche les éléments de la pile du fond vers le sommet,
- Une méthode qui nous permet d'ajouter un élément au sommet,
- Une méthode qui permet de retirer l'élément au sommet.

3.3 Détail d'implémentation

En python, le comportement d'une pile peut être reproduit en utilisant les méthodes append et pop d'une liste.

La documentation officielle de python nous fournit l'exemple suivant :

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
```

```
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Comme pour les files, nous pouvons approfondir le sujet en réalisant notre propre implémentation d'une pile. De la même manière nous pouvons créer une classe Pile qui possède une méthode *est_vide*, une méthode *nb_elements* et une méthode *afficher* :

```
class Pile:
    def __init__(self):
        """Initialise la pile comme une pile vide."""
        self.elements = []

    def est_vide(self):
        """Renvoie True si la liste est vide, False sinon."""
        if(len(self.elements) == 0):
            return True
        else:
            return False

    def nb_elements(self):
        """Renvoie le nombre d'éléments de la pile. """
        return len(self.elements)

    def afficher(self):
        """Affiche de gauche à droite les éléments de la pile, du fond
        de la pile vers son sommet. Le sommet est alors l'élément
        affiché le plus à droite. Les éléments sont séparés par une
        virgule. Si la pile est vide la méthode affiche « pile
        vide »."""
        if(self.est_vide()):
```

```
            print("pile vide")
        else:
            affichage = ", ".join(self.elements)
            print(affichage)
```

Enfin, nous ajoutons deux méthodes :

- *Empiler* qui permet d'ajouter un élément à la pile
- *Depiler* qui permet de retirer l'élément au sommet de la pile

```
    def empiler(self, e):
        """Ajoute l'élément e sur le sommet de la pile,
        ne renvoie rien."""
        self.elements.append(e)

    def depiler(self):
        """Retire l'élément au sommet de la pile et le renvoie."""
        return self.elements.pop()
```

Voici un exemple de l'utilisation de cette classe :

```
pile = Pile()
pile.est_vide() # True
pile.empiler("5")
pile.est_vide() # False
pile.empiler("2")
pile.empiler("8")
pile.nb_elements() # 3
pile.afficher() # 5, 2, 8
pile.depiler() # 8
pile.afficher() # 5, 2
```

Exercices non à soumettre

Pour ces exercices, nous reprendrons les implémentations de la pile et de la file réalisées lors du cours.

Exercice 1

1. En utilisant l'implémentation de la classe File, donner une suite d'instruction permettant de créer une file nommée ***file1*** et contenant les valeurs ***5, 8 et 3*** dans cet ordre. La dernière instruction doit permettre d'afficher « 5, 8, 3 ».
2. A partir de la file ***file1***, donnez le résultat de la suite d'instruction suivante :

```
element = file1.defiler()
file1.enfiler("7")
file1.afficher()
```

Exercice 2

Pour cet exercice nous utilisons une pile nommée ***pile1***. Cette pile contient les valeurs suivantes ***7, 3 et 6***.

1. Ecrivez une suite d'instructions qui permet de retirer l'élément au sommet de la pile ***pile1*** et qui permet de l'ajouter dans une nouvelle pile ***pile2***.
2. Ecrivez la fonction *transferer(pile1)* qui permet de transférer l'ensemble des éléments de la pile ***pile1*** dans une nouvelle pile ***pile2*** et qui retourne cette nouvelle pile.
3. Donnez le résultat de la suite d'instructions suivante :

```
pile1.depiler()
pile3 = transferer(pile1)
pile3.empiler('1')
pile3.afficher()
```

●●●●●