Structures de données linéaires 1 : les listes Python

En Python, une liste est une structure de donnée linéaire : elle contient différentes données que l'on peut représenter sur une ligne.
On peut accéder à n'importe quelle donnée d'une liste grâce à son **indice** : on dit que les listes Python sont **indexées**.
On peut aussi supprimer des données ou en rajouter, à la différence des **tableaux** qui sont des **structures linéaires indexées de taille fixe**.

En informatique, on utilise différents modèles de structure linéaire, notamment les tableaux, les listes chainées, les piles et les files. On peut aussi « customiser » un modèle en lui donnant les fonctions de plusieurs modèles basiques. C'est le cas des listes Python qui combine le concept historique de liste et la notion de tableau.

Structures mutables et structures immuables
En Python, une liste est mutable : on peut modifier ses données.
Un tuple est immuable : on ne peut pas le modifier, mais on peut en créer un nouveau.

Exemple 1
On veut doubler la valeur de chaque nombre d'une liste ou d'un tuple.
On peut conserver la même liste en changeant les valeurs :

```
for i in range(len(lst)):
    lst[i] = lst[i] * 2
```

On doit créer un nouveau tuple :

```
ntpl = tuple(tpl[i]*2 for i in range(len(tpl)))
```

lst est la même liste, ntpl est un nouveau tuple… que l'on aurait pu appeler tpl, écrasant l'ancien tuple.

Exemple 2
On veut rajouter une valeur à une liste :

```
lst.append(nouvelle_valeur)
```

On a ajouté une valeur à la même liste. C'est une opération en temps constant.

On veut rajouter une valeur à un tuple :

```
tpl = tpl + (nouvelle_valeur,)
```

On vient de créer un nouveau tuple, en recopiant toutes les valeurs du tuple précédent et en rajoutant une valeur. C'est une opération en temps linéaire : on a dû recopier tout l'ancien tuple, même si le nouveau tuple a le même nom que l'ancien.
Plutôt que « additionner » (ou concaténer) les tuples, une méthode classique consiste à les imbriquer :

```
tpl = (tpl, nouvelle_valeur)
```

Cette instruction crée un nouveau tuple constitué de l'ancien tuple et de la nouvelle valeur.
Un tuple créé en imbriquant des tuples peut ressembler à ((((1,), 2), 3), 4) ou à
(1, (2, (3, (4,)))).

Vous avez travaillé en première sur les tableaux et les listes Python.
On présente dans ce cours trois structures séquentielles : les listes chainées immuables, les piles mutables et les files mutables.

Structures de données linéaires 2 : les listes chainées,
appelées simplement listes dans ce chapitre

1. Description

On s'intéresse au type de liste popularisé par le langage de programmation Lisp (**Lis**t **p**rocessing), un des premiers langages de programmation de haut niveau.

Il s'agit de **listes chainées immuables**. Ce sont des structures de données linéaires avec un accès direct uniquement à l'élément de tête. On ne peut accéder à un élément de la liste qu'en passant par l'élément précédent, d'où le nom de structure séquentielle. Leur taille est variable, contrairement aux tableaux.

Plus précisément, une **liste** est caractérisée par un ensemble de **cellules.** Chaque **cellule** contient une **valeur** (la **tête**) et un **lien** vers la cellule suivante (la **queue**). Une liste peut être vide.

Une liste simplement chaînée, composée de trois éléments ayant respectivement la valeur : 12, 99 et 37.

2. Définition récursive

Une **liste** est soit une **liste vide**, soit un **couple** (*e*, *L*), appelé cellule, avec :

- *e* l'étiquette de la cellule (la valeur stockée)
- *L* le reste de la liste

Cette structure n'existe pas en Python, ni en C, ni en Java et doit être implémentée.

3. Définition par l'interface

On peut caractériser une structure de type liste par son interface, son jeu d'instructions. Ainsi, une structure de type liste disposera généralement des fonctions suivantes.

- liste_vide() : renvoie une liste vide
- ajoute(*lst*, *element*) : renvoie une copie de la liste *lst* en ayant ajouté *element* en tête de liste. Cette fonction est parfois appelée cons, comme constructeur.
- est_vide(*lst*) : renvoie un booléen indiquant si la liste *lst* est vide.
- valeur(*lst*) : renvoie la première valeur de la liste *lst*, à partir d'une suite lst = (e, lst2) renvoie la valeur e.
  Cette fonction peut être appelée premier_element ou tete (head).
- suite(*lst*) : à partir d'une suite lst = (e, lst2) renvoie la liste lst2.
  Si la fonction précédente est appelée tete, celle-ci est appelée queue (tail).

Ces fonctions, indispensables, sont dites fonctions primitives. On trouve parfois d'autres fonctions : longueur, dernier_element, inserer_element, cherche, min, max, tri… la plupart peuvent être programmées à partir des primitives.

Exemple - Implémentation de longueur

```
def longueur(lst):
    n = 0
    while not est_vide(lst):
        lst = suite(lst)
        n += 1
    return n
```

Cette fonction a une complexité en O(*n*) où *n* est la longueur de la liste. Dans la mesure du possible, on l'implémentera différemment. On rajoute généralement une variable, à maintenir à jour, contenant le nombre d'éléments de la liste. Plus besoin alors de parcourir toute la liste pour connaitre sa longueur !

Exercice
Écrire la fonction dernier_element en utilisant les fonctions primitives nécessaires.

4. Implémentation fonctionnelle (récursive)

```
def liste_vide():
    return None

def ajoute(lst, e):
    return (e, lst)

def valeur(lst):
    return lst[0]

def suite(lst):
    return lst[1]

def est_vide(lst):
    return lst is None
```

D'autres implémentations sont proposées en exercice.

Structures de données linéaires 3 : les piles

1. Description

Une **pile** (en anglais **stack**) est une structure de données linéaire fondée sur le principe « dernier arrivé, premier sorti » (**LIFO** pour **Last In, First Out**), ce qui veut dire que les derniers éléments ajoutés à la pile seront les premiers à être récupérés.
Le fonctionnement est celui d'une pile d'assiettes : on ajoute des assiettes sur la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée. Comme avec les listes, on a un accès direct qu'au premier élément de la pile, le sommet.
On utilisera des **piles mutables**.

Applications

Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile en cliquant le bouton « Précédent ».
La fonction « Annuler » (Undo) d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

2. Définition par l'interface

Une pile est une structure de donnée « LIFO » munie des fonctions primitives suivantes :

- pile_vide() : renvoie une pile vide
- est_vide(pile) : renvoie un booléen indiquant si la pile est vide
- empiler(pile, element) : rajoute un élément à la pile (push en anglais)
- depiler(pile) : enlève un élément à la pile et le renvoie (pop)

Remarque

Pour lire le sommet de la pile sans modifier la pile, on doit le dépiler et le rempiler :

```
def lire_sommet(pile):
    sommet = depiler(pile)
    empiler(pile, sommet)
    return sommet
```

3. Implémentation objet

On propose deux types d'implémentation objet des piles, une version à une classe et une version à deux classes.

La version à une classe est plus simple, elle peut être suffisante, mais les puristes préfèrent la version à deux classes qui colle davantage au modèle théorique proche des listes dans lequel une pile est soit une cellule, soit une pile vide.

Dans certaines implémentations, la pile vide est désignée par **None**. Ces implémentations ne sont pas adaptées à la programmation objet puisque dans ce cas, la pile vide n'est pas une instance de Pile et on ne peut pas lui appliquer les méthodes de la classe Pile (empiler, est_vide...).

Avec une classe	Avec deux classes

class Pile:
  def __init__(self, valeur=**None**, suivant=**None**):
  self.valeur = valeur
  self.suivant = suivant
 	
  def empiler(self, valeur):
  self.suivant = Pile(self.valeur, self.suivant)
  self.valeur = valeur
 	
  def depiler(self):
  valeur = self.valeur
  self.valeur = self.suivant.valeur
  self.suivant = self.suivant.suivant
  return valeur
 	
  def est_vide(self):
  return self.suivant is **None**
	class Cellule:
	  def __init__(self, valeur, suivant):
	  self.valeur = valeur
	  self.suivant = suivant
	
	class Pile:
	  def __init__(self, c=**None**):
	  self.cellule = c
	
	  def empiler(self, valeur):
	  self.cellule = Cellule(valeur, self.cellule)
	
	  def depiler(self):
	  valeur = self.cellule.valeur
	  self.cellule = self.cellule.suivant
	  return valeur
	
	  def est_vide(self):
	  return self.cellule **is None**

La pile vide s'obtient avec Pile(**None**) dans le modèle à deux classes ou Pile(**None**, **None**) dans le modèle à une classe. Comme **None** est un argument par défaut, une pile vide s'obtient simplement avec pile = Pile(). On peut créer une fonction renvoyant une pile vide :

```
def pile_vide():
    return Pile()
```

4. Utilisation rapide des piles en Python

Un des objectifs de ce cours est d'apprendre à construire soi-même ses structures de données, mais en cas de besoin dans d'autres contextes, pour simuler les piles, le plus simple est d'utiliser les listes Python existantes.

On utilise les fonctions suivantes :

```
def pile_vide():
    return []

def empiler(valeur, pile):
    pile.append(valeur)

def depiler(pile):
    return pile.pop()

def est_vide(pile):
    return pile == []
```

Exemple

pile = [1, 2, 3]
depiler(pile) renvoie 3 et modifie pile qui devient égale à [1, 2].
empiler(pile, 4) ne renvoie rien mais modifie pile qui devient égale à [1, 2, 3, 4].

Structures de données linéaires 4 : les files

1. Description

Une **file** (**queue** en anglais) est une structure de données basée sur le principe du Premier entré, premier sorti (**FIFO** : **First In, First Out**), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à être récupérés. Le fonctionnement ressemble à une file d'attente : les premières personnes à arriver sont les premières personnes à sortir de la file.
On accède en écriture, en ajout, à la queue de la file et en lecture à la tête de la file. La lecture d'un élément le supprime de la file.
On utilisera des **files mutables**.

Applications
- mémoriser temporairement des transactions qui doivent attendre pour être traitées
- mémoire tampons (buffers)
- serveurs d'impression (qui traitent les requêtes dans l'ordre dans lequel elles arrivent)

2. Définition par l'interface

Une file peut être définie par les fonctions primitives suivantes :
- file _vide() : renvoie une file vide
- est_vide(file) : renvoie un booléen indiquant si la file est vide
- enfiler(file, element) : rajoute un élément en queue de file (push)
- defiler(file) : enlève un élément en tête de file et le renvoie (pop)

Exercice
Écrire une fonction longueur qui détermine la longueur d'une file sans la modifier.

3. Implémentation objet

On peut utiliser une implémentation similaire à celle des piles, mais si defiler renvoie l'élément de tête, enfiler doit placer le nouvel élément à la queue de la file. Pour cela, on doit remonter toute la file. L'opération enfiler est alors en temps linéaire.

On peut envisager le contraire, avec enfiler en temps constant et defiler en temps linéaire.

Avec un attribut désignant la tête de la file et un attribut désignant la queue, les deux méthodes enfiler et defiler sont alors en temps constant.

Implémentation tête à une classe

```
class File:
    def __init__(self, valeur=None, suivant=None):
        self.valeur = valeur
        self.suivant = suivant

    def enfiler(self, valeur):
        if self.est_vide():
            self.suivant = File()
            self.valeur =  valeur
        else:
            cel = self
            while not cel.suivant.est_vide():
                cel = cel.suivant
            cel.suivant = File(valeur, File())

    def defiler(self):
        valeur = self.valeur
        self.valeur = self.suivant.valeur
        self.suivant  = self.suivant.suivant
        return valeur

    def est_vide(self):
        return self.suivant is None
```

Implémentation tête-queue à une classe

```
class File:
    def __init__(self, valeur=None, suivant=None):
        self.valeur = valeur
        self.suivant = self.queue = suivant

    def enfiler(self, valeur):
        if self.est_vide():
            self.valeur = valeur
            self.queue = File()
            self.suivant = self.queue
        else:
            self.queue.valeur = valeur
            self.queue.suivant = File()
            self.queue = self.queue.suivant

    def defiler(self):
        valeur = self.valeur
        self.valeur = self.suivant.valeur
        self.suivant = self.suivant.suivant
        return valeur

    def est_vide(self):
        return self.suivant is None
```

Implémentation tête à deux classes

```
class Cellule:
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class File:
    def __init__(self, c=None):
        self.cellule = c

    def enfiler(self, valeur):
        if self.est_vide():
            self.cellule =  Cellule(valeur, None)
        else:
            cel = self.cellule
            while cel.suivant != None:
                cel = cel.suivant
            cel.suivant = Cellule(valeur, None)

    def defiler(self):
        valeur = self.cellule.valeur
        self.cellule = self.cellule.suivant
        return valeur

    def est_vide(self):
        return self.cellule is None
```

Implémentation tête-queue à deux classes

```
class Cellule:
    def __init__(self, valeur, suivant):
        self.valeur = valeur
        self.suivant = suivant

class File:
    def __init__(self, c=None):
        self.tete = c
        self.queue = c

    def enfiler(self, valeur):
        if self.est_vide():
            self.queue = Cellule(valeur, None)
            self.tete = self.queue
        else:
            self.queue.suivant = Cellule(valeur, None)
            self.queue = self.queue.suivant

    def defiler(self):
        valeur = self.tete.valeur
        self.tete = self.tete.suivant
        return valeur

    def est_vide(self):
        return self.tete is None
```

Remarque
Dans l'implémentation tête-queue, la méthode defiler n'agit pas sur la queue, donc quand on défile le dernier élément, il va rester présent dans la queue. Cela n'altère pas le fonctionnement du code, mais si on veut retrouver queue = **None** après avoir enfilé/défilé sur une file vide, on peut rajouter dans la méthode defiler :

```
if tete is None:
    queue = None
```

4. Utilisation rapide des files en Python

4.1. Utilisation des listes Python

En Python, pour accéder rapidement à une structure de type liste, on peut utiliser les listes :
file = [] crée une file vide.
file.append(valeur) permet d'enfiler une valeur
file.pop(0) permet de défiler.
Une autre option est d'utiliser file.insert(0, valeur) pour enfiler et file.pop() pour défiler.
Dans le premier cas, enfiler est en temps constant et défiler en temps linéaire.
Dans le deuxième cas, enfiler est en temps linéaire et défiler en temps constant.
En effet, pop(0) ou insert(0) décalent tous les éléments de la liste.

4.2. Utilisation des deque

Le type deque à importer de la bibliothèque collections fournit une structure de file avec enfiler et defiler en temps constant.

```
from collections import deque
file = deque([]) : créer une file vide
file.append(s) : enfiler s
file.popleft() : défiler
```

Après **if** ou **while**, une file vide est interprétée comme False, une file non vide comme True. Ainsi, une boucle « **while** file: » va s'exécuter tant que la file n'est pas vide.

Il existe aussi les méthodes appendleft() et pop() pour ajouter à gauche et supprimer à droite.

Structures de données linéaires 5 :
implémentation fonctionnelle des piles et des files

En programmation fonctionnelle, les objets sont immuables.
Ainsi, les fonctions *empiler* et *depiler* ne modifient pas la pile en cours mais renvoient une nouvelle pile.

1. Piles immuables

Une pile immuable est en fait une liste immuable :

- empiler renvoie une copie de la pile avec un élément en plus en tête de pile,
- depiler renvoie le couple (tête de la pile, suite de la pile).

La fonction empiler remplace la fonction ajoute ; la fonction depiler remplace les fonctions valeur et suite.

Exemple
pile = (3, (2, (1, **None**)))
depiler(pile) renvoie 3 et (2, (1, **None**)).
empiler(pile, 4) renvoie (4, (3, (2, (1, **None**)))).

Constructeurs

```
def pile_vide():
    return None

def empiler(pile, valeur):
    return (valeur, pile)
```

Exercice
Programmer les fonctions depiler, est_vide, longueur (ou hauteur).

2. Files immuables

En programmation fonctionnelle, les files sont immuables :

- enfiler renvoie une copie de la file avec un élément en plus en queue de file,
- defiler renvoie le couple (tête de la file, suite de la file).

Exemple
file = (1, (2, (3, **None**)))
defiler(file) renvoie 1 et (2, (3, **None**)).
enfiler(file, 4) renvoie (1, (2, (3, (4, **None**))))).

Constructeur

```
def file_vide():
    return None
```

Exercice
Écrire les fonctions enfiler, defiler, est_vide, copier, minimum.