Partie 1

Types abstraits

La résolution d'un problème par un programme nécessite :

- de définir les données du problème
- d'utiliser des instructions du langage sur ces données

Souvent, la résolution du problème demande d'arranger ces données dans un *type abstrait*, afin d'avoir une résolution plus efficace.

Un type abstrait est défini par son *interface*, qui est indépendante de son *implémentation*.

Les types abstraits sont des types *structurés*. On a déjà vu des types *structurés* natifs : les types séquences de texte (string), les types séquentiels (listes, tuples), et les mappages (dictionnaires).

Un type abstrait est caractérisé par une *interface* de programmation qui permet de manipuler les données de ce type. Sans pour autant avoir connaissance du contenu des fonctions proposées par l'interface.

Spécifier un type abstrait, c'est définir son *interface*, sans prejuger de la façon dont ses opérations sont implémentées.

Implémenter, c'est fournir le code de ces opérations. Plusieurs implémentations peuvent correspondre à la même spécification. On verra dans ce chapitre la programmation dans un style fonctionnel. Il viendra plus tard la programmation par objet.

On verra dans un premier temps, deux exemples de types abstraits : Les *listes chainées* et les *tableaux*. Dans un chapitre ultérieur, nous verrons les types abstraits *Piles*, *Files*, et *Graphes*.

Partie 2

Listes chaînées

Une liste chaînée est une séquence ordonnée d'éléments. L'avantage sur un tableau (la Liste native en Python), c'est qu'en parcourant la Liste chainée, quelle que soit la profondeur de parcours, celle-ci présente la même structure.

On peut donc lui associer des méthodes ou fonctions, qui seront indépendantes de cette profondeur. Que l'on pourra utiliser de la même manière à chaque niveau de la liste chainée.

{{< img src="../images/liste_classes.png" caption="classes à parcourir après le collège" >}}

{{< img src="../images/liste_classes2.png" caption="classes à parcourir après la 2nde" >}}

La *liste L* précédente représente la chaine : `"2nde" -> "1ere" -> "Term" -> "Univ`

```
1 L = ("Univ",("Term",("1ere",("2nde", ()))))
```

La liste chainée L contient 2 éléments `(tete, queue)` et queue est elle-même une liste chainée, contenant aussi `(tete, queue)`. Le dernier élément : `(tete, ())`.

On utilisera par exemple une liste chainée lorsqu'il y a une filiation, une chronologie entre les éléments.

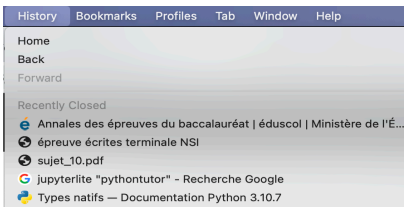

FIGURE 1 – l'historique du navigateur

2.1 L'interface d'une liste chainée

L'*interface* fournit certaines fonctions.

fonctions qui implémentent la liste chainée L	nom
créer une liste vide	`creer_liste()`
questionner si la liste est vide	`liste_vide(L)`
insérer un élément **e** en tête de liste et retourne une nouvelle liste	`inserer_tete(L,e)`
retourne l'élément de tête (premier élément)	`tete(L)`
retourne la liste privée de son premier élément (retourne donc le 2e élément)	`queue(L)`
insere un nouvel `element_a_inserer` juste avant l'élement recherché dans la liste	`insere(L,element_recherche, element_a_inserer)`
retourne une liste python avec tous les éléments de la liste chainée	`elements_liste(L)`

Le *contenu* de ces fonctions va dépendre de l'*implémentation* choisie par le programmeur.

Par exemple, si l'on choisit d'implémenter la liste chainée par un tuple, on aura pour la premiere fonction :

```
1 def creer_liste():
2   return ()
```

Alors que si l'on choisit plutôt une liste :

```
1 def creer_liste():
2   return []
```

2.2 Généralités sur les listes chainées

Une liste chainée est un objet :

- **non mutable** : on peut la modifier en partie (ajout/suppression d'un élément), mais sa copie se fait par valeur.
- **dynamique** : on peut modifier sa taille après création, par exemple en faisant : `L = inserer(L, e)`, mais cela va créer un nouvel objet.
- les éléments ont une relation d'ordre entre eux.
- on peut atteindre un élément au rang i en temps proportionnel à i, avec une boucle par exemple.

Mémoire : Contrairement aux tableaux, les éléments d'une liste chaînée ne sont pas placés côte à côte dans la mémoire. Chaque case pointe vers une autre case en mémoire, qui n'est pas nécessairement stockée juste à côté.

L'**interface** d'une liste chainée doit aussi proposer **l'insertion d'un élément au rang i, en temps constant**. (Intercaler cet élément entre 2 éléments de la liste). Ceci ne peut pas être réalisé avec l'implémentation vue plus haut.

Nous verrons d'autres implémentations pour ce type abstrait.

Attention : les listes chaînées et les Listes Python sont différentes, il ne s'agit pas des mêmes objets.

Partie 3

Tableaux statiques

3.1 Présentation

Les tableaux se comportent de manière très similaire aux listes, sauf que les types d'objets qui y sont stockés sont limités. (Array)

Un tableau peut être représenté en Python par un tuple contenant 2 éléments :

- la liste des valeurs, de dimension fixe. On mettra `None` pour les valeurs non renseignés.
- la valeur de la taille de la liste.

Par exemple :

```
1 T = ([6, 7, 8, None, None], 5)
```

C'est alors un objet qui est :

- **statique** : sa taille ne varie pas une fois celui-ci créé
- **non mutable** : mais avec un élément qui est lui *mutable* :

On ne peut pas modifier `T[0]` ou `T[1]`. Ce sont les éléments d'un tuple.

Mais on peut modifier en partie `T[0]`, par exemple avec une instruction du genre : `T[0][i] = x`. La copie de `T[0]` se fait par reference.

Un tableau peut servir à implémenter une grille de notes par exemple :

	C1	C2	C3
Kyle	6	7	8
Sean	10	0	10
Quentin	10	0	14
Zinedine	15	0	12

FIGURE 2 – Tableau de notes

Supposons, pour simplifier, que le tableau ne contient que les notes de Kyle :

	C1	C2	C3	C4	C5
Kyle	6	7	8	Non noté	Absent

On peut représenter cet ensemble de notes par le tableau :

```
T = ([6, 7, 8, None, None], 5)
```

3.2 L'interface d'un tableau

fonctions qui implémentent un tableau T (Array)	nom
taille du tableau	`taille(T)`
demander l'élément au rang i	`element(T,i)`
remplacer l'élément au rang i par e	`remplacer(T,i,e)`

3.3 Tableaux dynamiques et tableaux statiques

Les tableaux vus ci-dessus sont des tableaux *statiques* : leur taille ne peut pas être modifiée. Dans le cas où l'on ait besoin d'agrandir le tableau, il faut le copier dans un nouveau tableau, plus grand.

Python implémente naturellement un autre type de tableau, que l'on appelera *dynamique* : Les *Listes Python*. Ce problème de dimension n'apparait pas dans les Listes Python, qui apportent de surcroit des méthodes bien pratiques comme `append` et `pop`.

Partie 4

La liste python dynamique (type list)

Le type `list` python est dynamique. Il peut lui aussi implémenter une Liste chainée ou bien un Tableau.

Pour un Tableau, si on l'implémente avec le type `list`, la taille de celui-ci ne devra pas être modifiée à la fin du traitement.