Sujet n°3

Algorithmique & Programmation

Structures de données linéaires : suite et fin

`Temps de réalisation`[1]`: 3h (+1h ttp)`

L'objectif du TP°3 est de (◇ = algorithmique, ◦ = programmation) :

- ◇ se familiariser avec les listes chaînées, dans une version avec double chaînage
- ◇ évaluer le coût des opérations sur les listes chaînées
- ◇ comparer les listes et les tableaux
- ◇ concevoir une implémentation alternative des piles et des files d'attente
- ◇ appréhender les files d'attente à deux extrémités
- ◇ résoudre des problèmes en mobilisant des structures de données linéaires et leurs opérations
- ◦ découvrir les itérateurs et les générateurs en Python

● « obligatoire » ● « au choix » ● « supplémentaire » ● « en autonomie »

Ex.	01	02	03	04	05	06	07	08	09
	●	●	●	●	●	●	●	●	●

Objectif : **tout** (●) et **la moitié** (●)

ProTip : il est super-méga-extra-recommandé de se munir d'un papier et d'un crayon pour dessiner des tonnes de cases et de flèches qui représentent les listes chaînées en mémoire et leur évolution en fonction des scénarios étudiés.

Table des matières

1. Estimé pour un·e étudiant·e faisant preuve d'une efficacité « raisonnable »: (i) points du cours révisés, (ii) mise au travail en -5mn, (iii) recherche autonome dans les supports de cours et les ressources en ligne, et (iv) absence de distraction (téléphone portable, notifications, bavardage, etc.).

Le contenu intégral des exercices du TP3 doit figurer dans un paquetage `tp3` du *cahier d'exercices*.

1 Les structures de données

1.1 Les listes chaînées

Elles peuvent être vues comme une alternative aux tableaux pour l'implémentation de suites finies. Néanmoins, elles se distinguent des tableaux par (i) leur allocation dynamique, et (ii) leur mode d'accès proprement séquentiel. Il existe en pratique de multiples variantes de listes chaînées. Nous explorons l'une d'entre elles dans cette section.

Exercice 1 ● : Définition d'une liste doublement chaînée

Dans cet exercice, vous allez créer à l'aide de la construction `@dataclass` les structures de données et les opérations nécessaires à la gestion d'une liste de nombres entiers.

L'implémentation a lieu dans le fichier `tp3/linkedlist.py`, dont un squelette est fourni.

QUESTION 1.1. Créer les types `LinkedList` et `Cell` pour représenter une **liste doublement chaînée** (*Doubly Linked List*) de maillons[2] (*Cell*) contenant des nombres entiers.

Astuce : *Il est intéressant de considérer un maillon fictif, appelé **sentinelle**, comme point d'entrée dans la liste. Celui-ci est créé et détruit avec la liste. Il se place juste avant la véritable tête de liste si la liste est non vide. Dans le cas d'une liste doublement chaînée, il est également lié à la queue de liste. Le maillon sentinelle évite ainsi la maintenance de deux références nommées et offre « gratuitement » une forme de circularité ! En pratique, il se substitue avantageusement à toutes les références à `None` dans le code. Il permet également d'économiser un grand nombre de cas aux bornes – en tête et queue de liste –. Bien qu'elle soit recommandée, vous avez le choix d'implémenter la liste doublement chaînée avec ou sans la sentinelle.*

Remarque : *La définition du type `Cell` requiert le symbole `Cell` lui-même : elle est récursive. En Python, à partir de la version 3.7, ce mécanisme est disponible grâce à l'ajout en tête de fichier de la ligne suivante :*

2. Les maillons représentent les « places » dans la liste.

```python
from __future__ import annotations
```

En marge : *Il existe une version récursive de la définition de suites finies : $\ell = (h, t)$ où h est l'élément de tête, et t est la sous-liste de queue. Ce point de vue est notamment très utilisé dans les langages de programmation fonctionnelle. Il reviendrait ici à promouvoir le type `Cell` en type de liste à part entière.*

QUESTION 1.2. Écrire et tester[3] le « constructeur », et le test de liste vide :

```python
def ll_new(init_l: list[int] | None = None) -> LinkedList
def ll_is_empty(l: LinkedList) -> bool
```

Une liste vide est une liste sans maillon. Le constructeur de liste chaînée (`ll_new`) admet un paramètre `init_l` que l'on ignore pour le moment : il est initialisé à `None` par défaut et ne fait l'objet d'aucun traitement dans le corps de la fonction *new*.

QUESTION 1.3. Ajouter 3 « observateurs », i.e., des fonctions d'accès aux informations de la liste :

```python
def ll_head(l: LinkedList) -> Cell
def ll_tail(l: LinkedList) -> Cell
def ll_len(l: LinkedList) -> int
```

qui donnent respectivement, le premier et le dernier maillons de la liste, et le nombre d'éléments de la liste. Les fonctions *head* et *tail* produisent une erreur `IndexError` lorsque la liste est vide.

QUESTION 1.4. Écrire et tester la fonction

```python
def ll_append(l: LinkedList, item: int) -> Cell
```

qui ajoute un élément à la fin de la liste ℓ et retourne le maillon nouvellement créé. Déterminer son coût au mieux, en moyenne et au pire.

QUESTION 1.5. Grâce à la fonction *append*, il devient aisé d'initialiser la liste chaînée avec les éléments d'une liste Python `init_l` passée en paramètre. Modifier le constructeur *new* en conséquence.

Exercice 2 ● : Le long de la chaîne...

QUESTION 2.1. Munir la liste chaînée d'un itérateur de maillons, à l'aide de la fonction génératrice :

```python
def ll_iter(l: LinkedList, reverse: bool=False) -> Iterator[Cell]
```

qui permet de parcourir tous les maillons de la chaîne du début à la fin, par exemple à l'aide d'une boucle `for` :

```
>>> for c in ll_iter(l):
...     # traitement du maillon c
...     print(c)
```

3. Les jeux de test sont fournis. Le fichier `conftest.py`, qui contient du matériel partagé par tous les tests, doit figurer dans le même répertoire.

Lorsque le paramètre optionnel `reverse` est à `True`, le parcours s'effectue de la fin vers le début de la liste.

Remarque : *Une version martyre de cette fonction est fournie en commentaire dans le squelette de `tp3/linkedlist.py`. Vous n'avez qu'à la corriger et/ou la compléter.*

Remarque : *L'annexe A offre un premier aperçu de la notion d'itérateur en Python.*

QUESTION 2.2. Proposer une fonction de sérialisation des listes :

```
def ll_str(l: LinkedList) -> str
```

qui retourne la représentation sérialisée (textuelle) de la liste ℓ, principalement en vue d'un affichage sur la sortie standard (Terminal) :

```
>>> print(ll_str(l))
[1, 2, 3, 2, 1]
```

L'affichage de la liste doit reproduire l'affichage des listes natives de Python, e.g. `[1, 2, 3, 2, 1]`.

QUESTION 2.3. Écrire et tester les fonctions de recherche par valeur et par rang :

```
def ll_lookup(l: LinkedList, item: int) -> Cell | None
def ll_cell_at(l: LinkedList, i: int) -> Cell
```

La fonction *lookup* recherche la première occurrence de l'élément `item` et retourne – une référence vers – le maillon si la recherche aboutit, `None` sinon. La fonction *cell at* fournit le maillon de rang $0 \leq i \leq |\ell| - 1$. Si i n'est pas une position valide, la fonction produit une erreur `IndexError`.

Établir les coûts au mieux, en moyenne et au pire.

Exercice 3 ● : Les opérations d'édition

QUESTION 3.1. Écrire et tester la fonction

```
def ll_prepend(l: LinkedList, item: int) -> Cell
```

qui ajoute un élément au début de la liste ℓ.

Donner le coût au mieux, en moyenne et au pire.

QUESTION 3.2. Écrire et tester la fonction

```
def ll_insert(l: LinkedList, item: int, next_to: Cell) -> Cell
```

qui insère l'élément `item` *immédiatement après* le maillon `next_to`, et retourne le nouveau maillon ainsi créé.

Donner le coût au mieux, en moyenne et au pire.

QUESTION 3.3. Ajouter la fonction

```
def ll_remove(l: LinkedList, c: Cell) -> int
```

qui retire le maillon c de la chaîne et retourne la valeur entière ainsi supprimée. Pour quels coûts ?

QUESTION 3.4. **[bonus]** Écrire et tester la fonction

```
def ll_extend(l1: LinkedList, l2: LinkedList) -> None
```

qui concatène deux listes chaînées. Les données de la liste ℓ_2 sont ajoutées à la fin de la liste ℓ_1. La liste ℓ_2 *n'est pas modifiée*.

Quel sont les coûts de cette opération ?

QUESTION 3.5. Comparer et discuter les coûts relevés dans cet exercice, à ceux des opérations équivalentes sur les tableaux (cf. Sujet n°2).

1.2 Dèques, files et piles

Au cours de l'épisode précédent, nous avons vu comment implémenter les types abstraits *pile* et *file d'attente* à l'aide de tableaux et tampons circulaires. En fait, les files et les piles sont des cas particuliers de la *file d'attente à deux extrémités* mal nommée *dèque*[4].

Dans cette partie, nous allons donc nous intéresser à une implémentation alternative des piles et des files d'attente, à partir d'un nouveau type *dèque*.

Exercice 4 ● : Nouvelle implémentation des piles et files d'attente

Un dèque est une file d'attente dans laquelle il est possible d'entrer et de sortir aux deux extrémités. En cela, il combine les opérations sur les files d'attente et sur les piles.

Le fichier `tp3/deque.py` propose une implémentation du type `Deque` de file d'attente à deux extrémités pour des nombres entiers, par composition à partir des listes doublement chaînées `LinkedList`.

Il fournit également l'interface de programmation suivante :

```
def d_new() -> Deque                # "constructeur"
def d_is_empty(d: Deque) -> bool    # dèque vide ?
def d_len(d: Deque) -> int          # taille
def d_str(d: Deque) -> str          # représentation sérialisée
def d_front(d: Deque) -> int        # présente l'élément en tête de file
def d_rear(d: Deque) -> int         # présente l'élément en queue de file

# ajoute en tête (front) et en queue (rear)
def d_push_front(d: Deque, item: int) -> Deque
def d_push_rear(d: Deque, item: int) -> Deque

# supprime en tête (front) et en queue (rear)
def d_pop_front(d: Deque) -> Deque
def d_pop_rear(d: Deque) -> Deque
```

QUESTION 4.1. Compléter et tester l'implémentation du type `Deque` du module `tp3.deque`. En particulier, les fonctions *pop front* et *pop rear* sont à écrire.

QUESTION 4.2. Proposer, dans le fichier `tp3/stack.py`, une implémentation alternative du type personnalisé `Stack` à partir de `Deque`. Tous les opérateurs sur les piles doivent être redéfinis. Le squelette du fichier est fourni.

QUESTION 4.3. Vérifier que les programmes qui utilisent le type `tp2.stack.Stack` fonctionnent parfaitement avec cette implémentation alternative.

4. pour deque en anglais, la contraction de double-ended queue.

QUESTION 4.4. **[bonus]** Faire de même pour le type personnalisé `Queue`, dans un fichier `tp3/queue_.py`, dont le squelette est fourni.

QUESTION 4.5. Comparer les deux implémentations disponibles pour les types `Stack` et `Queue`. En particulier, analyser les coûts des opérations élémentaires.

2 Les problèmes

2.1 À résoudre avec un dèque

Exercice 5 ● : La valeur maximale d'une fenêtre glissante

Dans un fichier `tp3/slidingmax.py`.

Étant donné une séquence d'entiers, on souhaite fournir la valeur maximale de toute sous-séquence de k entiers consécutifs. Il s'agit en pratique de calculer un agrégat – le `max` – d'une fenêtre glissante de taille k sur un flux de données.

La séquence initiale est de type `list[int]` (ou `ArrayList` ou même `LinkedList`!).

QUESTION 5.1. Écrire une première version très simple du programme à l'aide d'une double boucle sur les éléments de la séquence et sur ceux de la fenêtre courante.

QUESTION 5.2. Écrire une version plus astucieuse, à l'aide d'un dèque (type `Deque`) en charge de mémoriser la valeur maximale courante et les éventuelles valeurs maximales suivantes.

Indice : *Le dèque contient toujours la valeur maximale en tête et insère les valeurs suivantes par la queue. Il est à noter que toute valeur du dèque qui serait moins grande que la valeur courante n'aurait aucune chance de figurer dans les réponses (le max) des fenêtres suivantes !*

QUESTION 5.3. Comparer les deux résolutions, en termes de coût.

2.2 À résoudre avec des listes chaînées

Exercice 6 ● : Topswops

Dans le fichier `tp3/topswops.py`.

Le défi *Topswops* figure dans la liste des *Al Zimmermann's Programming Contests* et consiste à trouver la permutation des entiers de 1 à n qui maximise le nombre d'itérations nécessaires pour atteindre une permutation commençant par 1, en appliquant une opération unique : inverser la sous-séquence préfixe, de taille égale à la valeur du premier élément. On réitère jusqu'à observer la valeur 1 en première position !

Par exemple, pour $n = 6$, la permutation (**3**, 6, 5, 1, 4, 2) requiert une inversion des **3** premiers éléments –la sous-séquence (3, 6, 5) est transformée en (5, 6, 3)– dans la première itération. L'intégralité du procédé donne lieu à 10 itérations :

```
0. [3, 6, 5], 1, 4, 2
1. [5, 6, 3, 1, 4], 2
2. [4, 1, 3, 6], 5, 2
3. [6, 3, 1, 4, 5, 2]
4. [2, 5], 4, 1, 3, 6
5. [5, 2, 4, 1, 3], 6
6. [3, 1, 4], 2, 5, 6
7. [4, 1, 3, 2], 5, 6
8. [2, 3], 1, 4, 5, 6
```

```
 9. [3, 2, 1], 4, 5, 6
10. 1, 2, 3, 4, 5, 6
```

La sous-séquence entre crochets indique les éléments à inverser à chaque itération. Après la dixième itération, le chiffre 1 se trouve en première position. L'algorithme s'arrête. Donc pour $n = 6$, il existe une permutation, $(3, 6, 5, 1, 4, 2)$, qui génère 10 itérations du mécanisme proposé. Le défi est de trouver, pour n donné, la permutation qui génère le maximum d'itérations !

QUESTION 6.1. Pour commencer, il faut être capable d'inverser les éléments dans une liste. Ajouter à la bibliothèque d'opérations sur les listes doublement chaînées, une fonction d'inversion des k premiers éléments :

```
def ll_reverse(l: LinkedList, k: int = 0) -> None
```

Remarque : *Il est possible soit d'inverser les valeurs des maillons, soit d'inverser les maillons eux-mêmes. C'est cette dernière approche qui est préconisée.*

QUESTION 6.2. Ensuite, nous avons besoin d'une fonction qui énumère une à une, toutes les permutations d'une liste doublement chaînée :

```
def permutations(ll: LinkedList) -> Iterator[LinkedList]
```

Écrire cette fonction génératrice.

Remarque : *L'énumération des permutations s'écrit aisément de manière récursive : l'étape d'induction considère successivement l'insertion de l'élément de tête à toutes les positions de chaque permutation du reste de la liste.*

QUESTION 6.3. Résoudre le problème *Topswops* par énumération exhaustive des permutations, pour $n \in [\![4, 9]\!]$.

Remarque : *La solution a été trouvée pour $n = 97$. N'essayez pas !*

3 Pour aller plus loin…

3.1 Les listes à enjambements

Les listes à enjambements (ou *skip list* en anglais) sont des listes chaînées triées multi-couches. La première couche consiste en une liste chaînée usuelle dont les éléments ont été ordonnés, tandis que chaque couche supérieure offre un « raccourci » pour la liste immédiatement inférieure :

```
couche 3:   A
couche 2:   A---C-------G
couche 1:   A---C-D-----G---I
couche 0:   A-B-C-D-E-F-G-H-I-J
```

En pratique, les couches supérieures ne sont constituées que de références vers les maillons de la première couche. La recherche dans une liste à enjambements s'opère de la couche la plus haute vers la plus basse, en se déplaçant au plus près de la valeur recherchée dans chaque couche. Par exemple, le chemin de recherche de la valeur H est :

```
   A[3]
-> A[2] -> C[2] -> G[2]
                -> G[1]
                -> G[0] -> H[0]
```

L'insertion procède de la même manière pour établir le rang du nouvel élément au sein de la liste ordonnée. On insère toujours l'élément dans la première couche. La décision de faire figurer un élément dans la couche supérieure est totalement aléatoire ! Les *skip list* sont en effet des structures probabilistes. Le paramètre clé d'une structure de *skip list* est donc la probabilité p de figurer dans la couche supérieure. En pratique, on utilise souvent la valeur $p = 0,5$ qui suggère que la moitié des éléments d'une couche est supposée constituer la couche supérieure. Le nombre de couches est alors variable et déterminé par les tirages aléatoires successifs lors de l'insertion de nouveaux éléments. Il est toutefois d'usage de fixer une borne supérieure ℓ au nombre de couches.

Pour une présentation détaillée des *skip list*, vous pouvez consulter ce diaporama, ou commencer simplement par l'entrée Wikipédia. Pour les plus hardis, il est possible de consulter l'article original de W. Pugh (1990) qui présente les *skip list*.

Exercice 7 ● : Première implémentation

QUESTION 7.1. Implémenter un type `SkipList` de nombres entiers, muni de ses opérations élémentaires : `new`, `is_empty`, `insert`, `delete`, `lookup`.

QUESTION 7.2. Établir les performances –complexités– des opérations sur les *skip list*, en tenant compte du paramètre p.

Exercice 8 ● : La médiane mobile

Il s'agit de calculer la valeur médiane pour chaque fenêtre glissante de taille w sur un tableau de valeurs entières de taille n.

Avec une liste à enjambements pour stocker les éléments de la fenêtre courante de manière ordonnée, le procédé requiert une recherche par index (recherche de l'élément au rang $w//2$). Or, cette recherche est a priori en $O(n)$, sauf si l'on ajoute une information de « rang relatif » avec toute référence vers un maillon.

L'exemple précédent devient :

```
couche 3:   A/0
couche 2:   A/0-----C/2-------------G/4
couche 1:   A/0-----C/2-D/1---------G/3-----I/2
couche 0:   A/0-B/1-C/1-D/1-E/1-F/1-G/1-H/1-I/1-J/1
```

Il s'interprète comme suit : le parcours de la couche 2 nous renseigne sur les rangs de A, C et G. Le rang de A est obtenu par un saut de 0 à partir du début de la liste et vaut donc 0 ! Le rang de C est alors rang(A) + 2 soit 2. Le rang de G est quant à lui rang(C) + 4 = 6. Ce procédé fonctionne également le long de chemins descendants. Par exemple, chercher l'élément de rang 7 consiste à parcourir le chemin :

```
   A[3] (0)
-> A[2] (0) -> C[2] (0+2=2) -> G[2] (2+4=6)
                            -> G[1] (6)
                            -> G[0] (6) -> H[0] (6+1=7!)
```

QUESTION 8.1. Modifier le type `SkipList` pour y intégrer l'information de rang relatif.

QUESTION 8.2. Proposer une résolution pour le calcul de la médiane mobile.

3.2 Autres problèmes

Exercice 9 ● : Algorithme de Graham

L'enveloppe convexe d'un ensemble de points est la plus petite forme convexe qui les contient tous. Pour un ensemble fini de points, cette enveloppe est un polygone. Il existe de nombreux algorithmes d'extraction d'enveloppe convexe. Celui de GRAHAM repose sur l'idée que les points d'un polygone convexe tournent dans le même sens. Une fois les points triés (ici, dans le sens trigonométrique), il suffit de supprimer ceux qui provoquent des concavités.

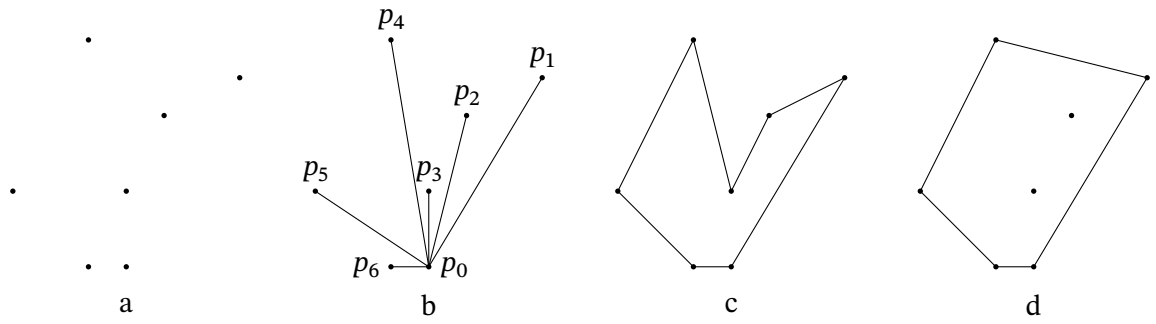

FIGURE 1 : Illustration du principe de l'algorithme de GRAHAM. (a) : un ensemble de points. (b) : choix du point pivot p_0 et tri angulaire des autres points. (c) : polygone non convexe formé par les points ordonnés. (d) : enveloppe convexe obtenue après suppression des concavités.

Cet algorithme met en jeu la recherche dans une liste, le tri d'une liste et la structure de pile.

QUESTION 9.1. On dispose de n points représentés par une liste de tuples `(x,y)`. Déterminer le point pivot p_0, situé en bas à droite de tous les autres (celui qui a la plus petite ordonnée ou, à ordonnée minimale, celui qui a la plus grande abscisse).

QUESTION 9.2. Trier les autres points selon le sens trigonométrique. p_i et p_j sont dans l'ordre trigonométrique si p_j est à gauche de $\overrightarrow{p_0p_i}$, autrement dit si le déterminant $\det(\overrightarrow{p_0p_i}, \overrightarrow{p_0p_j}) = (x_i - x_0)(y_j - y_0) - (y_i - y_0)(x_j - x_0)$ est positif. Écrire une fonction intermédiaire `determinant`, qui servira également à la question suivante.

QUESTION 9.3. Supprimer les concavités selon l'algorithme suivant. Les points p_0 et p_1 sont placés dans une pile qui contiendra finalement les points de l'enveloppe convexe. Par construction, p_0 et p_1 appartiennent nécessairement à l'enveloppe convexe. On considère le point p_k, k étant initialisé à 2. Tant que k est inférieur à n ($n = 7$ sur la Figure 1), on considère à chaque étape les deux points p_i et p_j en haut de la pile. Ils forment la dernière arête du polygone en construction. Si p_k est à gauche de $\overrightarrow{p_ip_j}$ alors on empile p_k et on passe au point suivant (on incrémente k), sinon, on enlève le point qui crée une concavité (on dépile p_j sans changer k). On arrête quand on a empilé le dernier point (à cette étape, $k = n - 1$).

4 Annexe

A. Les générateurs et les itérateurs

Une fonction génératrice – ou un générateur – se distingue d'une fonction usuelle par l'emploi du mot-clé `yield` « à la place de `return` ». Elle produit un objet nommé itérateur, qui s'interprète comme un curseur sur les éléments d'une séquence. L'activation de ce curseur est par exemple réalisé à l'aide d'une boucle `for` qui « consomme » les éléments un à un. Ci-dessous est présenté un exemple de générateur (la fonction `gen`), qu'il est recommandé de reproduire :

```
>>> from collections.abc import Iterator

>>> def gen(n: int) -> Iterator[int]:
...     yield 0
...     for i in range(n):
...         yield (i+1)**2

>>> for carré in gen(5):
...     print(carré, end=' ')
0 1 4 9 16 25
```

En définitive, l'appel à `gen(5)` dans la boucle `for` créé un curseur qui progresse à chaque itération, au gré des `yield` disséminés dans la fonction génératrice. Après le dernier `yield`, le curseur produit une erreur de type `StopIteration`, parfaitement comprise de la boucle `for` qui termine donc silencieusement.

En outre, l'itérateur peut être « activé » à l'aide de la fonction `next` :

```
>>> it_carré = gen(5)
... while (carré := next(it_carré, None)) is not None:
...     print(carré, end=' ')
0 1 4 9 16 25
```

Dès lors, on comprend que les fonctions génératrices du langage Python simplifient la création d'objets fondamentaux que sont les *itérateurs*.

Créer une fonction génératrice pour les éléments d'un ensemble E rend cet ensemble `Iterable`, comme c'est le cas des `list`, `set`, `tuple` et `dict` par exemple. Pour ces collections, la fonction Python `iter()` joue le rôle de générateur :

```
>>> iterateur_de_liste = iter([1, 2])
>>> next(iterateur_de_liste)
1
>>> next(iterateur_de_liste)
2
>>> next(iterateur_de_liste)
StopIteration
```

Pour terminer ce petit tour d'horizon, Python autorise la définition d'*expressions génératrices* qui produisent des itérateurs. Leur construction est très semblable aux listes en intension :

```
>>> liste_carrés = [i ** 2 for i in range(10)]  # liste en intension
>>> liste_carrés
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> iter_carrés = (i ** 2 for i in range(10))   # expression génératrice !
>>> iter_carrés
<generator object <genexpr> at 0x110bef030>
>>> next(iter_carrés)
0
>>> next(iter_carrés)
1
```