Algorithmique Programmation Objet

Python

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr

CM - Séance 5

Listes et itérateurs

Plan

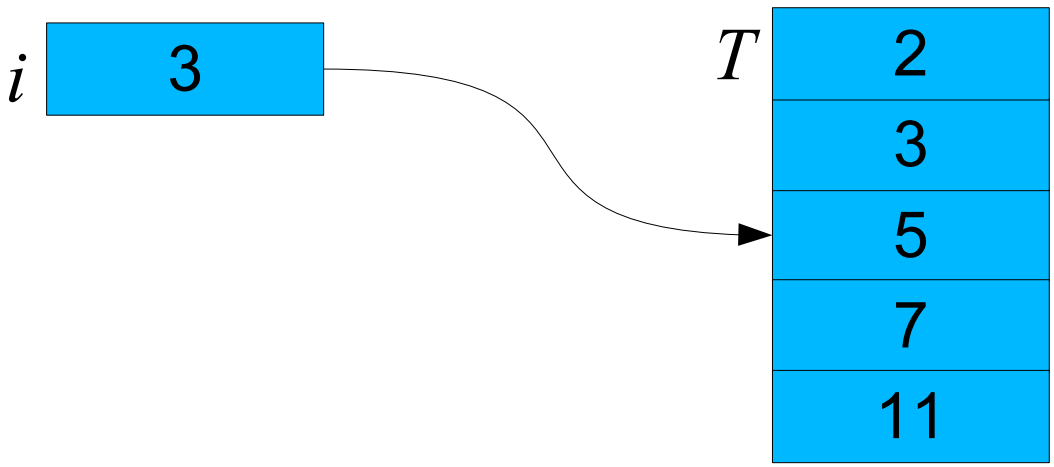
- Indirection et pointeurs
- Listes simplement chaînées
- Listes doublement chaînées
- Listes en Python
- Itérateurs

Introduction

- Les tableaux sont très pratiques, mais ils ont un gros défaut
- Insertion et élimination d'éléments sont des opérations O(n) !
- Il nous faut une structure de données pour laquelle ces opérations soient plus performantes
- On peut obtenir une telle structure en utilisant l'indirection

Indirection

- L'idée de l'indirection est d'utiliser la valeur d'une variable pour indiquer la position d'une autre variable (dans un tableau ou en mémoire)
- Nous avons déjà rencontré cette idée avec les variables indice :



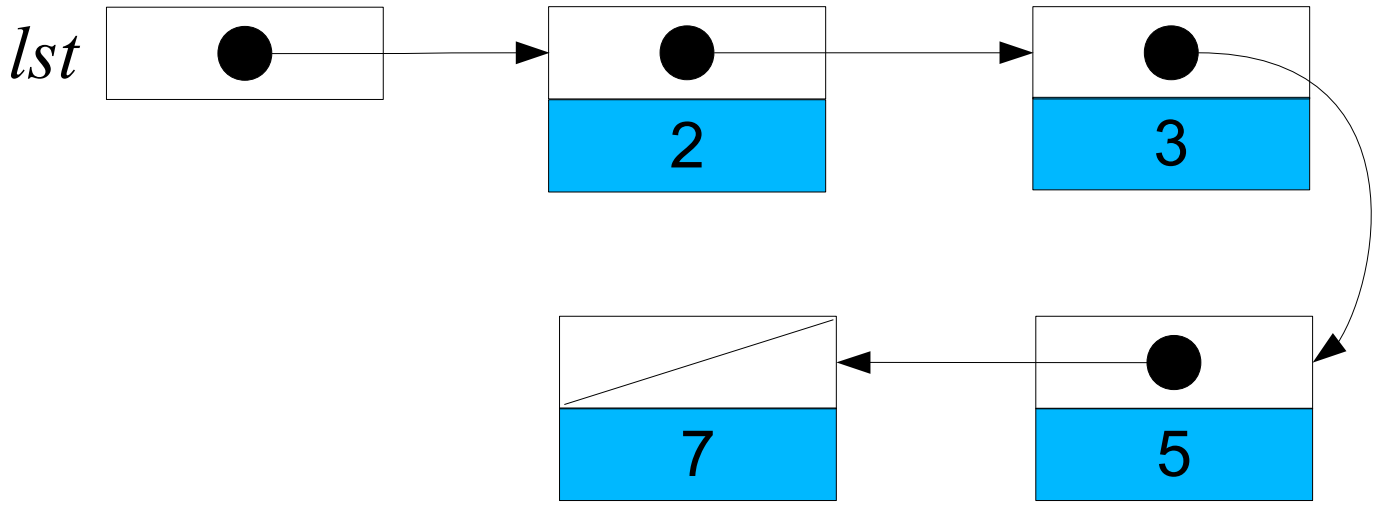
Pointeurs

- Un pointeur est un type de données dont la valeur fait référence (référencie) directement à (pointe vers) une autre valeur.
- Un pointeur référencie une valeur située quelque part d'autre en mémoire, habituellement en utilisant son adresse
- La mémoire est un grand tableau, donc un pointeur est une sorte de variable indice de la mémoire
- Obtenir la valeur vers laquelle un pointeur pointe est appelé « déréférencer le pointeur ».
- Un pointeur qui ne pointe vers aucune valeur aura la valeur nil

Liste chaînée

- Une liste chaînée désigne une structure de données représentant une collection ordonnée et de taille arbitraire d'éléments.
- L'accès aux éléments d'une liste se fait de manière séquentielle
- chaque élément permet l'accès au suivant (contrairement au cas du tableau dans lequel l'accès se fait de manière absolue, par adressage direct de chaque cellule dudit tableau).
- Un élément contient un accès vers une donnée
- Le principe de la liste chaînée est que chaque élément possède, en plus de la donnée, des pointeurs vers les éléments qui lui sont logiquement adjacents dans la liste.
- Liste simplement chaînée : un seul pointeur vers l'élément suivant
- Liste doublement chaînée : pointeurs vers précédent et suivant

Liste (simplement) chaînée



Liste chaînée

- Étant donnée une liste L
  - L.premier désigne le premier élément de la liste
  - nil désigne l'absence d'élément

- Liste simplement chaînée :
  - elt.donnée désigne la donnée associée à l'élément elt
  - elt.suivant désigne l'élément suivant de la liste

- Liste doublement chaînée :
  - elt.précédent désigne l'élément précédent de la liste

Remarque historique

- La représentation de listes chaînées à l'aide du diagramme avec une flèche vers le suivant a été proposé par Newell and Shaw dans l'article "Programming the Logic Theory Machine" Proc. WJCC, February 1957.
- Newell et Simon ont obtenu l'ACM Turing Award en 1975 pour avoir "*made basic contributions to artificial intelligence, the psychology of human cognition, and list processing*".

Liste : opérations

- Trois opérations principales
  - Parcours de la liste
  - Ajout d'un élément
  - Suppression d'un élément
- A partir de là, d'autres opérations vont être obtenues :
  - recherche d'une donnée,
  - Remplacement,
  - concaténation de liste,
  - fusion de listes,
  - etc.

Liste vs. Tableau

- Le principal avantage des listes sur les tableaux
  - L'ordre des éléments de la liste peut être différent de leur ordre en mémoire.
  - Les listes chaînées vont permettre l'ajout ou la suppression d'un élément en n'importe quel endroit de la liste en temps constant.
- En revanche, certaines opérations peuvent devenir coûteuses comme la recherche d'un élément contenant une certaine donnée.
  - Pas de recherche dichotomique dans une liste : on ne peut pas atteindre le ième élément sans parcourir !

Liste : parcours

```
entier nombreElements(L)
   compte ← 0;
   elt ← L.premier
   tant que elt ≠ nil
      compte ← compte + 1
      elt ← elt.suivant
   renvoyer compte
```

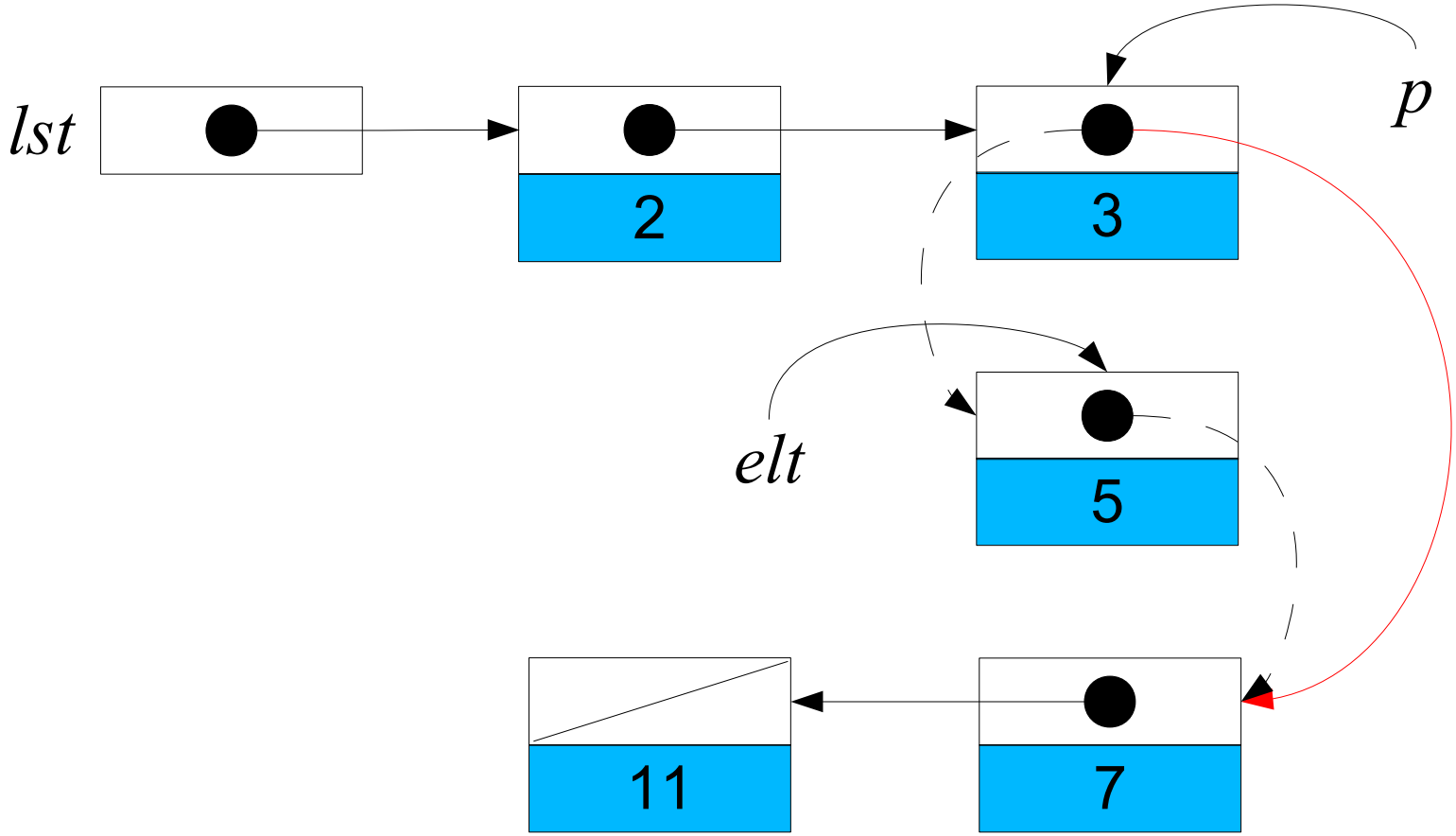
Liste : ajout d'un élément

- On ajoute un élément elt au début de la liste.
- On suppose qu'il n'est pas déjà dans la liste (sinon que se passe-t-il ?)
- Principes :
  - Le premier de la liste deviendra elt
  - Mais où est le premier ? Il devient le suivant de elt
  - Attention à l'ordre de mise à jour ! On ne doit pas perdre le premier. Donc
  - Le suivant de elt est mis à jour
  - Puis le premier de la liste

Liste : ajout d'un élément

```
ajouteAuDebut(elt, L)
  # elt n'est pas dans L
  elt.suivant ← L.premier
  L.premier ← elt
```

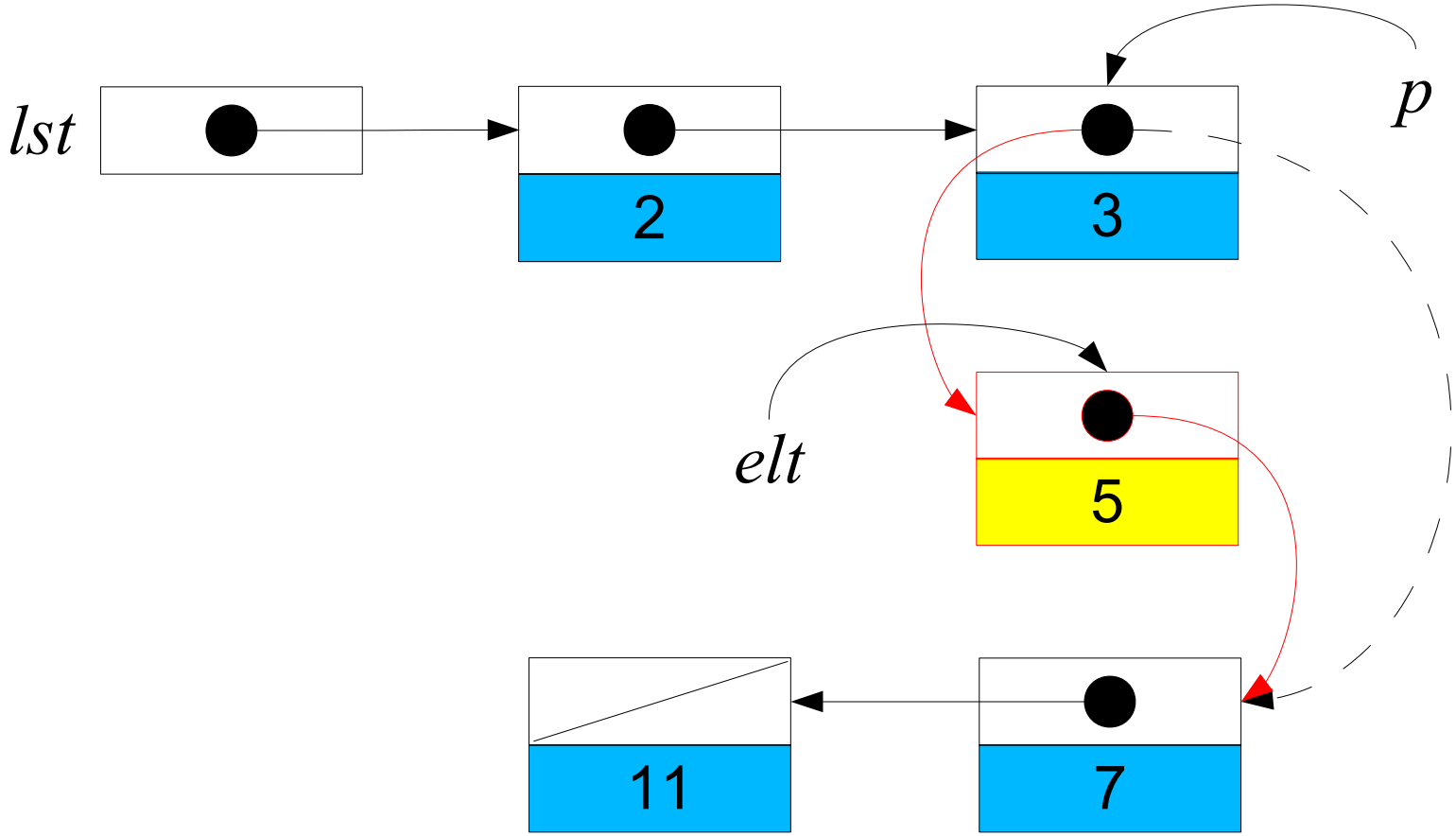
Liste : insertion d'un élément



Liste : insertion d'un élément

```
insèreAprès(elt, p)
  # elt n'est pas dans L
  # p est dans L
  elt.suivant ← p.suivant
  p.suivant ← elt
```

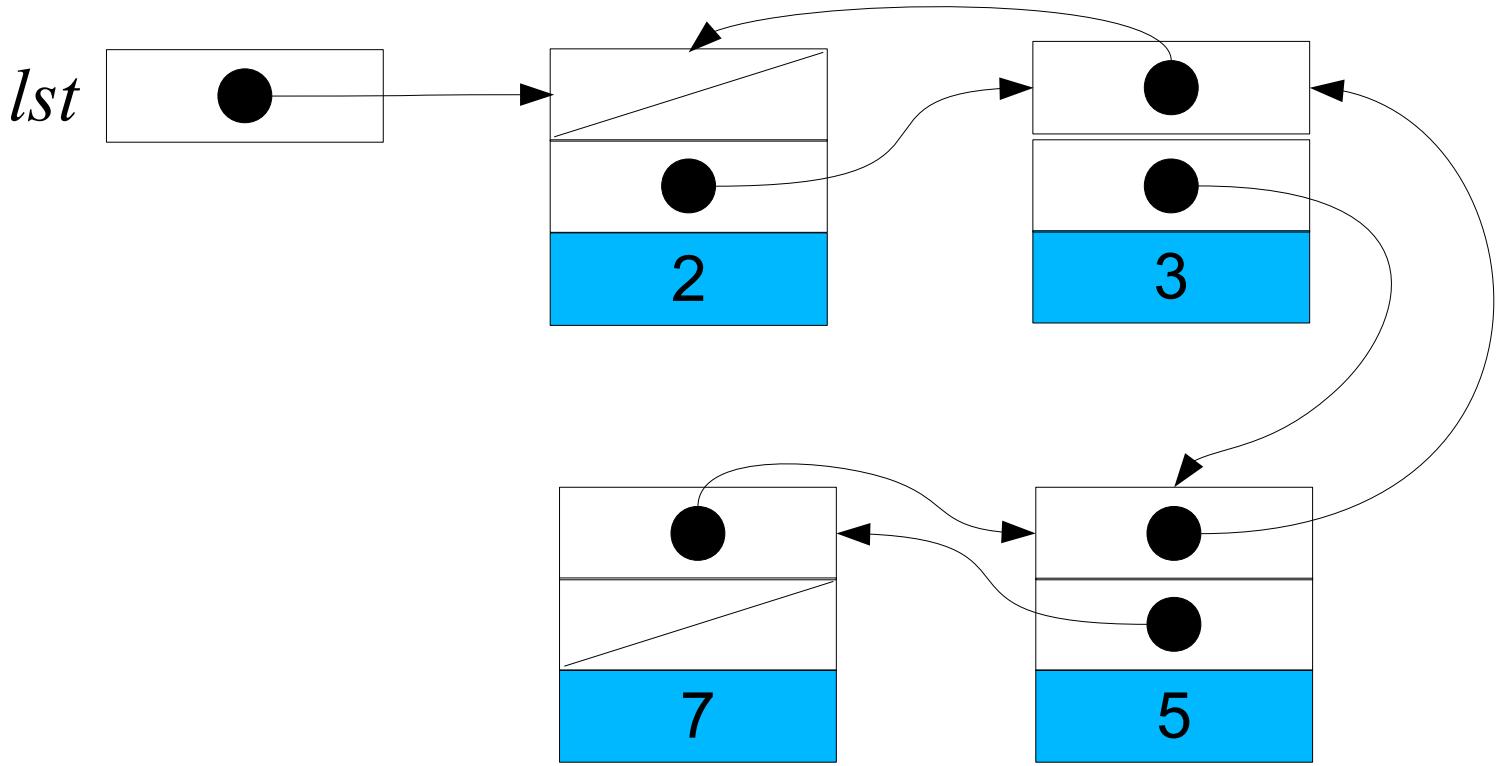
Liste : suppression d'un élément



Liste : suppression d'un élément

```
supprime(elt, p, L)
   # elt est dans L,
   # p est son précédent
   si elt = L.premier
      L.premier ← elt.suivant
   sinon si elt = p.suivant
      p.suivant ← elt.suivant
```

Liste doublement chaînée



Liste doublement chaînée

- Les opérations principales changes pour prendre en compte le fait que chaque élément pointe au précédent ainsi qu'au suivant
- Parcours
  - On peut parcourir la liste en avant et en arrière
- Insertion et suppression
  - Il faut modifier aussi les pointeurs au précédent

Liste double : ajout d'un élément

```
ajouteAuDebut(elt, L)
  # elt n'est pas dans L
  elt.suivant ← L.premier
  L.premier.suivant ← elt
  elt.précédent ← nil
  L.premier ← elt
```

Liste double : insertion d'un élément

```
insèreAprès(elt, p)
  # elt n'est pas dans L
  # p est dans L
  elt.suivant ← p.suivant
  elt.précédent ← p
  si p.suivant ≠ nil
    p.suivant.précédent ← elt
  p.suivant ← elt
```

Liste double : suppression d'un élément

```
supprime(elt, L)
  # elt est dans L
  s ← elt.suivant
  p ← elt.précédent
  si p = nil
    L.premier ← s
  sinon
    p.suivant ← s
  si s ≠ nil
    s.précédent ← p
```

Listes simplement chaînées en Python

- Ce que Python appelle « liste » c'est en effet un tableau !
- Comment est-ce qu'on peut réaliser des listes chaînées en Python, alors ?
- Il y a deux manières possibles
  - En utilisant les tuples et un style de programmation fonctionnel
  - En utilisant les objets et un style de programmation orienté objet

Listes en Python : style fonctionnel

- La brique de base pour construire une liste est une tuple contenant deux éléments :
  - La tête de liste (ce qu'on appelle CAR en Lisp)
  - La queue de liste (ce qu'on appelle CDR en Lisp)
- La liste vide est représentée par None.

```
def mklist(*args):
    result = None
    for element in reversed(args):
        result = (element, result)
    return result
```

```
mklist = lambda *args:
  reduce(lambda lst, el: cons(el, lst), reversed(args), None)
```

Listes en Python : style fonctionnel

```
cons = lambda el, lst: (el, lst)

car = lambda lst: lst[0] if lst else lst

cdr = lambda lst: lst[1] if lst else lst

nth = lambda n, lst:
  nth(n-1, cdr(lst)) if n > 0 else car(lst)

length = lambda lst, count=0:
  length(cdr(lst), count+1) if lst else count
```

Listes en Python : style orienté objet

```
class ListNode:
  def __init__(self, cargo=None, next=None):
    self.car = cargo
    self.cdr = next
  def __str__(self):
    return str(self.car)
```

Itérateur

- Un itérateur est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc). Un synonyme d'itérateur est curseur, notamment dans le contexte des bases de données.
- Un itérateur dispose essentiellement de deux primitives :
  - accéder à l'élément en cours dans le conteneur,
  - se déplacer vers l'élément suivant.
- Il faut aussi pouvoir créer un itérateur sur le premier élément ; ainsi que déterminer à tout moment si l'itérateur a épuisé la totalité des éléments du conteneur. Diverses implémentations peuvent également offrir des comportements supplémentaires.

Itérateur

- Le but d'un itérateur
  - permettre à son utilisateur de parcourir le conteneur, c'est-à-dire d'accéder séquentiellement à tous ses éléments pour leur appliquer un traitement,
  - isoler l'utilisateur de la structure interne du conteneur, potentiellement complexe.
- Avantage :
  - le conteneur peut stocker les éléments de la façon qu'il veut, tout en permettant à l'utilisateur de le traiter comme une simple liste d'éléments.
  - Le plus souvent l'itérateur est conçu en même temps que la classe-conteneur qu'il devra parcourir, et ce sera le conteneur lui-même qui créera et distribuera les itérateurs pour accéder à ses éléments

Itérateur : avantages

- On utilise souvent un index dans une simple boucle, pour accéder séquentiellement à tous les éléments, notamment d'un tableau; l'utilisation des itérateurs a certains avantages:
- Un simple compteur dans une boucle n'est pas adapté à toutes les structures de données, en particulier
  - celles qui n'ont pas de méthode d'accès à un élément quelconque
  - celles dont l'accès à un élément quelconque est très lent (c'est le cas des listes chaînées et des arbres).

Itérateur : avantages

- Les itérateurs fournissent un moyen cohérent d'itérer sur toutes sortes de structures de données, rendant ainsi le code client plus lisible, réutilisable, et robuste même en cas de changement dans l'organisation de la structure de données.
- Un itérateur peut implanter des restrictions additionnelles sur l'accès aux éléments, par exemple pour empêcher qu'un élément soit « sauté », ou qu'un même élément soit visité deux fois.

Itérateur

- Un itérateur peut dans certains cas permettre que le conteneur soit modifié, sans être invalidé pour autant.
- Par exemple, après qu'un itérateur s'est positionné derrière le premier élément, il est possible d'insérer d'autres éléments au début du conteneur avec des résultats prévisibles.
- Avec un index on aurait plus de problèmes, parce que la valeur de l'index devrait elle aussi être modifiée en conséquence.
- **Important** : il est indispensable de bien consulter la documentation d'un itérateur pour savoir dans quels cas il est invalidé ou non.

Itérateur

- Cela permet de faire des algorithmes sans connaître la structure de données sous-jacente.
- On recherche un plus court chemin dans un graphe :
  - On ne sait pas comment le graphe est représenté.
  - Cela ne nous empêche pas de faire un algorithme efficace

Itérateurs en Python

- Objets itérables
  - Listes, chaînes de caractères, tuples, dictionnaires et fichiers
  - Objet des classes ayant une méthode `__iter__()` ou `__getitem()__`.
- Quand la fonction primitive `iter()` est appliquée à un objet itérable, elle renvoie un itérateur
- L'instruction `for` applique `iter()` implicitement, en coulisse
- Un itérateur est un objet qui expose la méthode `__next__()`
- La fonction primitive `next()` appliquée à un itérateur renvoie le prochain élément ou lance l'exception `StopIteration` si l'itérateur est épuisé.

Générateurs

- Les générateurs sont des fonctions spéciales qui renvoient un itérateur
- Elles sont écrites comme des fonctions normales, sauf qu'elles utilisent l'instruction `yield` pour renvoyer le prochain élément
- Un générateur simple peut être codé comme expression, en utilisant la syntaxe : *expression* for *variables* in *objet_itérable*
- Exemples :

```
def reverse(data):
  for index in range(len(data)-1, -1, -1):
    yield data[index]

sum(i*i for i in range(10))
```

Merci de votre attention