Algorithmique et Programmation

Algorithmique et compréhensions de listes

Elise Bonzon

elise.bonzon@mi.parisdescartes.fr

LIPADE - Université Paris Descartes
http://www.math-info.univ-paris5.fr/∽bonzon/

Avant de commencer

Quelques informations :

- Le QCM3 aura lieu, en début de TD, la semaine du 9/12/2019
- Il y aura TD, mais pas de TP la semaine du 9/12/2019
- La contrôle machine aura lieu la semaine du 16/12/2019
  - *a priori le mardi 17/12 – à confirmer encore*
  - Les horaires de passage seront affichées sur Moodle. Il faudra **impérativement** arriver a minima 10mns en avance !
- Le cours la semaine prochaine aura lieu le mercredi 4/12 (échange avec le cours de MC2)

Encore des algorithmes ?

- Il existe de nombreux autres algorithmes sur les listes que la recherche et le tri
- Vous en avez déjà vu beaucoup : chercher le max, le min, calculer la moyenne, l'écart-type, la variance...
- Présentation dans ce cours d'autres algorithmes classiques
- Et d'une spécificité du langage Python : les compréhensions de listes

Algorithmique

1. Insertion d'un élément dans une liste triée
2. Suppression d'un élément dans une liste
3. Compréhensions de listes
4. Filtrage d'une liste
5. Compréhensions multiples
6. Pour conclure

Insertion d'un élément dans une liste triée

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?
- Nous voulons modifier la liste et ne pas en créer une nouvelle contenant cet élément

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?
- Nous voulons modifier la liste et ne pas en créer une nouvelle contenant cet élément
- Et ne pas avoir besoin de trier de nouveau toute la liste...

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?
- Nous voulons modifier la liste et ne pas en créer une nouvelle contenant cet élément
- Et ne pas avoir besoin de trier de nouveau toute la liste...
- **Idée :** Utiliser la méthode d'insertion déjà vu dans l'algorithme de **tri par insertion** :

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?
- Nous voulons modifier la liste et ne pas en créer une nouvelle contenant cet élément
- Et ne pas avoir besoin de trier de nouveau toute la liste...
- **Idée :** Utiliser la méthode d'insertion déjà vu dans l'algorithme de **tri par insertion** :
  - Ajouter l'élément en fin de la liste (méthode `append`)

Insertion d'un élément dans une liste triée

- Insérer un élément à la fin d'une liste : méthode `append`
- Mais comment insérer un élément *à sa place* dans une liste triée ?
- Nous voulons modifier la liste et ne pas en créer une nouvelle contenant cet élément
- Et ne pas avoir besoin de trier de nouveau toute la liste...
- **Idée :** Utiliser la méthode d'insertion déjà vu dans l'algorithme de **tri par insertion** :
  - Ajouter l'élément en fin de la liste (méthode `append`)
  - Décaler les valeurs de la liste vers la droite, jusqu'à trouver la place de l'élément à insérer

Insertion d'un élément dans une liste triée

- **Exemple** : insérer 16 dans la liste `[5, 9, 12, 18, 43]`

Insertion d'un élément dans une liste triée

- **Exemple** : insérer 16 dans la liste `[5, 9, 12, 18, 43]`

Indice	*0*	*1*	*2*	*3*	*4*
Elément	5	9	12	18	43

Insertion d'un élément dans une liste triée

- **Exemple** : insérer 16 dans la liste `[5, 9, 12, 18, 43]`
- Insérer 16 en fin de liste (méthode `append`)

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	18	43	16

Insertion d'un élément dans une liste triée

- **Exemple** : insérer 16 dans la liste `[5, 9, 12, 18, 43]`
- Insérer 16 en fin de liste (méthode `append`)
- Décaler les éléments de la liste vers la droite jusqu'à trouver la place de l'élément 16

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	18	43	16

Insertion d'un élément dans une liste triée

- **Exemple** : insérer `16` dans la liste `[5, 9, 12, 18, 43]`
- Insérer `16` en fin de liste (méthode `append`)
- Décaler les éléments de la liste vers la droite jusqu'à trouver la place de l'élément `16`

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	18	43	43

Insertion d'un élément dans une liste triée

- **Exemple** : insérer `16` dans la liste `[5, 9, 12, 18, 43]`
- Insérer `16` en fin de liste (méthode `append`)
- Décaler les éléments de la liste vers la droite jusqu'à trouver la place de l'élément `16`

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	18	43	43

Insertion d'un élément dans une liste triée

- **Exemple** : insérer 16 dans la liste `[5, 9, 12, 18, 43]`
- Insérer 16 en fin de liste (méthode `append`)
- Décaler les éléments de la liste vers la droite jusqu'à trouver la place de l'élément 16

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	18	18	43

Insertion d'un élément dans une liste triée

- **Exemple** : insérer `16` dans la liste `[5, 9, 12, 18, 43]`
- Insérer `16` en fin de liste (méthode `append`)
- Décaler les éléments de la liste vers la droite jusqu'à trouver la place de l'élément `16`

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	18	18	43

Insertion d'un élément dans une liste triée

- **Exemple** : insérer 16 dans la liste `[5, 9, 12, 18, 43]`
- Insérer 16 en fin de liste (méthode `append`)
- Décaler les éléments de la liste vers la droite jusqu'à trouver la place de l'élément 16

Indice	*0*	*1*	*2*	*3*	*4*	*5*
Elément	5	9	12	16	18	43

Insertion d'un élément dans une liste triée

```
def insertion(liste, elem):
    """List x Elem --> None
    Insère l'élément elem à sa position dans la liste triée"""
    liste.append(elem)
    n = len(liste)
    indice = n - 1
    while indice > 0 and liste[indice - 1] > elem :
        liste[indice] = liste[indice - 1]
        indice = indice - 1
    liste[indice] = elem
```

Suppression d'un élément dans une liste

Suppression d'un élément dans une liste : algorithmique

- D'un point de vue algorithmique :

Suppression d'un élément dans une liste : algorithmique

- D'un point de vue algorithmique :
  - Rechercher l'élément (recherche dichotomique dans une liste triée, recherche séquentielle dans une liste triée, recherche séquentielle dans une liste non triée...)

Suppression d'un élément dans une liste : algorithmique

- D'un point de vue algorithmique :
  - Rechercher l'élément (recherche dichotomique dans une liste triée, recherche séquentielle dans une liste triée, recherche séquentielle dans une liste non triée...)
  - Suppression de l'élément

Suppression d'un élément dans une liste : algorithmique

- D'un point de vue algorithmique :
  - Rechercher l'élément (recherche dichotomique dans une liste triée, recherche séquentielle dans une liste triée, recherche séquentielle dans une liste non triée...)
  - Suppression de l'élément
  - La méthode de suppression dépend du langage de programmation !

Suppression d'un élément dans une liste : en Python

- Deux fonctions prédéfinies :

Suppression d'un élément dans une liste : en Python

- Deux fonctions prédéfinies :
  - La **méthode** `remove` supprime la première occurence de l'élément donné en paramètre

```
>>> liste = ['a', 'b', 'c', 'a', 'd', 'b']
>>> liste.remove('b')
>>> liste
['a', 'c', 'a', 'd', 'b']
```

Suppression d'un élément dans une liste : en Python

- Deux fonctions prédéfinies :
  - La **méthode** `remove` supprime la première occurence de l'élément donné en paramètre

```
>>> liste = ['a', 'b', 'c', 'a', 'd', 'b']
>>> liste.remove('b')
>>> liste
['a', 'c', 'a', 'd', 'b']
```

  - La **procédure** `del` supprime l'élément positionné à l'indice donné en paramètre

```
>>> liste = ['a', 'b', 'c', 'a', 'd', 'b']
>>> del liste[1]
>>> liste
['a', 'c', 'a', 'd', 'b']
>>> del liste[2:4]
>>> liste
['a', 'c', 'b']
```

Suppression de toutes les occurences d'un élément dans une liste

- La **méthode** `remove` ne supprime que la première occurence d'un élément

Suppression de toutes les occurences d'un élément dans une liste

- La **méthode** `remove` ne supprime que la première occurence d'un élément
- Pour supprimer toutes les occurences : appliquer la méthode `remove` **tant que** l'élément appartient à la liste

```
def suppression_occurences(liste, elem):
    """List x Elem --> None
    Supprime toutes les occurences de l'élément
    elem dans la liste"""

    while elem in liste :
        liste.remove(elem)
```

Suppression de toutes les occurences d'un élément dans une liste

- La **méthode** `remove` ne supprime que la première occurence d'un élément
- Pour supprimer toutes les occurences : appliquer la méthode `remove` **tant que** l'élément appartient à la liste

```
def suppression_occurences(liste, elem):
    """List x Elem --> None
    Supprime toutes les occurences de l'élément
    elem dans la liste"""

    while elem in liste :
        liste.remove(elem)
```

- Une autre possibilité est d'utiliser les compréhensions de listes :

```
>>> liste = ['a', 'b', 'c', 'a', 'b', 'd']
>>> liste = [elem for elem in liste if elem != 'b']
>>> liste
['a', 'c', 'a', 'd']
```

Compréhensions de listes

Retour sur un problème simple : liste des n premiers entiers

```
def naturels_while(n) :
    """Int --> List.
    Retourne la liste des n premiers entiers non nuls"""
    i = 1
    liste = []
    while i <= n :
        liste.append(i)
        i = i + 1
    return liste
```

Retour sur un problème simple : liste des n premiers entiers

```
def naturels_while(n) :
    """Int --> List.
    Retourne la liste des n premiers entiers non nuls"""
    i = 1
    liste = []
    while i <= n :
        liste.append(i)
        i = i + 1
    return liste
```

```
def naturels_for(n) :
    """Int --> List.
    Retourne la liste des n premiers entiers non nuls"""
    liste = []
    for i in range(1, n + 1) :
        liste.append(i)
    return liste
```

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

  $\rightarrow$ On aimerait avoir une description plus littérale de la solution

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

  $\rightarrow$ On aimerait avoir une description plus littérale de la solution
- Similaire à la définition des ensembles en mathématique. Différence entre :

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

  $\rightarrow$ On aimerait avoir une description plus littérale de la solution
- Similaire à la définition des ensembles en mathématique. Différence entre :
  - $E = \{1, 2, 3, 4, 5\}$

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

  $\rightarrow$ On aimerait avoir une description plus littérale de la solution
- Similaire à la définition des ensembles en mathématique. Différence entre :
  - $E = \{1, 2, 3, 4, 5\}$
  - $E = \{n \in \mathbb{N}^* | n \leq 5\}$

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

  $\rightarrow$ On aimerait avoir une description plus littérale de la solution
- Similaire à la définition des ensembles en mathématique. Différence entre :
  - $E = \{1, 2, 3, 4, 5\}$
  - $E = \{n \in \mathbb{N}^* | n \leq 5\}$
- La première est une définition **explicite**

Définition explicite vs. implicite

- Les deux solutions sont frustrantes :
  - Description explicite du contenu de la liste
  - Besoin de trouver l'algorithme (simple) qui construit la liste

  $\rightarrow$ On aimerait avoir une description plus littérale de la solution
- Similaire à la définition des ensembles en mathématique. Différence entre :
  - $E = \{1, 2, 3, 4, 5\}$
  - $E = \{n \in \mathbb{N}^* | n \leq 5\}$
- La première est une définition **explicite**
- La seconde est une définition par compréhension

Construction par compréhension

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des i pour $i \in [1, 5]$

Construction par compréhension

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des i pour $i \in [1, 5]$

```
[]
```

Construction par compréhension

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des i pour $i \in [1, 5]$

```
[i]
```

Construction par compréhension

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des i pour $i \in [1, 5]$

```
[i for i ]
```

Construction par compréhension

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des i pour $i \in [1, 5]$

```
[i for i in range(1, 6)]
```

Construction par compréhension

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des i pour $i \in [1, 5]$

```
[i for i in range(1, 6)]
```

```
>>> liste = [i for i in range(1, 6)]
>>> liste
[1, 2, 3, 4, 5]
```

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- Construit la liste composée des éléments :

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- Construit la liste composée des éléments :
  1. Le **premier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **premier** élément de `<seq>`

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- Construit la liste composée des éléments :
  1. Le **premier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **premier** élément de `<seq>`
  2. Le **deuxième** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **deuxième** élément de `<seq>`

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- Construit la liste composée des éléments :
  1. Le **premier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **premier** élément de `<seq>`
  2. Le **deuxième** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **deuxième** élément de `<seq>`
  3. ...

Syntaxe d'une construction par compréhension

```
[<expr> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- Construit la liste composée des éléments :
  1. Le **premier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **premier** élément de `<seq>`
  2. Le **deuxième** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **deuxième** élément de `<seq>`
  3. ...
  4. Le **dernier** élément est la valeur de `<expr>` dans laquelle la variable `<var>` a pour valeur le **dernier** élément de `<seq>`

Retour sur la construction de la liste des n premiers entiers

```
def naturels_comprehension(n) :
    """Int --> List.
    Retourne la liste des n premiers entiers non nuls"""

    return [i for i in range(1, n+1)]
```

Autres exemples simples sur les compréhensions

```
def liste_carres_comprehension(liste) :
    """List --> List.
    Retourne la liste des carrés des éléments
    de la liste donnée en paramètre"""

    return [elem * elem for elem in liste]
```

Autres exemples simples sur les compréhensions

```
def liste_carres_comprehension(liste) :
    """List --> List.
    Retourne la liste des carrés des éléments
    de la liste donnée en paramètre"""

    return [elem * elem for elem in liste]
```

```
def liste_longueurs_comprehension(liste) :
    """List --> List.
    Retourne la liste des longueurs des chaines
    de la liste donnée en paramètre"""

    return [len(s) for s in liste]
```

Filtrage d'une liste

Schéma de filtrage

Schéma de filtrage

Le filtrage d'une liste retourne une sous-liste de la liste de départ, selon un prédicat donné.

Schéma de filtrage

Schéma de filtrage

Le filtrage d'une liste retourne une sous-liste de la liste de départ, selon un prédicat donné.

```
def liste_positifs(liste) :
    """list --> list
    Retourne la sous-liste des entiers positifs
    de la liste donnée en paramètre"""

    resultat = []
    for elem in liste :
        if elem > 0:
            resultat.append(elem)
    return resultat
```

Filtrage : construction par compréhension conditionnée

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des `elem` strictement positifs pour `elem` $\in$ `liste`

Filtrage : construction par compréhension conditionnée

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des `elem` strictement positifs pour `elem` $\in$ `liste`

```
[elem for elem in liste]
```

Filtrage : construction par compréhension conditionnée

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des `elem` strictement positifs pour `elem` $\in$ `liste`

```
[elem for elem in liste if elem > 0]
```

Filtrage : construction par compréhension conditionnée

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des `elem` strictement positifs pour `elem` $\in$ `liste`

```
[elem for elem in liste if elem > 0]
```

```
>>> liste = [-1, 2, 6, -15, -6, 3, 12, -4]
>>> res = [elem for elem in liste if elem > 0]
>>> res
[2, 6, 3, 12]
```

Syntaxe d'une construction par compréhension conditionnée

```
[<expr> for <var> in <seq> if <condition>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)

Syntaxe d'une construction par compréhension conditionnée

```
[<expr> for <var> in <seq> if <condition>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`

Syntaxe d'une construction par compréhension conditionnée

```
[<expr> for <var> in <seq> if <condition>]
```

- `<var>` : une variable de compréhension
- `<expr>` : une expression pouvant contenir `<var>`
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`
- Construit la liste de la même façon, en ne retenant que les éléments pour lesquels la `<condition>` est `True`

Autres exemples simples sur les compréhensions conditionnées

```
def liste_pairs(liste) :
    """List --> List.
    Retourne la sous-liste des entiers pairs
    de la liste donnée en paramètre"""

    return [elem for elem in liste if elem % 2 == 0]
```

Autres exemples simples sur les compréhensions conditionnées

```
def liste_pairs(liste) :
    """List --> List.
    Retourne la sous-liste des entiers pairs
    de la liste donnée en paramètre"""

    return [elem for elem in liste if elem % 2 == 0]
```

```
def longueur_min(liste) :
    """List --> List.
    Retourne la sous-liste des chaines de la liste
    donnée en paramètre contenant au moins 3 caractères"""

    return [s for s in liste if len(s) >= 3]
```

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?
Par exemple, comment calculer la liste qui contient le carré des nombres pairs, et le cube des nombres impairs ?

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?
Par exemple, comment calculer la liste qui contient le carré des nombres pairs, et le cube des nombres impairs ?
On aimerait pouvoir exprimer **littéralement** en Python :

> Liste des $\texttt{elem}^2$ pour $\texttt{elem} \in \texttt{liste}$ si `elem` est pair, $\texttt{elem}^3$ sinon

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?
Par exemple, comment calculer la liste qui contient le carré des nombres pairs, et le cube des nombres impairs ?
On aimerait pouvoir exprimer **littéralement** en Python :

Liste des $\texttt{elem}^2$ pour $\texttt{elem} \in \texttt{liste}$ si `elem` est pair, $\texttt{elem}^3$ sinon

```
[elem**2 for elem in liste]
```

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?
Par exemple, comment calculer la liste qui contient le carré des nombres pairs, et le cube des nombres impairs ?
On aimerait pouvoir exprimer **littéralement** en Python :

Liste des `elem`^{2} pour `elem` $\in$ `liste` si `elem` est pair, `elem`^{3} sinon

```
[elem**2 if elem % 2 == 0 else elem**3 for elem in liste]
```

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?
Par exemple, comment calculer la liste qui contient le carré des nombres pairs, et le cube des nombres impairs ?
On aimerait pouvoir exprimer **littéralement** en Python :

> Liste des $\texttt{elem}^2$ pour $\texttt{elem} \in \texttt{liste}$ si `elem` est pair, $\texttt{elem}^3$ sinon

```
[elem**2 if elem % 2 == 0 else elem**3 for elem in liste]
```

Attention, dans ce cas le `if ... else` est avant le `for` !

Et le `else` ?

Comment utiliser une conditionnelle `if ... else` dans une construction par compréhension ?
Par exemple, comment calculer la liste qui contient le carré des nombres pairs, et le cube des nombres impairs ?
On aimerait pouvoir exprimer **littéralement** en Python :

Liste des `elem`^{2} pour `elem` $\in$ `liste` si `elem` est pair, `elem`^{3} sinon

```
[elem**2 if elem % 2 == 0 else elem**3 for elem in liste]
```

Attention, dans ce cas le `if ... else` est avant le `for` !

```
>>> liste = [2, 3, 4, 5, 6]
>>> [elem**2 if elem % 2 == 0 else elem**3 for elem in liste]
[4, 27, 16, 125, 36]
```

Syntaxe d'une construction par compréhension conditionnée

```
[<expr1> if <condition> else <expr2> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`

Syntaxe d'une construction par compréhension conditionnée

```
[<expr1> if <condition> else <expr2> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`
- `<expr1>` : une expression pouvant contenir `<var>`
- `<expr2>` : une expression pouvant contenir `<var>`

Syntaxe d'une construction par compréhension conditionnée

```
[<expr1> if <condition> else <expr2> for <var> in <seq>]
```

- `<var>` : une variable de compréhension
- `<seq>` : une séquence (`range`, `str` ou `list`)
- `<condition>` : une expression booléenne portant sur `<var>`
- `<expr1>` : une expression pouvant contenir `<var>`
- `<expr2>` : une expression pouvant contenir `<var>`
- Si `<condition>` est `True`, `<expr1>` sera appliqué ; autrement `<expr2>` sera appliqué

Compréhensions multiples

Compréhensions multiples

Problème

Donner une définition de la fonction qui, étant donné un entier n, renvoie la liste des couples (i, j) dans l'intervalle $[1, n]$ avec $i \leq j$

Compréhensions multiples

Problème

Donner une définition de la fonction qui, étant donné un entier n, renvoie la liste des couples (i, j) dans l'intervalle $[1, n]$ avec $i \leq j$

```
>>> liste_couple(3)
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Compréhensions multiples

Problème

Donner une définition de la fonction qui, étant donné un entier n, renvoie la liste des couples (i, j) dans l'intervalle $[1, n]$ avec $i \leq j$

```
>>> liste_couple(3)
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

```
def liste_couples(n) :
    """Int --> List, avec n >= 0
    Retourne la liste des couples (i,j) sur l'intervalle
    [1, n] avec i <= j"""

    liste = []
    for i in range(1, n + 1) :
        for j in range(i, n+1) :
            liste.append((i, j))
    return liste
```

Construction par compréhensions multiples

On aimerait pouvoir exprimer **littéralement** en Python :

> Construire la liste des couples (i, j) pour $i \in [1, n]$ et $j \in [i, n]$

Construction par compréhensions multiples

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des couples (i, j) pour $i \in [1, n]$ et $j \in [i, n]$

```
[(i, j)]
```

Construction par compréhensions multiples

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des couples (i, j) pour $i \in [1, n]$ et $j \in [i, n]$

```
[(i, j) for i in range(1, n+1)]
```

Construction par compréhensions multiples

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des couples (i, j) pour $i \in [1, n]$ et $j \in [i, n]$

```
[(i, j) for i in range(1, n+1) for j in range(i, n+1)]
```

Construction par compréhensions multiples

On aimerait pouvoir exprimer **littéralement** en Python :

Construire la liste des couples (i, j) pour $i \in [1, n]$ et $j \in [i, n]$

```
[(i, j) for i in range(1, n+1) for j in range(i, n+1)]
```

```
>>> n = 3
>>> liste = [(i, j) for i in range(1, n+1) for j in range(i, n+1)]
>>> liste
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Syntaxe d'une construction par compréhension multiple

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

Syntaxe d'une construction par compréhension multiple

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)

Syntaxe d'une construction par compréhension multiple

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)

Syntaxe d'une construction par compréhension multiple

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- ...

Syntaxe d'une construction par compréhension multiple

```
[<expr> for <var1> in <seq1> for <var2> in <seq2> ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- ...
- `<expr>` : une expression pouvant contenir `<var1>`, `<var2>`, ...

Syntaxe d'une construction par compréhension complètes

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

Syntaxe d'une construction par compréhension complètes

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<cond1>` : une expression booléenne portant sur `<var1>`

Syntaxe d'une construction par compréhension complètes

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<cond1>` : une expression booléenne portant sur `<var1>`
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- `<cond2>` : une expression booléenne portant sur `<var2>`

Syntaxe d'une construction par compréhension complètes

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<cond1>` : une expression booléenne portant sur `<var1>`
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- `<cond2>` : une expression booléenne portant sur `<var2>`
- ...

Syntaxe d'une construction par compréhension complètes

```
[<expr> for <var1> in <seq1> if <cond1>
        for <var2> in <seq2> if <cond2>
        ... ]
```

- `<var1>` : une variable de compréhension
- `<seq1>` : une séquence (`range`, `str` ou `list`)
- `<cond1>` : une expression booléenne portant sur `<var1>`
- `<var2>` : une variable de compréhension
- `<seq2>` : une séquence (`range`, `str` ou `list`)
- `<cond2>` : une expression booléenne portant sur `<var2>`
- ...
- `<expr>` : une expression pouvant contenir `<var1>`, `<var2>`, ...

Exemple d'une compréhension complète

Liste des couples (i, j) pour $i, j \in [2, 10]$ tels que $i \neq j$ et i divise j

Exemple d'une compréhension complète

Liste des couples (i, j) pour $i, j \in [2, 10]$ tels que $i \neq j$ et i divise j

```
[(i, j) for i in range(2, 11)
        for j in range(2, 11)]
```

Exemple d'une compréhension complète

Liste des couples (i, j) pour $i, j \in [2, 10]$ tels que $i \neq j$ et i divise j

```
[(i, j) for i in range(2, 11)
        for j in range(2, 11)
        if (i != j)]
```

Exemple d'une compréhension complète

Liste des couples (i, j) pour $i, j \in [2, 10]$ tels que $i \neq j$ et i divise j

```
[(i, j) for i in range(2, 11)
        for j in range(2, 11)
        if (i != j) and (j % i == 0)]
```

Exemple d'une compréhension complète

Liste des couples (i, j) pour $i, j \in [2, 10]$ tels que $i \neq j$ et i divise j

```
[(i, j) for i in range(2, 11)
        for j in range(2, 11)
        if (i != j) and (j % i == 0)]
```

```
>>> [(i, j) for i in range(2, 11)
...          for j in range(2, 11)
...          if (i != j) and (j % i == 0)]
[(2, 4), (2, 6), (2, 8), (2, 10), (3, 6), (3, 9), (4, 8), (5, 10)]
```

Pour conclure

Résumé du cours

Aujourd'hui, on a vu :

- Deux algorithmes classiques sur les listes (ajouter et supprimer un élément)
- Les compréhensions de listes