

# Algorithmique avancée

2023-01-08

## Contents

<b>1. Nombres aléatoires</b>	<b>1</b>
Générer un nombre entier aléatoire . . . . .	1
Générer une liste de nombres aléatoires . . . . .	2
<b>2. Vitesse d'exécution</b>	<b>2</b>
Fonctionnement de la fonction <code>time</code> . . . . .	2
Temps d'exécution de la première méthode de génération . . . . .	2
Temps d'exécution avec des <i>list comprehension</i> . . . . .	3
Pour aller plus loin . . . . .	3
Pour aller plus loin . . . . .	3
<b>3. Inverser une liste</b>	<b>4</b>
Avec la méthode <code>reverse</code> . . . . .	4
Avec la fonction <code>reversed</code> . . . . .	4
Pour aller plus loin - Comprendre la méthode <code>reverse</code> . . . . .	4
Avec des <code>slice</code> . . . . .	4
Avec la méthode <code>pop</code> . . . . .	4
Avec la méthode <code>insert</code> + <code>Pop</code> . . . . .	5
<b>4. Un élément sur n dans une liste</b>	<b>5</b>
Avec une boucle et une nouvelle liste . . . . .	5
Avec une condition sur les indices . . . . .	5
En utilisant un pas sur <code>range</code> . . . . .	6
Avec une <i>list comprehension</i> . . . . .	6
Avec une condition sur les indices . . . . .	6
Retours à la ligne pour plus de clarté . . . . .	6
En utilisant un pas sur <code>range</code> . . . . .	6

## 1. Nombres aléatoires

On veut écrire une fonction :

```
generation : (int > 0)3 → list[int > 0]
             (nb_val, nb_min, nb_max) ↦ [x|x ∈ [nb_min,nb_max]]
```

Qui génère `nb_val` nombres aléatoires entiers entre `nb_min` et `nb_max`

### Générer un nombre entier aléatoire

Avec `random.random`, qui renvoie un réel aléatoire dans `[0;1[`

```
from random import random
from math import floor
```

```
def random_gen_with_random (nb_min: int, nb_max: int) -> int:
    return floor(random() * (nb_max - nb_min) + nb_min) # floor -> arrondi inferieur (int fait la meme cho
```

```
print(random_gen_with_random(5, 10))

Autre version avec la fonction randint

from random import randint

def random_gen (nb_min: int, nb_max: int) -> int:
    return randint(nb_min, nb_max)

# Note: cette fonction est un peu inutile, car elle est exactement similaire a
# randint.

print(random_gen(8, 18))
```

## Générer une liste de nombres aléatoires

On utilise la fonction `random_gen` que l'on a définie plus haut.

Première version, avec une boucle :

```
def generation(nb_val: int, nb_min: int, nb_max: int) -> list[int]:
    """Renvoie une liste de *nb_val* nombres entiers aleatoires entre nb_min,
    """
    new_list = []
    for i in range(nb_val):
        new_list.append(random_gen(nb_min, nb_max))
    return new_list

print(generation(10, 5, 10))
```

Autre version, avec une *list comprehension* :

```
def generation_comprehension(nb_val: int, nb_min: int, nb_max: int) -> list[int]:
    return [random_gen(nb_min, nb_max) for i in range(nb_val)]

print(generation_comprehension(10, 5, 10))
```

## 2. Vitesse d'exécution

On veut mesurer (et comparer) le temps d'exécution des fonctions de génération de nombres aléatoires créées précédemment.

On veut en fait comparer les temps de génération pour des listes contenant entre 10 et 1000 éléments (avec plusieurs valeurs intermédiaires).

Pour cela, on utilise la fonction `time.time()`, du module `time`

### Fonctionnement de la fonction `time`

La fonction `time` renvoie le nombre de secondes depuis le début de "l'époque" (Epoch en anglais), c'est-à-dire depuis la "date initiale" définie par votre système d'exploitation. Sur les systèmes UNIX et leurs dérivés, cette date est généralement fixée au 1<sup>er</sup> janvier 1970.

Ce qui est à comprendre, c'est que c'est un nombre qui augmente de 1 chaque seconde (les chiffres après la virgule augmentent continuellement pour avoir une mesure plus précise).

Donc, pour mesurer la durée d'exécution d'une fonction, il suffit de mémoriser dans une variable le résultat de `time` avant l'exécution, puis celui après, et de faire la différence entre ces deux nombres. On obtient ainsi la durée de l'exécution en secondes.

### Temps d'exécution de la première méthode de génération

Voici donc le code :

```

from time import time

# liste des nombres d'éléments dans la liste que l'on veut tester
# on peut aussi utiliser range(10, 10000, 10) par exemple
LIST_NUMBER_OF_ELEMENTS = [10, 100, 1000, 10000]

for number_of_elements in LIST_NUMBER_OF_ELEMENTS:
    # on stocke le moment de début de la génération
    start = time()

    # on génère des nombres aléatoires
    foo = generation(number_of_elements, 42, 73)

    # on stocke le moment de fin de la génération
    end = time()

    # la durée d'exécution est la différence entre le moment de début et de fin
    # Attention : si on inverse end et start, on obtient un nombre négatif
    duration = end - start

    # on arrondi la durée, pour que le tout soit plus lisible
    duration = round(duration, 5)

    # affichage du résultat
    print(f"générer {number_of_elements} à mis {duration} secondes")

```

## Temps d'exécution avec des *list comprehension*

On utilise exactement le même code, mais avec la fonction `generation_comprehension` au lieu de `generation` :

```

LIST_NUMBER_OF_ELEMENTS = [10, 100, 1000, 10000]

for number_of_elements in LIST_NUMBER_OF_ELEMENTS:
    start = time()
    foo = generation_comprehension(number_of_elements, 42, 73)
    end = time()

    duration = end - start
    duration = round(duration, 5)
    print(f"générer {number_of_elements} à mis {duration} secondes")

```

## Pour aller plus loin

### Pour aller plus loin

On peut, par exemple, définir une fonction qui mesure le temps d'exécution d'une autre fonction.

Pour cela, il faut que cette nouvelle fonction (appelons-la `temps_execution`), prenne en argument la fonction dont on mesure le temps d'exécution.

On obtient donc quelque chose comme ça :

```

def temps_execution(fonction_a_tester):
    # ici, on mesure le temps d'exécution (code à ajouter)
    # on peut notamment utiliser la fonction à tester :
    fonction_a_tester(42, 6, 28)
    # l'idéal est de retourner le temps d'exécution plutôt
    # que de mettre un print dans une fonction (ce qui est à
    # éviter en général)
    return 73

```

```
print(temps_execution(generation))
```

### 3. Inverser une liste

On veut écrire une fonction qui inverse l'ordre des éléments d'une liste

On note que, puisqu'en python, les listes ne sont pas modifiables, on devra nécessairement créer une nouvelle liste.

Une première solution fonctionne

#### Avec la méthode reverse

```
def reverse_list (liste: list) -> list:
    # retourne la liste (change le contenu de la variable)
    liste.reverse()
    return liste
```

```
print(reverse_list([1, 2, 3, 4, 5]))
```

#### Avec la fonction reversed

```
def reverse_list_reversed(liste: list) -> list:
    # on est obligé de mettre la fonction list pour que le
    # résultat soit bien une liste (voir le "pour aller plus
    # loin")
    return list(reversed(liste))
```

### Pour aller plus loin - Comprendre la méthode reverse

Si on exécute ce code :

```
l = [1, 2, 3, 5, 8, 13, 21]
r = reversed(l)
print(r)
```

On remarque que `r` n'est pas une liste, mais un itérateur.

Un itérateur est un objet que l'on parcourt (tous les itérateurs peuvent donc être mis dans une boucle `for`).

Le concept d'itérateur est très utile lorsque l'on crée soi-même un objet qui doit être parcouru, car python permet de créer assez facilement ses propres itérateurs.

#### Avec des slice

En python, on peut indexer des listes de façon assez riche. Cela s'appelle des *slices* (des parts en anglais, car on prend des "parts" de la liste).

```
def reverse_list_slice (liste: list) -> list:
    # :: car on prends toute la liste
    # -1 car on a un pas de -1 (donc on recule dans la liste)
    return liste[::-1]
```

```
print(reverse_list_slice([1, 2, 3, 4, 5]))
```

#### Avec la méthode pop

La méthode `pop` des listes permet de retirer le dernier élément d'une liste. Elle retourne l'élément qu'elle retire, ce qui permet d'utiliser cet élément dans une autre fonction

Si on répète l'opération de mettre le dernier élément de l'ancienne liste à la fin de la nouvelle, retourne bien la liste

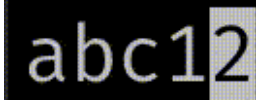


Figure 1: comment se passe l'inversion de la liste

```
def reverse_list_pop (liste: list) -> list:
    new_list = []
    for _ in range(len(liste)):
        new_list.append(liste.pop())
    return new_list
```

```
print(reverse_list_pop([1, 2, 3, 4, 5]))
```

#### Avec la méthode insert + Pop

La méthode `list.insert` permet d'insérer un élément dans une liste, avant l'élément à l'indice précisé.

Dans ce cas, on insère avant l'indice 0, donc au début de la liste. C'est pourquoi on utilise plus `pop()`, mais `pop(0)`, qui va retirer le premier élément au lieu du dernier.

```
def reverse_list_insert (liste: list) -> list:
    new_list = []
    for _ in range(len(liste)):
        new_list.insert(0, liste.pop(0))
    return new_list
```

```
print(reverse_list_insert([1, 2, 3, 4, 5]))
```

## 4. Un élément sur n dans une liste

On cherche à écrire une fonction qui, à partir d'une liste, sélectionne un élément sur `n` dans cette liste.

Par exemple, si  $n = 3$ , on veut transformer cette liste : `[3, 9, 2, 1, 7, 8, 4, 3, 0, 1, 9, 7, 5, 3, 1, 9]` en celle-ci : `[3, 1, 4, 1, 5, 9]`

Voici la liste de test que nous allons utiliser pour la suite :

#### Avec une boucle et une nouvelle liste

On peut utiliser une approche classique : créer la nouvelle liste au fur-et-à-mesure, en parcourant la liste de départ.

#### Avec une condition sur les indices

```
def un_sur_n_indices(n: int, liste: list) -> list:
    """Sélectionne un élément sur `n` dans `liste`"""
    new_list = []
    for i in range(len(liste)):
        # si i est divisible par n (une fois sur n)
        if 0 == i % n:
```

```

        # on ajoute l'élément à l'indice actuel dans la
        # nouvelle liste
        new_list.append(liste[i])
    return new_list

print(un_sur_n_indices(3, [2, 7, 1, 8, 2, 8, 1, 8]))

```

### En utilisant un pas sur range

Une technique plus simple (et plus efficace) est, plutôt que de tester pour tous les indices, d'utiliser un `range` dans lequel on met un pas de `n`.

Cela permet de n'avoir dans la boucle que les indices qui nous intéressent.

```

def un_sur_n_range(n: int, liste: list) -> list:
    new_list = []
    # on met un 0 pour que n soit bien le 3ème argument
    for i in range(0, len(liste), n):
        new_list.append(liste[i])
    return new_list

print(un_sur_n_range(3, [2, 7, 1, 8, 2, 8, 1, 8]))

```

### Avec une *list comprehension*

Pour être encore plus efficace, on peut simplement utiliser un *list comprehension*, en conjonction avec les techniques citées plus haut.

Le code est en fait équivalent, mais permet de créer la liste de façon plus efficace.

### Avec une condition sur les indices

```

def un_sur_n_comprehension_indices(n: int, liste: list) -> list:
    return [liste[i] for i in range(len(liste)) if 0 == i%n]

print(un_sur_n_comprehension_indices(3, [2, 7, 1, 8, 2, 8, 1, 8]))

```

### Retours à la ligne pour plus de clarté

Pour rendre le code plus clair, on peut mettre un retour à la ligne avant le `for` et le `if` :

```

def un_sur_n_comprehension_indices(n: int, liste: list) -> list:
    return [liste[i]
            for i in range(len(liste))
            if 0 == i%n]

```

Cela est très utile quand on construit des expressions complexes, avec par exemple des *list comprehension* à l'intérieur de *list comprehension*.

### En utilisant un pas sur range

On peut à nouveau utiliser un pas sur le `range` pour ne pas avoir à tester toutes les itérations.

```

def un_sur_n_comprehension_range(n: int, liste: list) -> list:
    return [liste[i] for i in range(0, len(liste), n)]

print(un_sur_n_comprehension_range(3, [2, 7, 1, 8, 2, 8, 1, 8]))

```