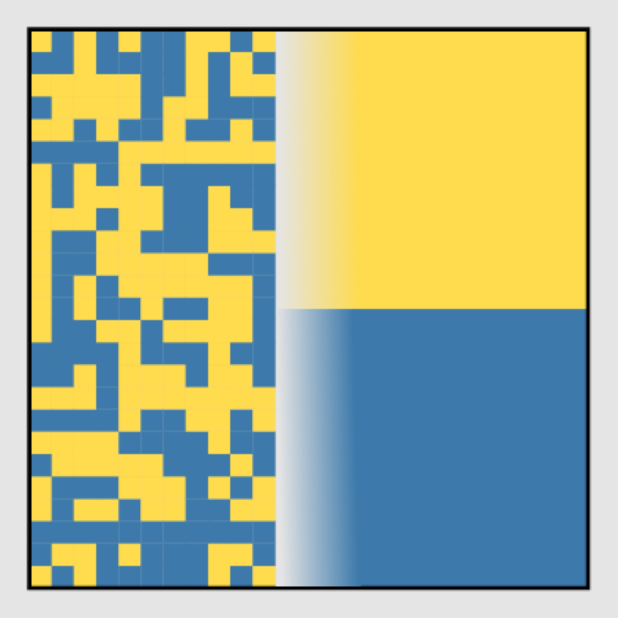
Pascal ORTIZ



Trier en Python

Version du 5 février 2023

Table des matières

La fonction `sorted` . . . 2
Trier avec la méthode `sort` . . . 2
Les fonctions intégrées `min` et `max` . . . 3
Trier « à l'envers » . . . 4
Trier des couples . . . 4
Tri d'une liste de listes de nombres . . . 5
Tri d'une liste de chaînes . . . 6
La fonction `sorted` et les conteneurs . . . 6
La méthode `sort` et les conteneurs . . . 7
Trier suivant une clé . . . 8
Maximum ou minimum suivant une clé . . . 9
Tri de chaînes et casse . . . 9
Tri de chaînes et particularités linguistiques . . . 9
Trier suivant une clé et fonction lambda . . . 10
Indice du maximum ou du minimum . . . 11
Tri avec comparaison complexe . . . 11
Notion de fonction de comparaison . . . 12
Ordre des caractères . . . 13
Trier suivant une clé en utilisant une fonction de comparaison . . . 13
Stabilité du tri en Python . . . 15
Trier des éléments comparables . . . 15
`sorted` et les itérables . . . 16
Performances de la fonction `max` . . . 17
Remarques générales sur le tri en Python . . . 19
Exercice : Deux mots sont-ils anagrammes l'un l'autre ? . . . 19
Exercice : Séparer entiers impairs et entiers pairs d'une liste . . . 20
Exercice : Plus petit multiple de 3 d'une liste d'entiers . . . 20
Exercice : Nombre dont la somme des chiffres est maximale . . . 21
Exercice : Trier suivant le nombre de chiffres . . . 22
Exercice : Trier des dates . . . 23
Exercice : Tri-casier . . . 24
Exercice : Suite croissante puis décroissante . . . 24
Exercice : Tri de fichiers par extension . . . 25
Exercice : Problème du drapeau hollandais . . . 25
Exercice : Trier une liste formée de listes ou de tuples de nombres . . . 33
Exercice : Indice maximum de la valeur minimum . . . 33
Exercice : Unique parmi des triplets . . . 34
Exercice : Trier suivant deux critères . . . 34
Exercice : Utiliser $\texttt{cmp}_{t}o_{k}eypourcomparerdescouples$ . . . 35
Exercice : Majuscule avant minuscule mais 35

La fonction `sorted`

Python dispose d'une fonction standard nommée `sorted` qui permet de trier toute collection d'objets comparables.

Exemple du tri d'une liste de nombres :

```
1 L=[65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3 LL=sorted(L)
4
5 print(LL)
6
7 print()
8
9 print(L)
```

```
10 [65, 31, 9, 32, 81, 82, 46, 12]
11 [9, 12, 31, 32, 46, 65, 81, 82]
12
13 [65, 31, 9, 32, 81, 82, 46, 12]
```

- Ligne 1 : la liste L à trier.
- Ligne 2 : La fonction `sorted` est une fonction built-in du langage Python et triant une liste donnée, ici L .
- Lignes 3 et 11 : la fonction `sorted` crée une **nouvelle** liste (ici LL), qui est effectivement triée.
- Lignes 13 et 10 : La fonction `sorted` ne modifie pas la liste initiale.

Exemple du tri d'une chaîne de caractères (qui est une collection) :

```
s = "nectarine"
L = sorted(s)
print(L)
```

```
['a', 'c', 'e', 'e', 'i', 'n', 'n', 'r', 't']
```

On voit que les lettres composant la chaîne sont écrites dans la liste par ordre alphabétique.

La fonction `sorted` renvoie toujours une liste même si on ne lui donne pas une liste en argument. Plus précisément, la fonction `sorted` crée une (nouvelle) liste qui est triée. Si on donne une liste à `sorted` en argument, elle n'est pas modifiée. Par défaut, la liste renvoyée est triée par ordre croissant.

Trier avec la méthode `sort`

Une **liste** peut être triée avec la **méthode** `sort` :

```
1 L=[65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3 x=L.sort()
4 print(L)
5 print(x)
```

```
6 [65, 31, 9, 32, 81, 82, 46, 12]
7 [9, 12, 31, 32, 46, 65, 81, 82]
8 None
```

— Ligne 3 : `sort` est une méthode qui s'applique à des listes.

— Ligne 3 : la méthode `sort` **modifie** la liste initiale et **ne** crée donc **pas** une nouvelle liste.

— Lignes 3 et 8 : la méthode `sort` ne renvoie rien.

En pratique, on n'écrit donc jamais une affectation de la forme `truc = ma_liste.sort()` et donc, la façon correcte d'utiliser la méthode `sort` est plutôt :

```
L=[65, 31, 9, 32, 81, 82, 46, 12]
print(L)
L.sort()
print(L)
```

```
[65, 31, 9, 32, 81, 82, 46, 12]
[9, 12, 31, 32, 46, 65, 81, 82]
```

La méthode `sort` ne crée pas de nouvelle liste mais en contre-partie, l'ordre initial de la liste est perdu, ce qui est parfois un inconvénient.

Les fonctions intégrées `min` et `max`

Python dispose nativement de fonctions de détermination du minimum et du maximum, par exemple d'une liste de nombres :

```
1 L=[65, 31, 9, 32, 81, 82, 46, 12]
2
3 print(L)
4 print(max(L))
5 print(min(L))
```

```
6 [65, 31, 9, 32, 81, 82, 46, 12]
7 82
8 9
```

En réalité, ces fonctions s'appliquent à n'importe quel conteneur voire n'importe quel itérable :

```
1 T=(65, 31, 9, 32, 81, 82, 46, 12)
2 S=set(T)
3 C= ["rose", "kiwi", "prune", "orange", "ananas"]
4 P= [[5, 4, 6], [6, 0, 0 ], [3, 1, 9]]
5 I=(k**2 for k in range(-4, 2))
6
7
8 print("T :", max(T))
9 print("S :", max(S))
10 print("C :", max(C))
11 print("P :", max(P))
12 print("I :", max(I))
```

```
13 T : 82
14 S : 82
15 C : rose
16 P : [6, 0, 0]
17 I : 16
```

— `T` : Maximum d'un tuple d'entiers

— `S` : Maximum des éléments d'un ensemble d'entiers

— `C` : Maximum d'une liste de chaînes de caractères

— `P` : Maximum d'une liste de listes d'entiers

— `I` : Maximum d'un itérateur qui n'est pas un conteneur.

Il est d'ailleurs possible d'écrire des éléments hors de tout conteneur ou itérateur pour en chercher le maximum :

```
1 m=max(65, 31, 9, 32, 81, 82, 46, 12)
2 print(m)
```

```
3 82
```

Trier « à l'envers »

Pour trier une liste dans l'ordre décroissant, il suffit de passer une « option » `reverse = True` à la fonction `sorted` :

```
1 L=[65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3 LL=sorted(L, reverse = True)
4 print(LL)
```

```
5 [65, 31, 9, 32, 81, 82, 46, 12]
6 [82, 81, 65, 46, 32, 31, 12, 9]
```

— Ligne 3 : on passe à la fonction `sorted` l'argument nommé `reverse` avec la valeur `True` et la liste est triée à l'envers (ligne 6).

C'est analogue pour la méthode `sort`.

Trier des couples

On appellera *couple* (disons d'entiers pour simplifier) toute liste de la forme `[a, b]` où a et b sont des entiers. Par défaut, Python comparera deux couples $[a, b]$ et $[c, d]$ pour **l'ordre lexicographique** autrement dit $[a, b] < [c, d]$ si et seulement si la condition suivante est vraie :

$$a < c \text{ ou } (a = c \text{ et } b < d)$$

Par exemple $[5, 4] < [6, 2]$ et $[5, 4] < [5, 7]$.

Si on demande à la fonction `sorted` d'effectuer un tri d'une liste de couples d'entiers, alors le tri sera effectué suivant le tri lexicographique ci-dessus :

```
L= [[2, 1], [4, 4], [1, 4], [0, 0], [2, 2], [2, 1]]
print(L)
LL=sorted(L,)
print(LL)
```

```
[[2, 1], [4, 4], [1, 4], [0, 0], [2, 2], [2, 1]]
[[0, 0], [1, 4], [2, 1], [2, 1], [2, 2], [4, 4]]
```

Si k est un entier tel que $k \geq 2$, on appelle k-uplet de nombres toute liste de k nombres. Python peut trier non seulement des listes de couples mais plus généralement des listes de k-uplets. Pour classer les k-uplets, Python utilise l'ordre lexicographique, comme pour le cas des couples :

```
L=[[2, 2, 2], [0, 1, 4], [3, 4, 0], [3, 4, 3], [2, 4, 0], [2, 4, 0]]
print(L)
LL=sorted(L)
print(LL)
```

```
[[2, 2, 2], [0, 1, 4], [3, 4, 0], [3, 4, 3], [2, 4, 0], [2, 4, 0]]
[[0, 1, 4], [2, 2, 2], [2, 4, 0], [2, 4, 0], [3, 4, 0], [3, 4, 3]]
```

Tout ce qui précède s'adapte à la méthode `sort`.

Tri d'une liste de listes de nombres

Pour Python, deux listes de nombres sont toujours comparables via l'ordre lexicographique, même si les deux listes n'ont pas même longueur. Il en résulte, même si l'intérêt ne semble pas apparent, que l'on peut trier toute liste formée de listes ou de tuples de nombres :

```
L = [[2, 5, 0, 4], [5, 4, 8, 4], [8, 5, 8, 8], [6, 4], [0, 8, 2, 8]]
print(sorted(L))
```

```
[[0, 8, 2, 8], [2, 5, 0, 4], [5, 4, 8, 4], [6, 4], [8, 5, 8, 8]]
```

Ce qui précède s'applique encore pour une liste (ou un tuple) de tuples de nombres :

```
L = [(2, 5, 0, 4), (5, 4, 8, 4), (8, 5, 8, 8), (6, 4), (0, 8, 2, 8)]
print(sorted(L))
```

```
[(0, 8, 2, 8), (2, 5, 0, 4), (5, 4, 8, 4), (6, 4), (8, 5, 8, 8)]
```

Toutefois, la liste ne peut pas contenir **à la fois** des listes et des tuples car ces derniers ne sont pas comparables :

```
L = [[2, 5, 0, 4], [5, 4, 8, 4], (8, 5, 8, 8), (6, 4), (0, 8, 2, 8)]
print(sorted(L))
```

```
TypeError: unorderable types: tuple() < list()
```

— L contient à la fois des listes et des tuples

Tout ce qui précède s'adapte à la méthode `sort`.

Tri d'une liste de chaînes

Un tri d'un conteneur de chaînes de caractères a toujours un sens. En effet, deux caractères étant toujours comparables via leur code unicode, deux chaînes sont toujours comparables pour l'ordre lexicographique qui est l'ordre utilisé par défaut par Python.

Voici un exemple de tri de chaînes :

```
1 L = ["rose", "kiwi", "prune", "orange", "ananas"]
2 print(sorted(L))
```

```
3 ['ananas', 'kiwi', 'orange', 'prune', 'rose']
```

Noter que l'ordre obtenu ne coïncide pas toujours avec l'ordre alphabétique usuel ; par exemple, une majuscule est toujours classée avant une minuscule :

```
1 L = ["rose", "kiwi", "Prune", "orange", "ananas"]
2 print(sorted(L))
```

```
3 ['Prune', 'ananas', 'kiwi', 'orange', 'rose']
```

Tout ce qui précède s'adapte à la méthode `sort`.

La fonction `sorted` et les conteneurs

La fonction `sorted` est susceptible de trier tout conteneur. Quel que soit le conteneur (liste, chaîne, etc), la fonction renvoie **toujours** une **liste**, par défaut, triée par ordre croissant des éléments du conteneur.

On peut ainsi trier des tuples ou des ensembles :

```
1 t=(65, 31, 9, 32, 81, 82, 46, 12)
2 print(sorted(t))
3
4 s=set([65, 31, 9, 32, 81, 82, 46, 12])
5 print(s)
6 print(sorted(s))
```

```
7 [9, 12, 31, 32, 46, 65, 81, 82]
8 {32, 65, 9, 12, 46, 81, 82, 31}
9 [9, 12, 31, 32, 46, 65, 81, 82]
```

On peut trier les lettres d'une chaîne :

```
1 print(sorted("fruit"))
```

```
2 ['f', 'i', 'r', 't', 'u']
```

Le tri d'un dictionnaire renvoie seulement la liste triée des **clés** du dictionnaire :

```
1 romains = {1:'I', 5:'V', 10:'X', 50:'L',  100:'C', 500:'D', 1000:'M'}
2 print(sorted(romains))
```

```
3 [1, 5, 10, 50, 100, 500, 1000]
```

— Ligne 2 : `sorted` prend en argument un dictionnaire, ici `romains`.
— Ligne 3 : seules les clés du dictionnaire sont triées.

La méthode `sort` et les conteneurs

La méthode `sort` appliquée à une liste **modifie** la liste en la triant. Par conséquent, la méthode `sort` s'applique essentiellement à des listes et ne s'applique pas à des conteneurs immutables ou non séquentiels. Ainsi, elle ne s'applique pas à des chaînes (car immutables) :

```
1 s="fruit"
2 s.sort()
```

```
3 Traceback (most recent call last):
4   File "trier_une_chaine_avec_sort.py", line 2, in <module>
5     s.sort()
6 AttributeError: 'str' object has no attribute 'sort'
```

Elle ne s'applique pas non plus à des ensembles (car non séquentiels) :

```
1 s = {65, 31, 12, 81, 81, 65, 81, 12}
2 s.sort()
3 print(s)
```

```
4 Traceback (most recent call last):
5   File "_.py", line 2, in <module>
6     s.sort()
7 AttributeError: 'set' object has no attribute 'sort'
```

Elle ne s'applique pas à des tuples (qui sont immutables) :

```
1 s = (65, 31, 12, 81, 81, 65, 81, 12)
2 s.sort()
3 print(s)
```

```
4 Traceback (most recent call last):
5   File "_.py", line 2, in <module>
6     s.sort()
7 AttributeError: 'tuple' object has no attribute 'sort'
```

Elle ne s'applique pas à des dictionnaires (car non séquentiels) :

```
1 romains = {1:'I', 5:'V', 10:'X', 50:'L',  100:'C', 500:'D', 1000:'M'}
2 print(romains.sort())
```

```
3 Traceback (most recent call last):
4   File "_.py", line 2, in <module>
5     print(romains.sort())
6 AttributeError: 'dict' object has no attribute 'sort'
```

Trier suivant une clé

Quand on effectue un tri d'une liste d'objets, on dispose implicitement ou explicitement d'une fonction dite *de comparaison* qui peut comparer deux objets quelconques et décider quel est le plus petit des deux.

Mais il est possible de proposer une fonction qui va modifier la comparaison par défaut, comme on va le voir sur un exemple. Soit une liste d'individus dont on connaît le nom et l'âge. Cette liste est donnée sous forme de couples [nom, age] comme `["marc", 24]`. On veut, par exemple trier cette liste suivant l'**âge de chaque individu**. Dans le jargon, on dit que l'âge constitue une *clé* du tri. Cela signifie que la fonction de comparaison du tri est en fait la fonction qui compare les âges. Par exemple, [”philippe”, 42] $\leq$ [”mireille”, 57] car $42 < 57$ alors que les prénoms sont dans un ordre différent.

Voici comment on peut trier en Python une liste d'individus par âges. Le tri utilise une clé de comparaison :

```
1 L=[ ["marc", 24], ["paul", 38], ["jeanne", 15],
2    ["jean", 21], ["philippe", 42], ["mireille", 57]]
3
4 def age(M):
5     return M[1]
6
7 print(sorted(L, key=age))
```

```
8 [['jeanne', 15], ['jean', 21], ['marc', 24],
9 ['paul', 38], ['philippe', 42], ['mireille', 57]]
```

— Lignes 7 et 8 : La fonction `sorted` trie une liste de paires. On observe que les paires sont bien triées par âges.

— Lignes 4-5 : on définit une fonction `age` qui renvoie l'élément sur lequel le tri est effectué : ici, le paramètre `M` est censé être une paire `(prénom, âge)` et la fonction va retourner l'âge, qui est ici le critère de tri.

— Ligne 7 : la fonction `sorted` est appelée avec un argument nommé `key` et qui prend la valeur d'une fonction qui retourne une valeur suivant laquelle le tri sera effectué, ici la fonction `age`. Autrement dit, le tri décide que le résultat de la comparaison (être plus petit ou être plus grand) de deux couples `M1` et `M2` est le résultat de la comparaison des nombres `age(M1)` et `age(M2)`.

D'une manière générale, si on utilise `sorted` en passant à `key` une fonction, disons `f`, pour trier une liste `L`, chaque fois que `sorted` devra comparer deux éléments de `L`, disons `x` et `y`, alors `x` sera considéré comme plus petit que `y` exactement lorsque `f(x) <= f(y)` où `<=` désigne l'ordre standard défini par Python.

En pratique, si le critère de tri n'est pas immédiat à appréhender, il pourra être délicat d'implémenter la fonction de comparaison du tri.

La documentation officielle précise que la fonction de comparaison doit obligatoirement être nommée, autrement dit, le paramètre n'est pas positionnel et l'option `key` doit apparaître forcément avec la syntaxe `key=` suivie d'une référence vers la fonction, cf. ligne 7 du code ci-dessus.

Tout ce qui précède s'adapte à la méthode `sort`.

Maximum ou minimum suivant une clé

Supposons qu'on ait une liste `L` d'individus donnés sous forme de paire [`nom`, `age`] et qu'on recherche une personne parmi elles qui soit la plus âgée. Il est possible de passer un argument particulier à `max` afin qu'il ne prenne en compte que l'âge dans la paire [`nom`, `ge`]. Voici un code qui utilise cette technique :

```
1 L=[ ["marc", 24], ["paul", 38], ["jeanne", 15],
2     ["jean", 21], ["philippe", 42], ["mireille", 57]]
3
4 def age(M):
5     return M[1]
6
7 print(max(L, key=age))
```

```
8 ['mireille', 57]
```

— Ligne 8 : on observe bien que la personne la plus âgée de la liste est `"mireille"`.
— Lignes 4-5 : `age(M)` renvoie l'âge de la paire `M` : c'est sur cette valeur que la maximum est recherché.
— Ligne 7 : l'appel utilise outre la liste, un argument nommé `key`.

Tout ce qui précède s'adapte à la fonction `min`.

Tri de chaînes et casse

Par défaut, le tri de chaînes n'ignore pas la casse :

```
1 print(sorted(["begonia", "Rose", "dahlia"]))
```

```
2 ['Rose', 'begonia', 'dahlia']
```

Pour ignorer la casse dans un tri de chaînes, il suffit de comparer les chaînes suivant leur écriture en minuscules. Il suffit donc d'utiliser `sorted` avec une clé qui convertit chaque chaîne en minuscules :

```
1 def clef_casse(s):
2     return s.lower()
3
4 print(sorted(["begonia", "Rose", "dahlia"], key=clef_casse))
```

```
5 ['begonia', 'dahlia', 'Rose']
```

— Ligne 5 : noter que les éléments de la liste initiale ne sont pas modifiés et gardent leurs majuscules s'ils en ont.

Tout ce qui précède s'adapte à la méthode `sort`.

Tri de chaînes et particularités linguistiques

Pour obtenir un tri qui tienne compte de l'ordre alphabétique en usage et en particulier des accents, des caractères spéciaux, il faut activer le module standard de localisation (qui permet

de tenir compte des conventions linguistiques) :

```
1 import locale
2 locale.setlocale(locale.LC_ALL, "")
3
4 L = ["Prune", "prune", "levées", "lève-glaces",
5      "lèvent", "leçon", "Lecteur"]
6
7 print(sorted(L, key=locale.strxfrm))
```

```
8 ['leçon', 'Lecteur', 'levées', 'lève-glaces',
9 'lèvent', 'prune', 'Prune']
```

— Ligne 1 : importation du module standard de localisation.

— Ligne 2 : activation de la locale.

— Ligne 6 : pour que le tri tienne compte des contraintes linguistiques, il faut utiliser la clé `strxfrm` du module `locale`

— Ligne 7 : on notera que le tri ignore la casse (puisque le mot *Lecteur* est placé après la mot *leçon*.

Tout ce qui précède s'adapte à la méthode `sort`.

Trier suivant une clé et fonction lambda

L'option key de la fonction `sorted` référence une fonction, définie avant l'appel de `sorted` :

```
1 L=[ ["marc", 24], ["paul", 38], ["jeanne", 15], ["jean", 21],
2     ["philippe", 42],  ["mireille", 57]]
3
4 def age(M):
5     return M[1]
6
7 print(sorted(L, key=age))
```

```
8 [['jeanne', 15], ['jean', 21], ['marc', 24], ['paul', 38],
9 ['philippe', 42], ['mireille', 57]]
```

Or, Python permet de définir certaines fonctions par des **expressions** dite `lambda`. L'avantage est que la fonction ainsi définie peut donc être directement placée lors de l'« affectation » `key=` dans l'appel.

Voici le code équivalent du code ci-dessus en utilisant une fonction `lambda` :

```
1 L=[["marc", 24], ["paul", 38], ["jeanne", 15], ["jean", 21],
2    ["philippe", 42], ["mireille", 57]]
3
4 print(sorted(L, key=lambda M : M[1]))
```

```
5 [['jeanne', 15], ['jean', 21], ['marc', 24], ['paul', 38],
6 ['philippe', 42], ['mireille', 57]]
```

— Ligne 4 : La définition de fonction age a disparue et le membre de droite de l'« affection » de key est une fonction (anonyme) définie « en ligne ».

Tout ce qui précède s'adapte à la méthode sort.

Indice du maximum ou du minimum

On dispose d'une liste L d'entiers et on cherche l'indice d'un élément de la liste L de valeur minimale (c'est analogue pour le maximum).

Pour cela, on peut générer les tuples (v, i) où v est un élément de la liste L et i l'indice corresponsant et prendre le minimum de ces tuples ce qui donnera un tuple (v, i) dont le 2e élément est un indice du minimum :

```
1 def imin(L):
2     return min((z,i) for i,z in enumerate(L))[1]
3
4 L=[40, 30, 10, 70, 10, 40, 10, 20]
5
6 i = imin(L)
7 print(i, L[i], min(L))
```

```
8 2 10 10
```

L'inconvénient de cette méthode est qu'elle génère une liste de tuples et qu'elle peut donc être assez lente sur une liste de grande taille.

Il existe une autre méthode, signalée dans ce message sur StackOverflow et utilisant astucieusement l'argument nommé key de la fonction min :

```
1 def imin(L):
2     return min(range(len(L)), key=L.__getitem__)
3
4 L=[40, 30, 10, 70, 10, 40, 10, 20]
5
6 i = imin(L)
7 print(i, L[i], min(L))
```

```
8 2 10 10
```

On parcourt la liste des indices de L et pour chaque indice i, on prend le minimum des valeurs de (L.__getitem__)(i) qui n'est autre que L[i] mais c'est l'indice qui est renvoyé puisqu'on parcourt la liste **indices** de L.

Cette dernière méthode est plus rapide que la première, d'un facteur de l'ordre de 2,5.

Tri avec comparaison complexe

Cette unité contient des informations occasionnellement utiles dans le codage de certains algorithmes et donc son intérêt est, ici, secondaire.

Pour ce qui suit, il pourra être utile de remarquer que pour trier une liste de nombres en Python dans l'ordre décroissant, il suffit de choisir la clé de comparaison qui compare les opposés. En

effet, pour des nombres a et b on $a \geq b$ si et seulement si $-a \leq -b$. Observons alors le code suivant :

```
1 L=[65, 31, 9, 32, 81, 82, 46, 12]
2 print(L)
3
4 def f(a):
5     return -a
6
7 LL=sorted(L, key = f)
8 print(LL)
```

```
9 [65, 31, 9, 32, 81, 82, 46, 12]
10 [82, 81, 65, 46, 32, 31, 12, 9]
```

En utilisant la clé de comparaison `f`, la liste `LL` a été triée par ordre décroissant.

On veut désormais trier une liste de couples de la forme `[x, y]` en triant d'abord suivant `x` dans l'ordre croissant puis en cas d'égalité, que le tri soit effectué suivant `y` et dans l'ordre décroissant. Pour cela, on s'appuie sur la façon dont Python trie par défaut des couples (l'ordre lexicographique) et sur l'astuce signalée ci-dessus. Voici un code permettant d'effectuer le tri indiqué :

```
1 L=[[5, 9], [8, 6], [9, 5], [3, 1], [4, 6], [7, 2], [1, 4], [4, 3],
2    [8, 6], [3, 6], [3, 5], [8, 2]]
3
4 def comp(x):
5     return [x[0], -x[1]]
6
7 print(sorted(L, key=comp))
```

```
8 [[1, 4], [3, 6], [3, 5], [3, 1], [4, 6], [4, 3], [5, 9], [7, 2], [8, 6],
9 [8, 6], [8, 2], [9, 5]]
```

— Lignes 1-2 : la liste à trier suivant la règle précisée ci-dessus.

— Ligne 8 : on observe par exemple que, dans le tri, `[4, 6]` est avant `[4, 3]` et en effet, `6>4`.

— Lignes 4-5 : La fonction de comparaison du tri compare d'abord les premières coordonnées et en cas d'égalité, compare les opposés.

Tout ce qui précède s'adapte à la méthode `sort`.

Notion de fonction de comparaison

Dans certaines circonstances, il peut être utile d'utiliser une fonction de comparaison `cmp(x, y)` qui traduira que l'objet `x` est, pour cet ordre, inférieur à l'objet `y`. Par convention,

— une valeur strictement négative de `cmp(x, y)` traduira que `x` $<$ `y`,

— une valeur strictement positive de `cmp(x, y)` traduira que `y` $<$ `x`,

— une valeur nulle de `cmp(x, y)` traduira que `x` $==$ `y`.

Ainsi, la comparaison usuelle de nombres est traduite par la fonction suivante :

```
def cmp_nb(x, y):
    return x - y
```

La comparaison usuelle de chaînes (l'ordre lexicographique) est traduite par la fonction récursive suivante :

```
def cmp_str(x, y):
    if 0 in (len(x),len(y)):
        return len(x)-len(y)
    x0=x[0]
    y0=y[0]
    if x0<y0:
        return -1
    if x0>y0:
        return 1
    return cmp(x[1:], y[1:])
```

Une fonction de comparaison pourra servir (mais c'est assez rare en pratique) pour créer une fonction à passer comme argument nommé `key` à la fonction `sorted` (et analogues comme la méthode `sort`, ou les fonctions `min` et `max`).

Ordre des caractères

Pour comparer des chaînes, Python utilise l'ordre lexicographique qui est lui-même basé sur l'ordre des caractères. Il se trouve que chaque caractère unicode a un numéro d'ordre (une valeur unicode) et c'est ce numéro qui est utilisé pour classer les caractères. Par exemple, la lettre minuscule `m` a pour valeur unicode l'entier 109. Pour obtenir la valeur unicode d'un caractère, on utilise la fonction standard `ord` :

```
print(ord("m"))
```

```
109
```

Les caractères `a`, `b`, etc reçoivent les valeurs consécutives 97, 98, etc. Les caractères majuscules `A`, `B`, etc reçoivent les valeurs 65, 66, etc. Donc, un caractère majuscule est toujours classé **avant** un caractère minuscule :

```
print("M" < "c")
```

```
True
```

Par conséquent, une fonction de comparaison des caractères pourrait être :

```
def cmp_car(x, y):
    return ord(x)-ord(y)
```

Trier suivant une clé en utilisant une fonction de comparaison

Lorsqu'on veut trier suivant une clé et que la règle de comparaison n'est pas simple, il peut être difficile de déterminer la fonction à donner à `sorted` en argument `key`. Par exemple, supposons que l'on veuille trier des mots en utilisant l'ordre suivant : c'est l'ordre alphabétique habituel, indépendamment du fait que les lettres soient majuscules ou pas, sauf que si on compare deux

lettres identiques, l'une en majuscule, l'autre en minuscule, la majuscule **est avant** la minuscule. Ainsi, selon cet ordre, les chaînes suivantes sont classées dans l'ordre croissant :

```
['ABB', 'aA', 'aa', 'aaB', 'aBab', 'BBA', 'BBAb', 'Bb', 'ba', 'bB']
```

Il n'est pas forcément évident de trouver la clé de comparaison.

Python donne la possibilité

(1) d'écrire directement une fonction de comparaison `cmp(x, y)` traduisant la comparaison de x et de y

(2) de transformer cette fonction de comparaison en une autre fonction qui sera la clé à donner à la fonction `sorted`.

Pour obtenir la fonction-clé, il faut utiliser la fonction `cmp_to_key` du module standard `functools`. Si `cmp` est la fonction de comparaison que nous avons écrite alors la clé sera le résultat de l'appel `cmp_to_key(cmp)`.

Ecrivons le code pour le tri ci-dessus :

```
1  from functools import cmp_to_key
2
3  def cmp(x, y):
4      if 0 in (len(x),len(y)):
5          return len(x)-len(y)
6      x0=x[0]
7      y0=y[0]
8
9      if x0.lower()!= y0.lower():
10         return ord(x0.lower())- ord(y0.lower())
11
12     if x0!=y0:
13         return ord(x0)-ord(y0)
14
15     return cmp(x[1:], y[1:])
16
17
18 from random import randrange
19 tests=list(set(['',.join(
20                          ['aAbB'[randrange(4)]
21                           for _ in range(randrange(2,5))])
22                 for i in range(10)]))
23
24 print(tests)
25 print(sorted(tests, key=cmp_to_key(cmp)))
```

```
26 ['bb', 'aa', 'ABBb', 'aaAB', 'aB', 'BA', 'aAa', 'bbB', 'Aa']
27 ['Aa', 'ABBb', 'aAa', 'aa', 'aaAB', 'aB', 'BA', 'bb', 'bbB']
```

La règle de construction de la fonction récursive `cmp` (lignes 3-15) est la suivante :

— une chaîne vide est toujours plus petite que n'importe quelle chaîne non vide ;

— pour comparer deux chaînes non vides, on examine leur premier caractère à chacune.

  — Si ces caractères sont différents (en faisant abstraction de la casse), comme "aB" et "BA"

mais pas comme "ABBb" et "aaAB", l'ordre va dépendre uniquement de l'ordre alphabétique de ce premier caractère en ignorant la casse (lignes 9-10) ; pour "aB" et "BA", on doit comparer "a" et "b" ce qui permet de dire que `"aB" < "BA"`.

— Si ces caractères, en faisant abstraction de la casse, sont les mêmes (comme "ABBb" et "aaAB"), alors si les caractères sont identiques (en tenant compte de la casse cette fois), il faut se décider en examinant les caractères suivants (ligne 15) comme c'est le cas pour "ABBb" et "aaAB", sinon, la majuscule sera placée avant la minuscule (ligne 13), par exemple `"ABBb" < "aAa"`.

Une fois écrite la fonction `cmp` (lignes 3-15), la fonction `cmp_to_key` (ligne 25) va se charger de convertir cette fonction en une fonction directement utilisable par la fonction `sorted` (ou ses analogues). Pour cela, il faut l'importer du module `functools` (ligne 1).

Stabilité du tri en Python

Lorsque la fonction `sorted` effectue un tri à l'aide d'une clé, les éléments ayant des clés de même valeur sont placés dans la liste triée dans le même ordre que dans la liste initiale (On dit que le tri effectué en Python est un *tri stable*).

Illustrons avec le code ci-dessous qui effectue un tri d'individus suivant l'âge :

```
1 L = [["marc", 24], ["paul", 38], ["jeanne", 15], ["jean", 24],
2      ["philippe", 42], ["mireille", 57]]
3
4 def age(M):
5     return M[1]
6
7 print(sorted(L, key=age))
```

```
8 [['jeanne', 15], ['marc', 24], ['jean', 24], ['paul', 38],
9 ['philippe', 42], ['mireille', 57]]
```

— Lignes 1-2 : les individus `"marc"` et `"jean"` ont même âge (24) et dans la liste `L`, `"marc"` est placé avant l'individu `"jean"`.

— Lignes 8-9 : dans la liste triée, `"marc"` est **encore placé avant** `"jean"` (alors que dans l'ordre alphabétique, c'est le contraire).

Tout ce qui précède s'applique à la méthode `sort`.

Trier des éléments comparables

Quand on trie (une liste par exemple), les éléments doivent être comparables. Or deux objets, pour Python, ne sont pas toujours comparables. Par exemple,

— un entier et une chaîne

— une liste et un tuple

ne sont pas comparables :

```
1 print("rose" < 2020)
```

```
Traceback (most recent call last):
  File "a.py", line 1, in <module>
    print("rose" < 2020)
TypeError: unorderable types: str() < int()
```

ou encore

```
print([42, 2020] < (42, 2020))
```

```
Traceback (most recent call last):
  File "a.py", line 1, in <module>
    print([42, 2020] < (42, 2020))
TypeError: unorderable types: list() < tuple()
```

Par conséquent, certaines listes ne peuvent pas être triées si elles comportent des éléments non comparables :

```
L=sorted(["rose", 2020])
print(L)
```

```
Traceback (most recent call last):
  File "a.py", line 1, in <module>
    L=sorted(["rose", 2020])
TypeError: unorderable types: int() < str()
```

Tout ce qui précède s'adapte à la méthode `sort`.

`sorted` et les itérables

`sorted` prend en réalité comme argument tout itérable. Par exemple, voici comment on peut trier les lignes d'un fichier texte.

Soit le fichier texte suivant :

```
rose
kiwi
prune
orange
ananas
```

et soit le code qui trie les lignes du fichier :

```
fruits=open("fruits.txt")
L=sorted(fruits)
print(L)
```

```
['ananas\n', 'kiwi\n', 'orange\n', 'prune\n', 'rose\n']
```

Pourtant un objet de type `file` n'est pas une liste (mais c'est un itérable sur les lignes du fichier).

Ce qui précède ne s'applique pas à la méthode `sort` : un itérable n'a aucune raison de disposer d'une méthode `sort`.

Performances de la fonction `max`

La fonction `max` est une fonction builtin et, en CPython (l'implémentation ultra-dominante de Python), elle est écrite en C. Elle n'est pourtant pas extraordinairement performante. Pour le voir, faisons quelques tests.

On implémente en Python **pur** une fonction `maxi` équivalente de la fonction `max`, on génère 5 millions d'entiers aléatoires et on compare les temps réalisés par ces deux fonctions appliquées à la liste :

```
from time import perf_counter
from random import randrange

def maketest(n, m):
    return list(randrange(m) for _ in range(n))

def maxi(L):
    a=L[0]
    for z in L:
        if z>a:
            a=z
    return a

n=5*10**7
m=10**7
L=maketest(n, m)

print("go!")

begin_perf = perf_counter()

m1=maxi(L)

delta = perf_counter() - begin_perf

print(f"maxi : {delta:.2f}s")

begin_perf = perf_counter()

m2=max(L)

delta = perf_counter() - begin_perf

print(f"max : {delta:.2f}s")

print(m1==m2)
```

qui affiche

```
go!
maxi : 0.96s
max : 0.54s
True
```

On voit que le temps de la fonction `max` est seulement `1,8` fois plus rapide que la fonction écrite en Python pur. Si on refait l'expérience avec la fonction `sum` au lieu de `max`, le ratio est beaucoup plus important, de l'ordre de `7,8`.

En réalité, si on appelle la fonction sous la forme `max(*L)` au lieu de `max(L)` à la ligne 30, le résultat est légèrement plus lent qu'en Python pur.

Pire, si on utilise la fonction `max` pour rechercher un **indice** de la liste où le maximum est atteint, le code en Python pur est tout bonnement plus rapide :

```
from time import perf_counter
from random import randrange

def maketest(n, m):
    return list(randrange(m) for _ in range(n))

n=5*10**7
m=10**7

L=maketest(n, m)

print("go!")

def index_max(L):
    a=L[0]
    imax=0
    for i,z in enumerate(L):
        if z>a:
            a, imax = z,i
    return imax


begin_perf = perf_counter()

i=index_max(L)

delta = perf_counter() - begin_perf

print(f"py : {delta:.2f}s")



begin_perf = perf_counter()

_,j=max((z,i) for i,z in enumerate(L))
```

```
delta = perf_counter() - begin_perf

print(f"builtin : {delta:.2f}s")

print(L[i] == L[j] == max(L))
```

qui affiche

```
go!
py : 2.05s
builtin : 4.78s
True
```

Remarques générales sur le tri en Python

L'algorithme de tri qu'utilisent la fonction `sorted` ou la méthode `sort` est spécifique à Python. Il porte le nom de *timsort* et a été inventé par un Python Core Developper du nom de Tim Peters. Cet algorithme est réutilisé dans l'implémentation Java du tri des objets non primitifs.

On trouvera une documentation sur le tri en Python dans le Sorting HOW TO.

Exercice : Deux mots sont-ils anagrammes l'un l'autre ?

Deux mots sont des anagrammes l'un de l'autre si en permutant les lettres du premier, on obtient le second. Par exemple, les deux mots NUMERISAIT et MINIATURES sont des anagrammes.

Ecrire une fonction `sontAnagrammes(a, b)`, utilisant la fonction `sorted` et qui renvoie `True` si les mots `a` et `b` sont anagrammes l'un de l'autre et `False` sinon.

Solution

Pour savoir si deux mots sont anagrammes, il suffit de trier leurs lettres dans l'ordre alphabétique et de comparer les listes obtenues. Les mots sont anagrammes si et seulement si les listes sont identiques. Et en Python, on peut comparer deux listes avec l'opérateur `==`.

D'où le code suivant :

```
1 def sontAnagrammes(a, b):
2     return sorted(a)==sorted(b)
3
4 a="NUMERISAIT"
5 b="MINIATURES"
6 print(a, b)
7 print(sontAnagrammes(a, b))
```

```
8 NUMERISAIT MINIATURES
9 True
```

Exercice : Séparer entiers impairs et entiers pairs d'une liste

On donne une liste L d'entiers et on demande de créer une nouvelle liste M où les entiers pairs de L viennent d'abord et sont suivis des entiers impairs de L. Par exemple :

```
L=[4, 8, 4, 3, 2, 9, 7, 4, 9]
M=[4, 8, 4, 2, 4, 3, 9, 7, 9]
```

On utilisera l'argument nommé key de la fonction `sorted`.

Solution

```
from random import randrange

L=[randrange(1, 10) for _ in range(9)]
print(L)
M=sorted(L, key=lambda z:z%2)
print(M)
```

```
[4, 8, 4, 3, 2, 9, 7, 4, 9]
[4, 8, 4, 2, 4, 3, 9, 7, 9]
```

Exercice : Plus petit multiple de 3 d'une liste d'entiers

On donne une liste L d'entiers et on demande de déterminer en utilisant la fonction `min` et une clé key de déterminer le plus petit multiple de 3 dans la liste L.

Par exemple, si

```
L = [97, 32, 46, 40, 67, 69, 35, 48, 99, 88]
```

alors le plus petit multiple de 3 de L est 48. On supposera que L contient au moins un multiple de 3.

(on pourra consulter par exemple le premier exercice des sélections de Prologin 2022)

Solution

Il s'agit de construire la fonction-clé, disons `f`, utilisée par la fonction `min` en sorte qu'un entier de la liste non multiple de 3 soit considéré comme « le plus grand possible ». Pour cela il suffit qu'il soit considéré comme infini. En Python, un nombre représentant $+\infty$ est défini par `float("+inf")` ou même `float("inf")`. Si l'entier est multiple de 3 alors la fonction `f` conservera l'entier.

D'où le code ci-dessous :

```
def f(k):
    if k % 3 != 0:
        return float("inf")
    else:
        return k


L = [97, 32, 46, 40, 67, 69, 35, 48, 99, 88]
```

```
print(min(L, key=f))
```

```
48
```

On a écrit d'abord la condition qui teste quand l'entier n'est pas multiple de 3 car elle est deux fois plus fréquente en moyenne.

Il est possible de simplifier le code de la fonction `f` en utilisant une *expression* `if/else` (et non plus une *instruction* `if/else`). En outre `k % 3 != 0` a même valeur booléenne que `k % 3`, d'où le code :

```
def f(k):
    return float("inf") if k % 3 else k


L = [97, 32, 46, 40, 67, 69, 35, 48, 99, 88]

print(min(L, key=f))
```

```
48
```

Il est même possible, en utilisant une expression `lambda`, d'écrire la définition de la fonction `f` à **l'intérieur** de l'appel de la fonction `min`. D'où le code encore simplifié suivant :

```
L = [97, 32, 46, 40, 67, 69, 35, 48, 99, 88]

print(min(L, key=lambda k: float("inf") if k % 3 else k))
```

```
48
```

Solution alternative

Bien sûr, il aurait été possible de procéder en deux temps :

— construire la liste des multiples de 3 appartenant la liste `L`,

— en prendre le minimum.

L'inconvénient est qu'on effectue deux parcours de liste au lieu d'un. Voici néanmoins le code correspondant :

```
L = [97, 32, 46, 40, 67, 69, 35, 48, 99, 88]

M = [k for k in L if k % 3 == 0]

print(min(M))
```

```
48
```

Exercice : Nombre dont la somme des chiffres est maximale

Soit à chercher un élément dans une liste de nombres entiers dont la somme des chiffres est la plus grande. Par exemple, si la liste à trier est constituée de `[42, 2020, 9, 38, 1024]` alors l'élément cherché est 38 car la somme de ses chiffres qui vaut $8 + 3 = 11$ est la plus grande de toutes les sommes des chiffres de nombres de la liste.

Pour calculer la somme des chiffres, on utilisera les constructeurs `int` et `str` ainsi que la fonction built-in `sum` et des listes en compréhension.

Solution

```
1 def somme_chiffres(n):
2     chiffres = [int(c) for c in str(n)]
3     return sum(chiffres)
4
5 L = [42, 2020, 9, 38, 1024]
6 m = max(L, key = somme_chiffres)
7 print(m)
```

```
8 38
```

— Lignes 1-3 : on définit, indépendamment de `L` et du tri, une fonction `somme_chiffres` qui renvoie la somme des chiffres de l'entier `n`. Pour cela, `somme_chiffres` convertit le nombre en chaîne avec `str`, en extrait la liste des chiffres sous forme de nombres (cf. ligne 2 avec `int`) et les somme (ligne 3).

— Ligne 7 : la fonction `somme_chiffres` est passée en argument à la fonction `max` avec le mot-clé `key`.

La syntaxe sans recours à une quelconque liste est également possible :

```
1 def somme_chiffres(n):
2     return sum(int(c) for c in str(n))
3
4 m = max(42, 2020, 9, 38, 1024, key = somme_chiffres)
5 print(m)
```

```
6 38
```

Exercice : Trier suivant le nombre de chiffres

On donne une liste d'entiers positifs, par exemple :

```
L = [2018, 0, 42, 1970, 1, 9, 42, 81, 10000000,
     99, 8, 2038, 421, 100, 33, 9999]
```

En utilisant l'argument nommé `key`, trier les entiers de `L` suivant leur nombre de chiffres ie d'abord les entiers ayant un chiffre, puis ceux auyant deux chiffres, etc.

Avec l'exemple ci-dessus, on doit obtenir :

```
[0, 1, 9, 8, 42, 42, 81, 99, 33,
421, 100, 2018, 1970, 2038, 9999, 10000000]
```

Solution

Le nombre de chiffres de l'entier $n \geq 0$ est obtenu en convertissant `n` en chaîne de caractères via la fonction str puis en en prenant la longueur via la fonction `len`, donc le nombre de chiffres de `n` est `len(str(n))`. Si $n > 0$ on peut aussi utiliser le logarithme décimal :

```
from math import log10

n=2020
N=int(1+log10(abs(n)))
print(N)
```

qui affiche

```
4
```

Il ne reste plus qu'à trier avec `sorted` (ou la méthode `sort`) en utilisant l'argument nommé `key` pointant vers une fonction calculant le nombre de chiffres. D'où le code :

```
L = [2018, 0, 42, 1970, 1, 9, 42, 81, 10000000,
     99, 8, 2038, 421, 100, 33, 9999]
print(L)
M=sorted(L, key=lambda n:len(str(n)))
print(M)
```

ce qui affiche

```
[2018, 0, 42, 1970, 1, 9, 42, 81,
10000000, 99, 8, 2038, 421, 100, 33, 9999]

[0, 1, 9, 8, 42, 42, 81, 99, 33,
421, 100, 2018, 1970, 2038, 9999, 10000000]
```

[Pour une meilleure lisibilité, l'affichage de la sortie a été légèrement modifié.]

Exercice : Trier des dates

On donne une liste de dates, comme celle-ci dessous :

```
dates = ['22-23/6/2024', '20-22/6/2024', '14-15/5/2024', '20/6/2024',
         '21/10/2024','19-21/6/2024', '21/6/2024', '20-21/6/2024']
```

Une date peut apparaître comme chaîne de caractères de la forme `j/m/a` ou de la forme `j-j/m/a` pour indiquer une plage de dates. On cherche à trier dans l'ordre croissant cette liste de dates, avec la règle que, pour les plages de dates, le classement se fait comme dans cet exemple : `20/10/2024 < 20-21/6/2024 < 21/6/2024`, autrement dit, c'est le premier jour qui prime.

Ecrire un code qui trie la liste de dates. Avec la liste ci-dessus, on devra trouver :

```
['14-15/5/2024', '19-21/6/2024', '20/6/2024', '20-21/6/2024',
'20-22/6/2024', '21/6/2024', '22-23/6/2024', '21/10/2024']
```

Cet exercice provient d'une question posé sur le forum Python d'OpenClassrooms

Solution

```
def cmp_dates(d):
    j, m, a = d.split("/")
    return (int(a), int(m), [int(z) for z in j.split('-')])
```

```
dates = ['22-23/6/2024', '20-22/6/2024', '14-15/5/2024', '20/6/2024',
         '21/10/2024','19-21/6/2024', '21/6/2024', '20-21/6/2024']
print(sorted(dates, key=cmp_dates))
```

Exercice : Tri-casier

On donne une liste `L` d'entiers positifs, par exemple

```
[47, 5, 5, 49, 23, 31, 13, 46, 48, 20,
9, 19, 19, 40, 42, 31, 38, 9, 45, 26]
```

On demande de construire une nouvelle liste `M` dont les éléments sont ceux de `L` mais triés suivant leur chiffre des unités.

Par exemple, dans le cas de la liste ci-dessus, `M` est la liste suivante :

```
[20, 40, 31, 31, 42, 23, 13, 5, 5, 45,
46, 26, 47, 48, 38, 49, 9, 19, 19, 9]
```

Explication : on lit d'abord les éléments de `L` qui se terminent par 0 (20, 40) puis ceux qui se terminent par 1 (31, 31), et ainsi de suite jusqu'aux éléments de `L` qui se terminent par 9.

Solution

```
L=[47, 5, 5, 49, 23, 31, 13, 46, 48, 20,
9, 19, 19, 40, 42, 31, 38, 9, 45, 26]

M=sorted(L, key=lambda z: z%10)

print(M)
```

Exercice : Suite croissante puis décroissante

Une liste `L` d'entiers comme

```
L = [2000, 2005, 2005, 2012, 2020, 2020, 2019, 2015, 2015, 2010]
```

qui commence en croissant et termine en décroissant est dite **unimodale**. La croissance est au sens large, de même pour la décroissance, autrement dit des valeurs successives peuvent être égales.

Ecrire une fonction `estUnimodale` qui renvoie `True` si et seulement si `L` est unimodale. On considérera qu'une suite croissante et, de même, qu'une suite décroissante, sont unimodales.

Voici quelques exemples de comportements de `estUnimodale(L)`

```
[2, 4, 8, 8, 5] -> True
[3, 8, 0, 8, 2] -> False
[7, 3, 3, 9, 6] -> False
[5, 6, 9, 1, 1] -> True
[2, 2, 3, 6, 7] -> True
[4, 1, 4, 3, 9] -> False
```

On pourra utiliser la fonction `max`, la méthode de liste `index` et des slices.

Solution

```
def estUnilodale(L):
    k=L.index(max(L))
    return sorted(L[:k])==L[:k] and sorted(L[k:], reverse=True)==L[k:]

from random import randrange

# Tests
for i in range(50):
    N=5
    L=[randrange(10) for _ in range(N)]
    print(L, '->', estUnilodale(L))
```

Exercice : Tri de fichiers par extension

On donne une liste de fichiers possédant une extension (par exemple, `.py`, `.cpp`, etc) et on veut trier ces fichiers suivant leur extension et par ordre alphabétique. Par exemple, si la liste de fichier est :

```
L=["orange.py", "salade.zip", "orange.cpp", "citron.png",
   "mangue.java", "oranges.cpp", "ananas.jpg", "avocat.png",
   "fruit.py", "compote.zip"]
```

la liste triée sera :

```
M=['orange.cpp', 'oranges.cpp', 'mangue.java', 'ananas.jpg',
   'citron.png', 'avocat.png', 'orange.py', 'fruit.py',
   'salade.zip', 'compote.zip']
```

Solution

```
L=["orange.py", "salade.zip", "orange.cpp", "citron.png",
   "mangue.java", "oranges.cpp", "ananas.jpg", "avocat.png",
   "fruit.py", "compote.zip"]

M=sorted(L, key=lambda name: name.split(".")[-1])

print(M)
```

Exercice : Problème du drapeau hollandais

On donne une liste `L` dont les éléments sont parmi les trois couleurs *bleu*, *blanc* ou *rouge* et on demande de déplacer les éléments de la liste pour que les couleurs apparaissent dans l'ordre *bleu*, *blanc* et *rouge*. Les éléments de `L` seront parmi les trois chaînes `"bleu"`, `"blanc"` et `"rouge"`. Par exemple, si initialement `L` vaut :

```
L = ['bleu', 'rouge', 'bleu', 'bleu', 'bleu', 'rouge', 'bleu',
'blanc', 'bleu', 'bleu', 'bleu', 'bleu', 'blanc']
```

alors, après échanges, L vaudra :

```
L = ['bleu', 'bleu', 'bleu', 'bleu', 'bleu', 'bleu', 'bleu',
     'bleu', 'bleu', 'blanc', 'blanc', 'rouge', 'rouge']
```

On ne demande pas de créer une nouvelle liste mais de modifier la position des éléments de L. Ce problème est connu sous le nom du problème du drapeau hollandais et sa version originale impose davantage de contraintes.

On peut résoudre ce problème facilement en utilisant la méthode sort de la classe `list` et en utilisant une clé de comparaison. Cette clé va associer les chaînes *bleu*, *blanc* et *rouge* aux nombres respectifs 0, 1 et 2.

1. Essayez différentes implémentations de la clé de comparaison.
2. Ecrire une fonction qui réalise le tri en parcourant une fois et une seule la liste et qui :
   - si l'élément courant est bleu, cet élément est échangé avec le premier blanc
   - si l'élément courant est rouge, cet élément est échangé avec l'élément qui précède le bloc final des rouges.
3. Comparer les performances.

Solution

1. Il s'agit d'écrire un code tel que celui-ci :

```
def cmp(col):
    pass

L= ['bleu', 'rouge', 'bleu', 'bleu', 'bleu', 'rouge',
    'bleu', 'blanc', 'bleu', 'bleu', 'bleu', 'bleu', 'blanc']

L.sort(key=cmp)
print(L)
```

où le corps de la fonction cmp est à trouver.
La première idée qui vienne à l'esprit est celle-ci :

```
def cmp(col):
    if col == "bleu":
        return 0
    if col == "blanc":
        return 1
    if col == "rouge":
        return 2
```

On peut essayer d'utiliser une expression conditionnelle (emboîtée) :

```
def cmp(col):
    return 0 if col=="bleu" else 1 if col =="blanc" else 2
```

et qui est un peu plus rapide car, sans doute, elle s'épargne le dernier test. En fait la fonction

```
def cmp(col):
    if col == "bleu":
        return 0
    if col == "blanc":
        return 1
    return 2
```

devrait avoir une performance comparable.
On peut aussi utiliser un dictionnaire :

```
def cmp(col):
    return {"bleu":0, "blanc":1, "rouge":2}[col]
```

avec le désavantage qu'à chaque comparaison, un dictionnaire toujours identique va être recréé. On peut donc songer à écrire un dictionnaire global :

```
DICO={"bleu":0, "blanc":1, "rouge":2}
def cmp(col):
    return DICO[col]
```

On peut aussi essayer une liste des couleurs et dont on cherche l'indice :

```
COLS=["bleu", "blanc", "rouge"]
def cmp(col):
    return COLS.index(col)
```

(2) Un tri d'une liste de n éléments étant de complexité $n\ln(n)$, on peut se dire qu'un ou plusieurs parcours de la liste devrait être plus rapide. Une solution assez naturelle consiste à parcourir la liste et à placer dans trois listes prédéfinies les références vers les éléments de même couleur. Ne reste plus qu'à fusionner ces listes dans l'ordre et que la fusion écrase le contenu de L (puisqu'on est censé garder le même conteneur), ce qui donne le code suivant :

```
def parcours(L):
    bleus=[]
    blancs=[]
    rouges=[]
    for col in L:
        if col=="bleu":
            bleus.append(col)
        elif col=="blanc":
            blancs.append(col)
        else:
            rouges.append(col)
    L[:]=bleus+blancs+rouges
```

La dernière ligne modifie le contenu de L. On peut lègèrement améliorer la vitesse d'exécution en remarquant qu'on appelle toujours une des trois méthodes et que cet appel a un coût (celui du coût de l'accès à un attribut), d'où la nouvelle version :

```
def parcours(L):
    bleus=[]
    blancs=[]
    rouges=[]
    bleu=bleus.append
    blanc=blancs.append
    rouge=rouges.append
    for col in L:
        if col=="bleu":
            bleu(col)
        elif col=="blanc":
            blanc(col)
        else:
            rouge(col)
    L[:]=bleus+blancs+rouges
```

L'autre méthode consiste à parcourir la liste, à construire au fur et à mesure le bloc des bleus à gauche, le bloc des rouges à droite (et en les plaçant en progressant de la droite vers la gauche) et à laisser les blancs à leur place car un blanc courant sera par contruction à droite du bloc des blancs. Cela donne le code suivant :

```
def sort(M):
    n=len(M)
    rouge=n-1
    bleu=0

    for i in range(n):
        if i>rouge:
            return
        if M[i]=="bleu":
            M[bleu], M[i]=M[i],M[bleu]
            bleu+=1
        elif M[i]=="rouge":
            while rouge> i and M[rouge]=="rouge":
                rouge-=1

            M[rouge], M[i]=M[i],M[rouge]
            rouge-=1
            if M[i]=="bleu":
                M[bleu], M[i]=M[i],M[bleu]
                bleu+=1
```

Un élément bleu est placé par échange avec un élément blanc et un élément rouge est placé par échange avec un élément de couleur autre que rouge.
Cette méthode est plus rapide que tout tri avec la méthode `sort` mais moins rapide que la fontion `parcours` précédente.

(3) On fait un test sur 18 millions de couleurs aléatoires suivi d'une vérification de la cohérence des résultats :

```python
from random import shuffle, randint
from time import perf_counter

dico={"bleu":0, "blanc":1, "rouge":2}
def cmp1(col):
    return dico[col]

def cmp2(col):
    return 0 if col=="bleu" else 1 if col =="blanc" else 2

COLS=["bleu", "blanc", "rouge"]
def cmp3(col):
    return COLS.index(col)

def cmp4(col):
    return {"bleu":0, "blanc":1, "rouge":2}[col]

def cmp5(col):

    if col == "bleu":
        return 0
    if col == "blanc":
        return 1
    if col == "rouge":
        return 2

def cmp5bis(col):

    if col == "bleu":
        return 0
    if col == "blanc":
        return 1
    return 2

def sort(M):
    n=len(M)
    rouge=n-1
    bleu=0

    for i in range(n):
        if i>rouge:
            return
        if M[i]=="bleu":
            M[bleu], M[i]=M[i],M[bleu]
            bleu+=1
        elif M[i]=="rouge":
            while rouge> i and M[rouge]=="rouge":
                rouge-=1
```

```
            M[rouge], M[i]=M[i],M[rouge]
            rouge-=1
            if M[i]=="bleu":
                M[bleu], M[i]=M[i],M[bleu]
                bleu+=1

def parcours(L):
    bleus=[]
    blancs=[]
    rouges=[]
    bleu=bleus.append
    blanc=blancs.append
    rouge=rouges.append
    for col in L:
        if col=="bleu":
            bleu(col)
        elif col=="blanc":
            blanc(col)
        else:
            rouge(col)
    L[:]=bleus+blancs+rouges

def parcours_bis(L):
    bleus=[]
    blancs=[]
    rouges=[]
    for col in L:
        if col=="bleu":
            bleus.append(col)
        elif col=="blanc":
            blancs.append(col)
        else:
            rouges.append(col)
    L[:]=bleus+blancs+rouges


N=10**7

M=["bleu"]*randint(1,N)+["blanc"]*randint(1,N)+["rouge"]*randint(1,N)
shuffle(M)

print("Mélange fait")
print("Nombre de couleurs :", len(M))


# ------------------------------------------------
Lcopy=M[:]
begin_perf = perf_counter()
```

```
parcours(Lcopy)

delta = perf_counter() - begin_perf

print(f"copy: {delta:.2f}s")

# ------------------------------------------------
Lcopy_bis=M[:]
begin_perf = perf_counter()

parcours_bis(Lcopy_bis)

delta = perf_counter() - begin_perf

print(f"copy_bis: {delta:.2f}s")

# ------------------------------------------------
L0=M[:]
begin_perf = perf_counter()


sort(L0)


delta = perf_counter() - begin_perf

print(f"for: {delta:.2f}s")

# ------------------------------------------------
L2=M[:]
begin_perf = perf_counter()


L2.sort(key=cmp2)


delta = perf_counter() - begin_perf

print(f"if/else : {delta:.2f}s")

# ------------------------------------------------
L3=M[:]
begin_perf = perf_counter()

L3.sort(key=cmp3)

delta = perf_counter() - begin_perf
```

```
print(f"index : {delta:.2f}s")

# ------------------------------------------------
L4=M[:]
begin_perf = perf_counter()


L4.sort(key=cmp4)


delta = perf_counter() - begin_perf

print(f"dico local : {delta:.2f}s")

# ------------------------------------------------
L5=M[:]
begin_perf = perf_counter()


L5.sort(key=cmp5)


delta = perf_counter() - begin_perf

print(f"if/return : {delta:.2f}s")

# ------------------------------------------------
L5bis=M[:]
begin_perf = perf_counter()


L5bis.sort(key=cmp5bis)


delta = perf_counter() - begin_perf

print(f"if/return bis: {delta:.2f}s")

# ------------------------------------------------
L1=M[:]
begin_perf = perf_counter()


L1.sort(key=cmp1)


delta = perf_counter() - begin_perf
```

```
print(f"Dico global: {delta:.2f}s")


# -----------------------------------------------
print(L0==L0==L1==L2==L3==L4==L5==L5bis==Lcopy==Lcopy_bis)
```

qui affiche :

```
Mélange fait
Nombre de couleurs : 18062926
copy: 1.34s
copy_bis: 1.71s
for: 1.94s
if/else : 2.85s
index : 4.24s
dico local : 3.94s
if/return : 2.97s
if/return bis: 2.84s
Dico global: 2.61s
True
```

Exercice : Trier une liste formée de listes ou de tuples de nombres

Soit une liste contenant à la fois des listes et des tuples de nombres. Trier cette liste en utilisant l'argument nommé key de la fonction `sorted`.

Solution

```
1 L = [[2, 5, 0, 4], [5, 4, 8, 4], (8, 5, 8, 8), (6, 4), (0, 8, 2, 8)]
2 print(sorted(L, key=lambda t: list(t)))
```

Exercice : Indice maximum de la valeur minimum

On donne une liste `L` d'entiers et on demande de déterminer le plus grand indice `m` d'un élément minimum de la liste `L`. Par exemple, si `L` est la liste des huit éléments suivants :

```
L=[40, 30, 10, 70, 10, 40, 10, 20]
```

alors `m=6` puisque `10` est le minimum de `L` et que parmi les trois indices où ce minimum est atteint, le plus grand indice est 6 (avant dernière position de `L`).

Solution

```
L=[40, 30, 10, 70, 10, 40, 10, 20]
G=((v, -i) for (i,v) in enumerate(L))

print(-min(G)[1])
```

Exercice : Unique parmi des triplets

On dispose d'une liste L d'entiers. Chaque entier de la liste est présent exactement trois fois sauf un entier qui, lui, n'est présent qu'une seule fois. On demande de trouver cet entier présent une seule fois.
Voici quelques exemples de comportements :

```
L = [2020, 2020, 50, 2024, 50, 50, 2020, 1000, 2024, 2024]
→ 1000
L = [2020, 2024, 50, 50, 2020, 2024, 42, 2020, 50, 2024]
→ 42
L = [2024, 50, 2020, 2042, 2020, 2020, 2024, 2024, 50, 50]
→ 2042
```

Enoncé original : LeetCode : Single Number II. Trier est une des solutions possibles, pas la plus rapide.

Solution

On trie la liste L en M et l'intrus est le premier élément à un indice multiple de 3 et qui est distinct de son suivant, à moins que l'intrus soit à la fin (il n'a pas de suivant). D'où le code :

```
tests=[
    [50,2020,2024,2024, 1000, 50, 2020,2020, 2024,50],
    [50,2020,42,2024,2024, 50, 2020,2020, 2024,50],
    [50,2020,2042,2024,2024, 50, 2020,2020, 2024,50]
    ]

for L in tests:
    print("L =", L)
    M=sorted(L)
    print('→', M[-1] if M[-1]!=M[-2] else
          M[min(k for k in range(0,len(L)-1, 3) if M[k]!=M[k+1])])
```

Exercice : Trier suivant deux critères

Soit une liste de chaînes, par exemple

```
BAB ACC AAA CC AAC B CAB BBAB ABB ABBA A AC BCC
```

Si on trie cette liste suivant le nombre de caractères de chaque mot, on obtient :

```
B A CC AC BAB ACC AAA AAC CAB ABB BCC BBAB ABBA
```

Mais, on voudrait en outre, qu'en cas d'ex aequo, les mots ainsi triés soient dans l'ordre l'alphabétique, ce qui donnerait plutôt le résultat suivant :

```
A B AC CC AAA AAC ABB ACC BAB BCC CAB ABBA BBAB
```

Ecrire un code qui réalise ce tri en utilisant une unique clé de comparaison. Le code devrait pouvoir tenir sur une seule ligne.

Solution

```
def cmp(s):
    return (len(s),s)

L=['BAB', 'ACC', 'AAA', 'CC', 'AAC', 'B', 'CAB',
    'BBAB', 'ABB', 'ABBA', 'A', 'AC', 'BCC']
print(L)

L.sort(key=cmp)
print(L)
```

Exercice : Utiliser `cmp_to_key` pour comparer des couples

Soit l'ordre $(a, b) \leq (c, d)$ signifiant que $a \leq c$ ou que si $a == c$ alors on a $b \geq d$. Ainsi, pour cet ordre, on a :

$$[7, 2] \leq [8, 6] \text{ et } [3, 6] \leq [3, 5]$$

Ecrire à l'aide de la fonction `cmp_to_key` du module functools une fonction `key` utilisable par la fonction `sorted`.

Solution

```
from functools import cmp_to_key

def cmp(x, y):
    a, b = x
    c, d = y
    if (a < c) or (a== c and b >= d):
        return -1
    if x==y:
        return 0
    return 1

f=cmp_to_key(cmp)

L=[[5, 9], [8, 6], [9, 5], [3, 1], [4, 6], [7, 2], [1, 4], [4, 3],
   [8, 6], [3, 6], [3, 5], [8, 2]]

print(sorted(L, key=f))
```

```
[[1, 4], [3, 6], [3, 5], [3, 1], [4, 6], [4, 3], [5, 9],
[7, 2], [8, 6], [8, 6], [8, 2], [9, 5]]
```

Exercice : Majuscule avant minuscule mais ...

Dans l'ordre usuel pour Python, une majuscule vient avant une minuscule (ça provient de l'ordre en ASCII). Plus précisément, toute majuscule vient avant toute minuscule :

```
print('"K" < "k" :', "K" < "k")
print('"K" < "a" :', "K" < "a")
print('"K" < "z" :', "K" < "z")
```

```
"K" < "k" : True
"K" < "a" : True
"K" < "z" : True
```

Soit maintenant l'ordre suivant : c'est l'ordre alphabétique habituel, sauf que si on compare deux lettres identiques, l'une en majuscule, l'autre en minuscule, la majuscule est avant la minuscule. Ainsi, selon cet ordre, les chaînes suivantes sont classées dans l'ordre croissant :

```
['ABB', 'aA', 'aa', 'aaB', 'aBab', 'BBA', 'BBAb', 'Bb', 'ba', 'bB']
```

Créer une fonction-clé pour la fonction sorted qui trie suivant l'ordre précédent.
Cette question est inspirée de cette question de concours : ACM-ICPC 5605 : Anagram.

Solution

```
from random import randrange

def f(x):
    return [(c.lower(), c) for c in x]

tests=list(set(['',join(['aAbB'[randrange(4)] for _
                in range(randrange(2,5))]) for i in range(10)]))

print(tests)
print(sorted(tests, key=f))
```

```
['BaB', 'aba', 'Aab', 'bBbb', 'BBb', 'AAba', 'bb', 'Aaa', 'aaAA', 'aAAa']
['AAba', 'Aaa', 'Aab', 'aAAa', 'aaAA', 'aba', 'BaB', 'BBb', 'bBbb', 'bb']
```