Les listes – premiers algorithmes

Le terme « informatique » est un mot-valise (constitué de « information » et « automatique ») décrivant la discipline visant à traiter de façon automatique et autonome des informations et données. Un tel traitement automatisé des données ne présente généralement d'intérêt que si le volume de données à traiter est conséquent.

Afin de manipuler de grandes quantités de données, il nous faudra une structure de données qui permette de les contenir. En informatique, c'est le role des *tableaux* (« *arrays* » en anglais), une structure capable de contenir plusieurs données. Dans ce chapitre, nous nous intéresserons à la structure Python appelée « *liste* » qui, pour l'essentiel, remplit cet office [1].

Nous profiterons par ailleurs de ce cours pour décrire quelques *algorithmes* élémentaires sur ces listes, un algorithme étant une suite d'opérations qui, à partir d'un ensemble de données, permet d'obtenir un résultat particulier.

1 Présentation des listes

1.1 Création

Une *liste* en Python est une collection *ordonnée* d'éléments. On représente la liste entre crochets, et les éléments à l'intérieur de celle-ci sont séparés par des virgules. On peut par exemple définir une liste de trois éléments, et lui associer le nom `L` de la façon suivante :

```
In [1]: L = [ 1, 3.14, 1+2j ]
```

1.2 Indexation des éléments

Pour faire référence à élément particulier d'une liste associée à un nom, on fait suivre le nom de crochets, à l'intérieur desquels on place l'*indice* (ou *index*) de l'élément qui nous intéresse, c'est-à-dire sa position dans la liste. Attention, comme souvent en informatique, **le premier élément correspond à l'indice 0** !

```
In [2]: L[0]
Out[2]: 1

In [3]: L[2]
Out[3]: (1+2j)
```

1. Nous reviendrons plus tard sur la terminologie, mais pour les lecteurs qui ont déjà eu l'occasion de travailler avec d'autres langages, les listes en Python sont essentiellement des tableaux redimensionnables.

On obtiendra une erreur si l'indice est trop grand pour correspondre à un élément :

```
In [4]: L[5]
IndexError: list index out of range
```

Pour obtenir le nombre d'éléments dans une liste, on utilise la fonction `len(liste)` qui prend un seul élément, la liste elle-même.

```
In [5]: len(L)
Out[5]: 3
```

L'indice du premier élément d'une liste `L` étant `0`, celui du dernier est donc `len(L)-1`. Pour accéder à ce dernier élément, on pourrait donc procéder de la sorte :

```
In [6]: L[ len(L)-1 ]
Out[6]: (1+2j)
```

Toutefois, Python permet d'y accéder plus facilement. L'indice -1 fait référence au dernier élément de la liste, -2 à l'avant dernier, et ainsi de suite :

```
In [7]: L[-1]
Out[7]: (1+2j)

In [8]: L[-2]
Out[8]: 3.14
```

Cette fois encore, il faut prendre garde à ne pas déborder de la liste :

```
In [9]: L[-5]
IndexError: list index out of range
```

1.3 Manipuler le contenu d'une liste

Que peut contenir une liste ? En fait, à peu près n'importe quoi que Python puisse manipuler. Des valeurs numériques, mais aussi des fonctions, comme dans cet exemple :

```
In [10]: L = [ 1, 3.14, sin ]

In [11]: L
Out[11]: [1, 3.14, <built-in function sin>]
```

On peut alors librement y accéder :

```
In [12]: L[2]
Out[12]: <built-in function sin>
```

Et même s'en servir directement, car `L[2]`, pour l'interpréteur Python, désigne la fonction `sin` :

```
In [13]: L[2](0)
Out[13]: 0.0
```

On peut donc aussi mettre une autre liste dans une liste :

```
In [14]: L = [ 1, 3.14, [ 42, sin ] ]
```

L'élément d'indice 2 est donc, sur cet exemple, lui-même une liste :

```
In [15]: L[2]
Out[15]: [42, <built-in function sin>]
```

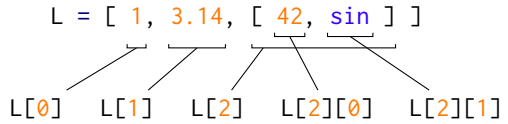
Et naturellement, on peut accéder aux éléments de cette sous-liste de `L` en ajoutant un second indice derrière le premier :

```
In [16]: L[2][0]
Out[16]: 42
```

Ainsi, on peut obtenir, puis utiliser, la fonction en seconde position de cette sous-liste :

```
In [17]: L[2][1]
Out[17]: <built-in function sin>

In [18]: L[2][1](0)
Out[18]: 0.0
```

On peut résumer les choses de la façon suivante :



Dans le cadre d'un programme, il n'est pas toujours pratique (principalement pour des questions de lisibilité du programme) de manipuler des indices. Supposons par exemple que la liste `Pos` contienne les coordonnées dans l'espace d'un point M :

```
In [19]: Pos = [ 2.71, 3.14, 4.09 ]
```

Plutôt que d'utiliser `Pos[0]`, `Pos[1]` et `Pos[2]` pour faire référence aux différentes coordonnées de M, il est plus clair d'utiliser par exemple des noms `x`, `y` et `z` dans les calculs.

Pour ce faire, il serait possible d'écrire `x = Pos[0]`, puis `y = Pos[1]` et `z = Pos[2]`. Mais on peut réaliser, d'un seul coup, ces trois affectations [2] :

```
In [20]: x, y, z = Pos

In [21]: z
Out[21]: 4.09
```

Une telle affectation fonctionne si l'on trouve, à gauche du signe égal, un nombre de noms qui correspond exactement au nombre d'éléments de l'objet situé à droite de ce même signe égal.

1.4 Itérations sur les éléments

Les listes en Python sont des ensembles ordonnés d'éléments, il est donc possible d'itérer les éléments d'une liste avec une boucle **`for`**, afin d'effectuer des opérations sur chacun de ces éléments.

Ainsi, afficher tous les éléments d'une liste `L` peut s'écrire avec une boucle :

```
for elem in L :
    print(elem)
```

Le nom `elem` reçoit ici tour à tour les différents éléments présents dans la liste, dans l'ordre, et il est possible d'en faire quelque chose. Tout se passe comme si l'on avait écrit :

```
elem = L[0]
print(elem)

elem = L[1]
print(elem)
...
```

De la même façon, on peut calculer et afficher la somme des termes de la liste :

```

```

2. L'écriture « `[ x, y, z ] = Pos` » convient aussi, si elle vous sied davantage.

Python fournit une fonction pour cette opération courante, `sum(liste)` :

```
In [22]: L = [1, 2, 5, 3, 4]

In [23]: sum(L)
Out[23]: 15
```

On peut aussi déterminer et afficher le plus grand de ses éléments :

```
plusGrand = L[0]           # plusGrand = plus grande valeur rencontrée

for elem in L :                # Pour tout les éléments "elem" dans L
    if elem > plusGrand :      # si elem est plus grand que plusGrand
        plusGrand = elem       # c'est le nouveau plus grand

print(plusGrand)           # On termine en affichant le résultat
```

On effectue ici une comparaison inutile, mais si la liste est longue, cette comparaison supplémentaire n'a guère d'importance.

Python fournit là encore une fonction pour faire cette même opération, `max(liste)` (de même que son pendant, `min(liste)`) :

```
In [24]: max(L)
Out[24]: 5

In [25]: min(L)
Out[25]: 1
```

On peut aussi vouloir déterminer l'*indice* du plus grand élément de la liste, ce qui est un peu plus délicat. Une première solution consiste à changer quelque peu la boucle, et itérer sur les indices plutôt que sur les éléments [3] :

```
plusGrand = L[0]           # plusGrand = plus grande valeur rencontrée
indicePlusGrand = 0        # indicePlusGrand son indice dans la liste

for i in range(len(L)) :       # Pour tout i de 0 à len(L)-1,
    if L[i] > plusGrand :      # si L[i] est plus grand que plusGrand
        plusGrand = L[i]       # c'est le plus grand élément rencontré
        indicePlusGrand = i    # et on mémorise son indice

print(indicePlusGrand)     # On termine en affichant le résultat
```

3. Certains concours pourraient vous demander de toujours itérer sur les indices, plutôt que sur les éléments de la liste, donc sachez manier les deux notations !

Ou bien, pour éviter de mettre à jour deux éléments : [4] :

```
indicePlusGrand = 0

for i in range(len(L)) :
    if L[i] > L[indicePlusGrand] :
        indicePlusGrand = i

print(indicePlusGrand)
```

On dispose cependant d'une fonction bien pratique, qui permet, lorsque l'on utilise une boucle for, de disposer à la fois de l'indice et de l'élément lui-même : la fonction `enumerate`. Elle s'utilise de la façon suivante :

```
for i, elem in enumerate(L) :
    print("L'élément d'indice", i, "est", elem)
```

On peut donc réécrire plus simplement le programme précédent :

```
plusGrand = L[0]
indicePlusGrand = 0

for i, elem in enumerate(L) :
    if elem > plusGrand :
        plusGrand = elem
        indicePlusGrand = i

print(indicePlusGrand)
```

Ce genre de construction est utilisé très très fréquemment, aussi est-il important de bien savoir les manier. Proposons un dernier exemple, un programme permettant de trouver, dans une liste, l'élément le plus proche de 0 :

```

```

4. Les deux approches sont à tous points de vue équivalentes en Python

2 Opérations sur les listes

2.1 Les opérateurs

L'opérateur `+`, utilisé sur deux listes, permet de créer une *nouvelle*[5] liste qui est la concaténation des deux listes en argument. Par exemple :

```
In [26]: [ 1, 2, 3 ] + [ 4, 5 ]
Out[26]: [1, 2, 3, 4, 5]

In [27]: L = [ 1, 2, 3 ]

In [28]: L + [ 4, 5 ]
Out[28]: [1, 2, 3, 4, 5]

In [29]: P = [ 6, 7 ]

In [30]: Q = P + L + P

In [31]: Q
Out[31]: [6, 7, 1, 2, 3, 6, 7]
```

L'opérateur `*` placé entre une liste et un entier positif permet de créer une *nouvelle* liste, dans laquelle tous les éléments de la liste fournie sont répétés un nombre de fois égal à l'entier :

```
In [32]: 2 * [ 1, 2, 3 ]
Out[32]: [1, 2, 3, 1, 2, 3]

In [33]: [ True ] * 4
Out[33]: [True, True, True, True]

In [34]: L = [ 0, 1 ]

In [35]: 2 * L * 2
Out[35]: [0, 1, 0, 1, 0, 1, 0, 1]
```

Notons que les priorités sont les mêmes que pour les calculs habituels :

```
In [36]: 2 * [ 0, 1 ] + [ 3 ] * 3
Out[36]: [0, 1, 0, 1, 3, 3, 3]
```

5. Nous discuterons de l'importance de ce point plus avant dans ce cours

2.2 Étendre une liste

Étendre une liste consiste à ajouter un élément à une liste existante. Supposons que l'on souhaite ajouter un 4 à la fin d'une liste désignée par le nom `L`. On pourrait envisager les choses de la façon suivante :

```
In [37]: L = [ 1, 2, 3 ]

In [38]: L = L + [ 4 ]

In [39]: L
Out[39]: [ 1, 2, 3, 4 ]
```

On voit que cela fonctionne. Toutefois, cette méthode peut avoir de nombreux inconvénients. Le principal (et les autres en découleront, comme nous le verrons). Ce que l'on fait ici, c'est que l'on crée une *nouvelle* liste qui est la concaténation de celle désignée par `L` et de la liste 4, et on demande que le nom `L` désigne dorénavant cette nouvelle liste.

On a donc fait plus que simplement ajouter un élément en fin de liste (il nous faut copier tous les éléments de `L`), ce qui nécessite du temps de calcul et de la mémoire.

Heureusement, il est possible d'*adjoindre* un élément *à la fin de la liste, sans en créer une nouvelle*, grâce à la fonction `list.append(liste, valeur)`, qui s'utilise ainsi :

```
In [40]: L = [ 1, 2, 3 ]

In [41]: list.append(L, 4)

In [42]: L
Out[42]: [ 1, 2, 3, 4 ]
```

De nombreuses fonctions Python commencent par « `list.` ». Cela signifie qu'elles ont été écrites pour fonctionner sur des listes. Le premier argument sera d'ailleurs toujours la liste sur laquelle on effectuera une opération. En fait, on dispose d'un raccourci pour utiliser de telles fonctions : il est possible de ne pas passer la liste comme premier argument de la fonction, mais à la place de substituer à `list` la liste elle-même, comme ceci :

```
In [43]: L = [ 1, 2, 3 ]

In [44]: L.append(4)

In [45]: L
Out[45]: [1, 2, 3, 4]
```

Pour Python, cela revient exactement au même, sous réserve bien entendu que `L` désigne une liste. Dans la suite, nous prendrons des exemples avec les deux écritures possibles.

Une autre fonction similaire est la fonction `list.extend(liste, liste_2)` qui permet d'*adjoindre* une copie du contenu de la seconde liste à l'extrémité de la première :

```
In [46]: L = [ 1, 2, 3 ]

In [47]: list.extend(L, [4, 5])

In [48]: L
Out[48]: [1, 2, 3, 4, 5]
```

Comme pour `list.append`, on peut aussi écrire, de façon équivalente :

```
In [49]: L.extend([6, 7])

In [50]: L
Out[50]: [1, 2, 3, 4, 5, 6, 7]
```

La liste insérée n'est elle pas modifiée par la fonction `extend` :

```
In [51]: P = [ 8, 9 ]

In [52]: L.extend(P)

In [53]: L
Out[53]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [54]: P
Out[54]: [8, 9]
```

2.3 Raccourcis d'écriture

Il existe quelques « raccourcis » de syntaxe en Python, comme dans d'autres langages. Par exemple, il existe un opérateur noté `+=` dit d'*affectation augmentée* (ou composée). Lorsqu'un nom `x` est associé à une valeur numérique, on peut écrire

```
x += 17
```

ce qui est équivalent à

```
x = x + 17
```

Cet opérateur effectue donc une opération (ici une somme), suivie d'une affectation. On économise ainsi quelques caractères ! On aura tôt fait de comprendre à quoi servent les opérateurs d'affectation augmentée `-=`, `*=`, `/=`, `**=`, `//=`, `%=`, etc.

Il est naturel de songer à essayer le même opérateur avec les listes, lequel est effectivement disponible. Mais **attention**, si `L` est une liste,

```
L += [ 1, 2 ]
```

est équivalent à

```
L.extend( [1, 2] )
```

et non à `L = L + [ 1, 2 ]` comme on aurait pu le penser [6] ! Cet opérateur est donc bien une façon efficace d'ajouter un ou plusieurs éléments à une liste existante.

Dans le cas où on rajoute un seul élément, il est impératif que celui-ci soit entouré de crochets, car c'est la fonction `list.extend` qui est appelée.

```
In [55]: L = [ 1, 2, 3 ]

In [56]: L += 4
TypeError: 'int' object is not iterable
```

Si l'on souhaite ajouter une *liste* dans une liste existante (et non son contenu), il faut donc des doubles crochets :

```
In [57]: L += [ 4 ]

In [58]: L += [ 5, 6 ]

In [59]: L
Out[59]: [ 1, 2, 3, 4, 5, 6 ]

In [60]: L += [ [ 7, 8 ] ]

In [61]: L
Out[61]: [1, 2, 3, 4, 5, 6, [7, 8]]
```

Si l'on veut être exhaustif, il existe également un opérateur `*=`, pour lequel

```
L *= 3
```

est « approximativement » équivalent [7] à

```
L.extend( L*2 )
```

6. Et comme le suggèrent quantité de tutoriels sur le net ainsi que quelques ouvrages, attention !
7. Si l'entier est stritement positif, en tout cas. S'il est négatif ou nul, cela revient à supprimer tous les éléments de la liste.

2.4 Exemples

Suite de Fibonacci

La suite de Fibonacci (u_n) est définie[8] par $u_0 = 0$, $u_1 = 1$ et la relation de récurrence $u_n = u_{n-1} + u_{n-2}$ pour tout $n \geq 2$.

Les premiers termes sont donc

0 1 1 2 3 5 8 13 21 34 55 89...

• Écrire un programme construisant la liste des vingts premiers termess de la suite de Fibonacci.

Triangle de Pascal

Le coefficient binomial $\binom{n}{k}$ correspond au nombre de façons de choisir k éléments parmi n, sans ordre.

Par exemple, $\binom{4}{2} = 6$.

Le triangle de Pascal regroupe les coefficients binomiaux $\binom{n}{k}$ de sorte que
- les coefficients de même n soient alignés sur une même ligne ;
- ceux de même k sur une même colonne.

Le résultat pour $0 \leq n \leq 6$ est donc :

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
1   6   15  20  15  6   1
```

Pour construire ce triangle, on peut remarquer que $\binom{n}{0} = \binom{n}{n} = 1$.

Pour les autres coefficients, on dispose de la relation $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

8. On trouve parfois la condition initiale $u_0 = 1$, mais cela conduit en fait à la même suite, simplement décalée d'un terme.

• Écrire un programme construisant la liste regroupant $n+1$ listes, contenant respectivement les coefficients des $n+1$ premières lignes du triangle de Pascal (on supposera $n \geq 1$).

2.5 Ajouter librement des éléments dans une liste

L'insertion d'un nouvel élément dans une liste peut également se faire ailleurs qu'à la fin de la liste, grâce à la fonction `list.insert(liste, index, valeur)` qui prend trois arguments : la liste dans laquelle ajouter un élément, l'index où doit se faire l'insertion (l'insertion se fera *avant* l'élément situé à cet index) et l'élément à insérer :

```
In [62]: L = [ 1, 2, 3 ]

In [63]: list.insert(L, 1, 4)

In [64]: L
Out[64]: [1, 4, 2, 3]
```

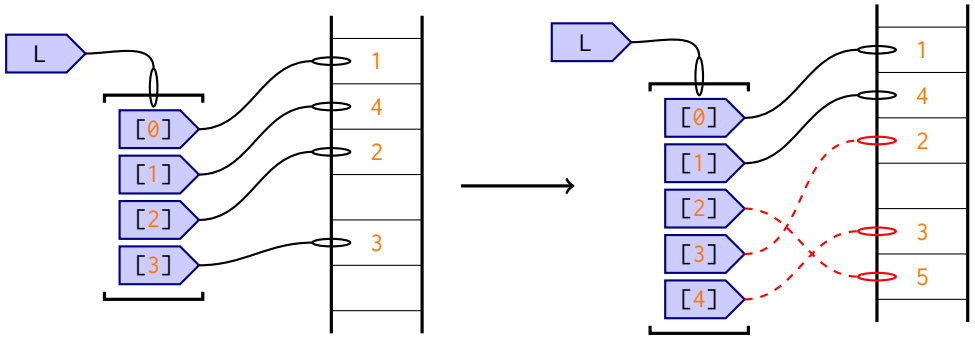
Bien entendu, on peut utiliser la forme raccourcie :

```
In [65]: L.insert(2, 5)

In [66]: L
Out[66]: [1, 4, 5, 2, 3]
```

Attention toutefois, en Python[9], **insérer un élément au milieu de la liste peut coûter très cher en temps de calcul**, car l'interpréteur ajoute en fait une place à la *fin* de la liste, puis décale les éléments un par un pour faire de la place au milieu pour la nouvelle valeur. Cela peut être très long si la liste est longue. On s'efforce donc d'éviter d'insérer des éléments au milieu d'une liste quand il est possible de faire autrement.

9. En tout cas, pour les implémentations courantes du langage.

On peut en effet se représenter une liste comme une collection d'étiquettes numérotées. L'ajout de 5 à la position d'index 2 nécessite, outre la création d'une étiquette supplémentaire [4], de réaffecter les étiquettes [2] et [3] :



On s'efforce donc, dans la mesure du possible, d'insérer des éléments dans une liste, en particulier vers le début d'une longue liste, ce qui nécessiterait beaucoup d'opérations !

Enfin, si l'index fourni est égal ou supérieur à la taille de la liste, l'élément est ajouté à la fin de celle-ci :

```
In [67]: L.insert(99, 6)

In [68]: L
Out[68]: [1, 4, 5, 2, 3, 6]
```

2.6 Supprimer des éléments

Pour extraire un élément situé à une place donnée, on peut utiliser la fonction `list.pop`, qui est le pendant de `list.insert` :

```
In [69]: L = [ 1, 4, 5, 2, 3 ]

In [70]: list.pop(L, 2)
Out[70]: 5

In [71]: L
Out[71]: [1, 4, 2, 3]

In [70]: L.pop(1)
Out[70]: 4

In [71]: L
Out[71]: [1, 2, 3]
```

L'élément retiré de la liste est retourné, ce qui permet d'en faire quelque chose. L'associer à un nom, par exemple, en écrivant `elem = L.pop(1)` :

```
In [70]: elem = L.pop(1)

In [70]: elem
Out[70]: 2

In [71]: L
Out[71]: [1, 3]
```

Si l'on ne fournit pas d'index à la fonction `list.pop`, c'est le dernier élément de la liste qui est retiré et renvoyé :

```
In [70]: L.pop()
Out[70]: 3

In [71]: L
Out[71]: [1]
```

Lorsque l'on n'a pas besoin de la valeur retirée, on peut utiliser le mot-clé [10] **`del`** :

```
In [69]: L = [ 1, 4, 5, 2, 3 ]

In [70]: del L[2]

In [71]: L
Out[71]: [1, 4, 2, 3]
```

Cette fois encore, on évitera de supprimer des éléments à l'intérieur d'une longue liste, car il se posera un problème similaire que pour une insertion : beaucoup d'opérations peuvent être nécessaires, pour « décaler » tous les éléments suivant celui qui est supprimé.

2.7 Itération, insertions et suppressions

Il convient d'être très prudent lorsque, dans une boucle par exemple, on souhaite retirer ou ajouter des éléments d'une liste. En effet, la suppression ou l'ajout d'un élément change la numérotation des autres, et la longueur de la liste. Par exemple, on ne peut pas retirer les zéros dans une liste de la sorte :

```
for i in range(len(L)) :
    if L[i] == 0 :
        del L[i]
```

10. **`del`** n'est pas une fonction mais un *mot-clé* du langage, il n'y a donc pas de parenthèses dans ce cas.

En effet, si L désigne une liste [17, 0, 0, 42, 54], à l'intérieur de la boucle **for**, le nom i prendra successivement les valeurs 0, 1, 2, 3 et 4.

Lors de la seconde itération, i vaut 1, et le zéro à la position d'index 1 est supprimé. La liste devient alors [17, 0, 42, 54].

Dans l'itération suivante, i est égal à 2, et l'élément considéré est donc le terme 42. Autrement dit le second zéro ne sera jamais supprimé !

Par ailleurs, dans la dernière itération, i prendra la valeur 4, ce qui cause une erreur car il n'y a plus d'élément à cette position dans la liste, qui a été raccourcie (les valeurs qui seront prises par i sont décidées une fois pour toutes à l'entrée dans la boucle).

Une façon correcte [11] de retirer les zéros pourrait par exemple être :

```
# On va gérer nous-même l'itération, en commençant à zéro
i = 0

while i < len(L) :   # Tant que l'on n'a pas atteint la fin de L
    if L[i] == 0 :   # Si l'élément de rang \p|i| est un zéro,
        del L[i]     # on supprime cet élément
    else :
        i = i + 1    # Sinon, on passe au rang suivant
```

2.8 Autres fonctions concernant le contenu de la liste

Un opérateur booléen supplémentaire est disponible pour les listes, très utile pour des tests ou des boucles conditionnelles, l'opérateur **in** qui renvoie **True** si l'élément à gauche de l'opérateur est présent dans la liste située à droite, et **False** dans le cas contraire :

```
In [72]: L = [ 1, 2, 3 ]

In [73]: 2 in L
Out[73]: True

In [74]: 42 in L
Out[74]: False
```

Précisons que l'élément doit être un élément de la liste (et non dans une sous-liste) :

```
In [75]: L = [ 1, 2, 3, [ 4, 5 ] ]

In [76]: 4 in L
Out[76]: False
```

11. Il en existe de meilleures, car supprimer un élément en milieu de liste est aussi coûteux qu'en insérer un.

Et si l'on met une liste à gauche de l'opérateur **in**, c'est une liste contenant les mêmes éléments qui doit se trouver dans la liste, et non les éléments proprement dits.

```
In [77]: [ 1, 2 ] in L
Out[77]: False

In [78]: [ 4, 5 ] in L
Out[78]: True
```

Lorsqu'un élément est présent dans une liste, on peut obtenir l'index de sa première occurrence dans la liste avec la fonction `list.index(liste, element)` :

```
In [79]: L = [ 1, 2, 3, 2 ]

In [80]: list.index(L, 2)
Out[80]: 1

In [81]: L.index(2)
Out[81]: 1
```

Cette même fonction retourne une erreur si l'élément n'est pas dans la liste :

```
In [82]: L.index(5)
ValueError: 5 is not in list
```

La fonction `list.remove(liste, element)` permet de retirer ce même élément (seulement sa première occurrence s'il est présent en plusieurs exemplaires) :

```
In [83]: L = [ 1, 2, 3, 2 ]

In [84]: list.remove(L, 2)

In [85]: L
Out[85]: [ 1, 3, 2 ]

In [86]: L.remove(2)

In [87]: L
Out[87]: [ 1, 3 ]x
```

L'interpréteur retournera cette fois encore une erreur si l'élément à retirer de la liste n'est pas présent :

```
In [88]: L.remove(42)
ValueError: list.remove(x): x not in list
```

Enfin, que l'élément soit présent ou non dans la liste, on dispose de la fonction `list.count(liste, element)` pour obtenir son nombre d'occurrences :

```
In [89]: L = [ 1, 2, 3 ]

In [90]: list.count(L, 2)
Out[90]: 1

In [91]: L.count(42)
Out[91]: 0
```

Il ne vous sera peut-être pas permis, lors des concours, d'utiliser ces fonctions (et d'autres dans ce chapitre), et vous aurez peut-être à les réécrire. Nous y reviendrons dans le prochain chapitre.

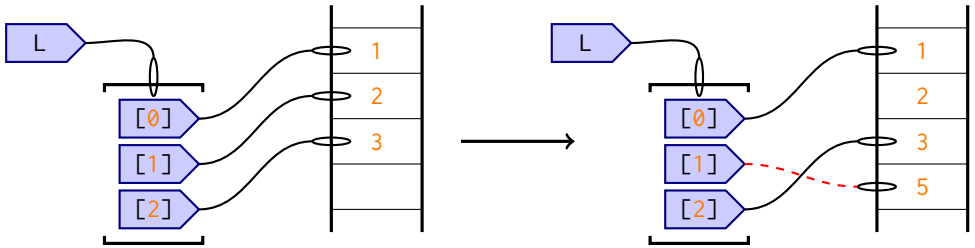
3 Modifier un élément d'une liste

3.1 Mutation d'un élément

Pour le moment, on sait accéder aux éléments d'une liste, en ajouter, en retirer, ou bien encore obtenir des informations sur le contenu de la liste. Il reste un des points les plus importants : la possibilité de « modifier » un élément dans une liste. Cette action est appelée *mutation*[12].

Elle ressemble à s'y méprendre à une affectation, puisqu'elle en reprend l'opérateur `=` :

```
In [92]: L = [ 1, 2, 3 ]

In [93]: L[1] = 5

In [94]: L
Out[94]: [1, 5, 3]
```



12. En Python, les objets qui, comme une liste, peuvent être modifiés de la sorte sont dit *mutables*. Les autres (comme les valeurs numériques mais aussi, on le verra plus tard, comme les chaînes de caractères) sont dit *immutables*.

Effectuer une mutation sur une liste vous paraît sans doute la chose la plus naturelle du monde, mais elle cache une difficulté qui peut parfois jouer de mauvais tours. Reprenons un exemple du premier chapitre :

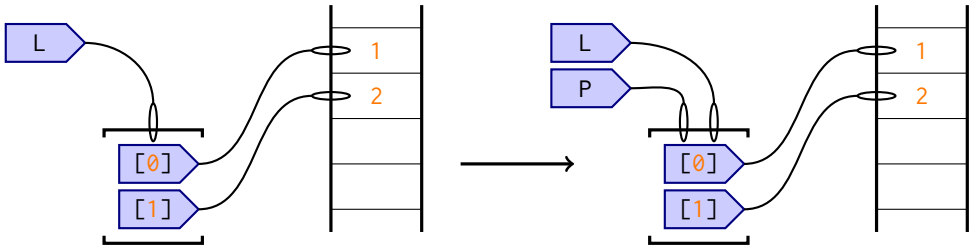
```
In [95]: x = 3

In [96]: y = x

In [97]: x = 5

In [98]: x
Out[98]: 5

In [99]: y
Out[99]: 3
```

Rien de bien surprenant ici. Lors de l'affectation `y = x`, on associe au nom `y` la valeur désignée par x, soit 3. Lorsque l'on réaffecte le nom `x`, cela n'a pas de conséquence sur ce que désigne le nom `y`.

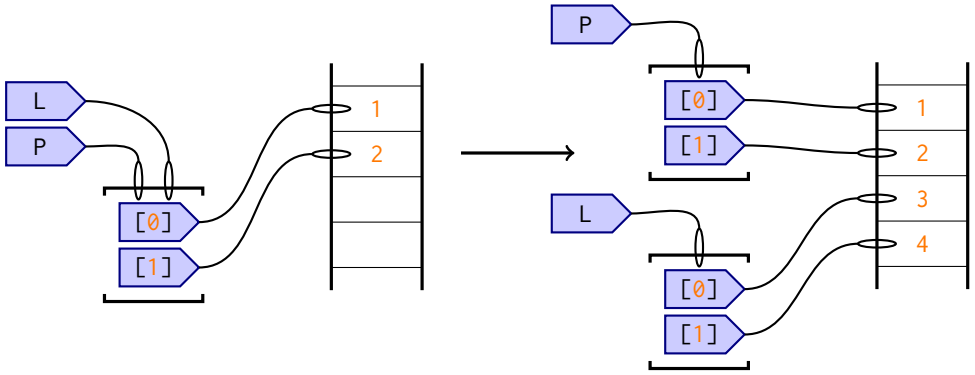
Les affectations sur les listes fonctionnent de la même façon :

```
In [100]: L = [ 1, 2 ]

In [101]: P = L

In [102]: P
Out[102]: [1, 2]

In [103]: L = [ 3, 4 ]

In [104]: L
Out[104]: [3, 4]

In [105]: P
Out[105]: [1, 2]
```

Après l'affectation `P = L`, le nom `P` désigne la *même liste* que le nom `L` :



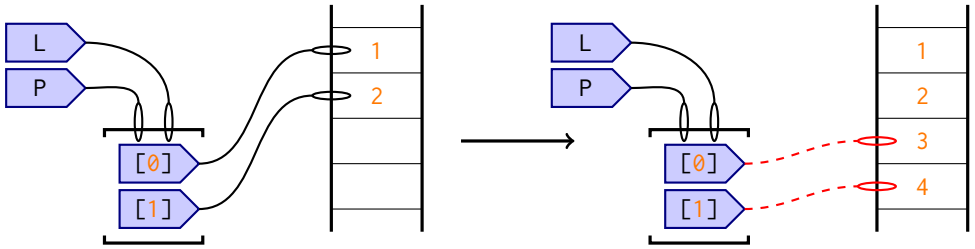
Après la seconde affectation, `L = [ 3, 4 ]`, le nom `L` est associé à une *nouvelle liste* contenant les éléments 3 et 4, le nom `P` désignant toujours la première liste :



Mais lorsque l'on effectue des *mutations*, les choses changent complètement, et l'on peut avoir de très mauvaises surprises :

```
In [106]: L = [ 1, 2 ]

In [107]: P = L

In [112]: P
Out[112]: [1, 2]

In [109]: L[0] = 3

In [110]: P[1] = 4

In [111]: L
Out[111]: [3, 4]

In [112]: P
Out[112]: [3, 4]
```

Après l'affectation `P = L`, puisque `P` et `L` désignent la *même* liste, les mutations `L[0] = 3` et `P[1] = 4` affectent toutes deux à la fois `L` et `P` !



Il n'est donc pas équivalent d'écrire

```
In [113]: L = [ 1, 2 ]

In [114]: P = L
```

et

```
In [115]: L = [ 1, 2 ]

In [116]: P = [ 1, 2 ]
```

En effet, dans le premier cas, `L` et `P` désignent la même liste (les mutations sur `L` auront un effet sur `P` et réciproquement). Dans le second, deux listes différentes, indépendantes, ayant simplement les mêmes éléments.

3.2 Comparer des listes

Que se passe-t-il si l'on compare deux listes avec l'opérateur `==` ? Par exemple si `L` et `P` désignent des listes et que l'on écrit un test tel que

```
if L == P :
```

L'opérateur `==` teste en fait l'*égalité*, et Python ne se soucie donc que du contenu des deux listes, et répondra que les deux listes sont *égales* si et seulement si elles sont de même longueur, et que leurs éléments sont *égaux* entre eux deux à deux.

On peut parfois vouloir savoir si deux noms désignent la *même* liste, et non pas simplement deux listes avec des éléments égaux entre eux. On dispose pour cela d'un autre opérateur, **`is`**, qui renvoie **`True`** si les deux éléments de part et d'autre ne sont pas seulement égaux, mais sont les mêmes [13].

Ainsi, on a

```
In [117]: L = [ 1, 2 ]

In [118]: P = L

In [119]: L == P
Out[119]: True

In [120]: L is P
Out[120]: True
```

13. Il fonctionne aussi sur d'autres éléments, mais pour des raisons assez particulières, notamment de problèmes dits de kerning, évitez de vous en servir sur des objets immutables car les résultats peuvent être inattendus au premier abord.

mais, en revanche :

```
In [121]: L = [ 1, 2 ]

In [122]: P = [ 1, 2 ]

In [123]: L == P
Out[123]: True

In [124]: L is P
Out[124]: False
```

Signalons au passage que l'on peut comparer deux listes avec les autres opérateurs de comparaison, tels que `!=`, `<`, `>=`, etc.

Pour déterminer la plus « grande » de deux listes, l'ordre lexicographique est utilisé (comme sont classés les mots dans un dictionnaire) : on compare d'abord les premiers éléments. Si l'un est plus grand que l'autre [14], alors la liste le contenant est la plus grande. Dans le cas contraire, on réessaie avec les éléments suivants.

Voici quelques exemples :

```
In [125]: [ 5, 1 ] > [ 3, 4, 5, 6 ]
Out[125]: True

In [126]: [ 5, 3, 3 ] > [ 5, 3, 1, 6 ]
Out[126]: True

In [127]: [ 5, 3, 2, 1 ] > [ 5, 3, 2 ]
Out[127]: True
```

Le dernier exemple est un peu particulier : si on ne rencontre que des éléments égaux jusqu'à parvenir à la fin d'une des listes, alors la plus longue est la plus grande (tout élément sera plus grand qu'une absence d'élément), là encore comme dans un dictionnaire.

Enfin, si l'opérateur `!=` est la négation de l'opérateur de `==`, celle de **`is`** est **`is not`** :

```
In [128]: L = [ 1, 2 ]

In [129]: P = [ 1, 2 ]

In [130]: L is not P
Out[130]: True
```

14. Attention à ce que cela ait toujours un sens de comparer les éléments deux à deux avec les opérateurs de comparaison. On peut par exemple écrire `sin > cos` car cela a un sens pour Python, mais sans doute pas le sens auquel on s'attend en mathématiques.

Profitons de l'occasion pour observer une autre situation courante dans laquelle on se retrouve avec la même liste répétée plusieurs fois : lorsque l'on utilise l'opérateur `*`, puisque celui-ci répète le *même* élément en construisant la liste résultat, comme on le constate dans l'exemple ci-dessous [15] :

```
In [131]: L = [ [ 1, 2 ] ] * 3

In [132]: L
Out[132]: [[1, 2], [1, 2], [1, 2]]

In [133]: L[0] is L[1]
Out[133]: True
```

Cela peut avoir des conséquences fâcheuses, puisque les sous-listes ne sont pas indépendantes. Par exemple :

```
In [134]: L[0][0] = 3

In [135]: L
Out[135]: [[3, 2], [3, 2], [3, 2]]
```

On voit que `L` contient ici trois fois la *même* liste, ce qui n'est souvent pas le résultat souhaité. Pour obtenir trois sous-listes indépendantes, on peut utiliser une boucle :

```
In [136]: L = []

In [137]: for _ in range(3) :
              L.append([1, 2])

In [138]: L
Out[138]: [[1, 2], [1, 2], [1, 2]]

In [139]: L[0] is L[1]
Out[139]: False
```

Cette fois-ci, les choses se passent donc mieux :

```
In [140]: L[0][0] = 3

In [141]: L
Out[141]: [[3, 2], [1, 2], [1, 2]]
```

15. On rappelle que `[ 1, 2 ]*3` donnerait `[ 1, 2, 1, 2, 1, 2 ]`.

3.3 Copier une liste

Il survient, dès lors, un problème important. Lorsque l'on veut travailler sur un document sans abimer l'original, on peut généralement en faire une (photo)copie.

Pour obtenir une copie d'une liste `L`, et non pas un autre nom pointant vers la *même* liste, on peut imaginer copier manuellement les éléments :

```
In [142]: L = [ 1, 2 ]

In [143]: P = []

In [144]: for elem in L :
              P.append(elem)

In [145]: P
Out[145]: [1, 2]

In [146]: P == L
Out[146]: True

In [146]: P is L
Out[146]: False

In [147]: P[1] = 4

In [148]: L
Out[148]: [1, 2]

In [148]: P
Out[148]: [1, 4]
```

Beaucoup de solutions, plus ou moins lisibles, permettent d'obtenir le même résultat. Il faut, d'une façon ou d'une autre, provoquer la *création d'une nouvelle liste.*

Par exemple avec une concaténation avec l'opérateur `+` :

```
In [149]: L = [ 1, 2 ]

In [150]: P = L + []

In [151]: P
Out[151]: [1, 2]

In [152]: P is L
Out[152]: False
```

Ou bien encore en partant d'une nouvelle liste vide, que l'on remplit :

```
In [153]: L = [ 1, 2 ]

In [154]: P = []

In [155]: P.extend(L)

In [156]: P
Out[156]: [1, 2]

In [157]: P is L
Out[157]: False
```

Aucune de ces trois méthodes n'est très claire quant aux intentions du programmeur (en particulier les deux dernières), ce qui peut être gênant dans un programme.

Aussi dispose-t-on d'une fonction `list.copy(liste)` qui a pour charge d'effectuer précisément ce que nous avons fait précédemment, en créant une nouvelle liste et en y copiant les éléments de la liste fournie en paramètre :

```
In [158]: L = [ 1, 2 ]

In [159]: P = L.copy()

In [160]: P
Out[160]: [1, 2]

In [161]: P is L
Out[161]: False
```

Malheureusement, `list.copy(liste)` n'est pas la panacée : comme dans nos propres exemples, cette fonction copie le contenu de la liste, mais si un des éléments de cette liste est mutable, par exemple une autre liste, les listes contiendront des choses en commun :

```
In [162]: L = [ [ 1, 2 ], 3 ]

In [163]: P = L.copy()

In [164]: P
Out[164]: [[1, 2]], 3]

In [165]: P is L
Out[165]: False
```

Jusque là, tout va bien...

Mais si l'on creuse...

```
In [166]: P[0] is L[0]
Out[166]: True
```

On s'aperçoit ici que si `list.copy(liste)` a bien copié les éléments de la liste `L` : `P[0]`, copie de `L[0]`, désigne la même sous-liste, donc les deux listes ne sont pas indépendantes.

Cela est encore plus évident lorsque l'on essaie de modifier un élément mutable d'une des deux listes :

```
In [167]: P[0][1] = 4

In [168]: P[1] = 5

In [169]: P
Out[169]: [[1, 4]], 5] # Deux éléments ont changé

In [170]: L
Out[170]: [[1, 4]], 3] # Le second élément a changé, pas le troisième
```

Cela arrivera non seulement pour les listes imbriquées, mais pour tous les éléments mutables qui se trouveraient dans la liste.

Pour (en partie) régler ce problème, il existe une *autre* fonction qui va copier récursivement les éléments de la liste, c'est-à-dire en faisant appel à elle-même pour copier aussi le contenu des éléments mutables de la liste, des éléments mutables à l'intérieur des éléments mutables de la liste et ainsi de suite : la fonction `copy.deepcopy(liste)` du module `copy`.

```
In [171]: import copy

In [172]: L = [ [ 1, 2 ], 3 ]

In [173]: P = copy.deepcopy(L)

In [174]: P
Out[174]: [[1, 2]], 3]

In [177]: P[0] is L[0]
Out[177]: False

In [178]: P[0][1] = 4

In [180]: L
Out[180]: [[1, 2]], 3]
```

Cette fonction pourrait paraître relativement simple, mais c'est en fait potentiellement un sac de nœuds que de copier intégralement une liste, et il peut arriver qu'on ne parvienne pas à le faire.

Sans chercher à trop entrer dans les détails (on pourra aisément ignorer en première lecture le reste de cette section), illustrons une de ces difficultés pour nos lecteurs les plus curieux.

Une liste pouvant contenir n'importe quoi, on peut donc y placer... la liste elle-même :

```
In [181]: L = [ 1, 2, 3 ]

In [182]: L[0] = L
```

Et là, les bizarreries commencent. L'affichage commence à être problématique [16] :

```
In [183]: L
Out[183]: [[...], 2, 3 ]
```

Le contenu de la liste est aussi assez inhabituel :

```
In [184]: L[0] is L
Out[184]: True

In [185]: L[0]
Out[185]: [[...], 2, 3 ]

In [186]: L[0][0][0][0][0][0][0][0][0][0][0][0]
Out[186]: [[...], 2, 3 ]
```

Et les mutations plus ou moins inattendues :

```
In [187]: L[0][1]=5

In [188]: L
Out[188]: [[...], 5, 3 ]
```

La fonction `copy.deepcopy` doit donc détecter qu'une liste fait référence à elle même, car sinon elle copierait infiniment des sous-listes. Elle doit aussi détecter d'éventuelles références circulaires (`L` contient une référence à `P`, laquelle contient une référence à `L`), ce qui est encore plus délicat. Prenons une aspirine, oublions ces difficultés en appréciant que quelqu'un les ait résolues à notre place, et passons à autre chose...

16. L'interpréteur Python identifiant que l'affichage de la liste risque de prendre une taille infinie, il se retrouve forcé de l'abréger.

4 Autres outils pour travailler avec les listes

4.1 Découpages (slicing) de listes

On l'a vu, l'outil `range` permet d'obtenir facilement des séquences d'entiers. Un mécanisme similaire permet de construire une nouvelle liste en extrayant une partie des données d'une liste existante. On parle de « *slicing* » ou de « *découpage* ».

Les règles sont exactement les mêmes que pour `range`, il suffit de les glisser entre les crochets, en les séparant par des doubles points à la place de virgules :

```
In [189]: L = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

In [190]: L[2:5]
Out[190]: [3, 4, 5]

In [191]: L[7:8]
Out[191]: [8]
```

Ici encore, le décompte commence à zéro, le premier index est inclus, le second exclu.

On peut, comme avec range, spécifier un pas avec un troisième paramètre :

```
In [192]: L[1:7:2]
Out[192]: [2, 4, 6]

In [193]: L[4:0:-1]
Out[193]: [5, 4, 3, 2]
```

Petite différence toutefois avec les range, on peut utiliser des indices négatifs pour désigner les éléments en partant de la fin de la liste :

```
In [194]: L[6:-1]
Out[194]: [7, 8, 9]

In [195]: L[9:-5:-1]
Out[195]: [10, 9, 8, 7]
```

Si l'élément indiquant le début ou la fin est omis, alors on ira jusqu'à l'extrémité correspondante de la liste :

```
In [196]: L[1::2]
Out[196]: [2, 4, 6, 8, 10]

In [197]: L[:4:-2]
Out[197]: [10, 8, 5]
```

On a ainsi une des manières les plus efficaces d'obtenir une liste retournée :

```
In [198]: L[::-1]
Out[198]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

A noter que cela construit une *nouvelle* liste dans laquelle les éléments se retrouvent dans l'ordre inverse. Si l'on souhaite inverser les éléments *en place*[17] dans une liste, on peut utiliser la fonction `list.reverse(liste)` :

```
In [199]: L.reverse()

In [200]: L
Out[200]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Tant que nous y sommes, signalons que puisque le slicing d'une liste crée une *nouvelle* liste, un autre moyen d'obtenir une copie[18] d'une liste `L` est d'écrire `L[:]` (copie qui contiendra tous les éléments de `L` puisque l'on les sélectionne du premier au dernier). Sans être la façon la plus claire d'obtenir une copie, comme la syntaxe est brève, on la trouve assez souvent utilisée.

4.2 Mutations et slices

Il est possible d'utiliser les slices dans une mutation, pour modifier plusieurs éléments d'un coup, comme dans l'exemple ci-dessous :

```
In [201]: L = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

In [202]: L[2:7:2] = [ 0, -1, -2 ]

In [203]: L
Out[203]: [1, 2, 0, 4, -1, 6, -2, 8, 9, 10]
```

Il est naturellement important qu'à droite du signe égal, on trouve autant d'éléments qu'en nécessite le slicing, sinon on aura une erreur. On peut donc placer à droite une liste avec le bon nombre d'éléments. On pourrait aussi utiliser l'opérateur `*` :

```
In [204]: L = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

In [205]: L[2:7:2] = [ 0 ] * 3

In [206]: L
Out[206]: [1, 2, 0, 4, 0, 6, 0, 8, 9, 10]
```

17. C'est-à-dire en changeant les éléments de place sans créer de nouvelle liste
18. Une copie simple, pas une copie profonde.

Ou bien un range :

```
In [207]: L = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

In [208]: L[2:7:2] = range(12, 17, 2)

In [209]: L
Out[209]: [1, 2, 12, 4, 14, 6, 16, 8, 9, 10]
```

Ou encore un autre slice :

```
In [210]: L = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

In [211]: L[2:7:2] = L[3:8:2]

In [212]: L
Out[212]: [1, 2, 4, 4, 6, 6, 8, 8, 9, 10]
```

4.3 Compréhensions de listes

Construire des listes avec des dizaines d'éléments, voire bien davantage, n'est pas chose facile (sauf dans le cas d'éléments tous identiques, ou périodiques, auquel cas on peut souvent utiliser l'opérateur `*`).

Il est toujours possible d'utiliser un boucle. Par exemple, pour créer une listes contenant les dix premiers nombres pairs, on peut écrire :

```
In [213]: L = []

In [214]: for i in range(0:20:2) :
              L.append(i)

In [215]: L
Out[215]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

On peut cependant arriver au même résultat en transformant un range en liste, grâce à la fonction `list` :

```
In [216]: L = list( range(0:20:2) )

In [217]: L
Out[217]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Il existe toutefois une méthode beaucoup plus puissante et souple pour construire des listes en Python, les *compréhensions de listes*.

En mathématiques, on peut désigner l'ensemble des dix premiers carrés non nuls en écrivant :

$$\{ k^2 \text{ pour } k \in [1..10] \}$$

En Python, il est de la même façon possible de construire une liste avec une telle règle avec la syntaxe suivante :

```
[ expression for nom in séquence ]
```

Par exemple, il est possible de construire une liste en Python contenant ces dix premiers carrés de la façon suivante :

```
In [218]: L = [ k**2 for k in range(1, 11) ]

In [219]: L
Out[219]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Techniquement, on n'a pas affaire ici à une boucle **for**, même si l'on retrouve la syntaxe **for** nom **in** séquence, avec un fonctionnement très similaire.

On peut calculer le sinus de ces dix valeurs en utilisant une autre compréhension :

```
In [220]: P = [ math.sin(x) for x in L ]
```

Ce genre de construction fonctionne notamment très bien avec les fonctions telles que `sum`. La somme des entiers de 1 à 100 peut par exemple s'écrire [19] :

```
In [221]: sum( list( range(1, 101) ) )
Out[221]: 5050
```

Quant à la somme des dix premiers carrés non nuls, $\sum_{i=1}^{10} i^2$, elle peut s'écrire [20] :

```
In [222]: sum( [ k**2 for k in range(1, 11) ] )
Out[222]: 385
```

On peut ainsi calculer simplement $\sum_{k=0}^{15} \sin\left(\frac{k\pi}{15}\right)$:

```

```

19. Il serait possible – et souhaitable – d'écrire ceci un peu différemment, comme cela est discuté à la fin de ce chapitre.
20. Là encore, voir la fin du chapitre pour une meilleure écriture de cette somme.

4.4 Compréhensions de listes avec conditions

Par ailleurs, on peut aussi ne garder qu'une partie des valeurs ainsi calculées, en ajoutant une condition, avec la syntaxe suivante :

```
[ expression for nom in séquence if expression_booléenne ]
```

Par exemple, on peut obtenir la liste des carrés inférieurs à 1000 qui se terminent par 9 :

```
In [223]: L = [ k**2 for k in range(1, int(sqrt(1000))+1)
                                                  if (k**2)%10 == 9 ]

In [224]: L
Out[224]: [9, 49, 169, 289, 529, 729]
```

De même, la liste des entiers inférieurs à 100 divisibles par 7 mais pas par 3 :

```
[ i for i in range(100) if i%7 == 0 and i%3 != 0 ]

# ou bien

[ i for i in range(0, 100, 7) if i%3 != 0 ]
```

Cette syntaxe permet par ailleurs de filtrer très simplement une liste. Par exemple, une façon d'obtenir une (nouvelle) liste ne contenant que les termes non nuls d'une liste `L`, il suffirait d'écrire

```
[ elem for elem in L if elem != 0 ]
```

C'est aussi un moyen supplémentaire de copier[21] une liste `L` :

```
[ elem for elem in L ]
```

crée en effet une nouvelle liste contenant *tous* les éléments de `L`.

4.5 Moyenne et variance

On considère à présent une liste `L` contenant un ensemble de N valeurs numériques, notées L_i (où $0 \leqslant i \leqslant N-1$).

La valeur moyenne m associée à cette liste de valeurs est définie mathématiquement par l'expression :

$$m = \frac{1}{N}\sum_{i=0}^{N-1} L_i$$

21. Une copie simple, cette fois encore, et non profonde.

En Python, le calcul de m peut se faire très simplement avec une boucle :

```
somme = 0

for elem in L :
    somme = somme + elem

moyenne = somme / len(L)
```

Ou bien plus simplement, en une seule ligne :

```
moyenne = sum(L) / len(L)
```

On souhaite également déterminer la variance V, définie par :

$$V = \frac{1}{N}\sum_{k=0}^{N-1} (L_i - m)^2$$

ainsi que l'écart-type σ, défini par $\sigma = \sqrt{V}$.

Le théorème de König-Huygens fournit une autre expression pour V :

$$V = \frac{1}{N}\left(\sum_{k=0}^{N-1} L_i^2\right) - m^2$$

Il suffit en effet de développer la première expression pour obtenir la seconde :

$$V = \frac{1}{N}\sum_{k=0}^{N-1}\left(L_i^2 - 2L_i m + m^2\right) = \frac{1}{N}\left(\sum_{k=0}^{N-1} L_i^2\right) - 2m\left(\frac{1}{N}\sum_{k=0}^{N-1} L_i\right) + m^2 = \frac{1}{N}\left(\sum_{k=0}^{N-1} L_i^2\right) - 2m^2 + m^2$$

Cette seconde expression demande un peu moins de calculs.

En effet, la première nécessite N soustractions, N mises au carré, N – 1 sommes et une division, soit 3N opérations. La seconde, N + 1 mises au carré, N – 1 sommes, une division et une soustraction, soit 2N + 2 opérations.

En supposant que la moyenne ait déjà été calculée, on peut déterminer la variance et l'écart-type, cette fois encore, avec une boucle :

```
sommecarres = 0

for elem in L :
    sommecarres = sommecarres + elem**2

variance = sommecarres / len(L) - moyenne**2

ecartType = math.sqrt(variance)
```

Ou bien, plus succintement, grâce aux outils de la section précédente :

```
variance = sum( [ elem**2 for elem in L ] ) / len(L) - moyenne**2

ecartType = math.sqrt(variance)
```

Avec un inconvénient toutefois, puisque l'on parcours *deux fois* l'ensemble de la liste pour obtenir moyennes et variance, alors qu'on pourrait ne le faire qu'une seule fois :

```
somme, sommecarres = 0, 0

for elem in L :
    somme = somme + elem
    sommecarres = sommecarres + elem**2

moyenne = somme / len(L)
ecartType = sqrt(sommecarres - moyenne**2)
```

Il peut être utile de ne parcourir qu'une seule fois la liste dans deux situation :

- lorsqu'il y a un gros volume de données, car elles peuvent être longues à charger, aussi s'efforce-t-on d'avoir à les charger deux fois ;
- lorsque les données arrivent, par exemple d'un capteur, sous la forme d'un « flot » continu de données, ce qui permet de calculer les deux grandeurs sans avoir à stocker les données.

Remarque : Toutes les formules proposées ici ne fonctionnent pas bien avec de grandes séries de données, car les additions successives peuvent causer des erreurs d'arrondis potentiellement importantes, comme nous le verrons un peu plus tard. Calculer correctement moyenne, variance et écart-type n'est pas un problème simple.

Il serait préférable ici d'utiliser la fonction `math.fsum` du module `math` de préférence à `sum`, car elle gère bien mieux éventuelles questions d'arrondis et de perte de précision.

Remarque (bis) : les formules données ici pour la variance sont celles qui visent à calculer la variance d'un échantillon. Vous rencontrerez probablement à un moment ou un autre (par exemple durant les travaux pratiques de sciences physiques lors des estimations des incertitudes) des formules très similaires, mais dans lesquelles figure un facteur $\frac{1}{N-1}$ en lieu et place de $\frac{1}{N}$:

$$V_{n-1} = \frac{1}{N-1}\left(\sum_{k=0}^{N-1} L_i^2\right) - m^2$$

Cela arrive lorsque l'on souhaite estimer la variance d'un processus aléatoire dont les données constituent des réalisations. Dans cette situation, on peut montrer que la formule avec le terme $\frac{1}{N}$ est un estimateur *biaisé* de la variance du processus aléatoire.

Au contraire, la formule précédente, avec le facteur $\frac{1}{N-1}$ constitue un bon estimateur de la variance du processus aléatoire, dépourvu de biais.

4.6 Tri des éléments d'une liste

Pour trier, *en place*, une liste par ordre croissant des éléments la constituant, on peut utiliser la fonction `list.sort(liste)` :

```
In [225]: L = [ 4, 10, 8, 9, 1, 5, 7, 2, 3, 6 ]

In [226]: L.sort()

In [227]: L
Out[227]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

On peut inverser l'ordre du tri en ajoutant en paramètre `reverse=True` :

```
In [228]: L = [ 4, 10, 8, 9, 1, 5, 7, 2, 3, 6 ]

In [229]: L.sort(reverse=True)

In [230]: L
Out[230]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Précisons dès à présent que les algorithmes permettant le tri de données, et leur efficacité, constituent une part particulièrement importante du programme de seconde année.

Les éléments à trier sont comparés grâce aux opérateurs de comparaison de Python tels que `<`, aussi peut-on trier tous les objets pour lesquels l'opérateur `<` a un sens.

On peut aussi préciser une fonction f, grâce à l'argument `key`, permettant de les comparer. x sera placé avant y si[22] $f(x) < f(y)$:

```
In [231]: L = [ 4, 10, 8, 9, 1, 5, 7, 2, 3, 6 ]

In [232]: L.sort(key=math.sin)

In [233]: L
Out[233]: [5, 4, 10, 6, 3, 9, 7, 1, 2, 8]
```

En effet, on a

$$\sin(5) < \sin(4) < \sin(10) < \sin(6) < \sin(3) < \sin(9) < \sin(7) < \sin(1) < \sin(2) < \sin(8)$$

22. En cas d'égalité de $f(x)$ et $f(y)$, l'ordre d'apparition de x et y dans la liste originale est préservé.

4.7 Mélange des éléments d'une liste et tirages aléatoires

Pour réaliser l'opération inverse et mélanger en place les éléments d'une liste, on peut utiliser la fonction `random.shuffle(liste)` du module `random` :

```
In [234]: L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

In [235]: import random

In [236]: random.shuffle(L)

In [237]: L
Out[237]: [10, 4, 7, 1, 9, 3, 5, 2, 8, 6]

In [637]: random.shuffle(L)

In [638]: L
Out[638]: [5, 8, 4, 3, 10, 2, 7, 6, 1, 9]
```

Le module `random` contient en fait de très nombreuses fonctions liées à l'aléatoire. Citons par exemple la fonction `random.sample(liste, k)` qui crée une nouvelle liste contenant k éléments pris au hasard, sans remise [23], dans `liste` :

```
In [238]: random.sample(L, 3)
Out[238]: [9, 4, 10]

In [239]: random.sample(L, 7)
Out[239]: [4, 1, 9, 8, 10, 2, 3]
```

5 Au-delà des compréhensions de listes, les générateurs (H. P.)

5.1 Mise en garde

Ce qui a été présenté dans ce chapitre jusqu'ici constitue une base concernant les listes en Python que vous devriez maîtriser d'ici aux concours, même si vous n'avez pas nécessairement à connaître rigoureusement toutes les fonctions (bien qu'elles constituent autant d'outils qui vous aideront à écrire plus simplement des algorithmes).

Le programme des classes préparatoires s'intéresse surtout à la création d'une liste, l'accès et la modification des éléments, l'ajout et la suppression (avec les problèmes éventuels que cela représente en terme de coût de calcul), et le principe des itérations sur une liste, que l'on utilisera très souvent par la suite.

23. Sans remise lors d'un appel à la fonction, mais le même élément peut évidemment apparaître dans des appels successifs à la fonction, comme sur l'exemple proposé !

Les compréhensions de listes ne figurent pas explicitement au programme, mais c'est un outil tellement utile qu'on les trouve dans la quasi-totalité des programmes Python, et possiblement au détour d'un sujet. Elles vous simplifieront grandement la vie une fois que vous les maîtriserez, aussi valent-elle la peine que l'on s'y attarde.

Le reste de ce chapitre introduit très brièvement des notions qui seront utiles à ceux qui décideront d'aller plus loin avec Python, mais qui vont bien au-delà de ce qui vous sera utile pour les concours. Tant que vous ne serez pas à l'aise avec ce qui précède, il est vraisemblablement préférable de l'ignorer intégralement.

5.2 Mieux utiliser `sum`, `max` et `min`

Plus tôt dans le chapitre, on a écrit la somme des entiers de 1 à 100 de cette façon :

```
sum( list(range(1, 101)) )
```

Et la somme des dix premiers carrés non nuls de la sorte :

```
sum( [ k**2 for k in range(1, 11) ] )
```

En fait, une fonction comme `sum` ne prend pas uniquement une liste comme argument. Elle a simplement besoin qu'on lui passe en paramètre un objet qui correspond à un ensemble de valeurs que l'on puisse égrenner une à une.

Un tel objet, en Python, est appelé un *itérable*. Une liste est un exemple d'itérable, mais ce n'est pas le seul. `range` fournit également un itérable, bien que ce ne soit pas une liste. On peut donc passer directement un `range` comme argument de `sum` :

```
In [240]: sum( range(1, 101) )
Out[240]: 5050
```

On voit que le résultat est le même. La façon dont l'interpréteur gère les choses est en revanche très différente.

En effet, dans le cas d'un range, Python ne réserve pas une zone en mémoire pour ranger l'ensemble des valeurs [24], comme il le ferait en construisant la liste. Elles sont produites une par une et fournies au fur et à mesure à la fonction `sum`.

La différence est notable : il n'y a probablement pas assez de mémoire dans la plupart des ordinateurs pour calculer `sum(list(range(1, 10**9)))`, mais `sum(range(1, 10**9))` ne pose pas de problème de mémoire (en revanche, le calcul sera sans doute *très* long).

Même lorsqu'il y a suffisamment de mémoire pour construire la liste, ne pas mémoriser toutes les valeurs avant de calculer leur somme peut également faire gagner un peu de temps. Cette seconde écriture est donc largement préférable.

24. En Python 3k. Le comportement de Python 2 est un peu différent, mais la plupart des idées présentées ici restent valables.

Il en est de même pour les compréhensions de listes. Lorsque l'on s'en sert comme argument d'une fonction comme `sum`, `max` ou `min`, il est inutile de stocker intégralement la liste en mémoire avant de la fournir à la fonction, comme on le fait en écrivant :

```
sum( [ k**2 for k in range(1, 11) ] )
```

On écrira, de préférence

```
sum( k**2 for k in range(1, 11) )
```

sans les crochets, ce qui a pour effet, comme dans le cas de `range` précédemment, de produire les valeurs une à une au fur et à mesure que `sum` les utilise, sans jamais les stocker toutes en même temps en mémoire, avec les mêmes bénéfices que précédemment.

En l'absence de crochets, on ne parle plus de compréhension de liste, mais de *générateur*, c'est-à-dire une expression qui va générer un ensemble de valeurs selon la règle fournie.

5.3 Générateurs dans les boucles

On pourrait vouloir faire une boucle sur une compréhension de liste, par exemple cette série d'instructions qui affiche les carrés inférieurs à 10000 qui se terminent par un 9 :

```
for carre in [ k**2 for k in range(1, 101) if k**2%10==9 ] :
    print(carre)
```

Cette fois encore, la boucle **for** a besoin des valeurs une par une, et il est inutile d'avoir à un instant donné toute la liste construite quelque part en mémoire (a fortiori si l'on peut éventuellement quitter la boucle prématurément avec un **break**, il n'est pas dit qu'il soit utile de calculer tous les éléments constituant la liste !).

Par conséquent, ici aussi, on souhaite retirer les crochets afin d'utiliser un générateur plutôt qu'une compréhension de liste. Mais pour éviter de se retrouver avec deux instructions **for** sur la même ligne, ce qui peut rendre la tâche un peu complexe à l'interpréteur, il faudra mettre des parenthèses à la place pour isoler le générateur :

```
for carre in ( k**2 for k in range(1, 100) if k**2%10==9 ) :
    print(carre)
```

Il est encore plus facile de voir pourquoi cela importe. Le programme suivant finira, après un certain temps, par déclencher une erreur pour raison de mémoire insuffisante :

```
for i in [ x**2 for x in range(10**20) ] :
    print(i)
    if i>=100 :
        break
```

alors que cette version fonctionne parfaitement bien :

```
for i in ( x**2 for x in range(10**20) ) :
    print(i)
    if i>=100 :
        break
```

5.4 Générateurs particuliers

On dispose par ailleurs de plusieurs générateurs spéciaux, qui sont notamment très utiles lorsque l'on crée une boucle **for** en Python, ou bien lorsque l'on écrit des compréhensions de listes.

On connait déjà l'un d'entre eux `enumerate`. Celui-ci prend en argument un itérable (qui peut être une liste, comme on l'a vu, mais aussi bien d'autres choses) et fournit successivement les couples [25] (indice, élément) pour tous les éléments de l'itérable fourni en argument :

```
In [241]: L = [ 14, 17, 22, 42, 49, 54 ]

In [242]: for elem in enumerate(L) :
              print(elem)
(0, 14)
(1, 17)
(2, 22)
(3, 42)
(4, 49)
(5, 54)
```

C'est encore plus pratique d'écrire les choses comme ceci, en attribuant un nom aux deux éléments du couple :

```
In [243]: for idx, elem in enumerate(L) :
              print("L'élément d'index", idx, "est", elem)
L'élément d'index 0 est 14
L'élément d'index 1 est 17
L'élément d'index 2 est 22
L'élément d'index 3 est 42
L'élément d'index 4 est 49
L'élément d'index 5 est 54
```

Nous avons déjà eu l'occasion de voir combien cette écriture peut simplifier la vie aux programmeurs Python.

25. Ces « couples » sont en fait appelés *tuples* en Python ; c'est un type un peu particulier, qui peut contenir plus de deux éléments, et qui se manipule comme une liste mais qui n'est pas mutable.

Il existe d'autres générateurs de ce genre. Le premier est `sorted`. Comme `enumerate`, il prend en argument un itérable, puis il fournit les valeurs contenues dans l'itérable dans l'ordre croissant, sans que l'on ait eu besoin de trier l'itérable lui-même :

```
In [244]: L = [ 49, 42, 17, 22, 14, 54 ] :

In [245]: for elem in sorted(L) :
              print(elem)
14
17
22
42
49
54

In [246]: L
Out[246]: [49, 42, 17, 22, 14, 54]
```

On voit ici que la liste désignée par `L` n'a pas été modifiée, même si Python a trié les valeurs (quelque part ailleurs dans la mémoire) pour nous fournir les éléments dans l'ordre, ce qui peut s'avérer très pratique.

On dispose également de `reversed`, qui a pour effet de fournir les éléments de l'itérable passé en paramètre dans l'ordre inverse :

```
In [247]: L = [ 49, 42, 17, 22, 14, 54 ]

In [248]: for elem in reversed(L) :
              print(elem)
54
14
22
17
42
49

In [249]: L
Out[249]: [49, 42, 17, 22, 14, 54]
```

Ici encore, `L` désigne toujours la liste originale, Python s'est simplement débrouillé de lui-même pour que l'on obtienne les éléments dans l'ordre inverse.

Ces deux dernières techniques « cachent » des calculs effectués par Python et un coût mémoire, et ce n'est pas toujours la façon de faire la plus efficace, mais dans bien des situations, cela simplifie l'écriture de certains programmes, et aussi grandement la vie d'un programmeur...

Enfin, vous aurez peut-être parfois envie d'égrenner plusieurs listes en même temps, par exemple lorsque vous aurez une liste `L1` et une liste `L2` de même longueur, et que vous voudrez construire la liste `P` contenant les produits des éléments des deux listes. Une façon « simple » de faire est la suivante :

```
P = []
for i in range(len(L1)) :
    P.append( L1[i] * L2[i] )
```

On peut également écrire

```
P = [ L1[i] * L2[i] for i in range(len(L1)) ]
```

Toutefois, il existe une fonction appelée `zip` qui permet de fabriquer des couples de deux valeurs (ou plus) prises dans deux itérables. Ainsi :

```
In [250]: L1 = [ 14, 17, 42, 54 ]

In [251]: L2 = [ 2, 3, 5, 7 ]

In [252]: for elem in zip(L1, L2) :
              print(elem)
(14, 2)
(17, 3)
(42, 5)
(54, 7)
```

On peut à nouveau choisir deux noms différents pour chacun des éléments du couple :

```
In [253]: L1 = [ 14, 17, 42, 54 ]

In [254]: L2 = [ 2, 3, 5, 7 ]

In [255]: for e1, e2 in zip(L1, L2) :
              print(e1, "fois", e2, "égale", e1*e2)
14 fois 2 égale 28
17 fois 3 égale 51
42 fois 5 égale 210
54 fois 7 égale 378
```

Et on peut donc construire la liste des produits ainsi :

```
P = [ e1*e2 for e1, e2 in zip(L1, L2) ]
```

On trouvera d'autres générateurs utiles dans le module `itertools`.

5.5 Listes de booléens

Il est bien évidemment possible de définir des listes contenant des booléens, notamment en utilisant des compréhensions de listes. Par exemple, on peut vouloir savoir quels sont les nombres entre 2 et 5 qui divisent 6 :

```
In [256]: [ 6%k == 0 for k in range(2, 6) ]
Out[256]: [True, True, False, False]
```

Deux fonctions sont très utiles pour travailler avec ce genre de liste : `all` et `any`.

La fonction `all` renvoie **`True`** si et seulement si l'ensemble des éléments de la liste passée en paramètre sont évalués à **`True`**.

La fonction `any`, à l'inverse, renvoie **`True`** si au moins un des éléments de la liste passée en paramètre est évalué à **`True`**, et **`False`** dans le cas contraire.

Ainsi, on peut déterminer si un nombre n est premier en écrivant :

```
if all( [ n%k != 0 for k in range(2, n) ] ) :
    print(n, "est premier")
```

Encore plus que précédemment, ce n'est pas la meilleure façon d'écrire ces tests, car la fonction `all`, par exemple, sait s'arrêter dès qu'un élément est égal à **`False`**, puisque cela impliquera que le résultat sera nécessairement **`False`**. De même, la fonction `any` s'arrête au premier élément égal à **`True`**, puisque le résultat sera alors nécessairement **`True`**.

Or l'utilisation d'une compréhension de liste crée nécessairement une liste avec les résultats de *tous* les tests, qu'ils soient ensuite utiles ou non.

Ainsi, une fois encore, il est préférable d'omettre les crochets afin de privilégier un générateur à une compréhension de liste, et d'écrire

```
if all( n%k != 0 for k in range(2, n) ) :
    print(n, "est premier")
```

car cette écriture fournit à la fonction `all` les booléens au fur et à mesure de ses besoins, ce qui permettra de ne pas prendre la peine d'essayer d'autres diviseurs possibles dès qu'on en a trouvé un, ce qui permet immédiatement de conclure que le nombre n'était pas premier.

On peut aussi écrire un test de primalité avec la fonction `any` ainsi :

```
if not any( n%k == 0 for k in range(2, n) ) :
    print(n, "est premier")
```

Exercices

Ex. 1 – Fond de caisse

Le prix d'entrée d'une manifestation est de cinq euros. Un groupe de personnes, placés dans une file d'attente, souhaite assister à la manifestation. Cependant, tous n'ont pas l'appoint pour payer l'entrée : chacune de ces personnes dispose d'un unique billet, de cinq, de dix ou de vingt euros. Ces informations sont regroupées dans une liste, telle que :

```
L = [ 5, 10, 5, 20 ]
```

Dans cet exemple, la première des personnes dans la file possède un billet de cinq euros, la seconde un billet de dix euros, etc. On souhaite savoir si la personne au guichet (dont la caisse initialement est vide) pourra rendre la monnaie à tout le monde. Ici, c'est en effet possible (on rendra la monnaie à la seconde personne avec le billet de la première entrée, et à la quatrième personne avec les billets des seconde et troisième entrées).

1. Écrire un programme déterminant si, pour une liste `L` donnée, il sera possible de faire entrer tous les spectateurs dans l'ordre où ils se présentent, ou si la personne au guichet se retrouvera dans l'impossibilité de rendre la monnaie.

2. Même question si l'on se permet de modifier l'ordre des personnes dans la file.

Ex. 2 – Suite des sommes des carrés de chiffres

Lorsque l'on somme les carrés des chiffres d'un entier positif strictement u_n, on obtient un nouvel entier positif u_{n+1}. En itérant le processus, on obtient une suite (u_n) d'entiers. Ces suites ont l'intéressante particularité de toujours conduire[26] soit vers 1, soit vers le cycle $89 \to 145 \to 42 \to 20 \to 4 \to 16 \to 37 \to 58 \to 89$.

1. Proposer un programme qui, partant d'un `u0` fourni, construit la liste des éléments de la suite jusqu'à trouver la valeur 1 ou la valeur 89[27]. Par exemple, pour les valeurs initiales 49 et 17, on devra obtenir les listes

```
[49, 97, 130, 10, 1]
[17, 50, 25, 29, 85, 89]
```

2. Déterminer, pour l'ensemble des valeurs initiales `u0` entre 1 à 10000, combien d'entre elles convergent vers 1 et combien « terminent » sur le cycle.

3. Pour quelle valeur de `u0` comprise entre 1 et 10000 le nombre d'itérations nécessaire pour parvenir à 1 ou à 89 est-elle le plus grand ?

26. Il est aisé de montrer qu'il n'est pas possible d'avoir une suite non-cyclique, si cela vous intéresse.
27. Traditionnellement, on choisit 89 parmi les valeurs du cycle, car c'est la valeur du cycle que l'on rencontre le plus souvent en premier.

Ex. 3 – Liste itérée de Conway

Le mathématicien britannique John Horton Conway, a proposé une règle permettant d'itérer des listes d'entiers. Lorsque l'on débute pour $n = 0$ avec la liste `[ 1 ]`, les premières itérations ($n = 1$ à $n = 4$) de cette série sont les suivants :

```
[ 1, 1 ]
[ 2, 1 ]
[ 1, 2, 1, 1 ]
[ 1, 1, 1, 2, 2, 1 ]
```

Pour passer à l'étape suivante, on décrit le contenu des listes : la liste `[ 2, 1 ]` pour $n = 2$ contient « un 2, un 1 », d'où la liste `[ 1, 2, 1, 1 ]` à l'ordre $n = 3$, et ainsi de suite.

1. Justifier que les listes ne contiendront que des 1, des 2 et des 3, et écrire un programme qui construit la liste de Conway à l'ordre n.

2. Construire une liste contenant les longueur l_k des listes de Conway à l'ordre k pour tous les ordres $0 \leq k \leq n$.

3. Construire de même une liste contenant les rapports de deux longueurs successives l_{k+1}/l_k pour tous les k vérifiant $0 \leq k \leq n-1$, et vérifier que la suite formée par ces termes semble tendre vers une constante.

4. Vérifier que cette constante ne dépend pas de la liste initiale choisie (on choisira une autre liste que `[ 1 ]` comme initialisation, à l'exception de `[ 2, 2 ]`).

Ex. 4 – Sommes minimales

On considère une liste `L` d'entiers positifs. On souhaite déterminer la plus petite somme de deux de ces entiers qui soit strictement supérieure à tous les éléments de la liste.

Par exemple, avec la liste `[ 3, 5, 8 ]`, cette somme serait $3+8=11$ ($3+5$ n'étant pas strictement supérieur à l'élément 8 présent dans la liste, et $5+8$ étant plus grand que $3+8$).

1. Proposer un programme déterminant cette somme de deux entiers.

2. On suppose à présent les éléments de la liste ordonnés par ordre croissant. Proposer un programme n'envisageant que $n-1$ sommes pour parvenir à ce même résultat.