3 Listes et tuples

La *liste* et le *tuple* sont deux types de variable permettant de stocker derrière un seul identifiant une collection ordonnée de plusieurs valeurs (non-nécessairement uniques, c'est-à-dire qu'une même valeur peut apparaître plusieurs fois).

3.1 Initialisation basique

Pour créer un **tuple**, on saisit ses éléments séparés par des virgules (le tout éventuellement entre parenthèses s'il est nécessaire de préciser les priorités). Deux cas particuliers :

— pour créer un tuple vide, on saisit une paire de parenthèses vide : `()` ,

— pour créer un tuple constitué d'une seule valeur, on laisse tout de même une virgule après cette valeur pour indiquer à Python qu'il s'agit bien d'un tuple constitué de cette valeur, et non pas de la valeur elle-même. Par exemple `3,` ou `(3,)` renvoient un tuple contenant seulement la valeur 3.

À tester. Saisissez les instructions suivantes et observez le type et le contenu des variables créées dans l'explorateur de variables. (Lorsqu'une variable contient une collection de valeurs, on peut double-cliquer dessus dans l'explorateur de variables pour afficher plus de détails.)

```
T1 = (3, 4, 10, 2)
T2 = 5, 6, 3, 6, 6
T3 = ()
T4 = 3,
T5 = 3
```

Pour créer une **liste**, on saisit ses éléments séparés par des virgules en les plaçant obligatoirement entre crochets. Notez que :

— pour créer une liste vide, on saisit une paire de crochets vide : `[]` ,

— pour créer une liste constituée d'une seule valeur, il n'est pas utile de laisser une virgule après cet élément dans la mesure où les crochets obligatoires indiquent déjà qu'il s'agit d'une liste. Par exemple, l'expression `[3]` renvoie une liste contenant seulement la valeur 3.

À tester. Saisissez les instructions suivantes et observez le type et le contenu des variables créées.

```
L1 = [3, 4, 10, 2]
L2 = 5, 6, 3, 6, 6   # Ceci n'est pas une liste !
L3 = []
L4 = [3]
```

Notez que rien n'impose que toutes les valeurs contenues dans un tuple ou dans une liste soient d'un même type. Il est tout à fait possible de créer des collections *hétérogènes* en stockant dans un tuple ou une liste des valeurs de types différents, y compris d'autres tuples et listes.

À tester. Examinez en détail la variable créez par l'instruction suivante.

```
L = [1, "Bonjour", 3.0, (0, True), ["3", 1, [2.0], (8, 3, 1)], 5, 5]
```

3.2 Propriétés des listes et tuples

Voici quelques propriétés des types `list` et `tuple`. Dans les exemples suivants, `C` désigne une variable de l'un de ces deux types, et on note N le nombre d'éléments qu'elle contient (on peut obtenir ce nombre à l'aide de l'instruction `len(C)`).

1) Les types `tuple` et `list` sont **indiçables** (*indexable* ou *subscriptable* en anglais). C'est-à-dire que l'on peut accéder à un élément ou à une partie de `C` à l'aide de la syntaxe `C[i]`.

   Plus particulièrement, `i` doit être au choix (voir figure 3.1) :

   — un entier compris entre 0 (inclus) et N (exclus), correspondant à une numérotation naturelle du premier élément jusqu'au dernier,

   — un entier compris entre -1 (inclus) et $-N$ (inclus), correspondant à une numérotation en sens inverse, du dernier élément jusqu'au premier,

   — un objet de type *slice* (« tranche » en français) obtenu à l'aide de la syntaxe `a:b` ou `a:b:h` pour désigner le sous-ensemble formé des éléments d'indices a (inclus) à b (exclus) avec un pas de h (valant 1 par défaut s'il est omis)[1]. Si on ne donne pas de valeur de a ou de b, alors la tranche se fait à partir de (ou jusqu'au) premier (ou dernier) élément (suivant le contexte).

À tester.

a) En introduisant au préalable la liste `L = [17, 24, 62, 38, 50, 87, 99]`, essayez d'anticiper puis vérifiez le résultat des instructions ci-dessous.

```
print(L[0])
print(L[1])
print(L[4])
print(L[6])
print(L[7])
```

```
print(L[-1])
print(L[-2])
print(L[-5])
print(L[-7])
print(L[-8])
```

```
print(L[1:4])
print(L[0:7:2])
print(L[4:0:-1])
print(L[:4])
print(L[2:])
```

b) Même question en définissant au préalable `L = (17, 24, 62, 38, 50, 87, 99)`.

c) Même question en définissant au préalable `L = "abcdefg"`.

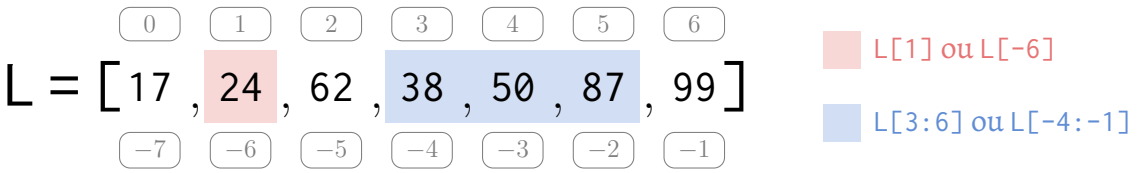

FIGURE 3.1 – Indexation des éléments d'une liste L (fonctionnement similaire pour un tuple).

1. Notez la similarité avec le fonctionnement de l'instruction `range(a, b, h)`.

2) Les types `tuple` et `list` sont **itérables**. C'est-à-dire que l'on peut parcourir les valeurs de `C` (dans l'ordre où elle sont stockées) à l'aide de la syntaxe `for x in C` .

À tester. Testez le code suivant.

```
T = (33, 17, 28, 96, 30, 44)
for x in T:
    print("x vaut " + str(x))
```

Exercice 3.1. Retour sur les algorithmes classiques. On suppose déjà définies une variable `values` de type `list` ou `tuple` contenant des nombres entiers, et une variable `x` de type `int`.

1. Écrire un programme qui détermine et stocke dans une variable `n` le nombre de fois où la valeur de `x` apparaît parmi les valeurs stockées dans `values`.
2. Écrire un programme qui détermine et stocke dans une variable `found` le booléen `True` si la valeur de `x` est présente dans `values`, et `False` sinon.
3. Écrire un programme qui détermine et stocke dans une variable `i` le plus petit indice positif tel que `values[i]` a même valeur que `x`. Si la valeur `x` n'apparaît pas dans `values` on posera `i = -1`.

Exercice 3.2. Encore des algorithmes classiques. On suppose déjà définie une variable `numbers` de type `list` ou `tuple` contenant des nombres entiers.

1. Écrire un programme qui détermine et stocke dans deux variables `m` et `M` la plus petite valeur et la plus grande valeur parmi celles présentes dans `numbers`.
2. Écrire un programme qui calcule et stocke dans une variable `S` la somme des éléments de `numbers`.

Vos programmes fonctionnent-ils si `numbers` ne contient plus des entiers mais des chaînes de caractères ?

3) Les types `list` et `tuple` supportent les opérations suivantes :

— si `C1` et `C2` sont deux tuples (ou deux listes), alors `C1 + C2` renvoie un nouveau tuple (ou une nouvelle liste) obtenu(e) en *concaténant*[2] `C1` et `C2`,

— si `C` est un tuple (ou une liste) et `n` un entier, alors `C * n` et `n * C` renvoient un nouveau tuple (ou une nouvelle liste) obtenu(e) en concaténant n copies de `C`.

À tester. Exécutez le programme suivant. Si un message d'erreur apparaît, essayez de le comprendre et supprimez la ligne causant l'erreur. Répétez l'opération jusqu'à ce que le programme se termine sans erreur, et observez alors les variables créées.

```
A = [1, 3, 6] + [7, 0, 8]
B = (17, 33, 20) + (12, 45, 9)
C = 17, 33, 20 + 12, 45, 9
D = [1, 3, 6] + (12, 45, 9)
E = [1, 3, 6] * 10
F = [2, 4, 1] * 0
G = [9, 0, 8] * 2.0
```

2. C'est-à-dire en « collant » bout à bout.

4) On peut appliquer aux types `list` et `tuple` les fonctions suivantes :
   — `len(C)` renvoie la taille (c'est-à-dire le nombre d'éléments) de `C`,
   — `min(C)` et `max(C)` renvoient respectivement le plus petit et le plus grand élément de `C`,
   — `sum(C)` renvoie la somme des éléments de `C`.

   De plus, les types `list` et `tuple` admettent les *méthodes* suivantes :
   — `C.count(x)` renvoie le nombre d'apparition de la valeur `x` dans `C`,
   — `C.index(x)` renvoie l'indice auquel la valeur `x` apparaît pour la première fois dans `C` (si `x` n'est pas présent dans `C`, alors une erreur est produite).

   Enfin, les expressions `x in C` et `x not in C` renvoient un booléen indiquant si `x` est présent (ou n'est pas présent) dans `C`.

Exercice 3.3. Utilisez les fonctions, méthodes, et l'opérateur `in` présentés ci-dessus pour vérifier que vos programmes créés dans les exercices précédents renvoient le bon résultat.

Attention. Comme toute fonction prédéfinie, ces fonctions ne servent qu'à effectuer exactement la tâche pour laquelle elles ont été créées, et rien d'autre. Il est donc important de savoir les recréer « à la main » de sorte à pouvoir les adapter à des tâches similaires, mais pas strictement identiques, par exemple :
— trouver le mot le plus court d'une liste de mots,
— compter le nombre d'éléments positifs d'une liste,
— déterminer si une tuple contient au moins un élément pair,
— etc.

Exercice 3.4. Reprenez votre code permettant de calculer la somme des éléments d'une liste `L` d'entiers, et modifiez-le pour l'adapter à chacun des cas suivants :
a) sommer seulement un élément sur deux de `L` (c'est-à-dire `L[0]` + `L[2]` + `L[4]` + ···),
b) sommer seulement les éléments pairs de `L` (en ignorant les éléments impairs),
c) calculer la somme des carrés des éléments de `L` (c'est-à-dire `L[0]`^{2}+`L[1]`^{2}+`L[2]`^{2} + ···).

3.3 Affectation de variables et muabilité

Dans cette section, nous abordons enfin la différence fondamentale entre listes et tuples : le type `list` est **muable** (*mutable* en anglais) tandis que le type `tuple` est **immuable** (*immutable* en anglais). Toutefois pour expliquer la signification de ces concepts, il faut revenir un peu plus en détail sur l'affectation de variables en Python.

3.3.1 Explications théoriques

Lorsque vous saisissez en Python l'instruction `x = 4`, voici ce qui se passe (de manière simplifiée) :
— un *objet* représentant l'entier 4 est créé dans la mémoire vive de votre ordinateur,
— l'identifiant `x` est ajouté à l'*espace des noms* (*namespace* en anglais) et pointe vers l'objet créé.

Si vous saisissez ensuite l'instruction `y = 4`, il y a alors deux scénarios possibles (voir figure 3.2) :

A) ou bien l'objet représentant l'entier 4 créé précédemment est réutilisé, et donc l'identifiant `y` ajouté au *namespace* pointe vers le même objet que `x` en mémoire,

B) ou bien un nouvel objet représentant l'entier 4 est créé en mémoire, et donc les identifiants `x` et `y` pointent vers deux objets différents (bien que représentant tous les deux une même valeur).

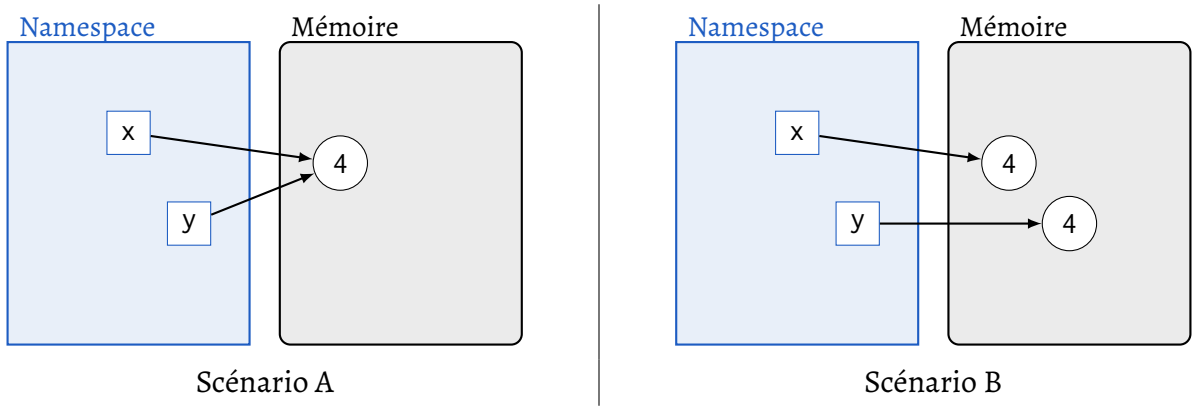

FIGURE 3.2 – Deux scénarios possibles lors de l'affectation de deux variables x et y à la valeur 4.

En théorie, ces deux scénarios sont envisageables et il est impossible de prédire lequel aura lieu ! Toutefois cela n'a aucune importance en pratique car le type entier est *immuable* : cela signifie qu'une fois qu'un objet représentant un entier est créé, il ne pourra jamais être modifié et représentera toujours la même valeur. Peu importe donc si x et y pointent vers un seul même objet ou vers deux objets différents : dans tous les cas on pourra les utiliser en étant certains qu'il pointent vers un objet qui représente le nombre 4.

Remarque. Pour information, il est possible de savoir *a posteriori* si x et y pointent vers le même objet en mémoire ou non en utilisant la fonction `id()`. Cette dernière renvoie un entier permettant d'identifier de manière unique l'objet auquel est rattaché l'identifiant passé en argument. Vous pouvez tester le programme suivant.

```
x = 4
y = 4
print(id(x))
print(id(y))
```

Si les deux dernières instructions affichent la même valeur, c'est que x et y pointent vers un seul même objet en mémoire (scénario A). Sinon, c'est que x et y pointent vers deux objets différents (scénario B).

En particulier, notez bien que si l'on exécutait ensuite l'instruction `x = 5`, cela ne modifierait pas l'objet auquel x est rattaché, mais effectuerait plutôt les opérations suivantes (voir figure 3.3) :

— création d'un objet en mémoire [3] représentant l'entier 5,

— suppression du lien entre l'identifiant x et l'ancien objet représentant 4,

— création d'un lien de l'identifiant x vers le nouvel objet représentant 5.

3. Ou, éventuellement, réutilisation d'un tel objet s'il existe déjà en mémoire (comme dans le scénario A).

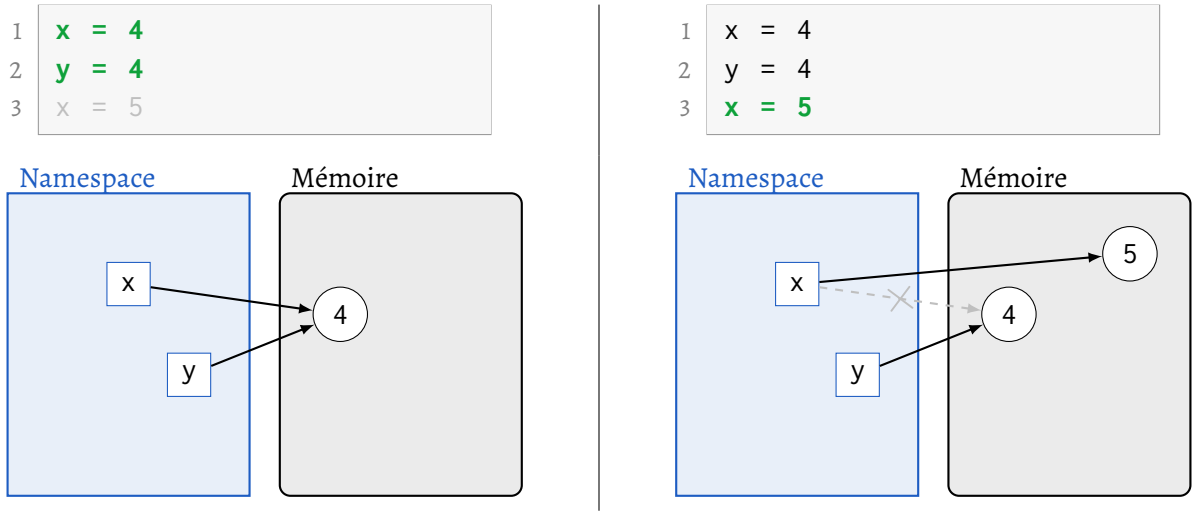

FIGURE 3.3 – Réaffectation d'une variable existante à une nouvelle valeur.

À tester. Vérifiez le comportement décrit ci-dessus à l'aide du programme suivant.

```
x = 4
print(id(x))
x = 5
print(id(x))
```

Remarque. Dans l'exemple de la figure 3.3, on est parti d'une situation dans laquelle x et y pointaient vers le même objet en mémoire (scénario A), mais ils auraient pu très bien pointer vers deux objets différents (scénario B). Dans ce cas, après l'instruction `x = 5`, l'ancien objet « 4 » (vers lequel x pointait) n'est plus relié à aucun identifiant et ne sert plus à rien. Sans que vous ne le sachiez, un processus appelé *garbage collector* (littéralement, « collecteur de déchets ») se charge tout au long du programme de supprimer de la mémoire de tels objets esseulés sans que vous n'ayez à vous en préoccuper.

Pour les listes en revanche, il est impératif de pouvoir anticiper quel scénario se produira, car les listes sont des objets *muables* : il existe des instructions qui modifient directement l'objet en mémoire. Il est donc nécessaire de savoir si x et y pointent vers un seul même objet ou vers deux objets différents. La convention est la suivante :

— les instructions ci-dessous créeront toujours deux objets **différents** en mémoire (scénario B) :

```
x = [4, 12, 9]
y = [4, 12, 9]
```

— les instructions ci-dessous feront toujours pointer y vers **le même objet** que x (scénario A) :

```
x = [4, 12, 9]
y = x
```

Ainsi, quand on exécutera ensuite une instruction telle que `x[1] = 3`, suivant comment la variable y a été définie on pourra savoir si la valeur de `y[1]` a été modifiée ou non.

À tester. Comparez (via l'explorateur de variables) les effets des deux programmes suivants, et essayez de les expliquer d'un point de vue théorique.

```
x = [4, 12, 9]
y = x
x[1] = 3
```

```
x = [4, 12, 9]
y = [4, 12, 9]
x[1] = 3
```

3.3.2 Conséquences importantes

Voici quelques conséquences concrètes du fait que le type `list` est muable, qu'il faudra idéalement garder à l'esprit lorsque à chaque fois que vous utiliserez les listes.

— **Affectation sur un élément d'indice donné.**
Si `L` est une liste, il est possible de faire des affectations de la forme `L[i] = ...` pour modifier un élément de la liste (comme vu plus haut), alors que ce n'est pas possible pour un tuple.

À tester. Essayez le code suivant.

```
T = (4, 12, 9)
T[1] = 3
```

— **Différence entre `x = x + y` et `x += y`.**
Rappelons que l'opération `x + y` entre deux listes `x` et `y` renvoie une *nouvelle* liste. En particulier les objets de type liste vers lequels pointent `x` et `y` ne sont pas modifiés. Une instruction du type `x = x + y` est alors interprétée de la manière suivante (voir figure 3.4) :

— d'abord, l'expression `x + y` est évaluée : le résultat est un nouvel objet de type liste créé en mémoire, mais les objets auxquels étaient rattachés `x` et `y` n'ont **pas** été modifiés lors du calcul,

— ensuite, le lien entre `x` et la liste vers laquelle pointait `x` est supprimé, afin de maintenant faire pointer `x` vers le nouvel objet créé en mémoire.

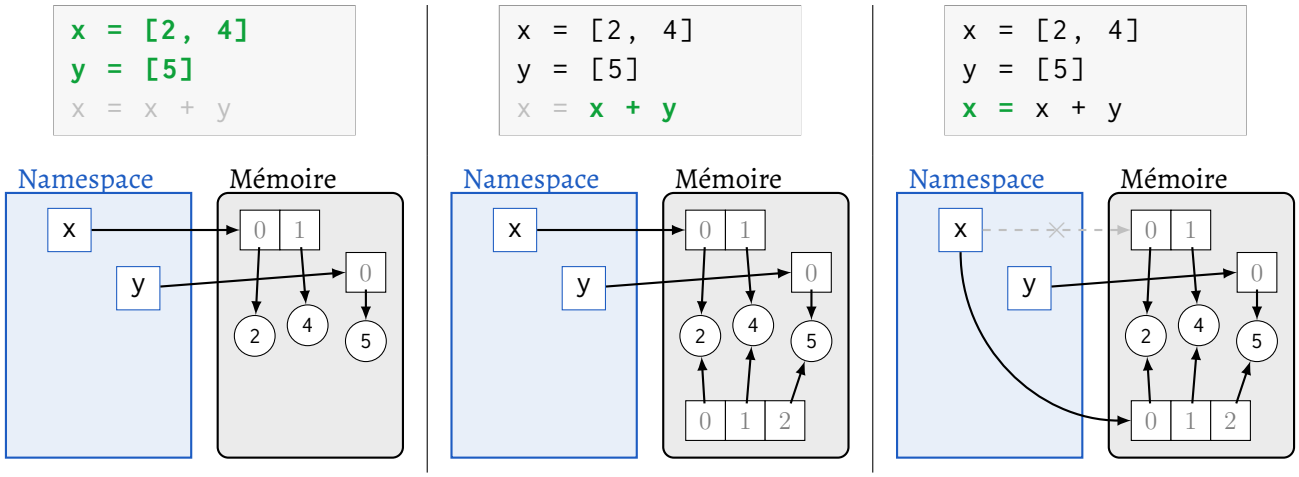

FIGURE 3.4 – Illustration des changements en mémoire provoqués par l'instruction `x = x + y` (pour des listes).

Toutefois, il existe aussi un opérateur += qui se comportera différemment suivant le type des objets manipulés. Plus précisément :

— si x est de type immuable, alors l'instruction `x += y` est strictement identique à l'instruction `x = x + y`,

— si x est de type muable, alors l'objet vers lequel pointe x est directement modifié en mémoire (voir figure 3.5). Il n'y a pas de nouvel objet créé comme lors de l'interprétation de `x = x + y`.

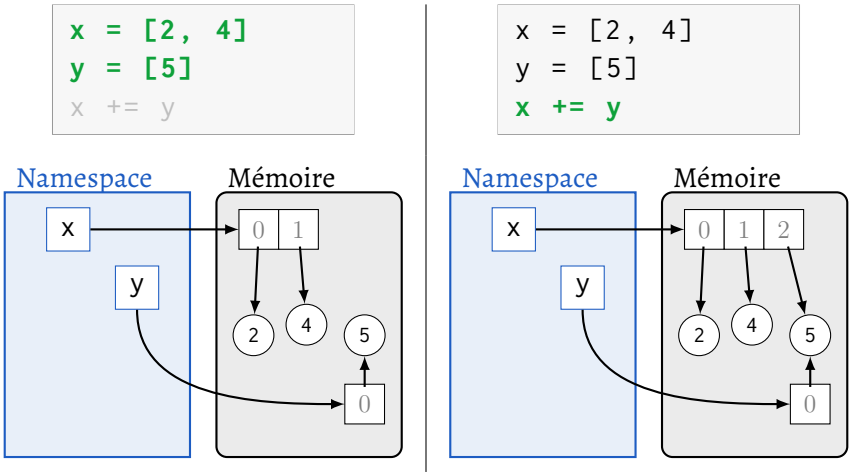

FIGURE 3.5 – Illustration des changements en mémoire provoqués par l'instruction `x += y` (pour des listes). Ces changements ne sont possibles que parce que le type `list` est muable. Pour un type immuable, on retrouverait le même déroulement que dans la figure 3.4.

À tester. Comparez et essayez d'expliquer les résultats des programmes suivants.

```
1 x = [4, 12, 9]
2 y = [2, 3]
3 z = x
4 x = x + y
5 print(x)
6 print(z)
```

```
1 x = [4, 12, 9]
2 y = [2, 3]
3 z = x
4 x += y
5 print(x)
6 print(z)
```

```
1 x = (4, 12, 9)
2 y = (2, 3)
3 z = x
4 x += y
5 print(x)
6 print(z)
```

Remarque. De manière générale, il existe de même les opérateurs `-=`, `*=`, `/=`, `//=`, `%=` et `**=` qui peuvent être utilisés dès lors que les opérations `-`, `*`, `/`, `//`, `%` et `**` respectivement on du sens. Ces opérateurs présentent les mêmes fonctionnement et subtilités que l'opérateur += introduit ci-dessus.

— **Attention lors de la manipulation de listes imbriquées.**

De manière plus générale, des comportements en apparence inattendus peuvent apparaître lors de la manipulation de listes contenant elles-mêmes des listes (ou d'autres types muables qui seront vus plus tard dans ce cours). En réalité, ils s'expliquent tous à l'aide des notions théoriques introduites dans cette section à condition d'être bien à l'aise avec ces concepts. Nous n'attendons pas pour le moment une telle maîtrise de ces notions, mais les plus curieux d'entre vous peuvent déjà se confronter à l'exemple ci-dessous.

À tester. *(Difficile)* Comparez et essayez d'expliquer les résultats des programmes suivants.

```
L = [[1, 2], [1, 2], [1, 2]]
print(L)
L[1][0] = 99
print(L)
```

```
L = [[1, 2]] * 3
print(L)
L[1][0] = 99
print(L)
```

```
L = [[1, 2]] * 3
print(L)
L[1] = [99, 2]
print(L)
```

3.4 Méthodes spécifiques aux listes

Les méthodes décrites dans la table 3.1 permettent de modifier directement une liste (elles n'auraient donc pas de sens sur un tuple [4] qui par définition n'est pas modifiable).

Syntaxe	Description
`L.append(x)`	Ajoute l'élément `x` à la fin de la liste `L`.
`L.remove(x)`	Retire la première apparition de la valeur `x` dans la liste `L`. *Déclenche une erreur si la valeur `x` n'apparaît nulle part dans la liste.*
`L.pop(i)`	Retire l'élément d'indice `i` de la liste `L` et renvoie cet élément. *Si l'indice `i` est omis, c'est le dernier élément qui est retiré.*
`L.insert(i, x)`	Insère la valeur `x` en position `i` dans la liste `L`. *Les valeurs d'indice $\geqslant$ `i` sont alors toutes décalées d'un cran vers la droite.*
`L.reverse()`	Inverse l'ordre des éléments de `L`.
`L.sort()`	Trie les éléments de la liste `L`.

TABLE 3.1 – Méthodes propres aux listes.

À tester. Exécutez les instructions suivantes **une par une** et observez à chaque fois les modifications dans l'explorateur de variables.

```
1 L = [12, 34, 98, 54, 34]
2 L.append(26)
3 L.remove(34)
4 a = L.pop(2)
```

```
5 b = L.pop()
6 L.insert(1, 66)
7 L.reverse()
8 L.sort()
```

La méthode `append()` est particulièrement utilisée car une façon courante de construire une liste répondant à un problème donné est de partir d'une liste vide puis la compléter au fur et à mesure jusqu'à arriver au résultat voulu. Par exemple, l'algorithme ci-dessous construit une liste contenant tous les multiples de 7 strictement inférieurs à 100.

```
numbers = []
n = 0
while n < 100:
    numbers.append(n)
    n = n + 7
```

4. Ni sur une chaîne de caractères, car le type `str` est lui-aussi immuable.

Exercice 3.5. Construire une liste contenant toutes les puissances de 2 (c'est-à-dire $2, 4, 8, 16, \ldots$) inférieures ou égales à 1000.

Exercice 3.6. On suppose déjà définie une liste `L` contenant des nombres entiers. Écrire un programme Python qui crée deux listes `L_pair` et `L_impair` contenant respectivement les valeurs paires et les valeurs impaires contenues dans `L`.

Exercice 3.7. On suppose déjà définie une liste `L`. Écrire un programme Python qui crée une liste `L_op` contenant les même valeurs que `L` mais dans l'ordre inverse. Il ne faudra pas modifier la liste `L` d'origine.

La méthode `reverse()` *ne sera donc pas votre alliée, mais vous êtes capables de vous débrouiller sans elle !*

Exercices supplémentaires

Exercice 3.8. On suppose déjà définie une liste `L` contenant des nombres entiers.

1. Écrire un programme qui remplace chaque élément de `L` par sa valeur absolue.
   Par exemple, si `L` *est la liste* `[12, -3, -50, 2, -9, 0, -7, -11, 5]`*, alors votre programme devra modifier* `L` *de sorte à ce qu'elle devienne la liste* `[12, 3, 50, 2, 9, 0, 7, 11, 5]`.
2. Écrire un programme qui crée une liste `pos` de même taille que `L`, et telle que la i-ème composante de `pos` est un booléen indiquant le signe de la i-ième composante de `L` (`True` pour positif, `False` pour strictement négatif).
   Par exemple, si `L` *est la liste* `[12, -3, -50, 2, -9, 0, -7, -11, 5]`*, alors votre programme devra créer la liste* `pos = [True, False, False, True, False, True, False, False, True]`.
3. Écrire un programme qui crée une variable `all_positive` contenant la valeur `True` si `L` ne contient que des valeurs strictement positives, et `False` sinon.

Exercice 3.9. On suppose déjà définie une liste `L` contenant des tuples, tous de la forme `(x, y)` (c'est-à-dire contenant deux éléments). Écrire un programme qui crée deux listes `L1` et `L2` indépendantes, l'une contenant seulement la première composante de chacun des tuples de `L`, et l'autre contenant la deuxième composante.

Par exemple, si `L` *est la liste* `[(12, 8), (33, 54), (87, 21), (50, 11)]`*, alors :*

— `L1` *devra être la liste* `[12, 33, 87, 50]`,

— `L2` *devra être la liste* `[8, 54, 21, 11]`.

Exercice 3.10. On suppose déjà définies deux listes `A` et `B` (non nécessairement de même taille). Écrire un programme Python qui créée une liste `P` contenant tous les tuples de la forme `(a, b)` avec `a` un élément de `A` et `b` un élément de `B`.

Vous pouvez lister les éléments de `P` *dans l'ordre que vous voulez pourvu qu'ils soient tous présents.*

Exercice 3.11. On suppose déjà définie une variable `N` contenant un entier. Écrire un programme qui créé une liste `primes` contenant (dans l'ordre croissant) tous les nombres premiers inférieurs ou égaux à N.

À retenir

— Connaître les syntaxes `C[i]` (avec `i` entier positif ou négatif), `C[a:b]` et `C[a:b:h]` lorsque `C` est un tuple, une liste ou une chaîne de caractères.

— Connaître et comprendre le vocabulaire : *indexable/subscriptable*, *iterable*, *(im)mutable*.

— Connaître les effets des opérateurs `+`, `*`, `+=`, `*=` sur les listes et les tuples.

— Savoir parcourir une liste ou un tuple élément par élément à l'aide d'une boucle.

— Savoir mettre en place un algorithme pour remplir une liste pas à pas (en fonction d'un critère donné).

— Fonctions à connaître : `len()`, `min()`, `max()`, `sum()`.

— Méthodes des types `list` et `tuple` à connaître : `count()`, `index()`.

— Méthodes spécifiques au type `list` à connaître : `append()`, `remove()`, `pop()`, `insert()`, `reverse()`, `sort()`.

Type	*Subscriptable*	*Iterable*	*Mutable*
`int`	✗	✗	✗
`float`	✗	✗	✗
`bool`	✗	✗	✗
`str`	✓	✓	✗
`list`	✓	✓	✓
`tuple`	✓	✓	✗