

Collections : généralités et listes

Polytech Marseille, GII, 3A

Séverine Dubuisson, Simon Vilmin
severine.dubuisson@univ-amu.fr,
simon.vilmin@univ-amu.fr

2023 - 2024



Plan

Généralités

- Ordonné vs. non-ordonné

- Mutable vs. immuable

- Quels éléments stocker

- Résumé

Listes : les bases

- Définition et syntaxe

- Parcours, accès, recherche

- Modification, ajout, suppression

- Opérations entre listes

- Résumé

Listes : aspects plus complexes

- Définition en compréhension

- Copies de listes

- Modifications et `for`

- Résumé

Généralités

Généralités

Ordonné vs. non-ordonné

Mutable vs. immuable

Quels éléments stocker

Résumé

Listes : les bases

Listes : aspects plus complexes

i Remarque : cette partie brosse un portrait *global* des *différentes collections* implémentées en Python. L'objectif *n'est pas de toutes les comprendre en profondeur* mais de *comprendre qu'elles ont chacune leur spécificités*.

Exercice


⚙️ **Exercice** : écrire un programme qui saisit 4 nombres et les affiche dans l'ordre inverse de leur saisie


```
# ==== Exemple
Entrez nombre 1 : 2
Entrez nombre 2 : 4
Entrez nombre 3 : 6
Entrez nombre 4 : 8
8 6 4 2
```


! **Attention** : interdit d'utiliser la concaténation de chaînes de caractères

⚙️ **Exercice** : en fait je me suis trompé, je voulais dire **8** nombres. Désolé.

Collections

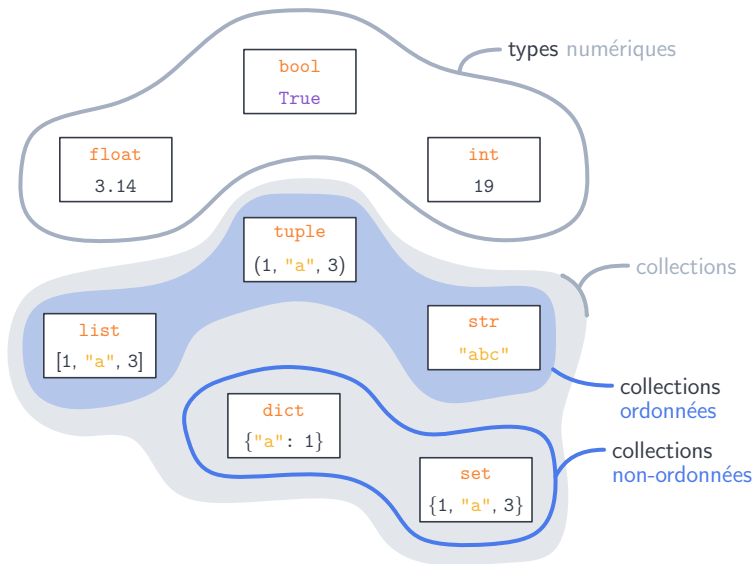
 **Problème** : on va pas pouvoir créer des nouvelles variables à chaque fois ...

 **Idée** : il faudrait pouvoir mettre ces valeurs dans une seule *collection d'objets*

 **Question** : ça tombe bien, Python propose *plusieurs types de collections*, mais ... *laquelle utiliser* du coup ?

```
liste = [2, 4, 6, 8] # type list ?
uplet = (2, 4, 6, 8) # type tuple ?
ensemble = {2, 4, 6, 8} # type set ?
dico = {"v1": 2, "v2": 4, "v3": 6, "v4": 8} # type dict ?
```

Galaxie (partielle) des types Python



Collections ordonnées, ou séquences

 **Définition** : une *collection ordonnée* ou *séquence* est une collection d'éléments indexés par un entier non-négatif (partant de 0).

i **Remarque** : dans une collection *ordonnée*

- l'*indexation* associe des *indices* aux éléments

```
collec[i] # element d'indice i de collec
```

- le *slicing* permet de sélectionner une partie des éléments

```
collec[i:j] # elements d'indices i a j - 1 de collec
```

- *collection ordonnée* : `str`, `tuple`, `list`
- *collection non-ordonnée* : `set`, `dict`

Ordonné vs. non-ordonné : égalité

Collections ordonnées :

```
In []: [1, 2, 3] == [3, 2, 1] # list
```

```
Out []: False
```

```
In []: (1, 2, 3) == (2, 1, 3) # tuple
```

```
Out []: False
```

```
In []: "abc" == "bca" # str
```

```
Out []: False
```

Collections non-ordonnées :

```
In []: {1, 2, 3} == {2, 1, 3} # set
```

```
Out []: True
```

```
In []: {"a": 1, "b": 2} == {"b": 2, "a": 1} # dict
```

```
Out []: True
```

i **Remarque** : en bref

- collections *ordonnées* égales : *mêmes éléments, même ordre*
- collections *non-ordonnées* égales : *mêmes éléments*

Ordonnées vs. non-ordonnées : indexation

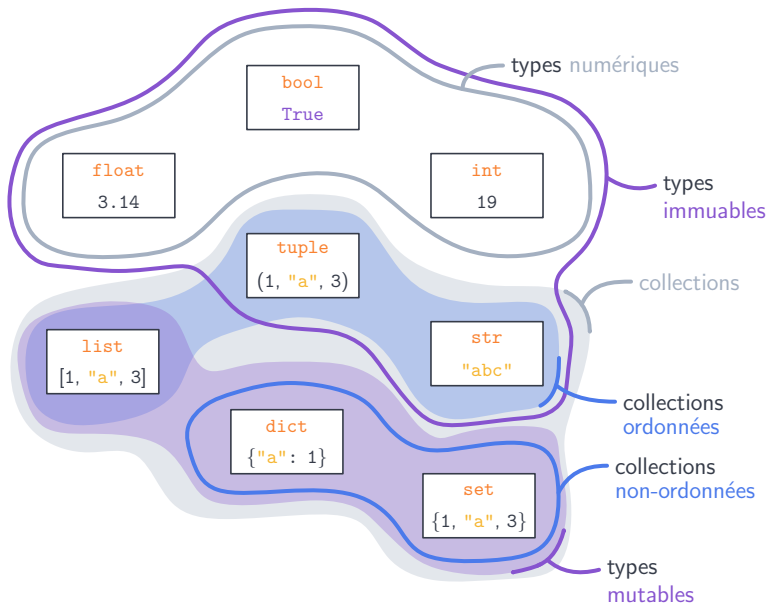
Collections ordonnées

```
In []: liste = ["a", "b", "c", "d", "e"] # list
In []: liste[0]
Out []: 'a'
In []: uplet = (12, 17, 284, 37, "bonjour")
In []: uplet[2:4]
Out []: 284, 37
```

Collections non-ordonnées

```
In []: ensemble = {"a", "b", "c", "d", "e"} # set
In []: ensemble[0]
Out []: TypeError: 'set' object is not subscriptable
In []: dico = {"a": 7, "b": 3.14}
In []: dico[0]
KeyError: 0
```

Galaxie (partielle) des types Python : le retour



Mutable vs. immuable : définitions

Définition :

- un objet est *(de type) immuable* si on ne peut pas changer sa valeur (« l'écraser ») directement à son emplacement mémoire
- un objet est *(de type) mutable* si, au contraire, on peut changer sa valeur directement à son emplacement mémoire.

Astuce : en clair,

- *mutable* : on peut changer un objet (valeur, variable, collection, ...) sans en créer un nouveau
- *immuable* : on ne peut pas

- *mutable* : `list`, `dict`, `set`
- *immuable* : `str`, `tuple`, `int`, `float`, `bool`

Mutable vs. immutable : exemple

Objets mutables :

```
In []: liste = [1, 2, 3]
```

```
In []: liste[0] = 7
```

```
In []: liste
```

```
Out []: [7, 2, 3]
```

```
In []: ens = {1, 2, 3}
```

```
In []: ens.add(4)
```

```
In []: ens
```

```
Out []: {1, 2, 3, 4}
```

Objets immuables :

```
In []: chaine = "bonjour"
```

```
In []: chaine[0] = "B"
```

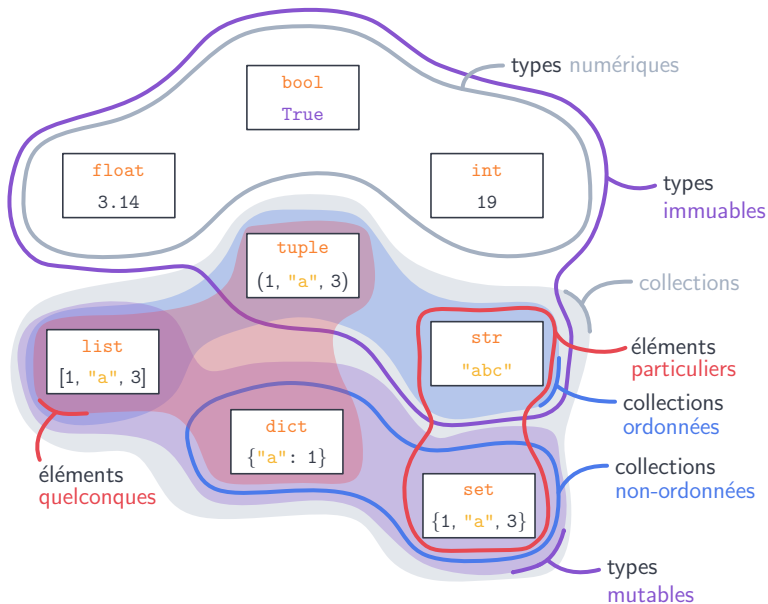
```
Out []: TypeError: 'str' object does not support item assignment
```

```
In []: uplet = (1, 2, 3)
```

```
In []: uplet[0] = 7
```

```
Out []: TypeError: 'tuple' object does not support item assignment
```

Galaxie (partielle) des types Python : le retour de la mort



Collectionner quoi ?

? **Question** : peut-on stocker n'*importe quoi* dans une collection ?

✓ **Réponse** : ça dépend ...

```
In []: l = [[1, 2], "tulipe", 3.14, {"a", "b"}] # list ok
In []: t = ([1, 2], "tulipe", 3.14, {"a", "b"}) # tuple ok
In []: s = {[1, 2], "tulipe", 3.14, {"a", "b"}}
Out []: TypeError: unhashable type: 'list' # ah ! set pas ok
```

i **Remarque** :

- **list** et **tuple** peuvent contenir n'importe quoi
- **set** et **str** non
- quid des **dict** ?

Notion de hash

i Remarque : un ensemble (**set**) ne peut contenir que des éléments dits *hashables*.

📖 Définition : une *fonction de hash* associe un nombre à un objet à partir de sa valeur. Un objet est *hashable* s'il a une fonction de hash.



- en Python, la méthode de hash est `hash`
- le hash est calculé à partir de la valeur d'un objet
- il ne change pas au cours de la vie de l'objet

Types prédéfinis

- *hashable* : `int`, `float`, `bool`, `str`, `tuple`
- *non-hashable* : `dict`, `list`, `tuple`

Le cas de `tuple` dépend : *hashable* s'il ne contient que des éléments hashable, *non-hashable* sinon

```
In []: t1 = (1, 2, "pouet")
```

```
In []: hash(t1)
```

```
Out []: 5038968233157651535
```

```
In []: t2 = (1, [2, 3], "bonjour")
```

```
In []: hash(t2)
```

```
Out []: TypeError: unhashable type: 'list'
```



Astuce : à part `tuple` les types *hashables* sont les types *immuables*

Cas des dictionnaires

 **Définition** : un *dictionnaire* (**dict**) est une collection de paires

cle: valeur

la *clé* doit être *hashable*, la *valeur* peut être *quelconque*

```
In []: d1 = {"a": 1, (2, 3): [1, 2], True: "sapin"}
```


```
In []: d2 = {[1, 2]: "bonjour"}
```

```
Out []: TypeError: unhashable type: 'list'
```



Astuce : intuitivement, un **set** peut être vu comme un **dict** mais sans valeurs (que des clés)

Bon, du coup qui stocke quoi ?

 **Astuce** : bon, en résumé

- le *hash* est la *représentation sous forme d'entier* (avec collisions parfois ...) d'un objet.
- le fait qu'un *objet soit hashable* est important pour pouvoir le stocker dans *certaines collections*

Stocker des valeurs dans une collection :

- **str** : caractères uniquement
- **list** : tout type d'éléments
- **tuple** : tout type d'éléments, mais parfois au prix du hash
- **dict** : clés hashables, tout type d'éléments,
- **set** : éléments hashables uniquement

Résumé

Rappel :

- les *collections* permettent de stocker plusieurs éléments
- Python propose plusieurs collections aux *comportements différents* !

collection	ordonnée	mutable	éléments stockés
<code>str</code>	✓	✗	<i>caractères</i>
<code>list</code>	✓	✓	<i>quelconques</i>
<code>tuple</code>	✓	✗	<i>quelconques</i>
<code>dict</code>	✗	✓	<i>quelconques</i> (mais clés <i>hashables</i> !)
<code>set</code>	✗	✓	<i>hashables</i>

Listes : les bases

Généralités

Listes : les bases

Définition et syntaxe


Parcours, accès, recherche

Modification, ajout, suppression

Opérations entre listes


Résumé

Listes : aspects plus complexes

 **Remarque** : dans cette partie, on fait *seulement un tour d'horizon* des opérations de base sur les listes. Il faut aller dans la [doc](#) pour plus de détails !

Définition et syntaxe

 **Définition** : une *liste* (type `list`) est une *collection ordonnée mutable* d'objets de tout types.

 **Syntaxe** : une liste est définie avec des crochets [], on y sépare les éléments par des virgules ,

```
liste = [element1, element2, ...]
```

```
L = [ ] # liste vide
```

```
L = [1, 2, ["c", "du", ["flan"]]] # listes imbriquées
```

```
matrice = [[1, 2, 0], [-7, 4, 10], [5, 2, 1]]
```

Accès, parcours basique

les listes sont des *séquences* : ses éléments sont indexés par des entiers.


```
In []: L = ["a", "b", "c", "d", "e"]
In []: L[2] # element d'indice 2
Out []: 'c'
In []: L[1:3] # slicing, liste des elements d'indices 1 a 2
Out []: ['b', 'c']
```

En utilisant la fonction `len`, on peut parcourir tous les éléments d'une liste

```
In []: L = ["a", "b", "c", "d", "e"]
In []: for i in range(0, len(L)):
    print(L[i], end=" ")
Out []: a b c d e
```

! **Attention** : parcourir une liste par les indices n'est pas toujours très pythonesque!!!

Itérables

 **Idée** : il vaut mieux utiliser le fait que les listes sont *itérables*

 **Remarque** : les listes, comme *toutes les collections*, sont *itérables* : on peut parcourir ses éléments directement avec un `for`

```
In []: L = ["a", "b", "c", "d", "e"]
```

```
In []: for car in L:  
    print(car, end=" ")
```


```
Out []: a b c d e
```


```
In []: phrase = ["messire,", "les", "anglais",  
    "nous", "envahissent", "!"]
```

```
In []: for mot in phrase:  
    print(mot, end=" ")
```

```
Out []: messire, les anglais nous envahissent !"
```

Énumération

 **Idee** : on a parcouru une liste par ses *indices* (avec `range`) et par ses valeurs (avec `in`) ... mais on peut aussi parcourir les *deux à la fois*!

 **Remarque** : la fonction `enumerate` permet de parcourir *à la fois les indices* d'une liste *et ses éléments*

```
In []: L = ["a", "b", "c", "d", "e"]
In []: for (indice, valeur) in enumerate(L):
    print(F"valeur a l'indice {indice} : {valeur}")
Out[]: valeur a l'indice 0 : a
valeur a l'indice 1 : b
valeur a l'indice 2 : c
valeur a l'indice 3 : d
valeur a l'indice 4 : e
```


Qui fait quoi ?

```
L = ["tu", "es", "un", "hibou"]
for i in range(len(L)):
    print(i, end=" ")
```

```
# ==== resultat
0 1 2 3
```

```
L = ["tu", "es", "un", "hibou"]
for i in range(len(L)):
    print(L[i], end=" ")
```

```
# ==== resultat
tu es un hibou
```

```
L = ["tu", "es", "un", "hibou"]
for i, j in enumerate(L):
    print(F"({i}, {j})", end=" ")
```

```
# ==== resultat
(0, tu) (1, es) (2, un) (3, hibou)
```

```
L = ["tu", "es", "un", "hibou"]
for i in L:
    print(i, end=" ")
```

```
# ==== resultat
tu es un hibou
```

```
L = ["tu", "es", "un", "hibou"]
for i in L:
    print(L[i], end=" ")
```

```
# ==== resultat
TypeError: list indices must
be integers or slices, not str
```

Recherche

On peut tester la présence d'un élément dans une liste grâce à `in`

```
In []: 4 in [2, 8, 6, 7]
Out []: False
In []: 9 in [1, 3, 5, 7, 9]
Out []: True
```

On peut également :

- obtenir la position de la première occurrence d'un élément avec `index`
- compter le nombre d'occurrences d'un élément avec `count`

```
In []: L = ["a", "b", "a", "b", "a"]
In []: L.index("a")
Out []: 0
In []: L.count("b")
Out []: 2
```

Exercices

⚙️ **Exercice** : écrire un programme qui parcourt une liste de lettres, et qui affiche les positions des voyelles dans la liste.

```
# ==== Exemple
pour L = ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
0 4 8
```

⚙️ **Exercice** : écrire un programme qui parcourt une liste de nombres et qui vérifie que cette liste est triée par ordre croissant

```
# ==== Exemple
pour L = [1, 2, 3, 0]
pas trieé !
```

⚙️ **Exercice** : écrire un programme qui parcourt une liste de mots et qui affiche les voyelles de chaque mots.

```
# ==== Exemple
pour phrase = ["tu", "m'as", "roule", "dans", "la", "frangipane"]
u a oue a a aiae
```

Modification

 **Idée** : puisque les listes sont *mutables*, on peut *modifier*, *ajouter* ou *supprimer* des éléments

modifications : on utilise l'*indexation* et le *slicing*

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

```
In []: L[0] = "a" # modif du premier element
```

```
In []: L[2:4] = ["c", "d"] # modif des items d'indices 2 et 3
```

```
In []: L
```

```
Out []: ["a", 2, "c", "d", 5]
```

Ajout

- `append` ajoute un élément à la *fin de la liste*
- `insert` insère un élément ou une liste d'éléments *avant une position donnée*
- `extend` ajoute une *liste* à la *fin de la liste*

```
In []: L = ["a", "b", "c", "d", "e"]
```

```
In []: L.append("f")
```

```
In []: L
```

```
Out []: ["a", "b", "c", "d", "e", "f"]
```

```
In []: L.insert(0, "alphabet")
```

```
In []: L
```

```
Out []: ["alphabet", "a", "b", "c", "d", "e", "f"]
```

```
In []: L.extend(["g", "h", "i"])
```

```
In []: L
```

```
Out []: ["alphabet", "a", "b", "c", "d", "e", "f", "g", "h", "i"]
```

Suppression

- `pop` *supprime* et *renvoie* un élément à une *position donnée*
- `remove` *supprime* la *première occurrence* d'une *valeur*
- `del` permet de *supprimer* des éléments ou des intervalles *via leur position*

```
In []: L = ["a", "b", "a", "c", "d", "e", "f"]
```

```
In []: L.pop(1)
```

```
In []: L
```

```
Out []: ["a", "a", "c", "d", "e", "f"]
```

```
In []: L.remove("a")
```

```
In []: L
```

```
Out []: ["a", "c", "d", "e", "f"]
```

```
In []: del L[2:4]
```

```
In []: L
```

```
Out []: ["a", "c", "f"]
```

Exercice

⚙️ **Exercice** : que vaut L à la fin de ce programme

```
L = [0]
for i in range(5):
    L.append(i)
    L.insert(i, i)
L.pop(0)
L.remove(0)
L.extend(["a", "b", "c"])
L.pop(1)
del L[0]
del L[2:4]
L.insert(3, [0, 0, 0])
```

Quelques opérations entre listes

i Remarque : comme pour `str`, on peut *multiplier* une liste par un entier et *ajouter des listes* (les *concaténer*)

```
L1 = ["pon"]
L2 = ["pata"]
L3 = (2 * L1) + L2 + L1 + (3 * L2) + L1
for mot in L3:
    print(mot, end=" ")
pon pon pata pon pata pata pata pon
```

! Attention : au contraire des opérations de modifications comme `append`, `pop`, ..., les opérations `+` et `*` renvoient *une nouvelle liste* !

Exercices

⚙️ **Exercice** : écrire un programme qui demande à l'utilisateur.ice de saisir une série de n prénoms, et qui les affiche dans l'ordre alphabétique :

```
# ==== Exemple
nombre de prenom : 3
1 : Athenais
2 : Clementine
3 : Azrael
Athenais Azrael Clementine
```

⚙️ **Exercice** : écrire un programme qui prend en entrée une matrice carrée composée de lettres, et qui vérifie que toutes les lignes, les colonnes et les diagonales sont des palindromes

```
# ==== Exemple
pour m =
c b c
a d a
c b a
non !
```

Résumé

Rappel :

- une liste est une *collection ordonnée, mutable* d'objets *quelconques*
- on les utilise avec des *crochets*, []
- on peut *parcourir* les éléments d'une liste directement via `for`

 **Attention :** `append`, `remove`, ... *modifient la liste* alors que `+` et `*` *non*

```
L = [0, [ ], "biscuit", [0, [ ], "biscotte"]  
for element in L:  
    print(element)
```

Listes : aspects plus complexes

Généralités

Listes : les bases

Listes : aspects plus complexes


- Définition en compréhension

- Copies de listes

- Modifications et `for`

- Résumé

Définition d'ensembles

 **Définition** : en mathématiques, il existe deux grandes manières de décrire les éléments d'un ensemble E

- en *extension* : on liste explicitement tous les éléments de E
- en *compréhension* : on donne une propriété (un prédicat) qui caractérise les éléments de E

Exemple

L'ensemble des nombres entiers impairs entre 1 et 30

- en *extension*, $\{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29\}$
- en *compréhension*, $\{n : n, k \in \mathbb{N}, 1 \leq n \leq 30, n = 2k + 1\}$

L'ensemble 2^U des parties de l'ensemble $U = \{1, 2, 3\}$:

- en *extension*, $2^U = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
- en *compréhension*, $2^U = \{X : X \subseteq U\}$

! **Attention** : on parle d'ensembles au sens maths, pas des `set` Python !


Définition de listes en compréhension

 **Syntaxe** : Python permet de définir des listes en *compréhension*, à l'image des ensembles en mathématiques, avec `for` (voir [doc](#))

```
liste = [expression for item in iterable if condition]
```

Remarque :

- `expression` est une expression qui *peut dépendre* de `item`
- un *itérable* est un objet que l'on peut parcourir (avec un `for`) : pour nous, `range` ou les collections (`str`, `list`, `set`, ...)
- `if condition` permet de filtrer les éléments à sélectionner dans l'itérable

 **Astuce** : l'écriture en compréhension est souvent plus *rapide et efficace* qu'un bloc `for`

Exemple

```
In []: L1 = [x for x in range(5)]
```

```
In []: L1
```

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

```
In []: L2 = [x**2 for x in range(5)]
```

```
In []: L2
```

```
Out[]: [0, 1, 4, 9, 16]
```

```
In []: L3 = [2 * c for c in "arg"]
```

```
In []: L3
```

```
Out[]: ["aa", "rr", "gg"]
```

```
In []: L4 = [x for x in L2 if x % 2 == 0] # condition de filtrage
```

```
In []: L4
```

```
Out[]: [0, 4, 16]
```

```
In []: L5 = [2 * [x] for x in L3]
```

```
In []: L5
```

```
Out[]: [["aa", "aa"], ["rr", "rr"], ["gg", "gg"]]
```

Exercice

⚙️ **Exercice** : utiliser la *compréhension* pour chacun des cas suivants :

1. étant donnée une liste L de lettres, construire la liste des voyelles de L
2. étant donnée une liste L d'entiers, construire la liste Lf des paires $(x, f(x))$ où $f(x) = 3x^2 - x + 1$, pour tout x de L
3. étant données deux listes L1, L2, trouver les éléments de L1 qui ne sont pas dans L2
4. étant donnée une liste L, construire la liste des éléments de L de type **str**
5. étant donnée une liste phrase de mots, construire la liste decoupage des listes [voyelles(mot), consonnes(mot)] pour chaque mot de phrase

```
# ==== Exemple
phrase = ["le", "cote", "obscuuuuuur"]
decoupage = [[['l'], ['e']],
             [['o', 'e'], ['c', 't']],
             [['o', 'u', 'u', 'u', 'u', 'u', 'u'], ['b', 's', 'c', 'r']]]
```


Exercice

⚙️ **Exercice** : à la fin du programme suivant, que valent L1 et L2 ?

```
L1 = [c for c in "abc"]
```

```
L2 = L1
```

```
L1.append("L1")
```

```
L2.append("L2")
```

Exercice

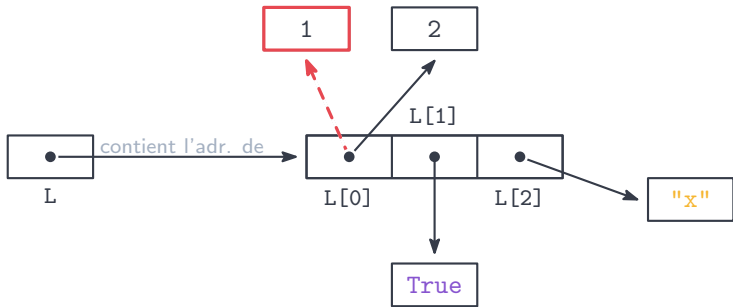
⚙️ Exercice : à la fin du programme suivant, que vaut L1 ?

```
def bidouille(L):  
    L.append("bidouille")  
  
def magouille(L):  
    Lc = L  
    Lc.append("magouille")  
  
def machination(L):  
    L = []  
    L.append(0)  
  
# ==== Programme principal  
L1 = [1, 2, 3]  
bidouille(L1)  
magouille(L1)  
machination(L1)
```

Représentation des listes en mémoire

! Attention : en fait, les listes sont proches des *tableaux contigus* et pas des *listes chaînées* ...

```
L = [1, True, "x"]  
L[0] = 2
```



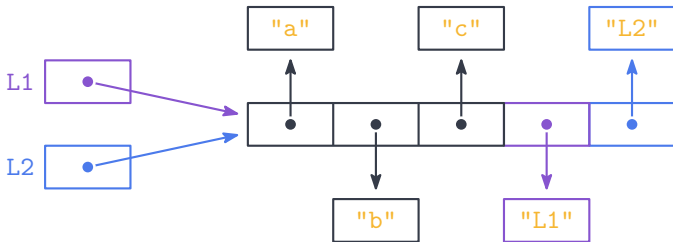
Assignement de listes

```
L1 = [c for c in "abc"]
```

```
L2 = L1
```

```
L1.append("L1")
```

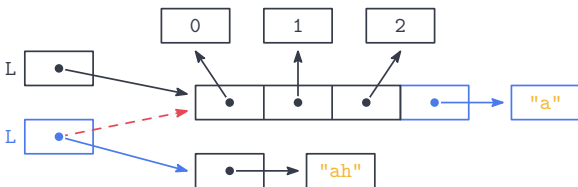
```
L2.append("L2")
```



⚠ Attention : quand on fait `L1 = L2` on copie juste l'*adresse* de L1 et *pas les éléments de la liste* !

Listes et portée des variables

```
def bidouille(L):  
    L.append("a")  
    L = ["ah"]  
  
# ==== Programme  
# principal  
L = [0, 1, 2]  
bidouille(L)
```



! Attention :

- quand on passe une liste en arguments, on passe une *copie de son adresse*, modifier la liste modifie donc la liste principale
- par contre, *ré-affecter* la liste crée une *variable locale*

Copie de listes

Pour éviter de modifier par inadvertance une liste, il faut *copier tous ses éléments* dans une nouvelle liste :

- si la liste *ne contient pas* de sous-collections (mutables)

```
L_copie = L[:] # slicing
```

- si la liste *contient* des sous-collections (mutables)

```
from copy import deepcopy  
L_copie = deepcopy(L) # fonction de copie profonde recursive
```

? Question : pourquoi aller chercher `deepcopy` ?

✓ Réponse : si une liste contient une sous-liste, le slicing va juste copier l'adresse de la sous-liste, et on aura les mêmes problèmes qu'avant !

Question

? **Question** : que se passe-t-il quand on *ajoute/supprime des éléments d'une liste pendant qu'on la parcourt* ?

```
In []: l = [c for c in "abcdef"]
```

```
In []: for c in l:
```

```
    l.remove(c)
```

```
In []: l
```

```
Out []: ["b", "d", "f"]
```

```
In []: l = [c for c in "abcdef"]
```

```
In []: for c in l:
```

```
    l.append(c)
```

```
In []: l
```

```
# boucle infinie !
```

```
In []: l = [c for c in "abcdef"]
```

```
In []: for c in l:
```

```
    l.pop(0)
```

```
In []: l
```

```
Out []: ["d", "e", "f"]
```



! Attention : *éviter d'ajouter/retirer* les items d'une liste qu'on *parcourt*

Exercices

⚙️ **Exercice** : écrire un programme qui, en une ligne, calcule le nombre de multiples de 7 entre 4 et 100000

⚙️ **Exercice** : écrire un programme qui renverse une liste ne contenant que des types numériques, sans utiliser la méthode `reverse`

```
# ==== Exemple
pour L = [1, 2, 3, 4]
[4, 3, 2, 1]
```

⚙️ **Exercice** : écrire un programme qui renverse une liste contenant des types numériques et des listes. Il peut y avoir un nombre arbitraire d'imbrications et toutes les sous-listes doivent être inversées aussi.

```
# ==== Exemple
pour L = [[1, 2], [3, 4], 5, [1.0, [0, -1, -2]]]
L = [[-2, -1, 0], 1.0], 5, [4, 3], [2, 1]]
```

Résumé

Rappel :

- on peut définir une liste en *compréhension*
- on peut copier les éléments d'une liste avec le *slicing* et *deepcopy*

Attention :

- L1 = L2 crée juste *un autre* « *pointeur* » vers L1
- éviter de *modifier* une liste pendant qu'on la *parcourt*

```
L1 = [x for i in range(0, 10, 2)]
```

```
L2 = [x for x in [3.14, 7.0, -8.3, 2.5, 1.0] if x.is_integer()]
```