

# Collections : généralités et listes

Polytech Marseille

Séverine Dubuisson, Simon Vilmin

[severine.dubuisson@univ-amu.fr](mailto:severine.dubuisson@univ-amu.fr),

[simon.vilmin@univ-amu.fr](mailto:simon.vilmin@univ-amu.fr)

2024 - 2025

**amU** Aix  
Marseille  
Université



# Plan

## Généralités

- Séquences, tableaux associatifs, ensembles
- Éléments de comparaison
- 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

Séquences, tableaux associatifs, ensembles

Éléments de comparaison

Résumé

Listes : les bases

Listes : aspects plus complexes

## 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

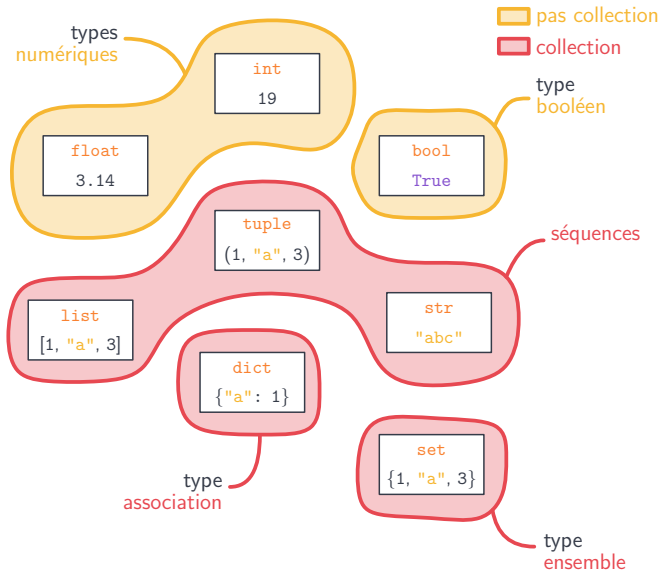
❌ **Problème** : on va pas créer des nouvelles variables à chaque fois ...

💡 **Idée** : il faudrait 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



# Séquences



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



**Remarque** : dans une *séquence*

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

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

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

```
seq[start:stop:step] # de start a stop de step en step
```

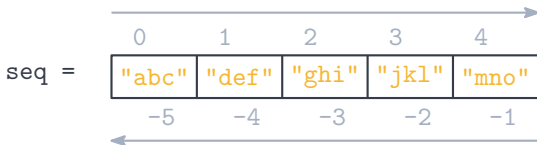
*séquences* : **str**, **tuple**, **list**

## Séquences : indexation

**i Remarque :** En Python, l'indexation d'une séquence `seq` est bidirectionnelle

1. de 0 à `len(seq) - 1`
2. de -1 à `-len(seq)`

```
>>> seq = ["abc", "def", "ghi", "jkl", "mno"]
>>> seq[1]
"def"
>>> seq[-3]
"ghi"
```





## Séquences : slicing

 **Syntaxe** : le *slicing* permet de sélectionner une partie d'une *séquence*

```
seq[start:stop]          # de start a stop de 1 en 1  
seq[start:stop:step]    # de start a stop de step en step
```



**Astuce** : le slicing ressemble beaucoup à un *range*



**Remarque** :

- par défaut, start=0, stop=len(seq) - 1, step=1
- tous les paramètres sont donc *facultatifs*
- possible d'utiliser l'*indexation négative*



**Attention** : si start + step « s'éloigne » de stop, le slicing renvoie la *séquence vide*

## Séquences : slicing

```
>>> seq = ["un", "deux", "trois", "quatre", "cinq", "six"]
>>> seq[1:3]
['deux', 'trois']
>>> seq[-3]
'quatre'
>>> seq[1:5:2]
['deux', 'quatre']
>>> seq[-2:-5:-1]
['cinq', 'quatre', 'trois']
>>> seq[-1:1:-1]
['six', 'cinq', 'quatre', 'trois']
>>> seq[2:-2]
['trois', 'quatre']
```



**Astuce :** le slicing est *très flexible* !

## Tableau associatif : dictionnaire

Un *tableau associatif* (ou *mapping*) est une structure abstraite qui associe des *valeurs* à des *clés*. À une clé est associée au plus une valeur.

**i Remarque :** En Python, le type qui implémente le tableau associatif est le *dictionnaire* (`dict`)

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

cle: valeur

la *clé* doit être *hashable*, la *valeur* peut être *quelconque*. Toutes les clés d'un dictionnaire sont *uniques*

```
>>> d = {"k": "v", "b": 2, 54: "pouet"}
>>> d["k"] # on veut la valeur associee a la cle "k"
'v'
```

## Tableau associatif : pourquoi faire ?



**Astuce :** un dictionnaire est pratique quand on veut accéder *très rapidement* à des données caractérisées par des *identifiants uniques*.


Quelques idées d'usages :

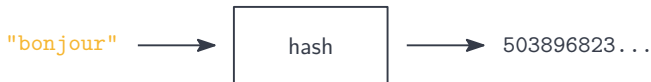
- informations sur des livres (valeurs) identifiés par leurs ISBN (clé)
- mots (clés) et leurs définitions (valeurs)
- informations sur des gens (valeurs) identifiés par leurs numéro de sécu (clé)



**Remarque :** teaser du cours sur les dictionnaires

## Notion de hash : quoi ?

 **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

## Notion de hash : 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

```
>>> t1 = (1, 2, "pouet")
>>> hash(t1)
5038968233157651535
>>> t2 = (1, [2, 3], "bonjour")
>>> hash(t2)
TypeError: unhashable type: 'list'
```


## Notion de hash : pourquoi ?

✓ **Réponse :** intuitivement, les dictionnaires sont une « surcouche » aux listes : la fonction de hash *transforme une clé en un index d'une liste* dans laquelle sont stockées les valeurs

```
>>> d = {}  
>>> d["a"] = 5  
>>> d[[1, 2]] = "essai liste"  
TypeError: unhashable type: 'list'
```

**i Remarque :** teaser du cours sur les dictionnaires

# Ensembles

 **Définition :** un *ensemble* (type `set`) est une *collection non-ordonnée* d'objets *hashables distincts*.

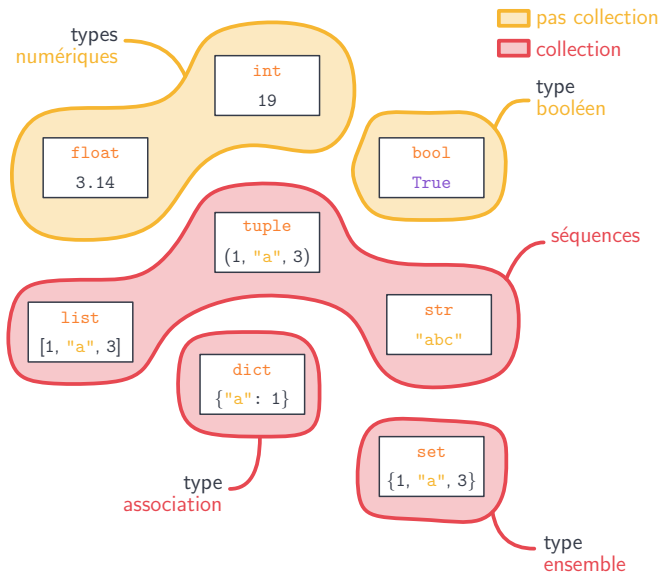
```
>>> ens = {1, 2, "a"}
>>> ens.add(3)
>>> ens
{1, 2, "a", 3}
```

 **Astuce :** `set`  $\simeq$  `dict` sans valeurs (que des clés)

 **Remarque :** teaser du cours sur les ensembles



# Galaxie : le retour



## Quelques éléments de comparaison

**i Remarque :** toutes les collections sont *itérables*, i.e. on peut les parcourir directement avec un *for*

Les éléments que l'on va considérer ici :

- l'*indexation* : la position des éléments est-elle définie et importante ?
- la *mutabilité* : est-ce qu'on peut changer le contenu de la collection ?
- ce qu'on peut *stocker* : la collection peut-elle contenir tout type d'objet ?

# Séquences vs. les autres : indexation

## Séquences :

```
>>> liste = [1, "deux", 3, "quatre", 5]
>>> liste[0]
1
>>> chaine = "attention ! Un bernard l'hermite !"
>>> chaine[-8:9:-1]
"eh'l dranreb nU !"
```

## Ensembles et dictionnaires

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

# Séquences vs. les autres : égalité

## Séquences :

```
>>> [1, 2, 3] == [3, 2, 1] # list
False
>>> (1, 2, 3) == (2, 1, 3) # tuple
False
>>> "abc" == "bca" # str
False
```

## Ensembles et dictionnaires :

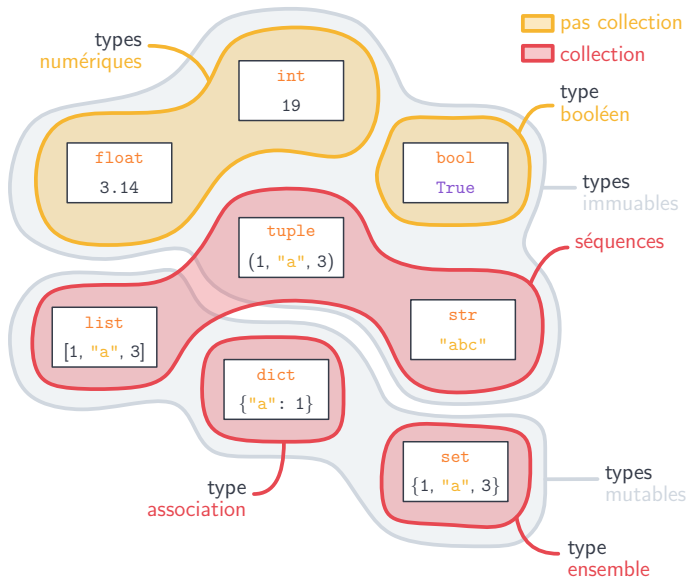
```
>>> {1, 2, 3} == {2, 1, 3} # set
True
>>> {"a": 1, "b": 2} == {"b": 2, "a": 1} # dict
True
```



**Remarque :** en bref

- *séquences* égales : *mêmes éléments, même ordre (indexation)*
- dictionnaires/ensembles égaux : *mêmes éléments*

# Galaxie : la vengeance



# 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, un objet est *mutable* si on peut le modifier, *immuable* sinon
- à part *tuple* les types *hashables* sont les types *immuables*

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

## Mutable vs. immutable : exemple

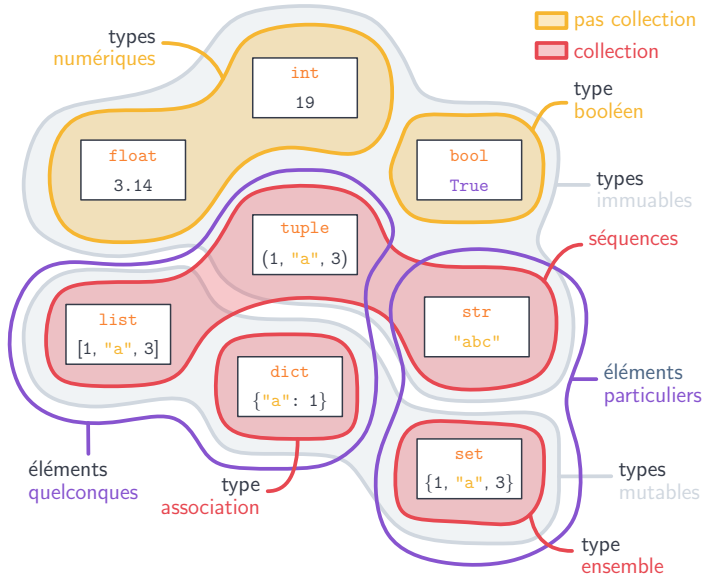
Objets mutables :

```
>>> liste = [1, 2, 3]
>>> liste[0] = 7
>>> liste
[7, 2, 3]
>>> ens = {1, 2, 3}
>>> ens.add(4)
>>> ens
{1, 2, 3, 4}
```

Objets immuables :


```
>>> chaine = "bonjour"
>>> chaine[0] = "B"
TypeError: 'str' object does not support item assignment
>>> uplet = (1, 2, 3)
>>> uplet[0] = 7
TypeError: 'tuple' object does not support item assignment
```

# Galaxie : le retour de la vengeance qui tue





# Collectionner quoi ?

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

 **Réponse :** ça dépend ...

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

 **Astuce :** en résumé

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

## Bon, du coup qui stocke quoi ?

Stocker des valeurs dans une collection :

- `str` : caractères uniquement (en fait des points de code unicode)
- `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	indexée	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

**i Remarque :** 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 *séquence mutable* d'objets de tout types.

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

```
liste = [elt1, elt2, ...]
```

```
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.

```
>>> L = ["une", "liste", "de", 5, "elements"]
>>> L[2] # element d'indice 2
'de'
>>> L[1:3] # slicing, liste des elements d'indices 1 a 2
['liste', 'de']
```


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

```
>>> L = ["une", "liste", "de", 5, "elements"]
>>> for i in range(0, len(L)):
    print(L[i], end=" ")
une liste de 5 elements
```



**Attention :** souvent parcourir les *indices* n'est *pas le plus pertinent*


# Itérables


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

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

```
>>> phrase = ["messire,", "les", "anglois",  
              "nous", "envahissent", "!"]  
>>> for mot in phrase:  
    print(mot, end=" ")  
messire, les anglois nous envahissent !"
```

# Énumération

 **Idée** : 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*

```
>>> L = [ True, False, 0, 1, None ]
>>> for (indice, valeur) in enumerate(L):
    print(f"valeur a l'indice {indice} : {valeur}")
valeur a l'indice 0 : True
valeur a l'indice 1 : False
valeur a l'indice 2 : 0
valeur a l'indice 3 : 1
valeur a l'indice 4 : None
```



# 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`

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

```
>>> L = [3, "perruches", "avec", 3, "tongues"]
>>> L.index("perruches")
1
>>> L.count(3)
2
```

## Exercices

⚙️ **Exercice** : écrivez une fonction `verif_ordre` qui prend en entrée une liste et qui vérifie que les nombres qui s'y trouvent sont triés par ordre croissant. Par exemple pour

```
L1 = ["oui", 1, "non", True, False, 2, "jsp", 12]
L2 = ["que", 2, "ne", 1, "fusse-je", "offusque", 3]
```

La fonction doit renvoyer `True` pour L1 et `False` pour L2

⚙️ **Exercice** : Écrivez une fonction `extraire_voyelle` qui prend en entrée une liste de mots et qui renvoie la chaîne de caractères contenant toutes les voyelles des mots de la liste. Par exemple pour

```
["tu", "m'as", "roule", "dans", "la", "frangipane"]
```

La fonction doit renvoyer

```
"u a oue a a aiae"
```

## Exercices

⚙ **Exercice :** Écrivez une fonction `epeler` qui prend en paramètre un nombre flottant et qui renvoie la chaîne de caractères épelant chacun des chiffres de ce nombre. Par exemple pour

14.27

La fonction doit renvoyer

"un quatre virgule deux sept"

⚙ **Exercice :** Écrivez une fonction récursive `tri_fusion` qui prend en paramètres une liste et qui trie la liste avec l'algorithme de fusion.

# Modification

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

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

```
>>> L = [1, 2, 3, 4, 5]
>>> L[0] = "un"
>>> L[2:4] = ["trois", "quatre"]
>>> L
["un", 2, "trois", "quatre", 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*

```
>>> L = ["a", "b", "c", "d", "e"]
>>> L.append("f")
>>> L
["a", "b", "c", "d", "e", "f"]
>>> L.insert(0, "alphabet")
>>> L
["alphabet", "a", "b", "c", "d", "e", "f"]
>>> L.extend(["g", "h", "i"])
>>> L
["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*
- `clear` *vide* la liste

```
>>> L = ["a", "b", "a", "c", "d", "e", "f"]
>>> b = L.pop(1)
>>> L, b
["a", "a", "c", "d", "e", "f"], "b"
>>> L.remove("a")
>>> L
["a", "c", "d", "e", "f"]
>>> del L[2:4]
>>> L
["a", "c", "f"]
>>> L.clear()
>>> L
[]
```

## Quelques opérations entre listes

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

```
>>> L1 = ["lun", "mar", "mer", "jeu", "ven"]
>>> L2 = ["sam", "dim"]
>>> L3 = 2 * (L1 + L2)
>>> L3
["lun", "mar", "mer", "jeu", "ven", "sam", "dim",
"lun", "mar", "mer", "jeu", "ven", "sam", "dim"]
```

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



## Quelques opérations entre listes

**!** **Attention :** par contre, `*` et `+=` *modifient* effectivement la liste initiale !

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L2 = 2 * L2
>>> L2 is L1 # L2 et L1 pointent sur la meme zone ?
False
>>> L3 = L1
>>> L3 *= 2
>>> L3 is L1 # L3 et L1 pointent sur la meme zone ?
True
```

## Exercices

⚙️ **Exercice** : Écrivez une fonction 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** : Écrivez une fonction qui prend en paramètres une matrice carrée composée de lettres, et qui vérifie que toutes les lignes, les colonnes et les diagonales sont des palindromes. Par exemple pour

```
m = ["cbc", "ada", "cba"]]
```

La fonction doit renvoyer **False**

# Résumé



## Rappel :

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



**Attention :** `append`, `+=`, `*=`, ... *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` filtre les éléments à sélectionner dans l'itérable

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

## Exemple

```
>>> L1 = [x for x in range(5)]
>>> L1
[0, 1, 2, 3, 4]

>>> L2 = [x**2 for x in range(5)]
>>> L2
[0, 1, 4, 9, 16]

>>> L3 = [2 * c for c in "arg"]
>>> L3
["aa", "rr", "gg"]

>>> L4 = [x for x in L2 if x % 2 == 0] # condition de filtrage
>>> L4
[0, 4, 16]

>>> L5 = [2 * [x] for x in L3]
>>> L5
[["aa", "aa"], ["rr", "rr"], ["gg", "gg"]]
```



## Exercice

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

1. étant donnée une liste  $L$  d'entiers, construire la liste  $L_f$  des paires  $(x, f(x))$  où  $f(x) = 3x^2 - x + 1$ , pour tout  $x$  de  $L$
2. étant donnée une liste  $L$ , construire la liste des éléments de  $L$  de type `str`
3. étant données deux listes  $L1$ ,  $L2$ , trouver les éléments de  $L1$  qui ne sont pas dans  $L2$
4. étant données deux listes  $L1$ ,  $L2$ , construire la différence symétrique de  $L1$  et  $L2$
5. étant donné un entier  $n$ , construire une liste de  $n$  listes où la  $i$ ème liste est la liste des entiers de 1 à  $i$

## Exercice

⚙️ **Exercice** : en utilisant les *comprehension list*, écrivez :

- une fonction `trouve_espace` qui prend en paramètre une phrase `p` et qui renvoie les indices auxquels se trouve des espaces dans `p`
- une fonction `trouve_mots` qui prend en paramètre une phrase `p` et qui renvoie la liste de mots de `p`

Par exemple pour

```
p = "une moquette en mimolette"
```

Les fonctions `espace` et `mots` renvoient respectivement

```
[3, 12, 15]  
["une", "moquette", "en", "mimolette"]
```

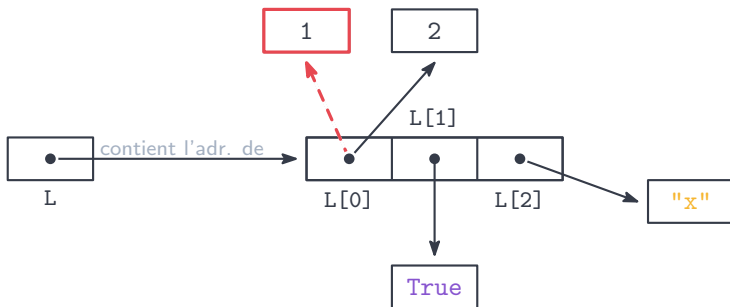
### **Astuce :**

- on suppose que `p` ne contient pas plusieurs espaces à la suite, et qu'elle commence et termine par une lettre
- pour la question 2, réutilisez la question 1.

# 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
```



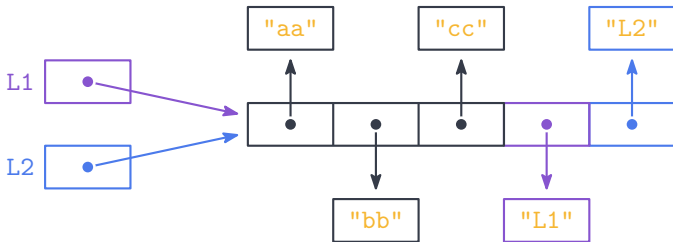
## Assignement de listes

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

```
L2 = L1
```

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

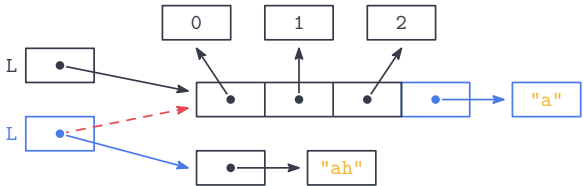
```
L2.append("L2")
```



**! Attention :** quand on fait `L2 = L1` 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* ?

```
>>> l = [c for c in "abcdef"]
>>> for c in l:
    l.remove(c)
>>> l
["b", "d", "f"]
```

```
>>> l = [c for c in "abcdef"]
>>> for c in l:
    l.append(c)
>>> l
# boucle infinie !
```

```
>>> l = [c for c in "abcdef"]
>>> for c in l:
    l.pop(0)
>>> l
["d", "e", "f"]
```



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



## Exercices

⚙️ **Exercice** : Écrivez une fonction `display` qui prend en paramètres une matrice `M` d'entiers positifs (liste de listes) et une chaîne de caractère `s` et qui affiche le dessin représenté par `M` et `s` en faisant correspondre à un élément `m` de la matrice le `m`-ème caractère de `s`. Par exemple, on aura

```
>>> M = [[0, 0, 1, 2, 0, 0],  
         [0, 1, 3, 3, 2, 0],  
         [1, 0, 0, 0, 0, 2]]  
>>> s = " /-\\ "  
>>> display(M, s)  
  /\n /\n /--\\  
/      \\
```



**Astuce** : allez chercher la `table Unicode` pour rigoler un peu !

## Exercices

⚙️ **Exercice** : Écrivez une fonction qui prend en paramètres deux entiers  $n$  et  $m$  ( $n < m$ ) et qui, *en une ligne*, renvoie le nombre de multiples de 7 entre  $n$  et  $m$

⚙️ **Exercice** : Écrivez une fonction `copie_profonde` qui prend en entrée une liste  $L$  et qui renvoie une copie profonde de  $L$ .

**i Remarque** : Pour simplifier, on ne considère que des listes contenant des listes ou des entiers, et *ne contenant pas* d'autoréférences, i.e.  $L$  ne contient pas  $L$  (pointeur), mais peut contenir une sous-liste avec exactement les mêmes éléments que  $L$ .

# Résumé

## Rappel :

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

## Attention :

- L2 = L1 crée *un « 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()]
```