



UNIVERSITÉ  
CÔTE D'AZUR

# Programmation impérative en Python

## Cours 5. Listes, tuples et gestion mémoire

---

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : `https://upinfo.univ-cotedazur.fr/~obaldellon/`

LICENCE I — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- ▶ Examen le jeudi 21 mars de 15h30 à 17h30
  - ▶ Sur les chapitres 1 à 5
  - ▶ On vous tiendra au courant du lieu

- ▶ Examen le jeudi 21 mars de 15h30 à 17h30
  - ▶ Sur les chapitres 1 à 5
  - ▶ On vous tiendra au courant du lieu
- ▶ En cas d'incompatibilité avec une autre UE.
  - ▶ Débrouillez-vous avec l'autre enseignant :)
  - ▶ Me contacter par mail

- ▶ Examen le jeudi 21 mars de 15h30 à 17h30
  - ▶ Sur les chapitres 1 à 5
  - ▶ On vous tiendra au courant du lieu
- ▶ En cas d'incompatibilité avec une autre UE.
  - ▶ Débrouillez-vous avec l'autre enseignant :)
  - ▶ Me contacter par mail
- ▶ Il n'y aura pas de cours de Python la semaine du 20 Mars.
  - ▶ :(
  - ▶ Ni CM, ni TP, ni TD

- 🍃 Partie I. Séquences
- 🍃 Partie II. Accès et constructeur
- 🍃 Partie III. Mutabilité des listes
- 🍃 Partie IV. Petits algorithmes
- 🍃 Partie V. Gestion de la mémoire
- 🍃 Partie VI. Table des matières

- ▶ Une **séquence** est une suite finie de valeurs indexées.

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.



- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>>
```

```
SHELL
```

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
```

```
SHELL
```

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>>
```

**SHELL**

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>>
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>>
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
```

SHELL



- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une séquence peut contenir des éléments de types distincts :

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une séquence peut contenir des éléments de types distincts :

```
>>>
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type str (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une séquence peut contenir des éléments de types distincts :

```
>>> t = (2.0 , 'Salut' , math.cos)# tuple de 3 éléments
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type str (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une séquence peut contenir des éléments de types distincts :

```
>>> t = (2.0 , 'Salut' , math.cos)# tuple de 3 éléments
>>>
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type `str` (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une séquence peut contenir des éléments de types distincts :

```
>>> t = (2.0 , 'Salut' , math.cos) # tuple de 3 éléments
>>> [1, 'B', t , [3,4,5]]        # liste de 4 éléments
```

SHELL

- ▶ Une **séquence** est une suite finie de valeurs indexées.
  - ▶ On connaît déjà le type str (**chaîne de caractères**)
  - ▶ Il y a aussi les **tuples** et les **listes**.

```
>>> t = (1,2,3)
>>> type(t)
<class 'tuple'>
>>> l = [1,2,3]
>>> type(l)
<class 'list'>
```

SHELL

- ▶ Une séquence peut contenir des éléments de types distincts :

```
>>> t = (2.0 , 'Salut' , math.cos) # tuple de 3 éléments
>>> [1, 'B', t , [3,4,5]]         # liste de 4 éléments
[1, 'B', (2.0, 'Salut', <built-in function cos>), [3, 4, 5]]
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet



- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
    # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

&gt;&gt;&gt;

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
    # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
    # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)  
>>>
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
    # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)  
>>> t # 37 = 5*7 + 2
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
        # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)  
>>> t # 37 = 5*7 + 2  
(5, 2)  
>>>
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
    # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)  
>>> t # 37 = 5*7 + 2  
(5, 2)  
>>> t + (3,4) # concaténation
```

SHELL



- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
        # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)  
>>> t # 37 = 5*7 + 2  
(5, 2)  
>>> t + (3,4) # concaténation  
(5, 2, 3, 4)  
>>>
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
        # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

```
>>> t = division(37,7)  
>>> t # 37 = 5*7 + 2  
(5, 2)  
>>> t + (3,4) # concaténation  
(5, 2, 3, 4)  
>>> t[0]=17 # les tuples ne sont pas modifiables
```

SHELL

- ▶ Les **tuples** sont une généralisation des couples.
  - ▶ couple, triplet (3-uplet), quadruplet (ou 4-uplet), ..., n-uplet

Une fonction ne peut retourner  
qu'une valeur,  
mais peut retourner un tuple.

```
def division(a,b):  
    q = 0  
    r = a  
    while r >= b:  
        r = r - b  
        q = q + 1  
        # q=a//b et r=a%b  
    return (q, r)
```

SCRIPT

SHELL

```
>>> t = division(37,7)  
>>> t # 37 = 5*7 + 2  
(5, 2)  
>>> t + (3,4) # concaténation  
(5, 2, 3, 4)  
>>> t[0]=17 # les tuples ne sont pas modifiables  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

- ▶ Les **listes** sont une généralisation des tuples.

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>>
```

```
SHELL
```

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']
```

**SHELL**

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>>
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']
```

SHELL



- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>>
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

```
>>>
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

```
>>> club[4]='Étienne'
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

```
>>> club[4]='Étienne'  
>>>
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

```
>>> club[4]='Étienne'  
>>> club
```

SHELL



- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

```
>>> club[4]='Étienne'  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Étienne', 'Fatou']
```

SHELL

- ▶ Les **listes** sont une généralisation des tuples.
- ▶ Exemple :
  - ▶ On souhaite conserver les prénoms des membres d'un club
  - ▶ On peut utiliser les listes pour les stocker.

```
>>> club = ['Alix', 'Bianca', 'Carlos', 'Dagobert']  
>>> club = club + ['Etienne', 'Fatou']  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Etienne', 'Fatou']
```

SHELL

- ▶ Zut alors! On a oublié l'accent sur Étienne!!!

```
>>> club[4]='Étienne'  
>>> club  
['Alix', 'Bianca', 'Carlos', 'Dagobert', 'Étienne', 'Fatou']
```

SHELL

- ▶ Les listes sont **mutables**.

- 🍃 Partie I. Séquences
- 🍃 **Partie II. Accès et constructeur**
- 🍃 Partie III. Mutabilité des listes
- 🍃 Partie IV. Petits algorithmes
- 🍃 Partie V. Gestion de la mémoire
- 🍃 Partie VI. Table des matières

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>>
```

```
SHELL
```

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>> len('')
```

SHELL



- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>> len('')  
0  
>>>
```

SHELL

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>> len('')  
0  
>>> len((2,))
```

SHELL

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>> len('')
0
>>> len((2,))
1
>>>
```

SHELL

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>> len('')
0
>>> len((2,))
1
>>> len([0,1,2])
```

SHELL

- ▶ Construction en extension
  - ▶ **tuples** : les éléments entre **parenthèses** séparés par des virgules
  - ▶ **liste** : les éléments entre **crochets** séparés par des virgules

Nombre d'éléments	0	1	cas général (ici 5 éléments)
Tuple	()	(2,)	(3,4,1,(1,3),-6)
Liste	[]	[2]	[3,4,1,(1,3),-6]
Chaîne	''	'@'	'Salut'

- ▶ Un 1-uplet est possible mais peu utile : **attention à la syntaxe** (2,).
- ▶ La fonction `len(S)` donne la longueur (nombre d'éléments de S)

```
>>> len('')
0
>>> len((2,))
1
>>> len([0,1,2])
3
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>>
```

```
SHELL
```

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'  
'Bonjour'  
>>>
```

SHELL



- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'  
'Bonjour'  
>>> (1,2) + (3,4)
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'  
'Bonjour'  
>>> (1,2) + (3,4)  
(1, 2, 3, 4)  
>>>
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'  
'Bonjour'  
>>> (1,2) + (3,4)  
(1, 2, 3, 4)  
>>> [1,2,3] + [7] + [8.5]
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a  
 $\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>>
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>> (1,2)*5
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>>
```

SHELL



- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> 4 * [ 3,4,5]
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> 4 * [ 3,4,5]
[3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5]
>>>
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> 4 * [ 3,4,5]
[3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5]
>>> 'HA ' * 10 + 5*'!'
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> 4 * [ 3,4,5]
[3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5]
>>> 'HA ' * 10 + 5*'!'
'HA HA HA HA HA HA HA HA HA HA HA !!!!!'
```

SHELL

- ▶ On peut **concaténer** deux séquences avec l'opérateur +

```
>>> 'Bon' + 'jour'
'Bonjour'
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> [1,2,3] + [7] + [8.5]
[1, 2, 3, 7, 8.5]
```

SHELL

- ▶ En particulier, si S1 et S2 sont deux séquences on a

$$\text{len}(S1+S2) == \text{len}(S1) + \text{len}(S2)$$

- ▶ On peut **multiplier** une séquence par un entier :

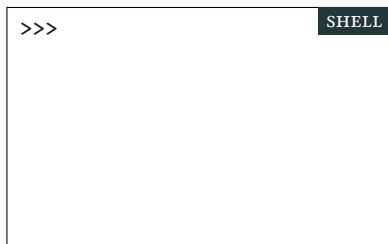
```
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> 4 * [ 3,4,5]
[3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5]
>>> 'HA '* 10 + 5*'!!'
'HA HA HA HA HA HA HA HA HA HA !!!!!'
```

SHELL

- ▶ Les opérations sont **communes** à ces trois types de séquence.

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>>
```



- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'  
>>>
```

SHELL



- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'  
>>> c[0] # Premier
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'  
>>> c[0] # Premier  
'S'  
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'  
>>> c[0] # Premier  
'S'  
>>> c[len(c)-1] # Dernier
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>>
```

SHELL



- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

&gt;&gt;&gt;

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(1)
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(1)
3
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(1)
3
>>> l[0] # Mon 1er est un chiffre
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(1)
3
>>> l[0] # Mon 1er est un chiffre
6
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> l = [1, 6]
>>> l[0] # Mon 1er est un chiffre
6
>>> l[1] # Mon 2nd est un booléen
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> l = [1, True, 'badour']
>>> len(l)
3
>>> l[0] # Mon 1er est un chiffre
1
>>> l[1] # Mon 2nd est un booléen
True
>>>
```

SHELL



- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(l)
3
>>> l[0] #Mon 1er est un chiffre
6
>>> l[1] #Mon 2nd est un booléen
True
>>> l[2] #Mon 3ème est une chaîne
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> l = [1, True, 'Yeux']
>>> len(l)
3
>>> l[0] # Mon 1er est un chiffre
1
>>> l[1] # Mon 2nd est un booléen
True
>>> l[2] # Mon 3ème est une chaîne
'Yeux'
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> l = [1, True, 'Yeux']
>>> len(l)
3
>>> l[0] # Mon 1er est un chiffre
1
>>> l[1] # Mon 2nd est un booléen
True
>>> l[2] # Mon 3ème est une chaîne
'Yeux'
>>> l
[1, True, 'Yeux']
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> l = [6, True, 'Yeux']
>>> len(l)
3
>>> l[0] # Mon 1er est un chiffre
6
>>> l[1] # Mon 2nd est un booléen
True
>>> l[2] # Mon 3ème est une chaîne
'Yeux'
>>> l
[6, True, 'Yeux']
```

SHELL

```
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(l)
3
>>> l[0] #Mon 1er est un chiffre
6
>>> l[1] #Mon 2nd est un booléen
True
>>> l[2] #Mon 3ème est une chaîne
'Yeux'
>>> l
[6, True, 'Yeux']
```

SHELL

- ▶ Comme pour les chaînes, on peut utiliser des indices négatifs.

```
>>> l[-1] , t[-2] , c[-3] # Sans parenthèses
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(l)
3
>>> l[0] #Mon 1er est un chiffre
6
>>> l[1] #Mon 2nd est un booléen
True
>>> l[2] #Mon 3ème est une chaîne
'Yeux'
>>> l
[6, True, 'Yeux']
```

SHELL

- ▶ Comme pour les chaînes, on peut utiliser des indices négatifs.

```
>>> l[-1] , t[-2] , c[-3] # Sans parenthèses
('Yeux', True, 'n')
>>>
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(1)
3
>>> l[0] # Mon 1er est un chiffre
6
>>> l[1] # Mon 2nd est un booléen
True
>>> l[2] # Mon 3ème est une chaîne
'Yeux'
>>> l
[6, True, 'Yeux']
```

SHELL

- ▶ Comme pour les chaînes, on peut utiliser des indices négatifs.

```
>>> l[-1] , t[-2] , c[-3] # Sans parenthèses
('Yeux', True, 'n')
>>> t[666] # Horreur, une erreur
```

SHELL

- ▶ L'accès est commun aux chaînes, tuples et listes.

```
>>> c = 'Saluton !'
>>> c[0] # Premier
'S'
>>> c[len(c)-1] # Dernier
'!'
>>> t = (1, True, 'badour')
>>> len(t)
3
>>> t[1]
True
```

SHELL

```
>>> len(1)
3
>>> l[0] #Mon 1er est un chiffre
6
>>> l[1] #Mon 2nd est un booléen
True
>>> l[2] #Mon 3ème est une chaîne
'Yeux'
>>> l
[6, True, 'Yeux']
```

SHELL

- ▶ Comme pour les chaînes, on peut utiliser des indices négatifs.

```
>>> l[-1] , t[-2] , c[-3] # Sans parenthèses
('Yeux', True, 'n')
>>> t[666] # Horreur, une erreur
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: tuple index out of range
```

SHELL



- ▶ Pour parcourir une séquence, on peut utiliser des boucles.

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant sur les indices de 0 à `len(Seq)-1`

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

&gt;&gt;&gt;

SHELL

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)
```

SHELL

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)  
S -> a -> l -> u -> t -> o -> n ->   -> ! ->  
>>>
```

SHELL

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)  
S -> a -> l -> u -> t -> o -> n ->   -> ! ->  
>>> affiche(t)
```

SHELL

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)  
S -> a -> l -> u -> t -> o -> n ->   -> ! ->  
>>> affiche(t)  
1 -> True -> badour ->  
>>>
```

SHELL

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)  
S -> a -> l -> u -> t -> o -> n ->   -> ! ->  
>>> affiche(t)  
1 -> True -> badour ->  
>>> affiche(l)
```

SHELL



- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)  
S -> a -> l -> u -> t -> o -> n ->   -> ! ->  
>>> affiche(t)  
1 -> True -> badour ->  
>>> affiche(l)  
6 -> True -> Yeux ->
```

SHELL

- ▶ Pour parcourir une séquence, on peut utiliser des boucles.
  - ▶ en itérant **sur les indices** de 0 à `len(Seq)-1`
  - ▶ en itérant directement **sur les éléments** de la séquence.

```
def affiche(Seq):  
    n=len(Seq)  
    for i in range(n):  
        print(Seq[i],end=' -> ')  
    print('')
```

SCRIPT

```
def affiche(Seq):  
    for e in Seq:  
        print(e,end=' -> ')  
    print('')
```

SCRIPT

```
>>> affiche(c)  
S -> a -> l -> u -> t -> o -> n ->   -> ! ->  
>>> affiche(t)  
1 -> True -> badour ->  
>>> affiche(l)  
6 -> True -> Yeux ->
```

SHELL

- ▶ C'est le même programme pour les trois types de séquences !

- ▶ On souhaite connaître l'indice d'un élément d'une séquence

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

&gt;&gt;&gt;

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep='')
```

SHELL



- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>>
```

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>> index(c,'o')
```

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>> index(c,'o')  
5  
>>>
```

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>> index(c,'o')  
5  
>>> index(t,False)
```

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>> index(c,'o')  
5  
>>> index(t,False)  
-1  
>>>
```

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>> index(c,'o')  
5  
>>> index(t,False)  
-1  
>>> index(l,True)
```

SHELL

- ▶ On souhaite connaître l'indice d'un élément d'une séquence
  - ▶ On retourne le premier indice qui convient
  - ▶ Si aucun indice ne convient, on renvoie -1

```
def index(Seq,x):  
    for i in range(len(Seq)):  
        if Seq[i]==x:  
            return i  
    # On sort de la boucle si on n'a pas trouvé x  
    return -1
```

SCRIPT

- ▶ Et ça marche avec tous les types de séquence !

```
>>> print(c,t,l,sep=' ')  
Saluton ! (1, True, 'badour') [6, True, 'Yeux']  
>>> index(c,'o')  
5  
>>> index(t,False)  
-1  
>>> index(l,True)  
1
```

SHELL

- ▶ Une méthode similaire existe déjà.



- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même !

```
>>>
```

```
SHELL
```

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]  
>>>
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]  
>>> liste.index('Coucou')
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]  
>>> liste.index('Coucou')  
3
```

SHELL

```
>>>
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]  
>>> liste.index('Coucou')  
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.

```
>>> liste.index(-2)
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]  
>>> liste.index('Coucou')  
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.

```
>>> liste.index(-2)  
1  
>>>
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
```

SHELL



- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>>
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ
- ▶ Si aucun n'est valide, le programme se termine par un erreur.

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ
- ▶ Si aucun n'est valide, le programme se termine par un erreur.

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: 5 is not in list
>>>
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ
- ▶ Si aucun n'est valide, le programme se termine par un erreur.

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: 5 is not in list
>>> (10,20,30).index(30) # marche avec tuples et chaînes
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ
- ▶ Si aucun n'est valide, le programme se termine par un erreur.

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: 5 is not in list
>>> (10,20,30).index(30) # marche avec tuples et chaînes
2
>>>
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ
- ▶ Si aucun n'est valide, le programme se termine par un erreur.

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: 5 is not in list
>>> (10,20,30).index(30) # marche avec tuples et chaînes
2
>>> 'abcdef'.index('bcd') # marche avec des sous-chaînes
```

SHELL

- ▶ Une méthode similaire existe déjà.
  - ▶ Mais il faut savoir l'écrire soit même!

```
>>> liste = [4, -2, False, 'Coucou', -2, (3,5)]
>>> liste.index('Coucou')
3
```

SHELL

- ▶ En cas de plusieurs index valides, c'est le premier qui est renvoyé.
- ▶ On peut donner un indice de départ
- ▶ Si aucun n'est valide, le programme se termine par un erreur.

```
>>> liste.index(-2)
1
>>> liste.index(-2,2) # on cherche à partir de liste[2]
4
>>> liste.index(5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: 5 is not in list
>>> (10,20,30).index(30) # marche avec tuples et chaînes
2
>>> 'abcdef'.index('bcd') # marche avec des sous-chaînes
1
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.

```
>>>
```

SHELL

```
>>>
```

SHELL



- ▶ On peut **déstructurer** un tuple ou une liste.

```
>>> liste=[1,2,3]
```

SHELL

```
>>> uplet=(1,2,3)
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.

```
>>> liste=[1,2,3]  
>>>
```

SHELL

```
>>> uplet=(1,2,3)  
>>>
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>>
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>>
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>>
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>>
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>>
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>>
```

SHELL



- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

SHELL

- ▶ Il faut la même structure des deux côtés.

```
>>>
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

SHELL

- ▶ Il faut la même structure des deux côtés.

```
>>> liste=[1,2,3]
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

SHELL

- ▶ Il faut la même structure des deux côtés.

```
>>> liste=[1,2,3]
>>>
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

SHELL

- ▶ Il faut la même structure des deux côtés.

```
>>> liste=[1,2,3]
>>> [a,b] = liste
```

SHELL

- ▶ On peut **déstructurer** un tuple ou une liste.
  - ▶ nécessite de connaître la taille en écrivant le programme.
  - ▶ permet de définir plusieurs variables d'un coup

```
>>> liste=[1,2,3]
>>> [a,b,c] = liste
>>> a
1
>>> b
2
>>> c
3
```

SHELL

```
>>> uplet=(1,2,3)
>>> (a,b,c) = uplet
>>> a
1
>>> b
2
>>> c
3
```

SHELL

- ▶ Il faut la même structure des deux côtés.

```
>>> liste=[1,2,3]
>>> [a,b] = liste
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>>
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>> c = 'Salut à toi'
```

SHELL



- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>> c = 'Salut à toi'  
>>>
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>>
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>>
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation `[i]`

```
>>> c = 'Salut à toi'
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]
'à'
>>>
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]
```

SHELL



- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation `[i]`
  - ▶ On accède à une suite d'éléments avec la notation `[i:j]`

```
>>> c = 'Salut à toi'
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]
'à'
>>> l[1:4]
['Un', 'Deux', 'Trois']
>>>
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]
'à'
>>> l[1:4]
['Un', 'Deux', 'Trois']
>>> t[1:6:2]
```

SHELL

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]  
['Un', 'Deux', 'Trois']  
>>> t[1:6:2]  
(1, 3, 5)
```

SHELL

&gt;&gt;&gt;

SHELL

- ▶ Exercice : Écrire la fonction extraire.

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]  
['Un', 'Deux', 'Trois']  
>>> t[1:6:2]  
(1, 3, 5)
```

SHELL

```
>>> extraire(c,6,7,1)
```

SHELL

- ▶ Exercice : Écrire la fonction extraire.

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]  
['Un', 'Deux', 'Trois']  
>>> t[1:6:2]  
(1, 3, 5)
```

SHELL

```
>>> extraire(c,6,7,1)  
'à'  
>>>
```

SHELL

- ▶ Exercice : Écrire la fonction extraire.

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]  
['Un', 'Deux', 'Trois']  
>>> t[1:6:2]  
(1, 3, 5)
```

SHELL

```
>>> extraire(c,6,7,1)  
'à'  
>>> extraire(l,1,4,1)
```

SHELL

- ▶ Exercice : Écrire la fonction extraire.

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]  
['Un', 'Deux', 'Trois']  
>>> t[1:6:2]  
(1, 3, 5)
```

SHELL

```
>>> extraire(c,6,7,1)  
'à'  
>>> extraire(l,1,4,1)  
['Un', 'Deux', 'Trois']  
>>>
```

SHELL

- ▶ Exercice : Écrire la fonction extraire.

- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]
'à'
>>> l[1:4]
['Un', 'Deux', 'Trois']
>>> t[1:6:2]
(1, 3, 5)
```

SHELL

```
>>> extraire(c,6,7,1)
'à'
>>> extraire(l,1,4,1)
['Un', 'Deux', 'Trois']
>>> extraire(t,1,6,2)
```

SHELL

- ▶ Exercice : Écrire la fonction extraire.



- ▶ Comme pour les chaînes, on peut **extraire des tranches**.
  - ▶ On accède à un élément d'une séquence avec la notation [i]
  - ▶ On accède à une suite d'éléments avec la notation [i:j]
  - ▶ On accède à une suite d'éléments avec la notation [i:j:p]

```
>>> c = 'Salut à toi'  
>>> l = ['Zéro', 'Un', 'Deux', 'Trois', 'Quatre', 'Cinq']  
>>> t = (0,1,2,3,4,5)
```

SHELL

```
>>> c[6]  
'à'  
>>> l[1:4]  
['Un', 'Deux', 'Trois']  
>>> t[1:6:2]  
(1, 3, 5)
```

SHELL

```
>>> extraire(c,6,7,1)  
'à'  
>>> extraire(l,1,4,1)  
['Un', 'Deux', 'Trois']  
>>> extraire(t,1,6,2)  
(1, 3, 5)
```

SHELL

- ▶ Exercice : Écrire la fonction extraire.

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>>
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>>
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>> [c for c in 'azerty']
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

- ▶ Une expression **if** suivant le **for** permet de filtrer les éléments.

```
>>>
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

- ▶ Une expression **if** suivant le **for** permet de filtrer les éléments.

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

- ▶ Une expression **if** suivant le **for** permet de filtrer les éléments.

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

```
>>>
```

SHELL



- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

- ▶ Une expression **if** suivant le **for** permet de filtrer les éléments.

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Pour les tuples la syntaxe est :

```
>>> tuple(x*x for x in range(20) if x%2==0 and x*x>20)
```

SHELL

- ▶ Une **construction par compréhension** (de liste) consiste à imiter la notation mathématique

$$\{f(x) \text{ tels que } x \in E \text{ et } P(x)\}$$

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]  
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

- ▶ Une expression **if** suivant le **for** permet de filtrer les éléments.

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Pour les tuples la syntaxe est :

```
>>> tuple(x*x for x in range(20) if x%2==0 and x*x>20)  
(36, 64, 100, 144, 196, 256, 324)
```

SHELL

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>>
```

```
SHELL
```

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]
```

**SHELL**

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>>
```

SHELL

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']
```

SHELL

- Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>>
```

SHELL

- Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]
```

SHELL



- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Même question avec une double boucle `for`

```
>>>
```

SHELL

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Même question avec une double boucle `for`

```
>>> L = [ (1,2), (3,4,5), (6,) ]
```

SHELL

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Même question avec une double boucle `for`

```
>>> L = [ (1,2), (3,4,5), (6,) ]  
>>>
```

SHELL

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Même question avec une double boucle `for`

```
>>> L = [ (1,2), (3,4,5), (6,) ]  
>>> [x for tuple in L for x in tuple]
```

SHELL

- ▶ Traduire les compréhensions suivantes en boucle `for`.

```
>>> [x * x for x in range(6)]  
[0, 1, 4, 9, 16, 25]
```

SHELL

```
>>> [c for c in 'azerty']  
['a', 'z', 'e', 'r', 't', 'y']
```

SHELL

```
>>> [x * x for x in range(20) if x%2==0 and x*x>20]  
[36, 64, 100, 144, 196, 256, 324]
```

SHELL

- ▶ Même question avec une double boucle `for`

```
>>> L = [ (1,2), (3,4,5), (6,) ]  
>>> [x for tuple in L for x in tuple]  
[1, 2, 3, 4, 5, 6]
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>>
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>>
```

SHELL



- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>> tuple('Découpage')
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>> tuple('Découpage')  
( 'D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

SHELL

- ▶ On peut convertir les listes en tuples et vice-versa

```
>>>
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>> tuple('Découpage')  
( 'D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

SHELL

- ▶ On peut convertir les listes en tuples et vice-versa

```
>>> list((1,2,3,4))
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>> tuple('Découpage')  
( 'D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

SHELL

- ▶ On peut convertir les listes en tuples et vice-versa

```
>>> list((1,2,3,4))  
[1, 2, 3, 4]  
>>>
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>> tuple('Découpage')  
( 'D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

SHELL

- ▶ On peut convertir les listes en tuples et vice-versa

```
>>> list((1,2,3,4))  
[1, 2, 3, 4]  
>>> tuple([1,2,3,4])
```

SHELL

- ▶ On peut convertir une chaîne en tuple ou en liste.

```
>>> list('Saucisson')  
['S', 'a', 'u', 'c', 'i', 's', 's', 'o', 'n']  
>>> tuple('Découpage')  
( 'D', 'é', 'c', 'o', 'u', 'p', 'a', 'g', 'e')
```

SHELL

- ▶ On peut convertir les listes en tuples et vice-versa

```
>>> list((1,2,3,4))  
[1, 2, 3, 4]  
>>> tuple([1,2,3,4])  
(1, 2, 3, 4)
```

SHELL

- ▶ Exercice : écrire les deux fonctions `tuple` et `list`.

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

[[a,b,c], [d,e,f], [g,h,i]]

```
>>>
```

SHELL



Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c],[d,e,f],[g,h,i]]`

```
>>> M = [[0,1,2],[10,11,12],[20,21,22]]
```

SHELL

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]  
>>>
```

SHELL

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]  
>>> M[1]
```

SHELL

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]
>>> M[1]
[10, 11, 12]
>>>
```

SHELL

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]
>>> M[1]
[10, 11, 12]
>>> M[1][2]
```

SHELL

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]
>>> M[1]
[10, 11, 12]
>>> M[1][2]
12
```

SHELL

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]
>>> M[1]
[10, 11, 12]
>>> M[1][2]
12
```

SHELL

- ▶ Exercice 1 : Trouver le maximum d'une matrice

Une matrice est un tableau de nombres.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- ▶ On représente souvent les **matrices** comme une **liste de listes**

`[[a,b,c], [d,e,f], [g,h,i]]`

```
>>> M = [[0,1,2], [10,11,12], [20,21,22]]
>>> M[1]
[10, 11, 12]
>>> M[1][2]
12
```

SHELL

- ▶ Exercice 1 : Trouver le maximum d'une matrice
- ▶ Exercice 2 : Ajouter case par case deux matrices

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 50 & 60 \\ 70 & 80 \end{pmatrix} = \begin{pmatrix} 51 & 62 \\ 73 & 84 \end{pmatrix}$$



- 🍃 Partie I. Séquences
- 🍃 Partie II. Accès et constructeur
- 🍃 **Partie III. Mutabilité des listes**
- 🍃 Partie IV. Petits algorithmes
- 🍃 Partie V. Gestion de la mémoire
- 🍃 Partie VI. Table des matières

- ▶ On pose `liste=[11,22,33]` ; `s='Chaine'`, et `t = ('t', 'U', 'p', 'l', 'e')`

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>>
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaine'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaine'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>>
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>>
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'
```

SHELL



- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>>
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>> t[1]='u'
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>> t[1]='u'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

SHELL

```
>>>
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>> t[1]='u'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

SHELL

- ▶ On peut cependant réaffecter une chaîne et un tuple

```
>>> s='Chaîne'
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>> t[1]='u'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

SHELL

- ▶ On peut cependant réaffecter une chaîne et un tuple

```
>>> s='Chaîne'  
>>>
```

SHELL

- ▶ On pose `liste=[11,22,33]` ; `s='Chaîne'`, et `t = ('t', 'U', 'p', 'l', 'e')`
- ▶ On peut modifier les listes.

```
>>> liste[1] = 'Choucroute'  
>>> liste  
[11, 'Choucroute', 33]
```

SHELL

- ▶ On ne peut pas modifier les tuples et les chaînes

```
>>> s[3]='î'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'str' object does not support item assignment  
>>> t[1]='u'  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

SHELL

- ▶ On peut cependant réaffecter une chaîne et un tuple

```
>>> s='Chaîne'  
>>> t=('t', 'u', 'p', 'l', 'e')
```

SHELL

```
>>> c1 = 'Chaîne vachement très très longue'  
>>>
```

SHELL

```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL



```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>>
```

SHELL

variables



```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]
```

SHELL

variables



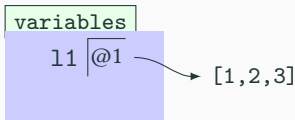
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>>
```

SHELL



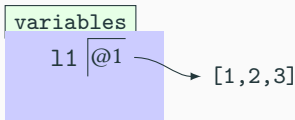
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1
```

SHELL



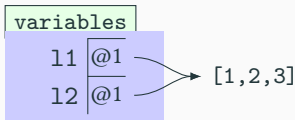
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>>
```

SHELL



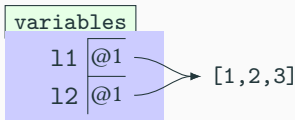
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>> l1[2]=4
```

SHELL



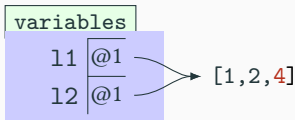
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>> l1[2]=4  
>>>
```

SHELL



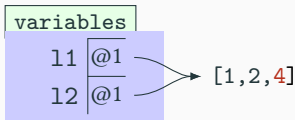
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>> l1[2]=4  
>>> l1      # l1 modifié : logique...
```

SHELL





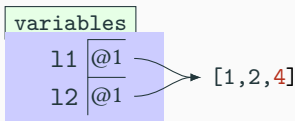
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>> l1[2]=4  
>>> l1 # l1 modifié : logique...  
[1, 2, 4]  
>>>
```

SHELL



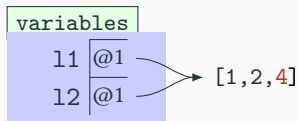
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>> l1[2]=4  
>>> l1      # l1 modifié : logique...  
[1, 2, 4]  
>>> l2      # Surprise !!!
```

SHELL



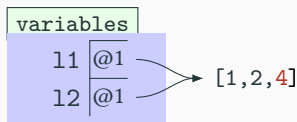
```
>>> c1 = 'Chaîne vachement très très longue'  
>>> c2=c1
```

SHELL

- ▶ La chaîne 'Chaîne ... longue' est stockée une fois en mémoire.
  - ▶ On dit que c1 et c2 pointent vers la même chaîne de caractères.
  - ▶ l'affectation c2=c1 est instantanée.

```
>>> l1=[1,2,3]  
>>> l2=l1  
>>> l1[2]=4  
>>> l1 # l1 modifié : logique...  
[1, 2, 4]  
>>> l2 # Surprise !!!  
[1, 2, 4]
```

SHELL



```
>>> L = [0,1,2]  
>>>
```

SHELL

```
>>> L = [0,1,2]  
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL



```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

▶ Le premier programme

- ▶ **recrée une liste similaire** à L (peut être long si L est grand)
- ▶ ajoute 3 à la fin de la nouvelle liste.

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas recopiée
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas recopiée
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>>
```

SHELL



```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>> l1=l1+[3] # On crée une nouvelle liste
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas recopiée
>>> m2=l2 # La liste n'est pas recopiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>> l2.append(3) # on modifie la liste
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>> l2.append(3) # on modifie la liste
>>>
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas recopiée
>>> m2=l2 # La liste n'est pas recopiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>> l2.append(3) # on modifie la liste
>>> print(l2,m2)
```

SHELL

```
>>> L = [0,1,2]
>>> L = L + [3]
>>> L
[0, 1, 2, 3]
```

SHELL

```
>>> L = [0,1,2]
>>> L.append(3)
>>> L
[0, 1, 2, 3]
```

SHELL

- ▶ Le premier programme
  - ▶ **recrée une liste similaire** à L (peut être long si L est grand)
  - ▶ ajoute 3 à la fin de la nouvelle liste.
- ▶ La méthode `append`
  - ▶ **modifie** la liste L directement.

```
>>> l1=[0,1,2] ; l2=[0,1,2]
>>> m1=l1 # La liste n'est pas copiée
>>> m2=l2 # La liste n'est pas copiée
>>> l1=l1+[3] # On crée une nouvelle liste
>>> print(l1,m1)
[0, 1, 2, 3] [0, 1, 2]
>>> l2.append(3) # on modifie la liste
>>> print(l2,m2)
[0, 1, 2, 3] [0, 1, 2, 3]
```

SHELL



- ▶ Python ne stocke pas des listes dans des variables (manque de place)

```
l1=[0,1,2]
l2=[0,1,2]
m1=l1
m2=l2
l1=l1+[3]
l2.append(3)
```

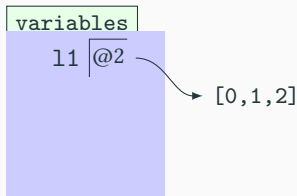
SCRIPT

variables

- ▶ Python ne stocke pas des listes dans des variables (manque de place)
- ▶ Python stocke les adresses en mémoire des listes.

```
11=[0,1,2]
12=[0,1,2]
m1=11
m2=12
11=11+[3]
12.append(3)
```

SCRIPT

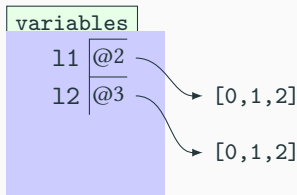


- ▶ 11 contient l'adresse @2 qui pointe vers une liste [0, 1, 2]

- ▶ Python ne stocke pas des listes dans des variables (manque de place)
- ▶ Python stocke les adresses en mémoire des listes.

```
11=[0,1,2]
12=[0,1,2]
m1=11
m2=12
11=11+[3]
12.append(3)
```

SCRIPT

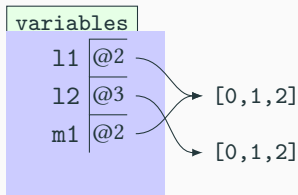


- ▶ 11 contient l'adresse @2 qui pointe vers une liste [0, 1, 2]
- ▶ Remarquons que :
  - ▶ 11≠12 : adresse différente

- ▶ Python ne stocke pas des listes dans des variables (manque de place)
- ▶ Python stocke les adresses en mémoire des listes.

```
11=[0,1,2]
12=[0,1,2]
m1=11
m2=12
11=11+[3]
12.append(3)
```

SCRIPT

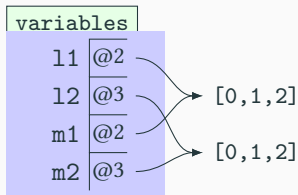


- ▶ 11 contient l'adresse @2 qui pointe vers une liste [0,1,2]
- ▶ Remarquons que :
  - ▶ 11≠12 : adresse différente
  - ▶ 11=m1 : même adresse

- ▶ Python ne stocke pas des listes dans des variables (manque de place)
- ▶ Python stocke les adresses en mémoire des listes.

```
11=[0,1,2]
12=[0,1,2]
m1=11
m2=12
11=11+[3]
12.append(3)
```

SCRIPT



- ▶ 11 contient l'adresse @2 qui pointe vers une liste [0,1,2]
- ▶ Remarquons que :
  - ▶ 11≠12 : adresse différente
  - ▶ 11=m1 : même adresse

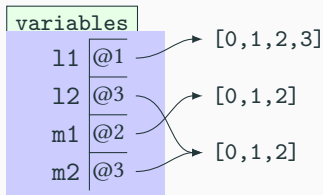
- ▶ Python ne stocke pas des listes dans des variables (manque de place)
- ▶ Python stocke les adresses en mémoire des listes.

```

11=[0,1,2]
12=[0,1,2]
m1=11
m2=12
11=11+[3]
12.append(3)

```

SCRIPT



- ▶ 11 contient l'adresse @2 qui pointe vers une liste [0, 1, 2]
- ▶ Remarquons que :
  - ▶ 11≠12 : adresse différente
  - ▶ 11=m1 : même adresse
- ▶ 11=... on modifie 11 (on remplace @2 par @1). Maintenant 11≠m1

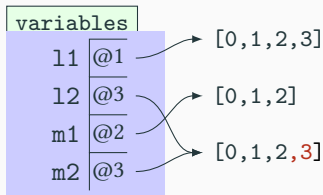
- ▶ Python ne stocke pas des listes dans des variables (manque de place)
- ▶ Python stocke les adresses en mémoire des listes.

```

11=[0,1,2]
12=[0,1,2]
m1=11
m2=12
11=11+[3]
12.append(3)

```

SCRIPT



- ▶ 11 contient l'adresse @2 qui pointe vers une liste [0, 1, 2]
- ▶ Remarquons que :
  - ▶ 11≠12 : adresse différente
  - ▶ 11=m1 : même adresse
- ▶ 11=... on modifie 11 (on remplace @2 par @1). Maintenant 11≠m1
- ▶ On ne modifie pas 12 (toujours égal à @3). On a toujours 12=m2

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de  $n$ .



- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de  $n$ .
  - ▶ uniquement ceux non triviaux (distincts de 1 et  $n$ )
  - ▶ on impose à  $n$  d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de  $n$ .
  - ▶ uniquement ceux non triviaux (distincts de 1 et  $n$ )
  - ▶ on impose à  $n$  d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat
  - ▶ On va donc utiliser une liste plutôt qu'un tuple

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat
  - ▶ On va donc utiliser une liste plutôt qu'un tuple

```
def diviseurs(n):  
    res = []  
    for d in range(2, n):  
        if n % d == 0 : # Si d divise n  
            res.append(d)  
    return res
```

SCRIPT

```
>>>
```

SHELL



- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat
  - ▶ On va donc utiliser une liste plutôt qu'un tuple

```
def diviseurs(n):  
    res = []  
    for d in range(2, n):  
        if n % d == 0 : # Si d divise n  
            res.append(d)  
    return res
```

SCRIPT

```
>>> d = diviseurs(500)
```

SHELL

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat
  - ▶ On va donc utiliser une liste plutôt qu'un tuple

```
def diviseurs(n):  
    res = []  
    for d in range(2, n):  
        if n % d == 0 : # Si d divise n  
            res.append(d)  
    return res
```

SCRIPT

```
>>> d = diviseurs(500)  
>>>
```

SHELL

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat
  - ▶ On va donc utiliser une liste plutôt qu'un tuple

```
def diviseurs(n):  
    res = []  
    for d in range(2, n):  
        if n % d == 0 : # Si d divise n  
            res.append(d)  
    return res
```

SCRIPT

```
>>> d = diviseurs(500)  
>>> (len(d) , d)
```

SHELL

- ▶ La fonction `diviseurs(n)` va retourner la liste des diviseurs de `n`.
  - ▶ uniquement ceux non triviaux (distincts de 1 et `n`)
  - ▶ on impose à `n` d'être un entier strictement positif.
- ▶ Liste ou tuple ?
  - ▶ On peut efficacement ajouter des éléments à une liste : `append`
  - ▶ Il n'y a pas d'équivalent pour les tuples (immutable/immuable)
  - ▶ On ne connaît pas à l'avance le nombre d'éléments du résultat
  - ▶ On va donc utiliser une liste plutôt qu'un tuple

```
def diviseurs(n):  
    res = []  
    for d in range(2, n):  
        if n % d == 0 : # Si d divise n  
            res.append(d)  
    return res
```

SCRIPT

```
>>> d = diviseurs(500)  
>>> (len(d), d)  
(10, [2, 4, 5, 10, 20, 25, 50, 100, 125, 250])
```

SHELL

<https://docs.python.org/fr/3.7/tutorial/datastructures.html>

<code>list.append(x)</code>	Ajoute un élément à la fin de la liste.
<code>list.extend(L)</code>	Étend la liste en y ajoutant tous les éléments de l'itérable.
<code>list.insert(i, x)</code>	Insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer.
<code>list.remove(x)</code>	Supprime de la liste le premier élément dont la valeur est égale à <code>x</code> . Erreur s'il n'existe aucun élément avec cette valeur.
<code>list.pop(i)</code>	Enlève de la liste l'élément situé à la position indiquée et le renvoie en valeur de retour. Si aucune position n'est spécifiée, <code>a.pop()</code> enlève et renvoie le dernier élément de la liste.
<code>list.index(x)</code>	Renvoie la position du premier élément de la liste dont la valeur égale <code>x</code> . Erreur si aucun élément n'est trouvé.
<code>list.count(x)</code>	Renvoie le nombre d'éléments ayant la valeur <code>x</code> dans la liste.
<code>list.sort()</code>	Ordonne les éléments dans la liste, en place.
<code>list.reverse()</code>	Inverse l'ordre des éléments dans la liste, en place.

► Exercices : à réécrire chez soi :)

- 🍃 Partie I. Séquences
- 🍃 Partie II. Accès et constructeur
- 🍃 Partie III. Mutabilité des listes
- 🍃 **Partie IV. Petits algorithmes**
- 🍃 Partie V. Gestion de la mémoire
- 🍃 Partie VI. Table des matières

- ▶ Calculons la liste des chiffres d'une chaîne  $s$ .

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.



- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ **Vous devez être capable** d'écrire `isdigit!` (*cf.* TP 3)

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ **Vous devez être capable** d'écrire `isdigit()`! (cf. TP 3)

```
>>>
```

```
SHELL
```

```
>>>
```

```
SHELL
```

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ Vous devez être capable d'écrire `isdigit()`! (cf. TP 3)

```
>>> '456'.isdigit() SHELL
```

```
>>> '45a6'.isdigit() SHELL
```

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ Vous devez être capable d'écrire `isdigit()`! (cf. TP 3)

```
>>> '456'.isdigit() SHELL  
True
```

```
>>> '45a6'.isdigit() SHELL  
False
```

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ Vous devez être capable d'écrire `isdigit()`! (cf. TP 3)

```
>>> '456'.isdigit() SHELL  
True
```

```
>>> '45a6'.isdigit() SHELL  
False
```

- ▶ Méthode classique avec boucle `for`.

```
def chiffres(chaîne):  
    liste = []  
    for c in chaîne:  
        if c.isdigit():  
            liste.append(c)  
    return liste
```

SCRIPT

```
>>>
```

SHELL

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ Vous devez être capable d'écrire `isdigit()`! (cf. TP 3)

```
>>> '456'.isdigit() SHELL  
True
```

```
>>> '45a6'.isdigit() SHELL  
False
```

- ▶ Méthode classique avec boucle `for`.

```
def chiffres(chaîne):  
    liste = []  
    for c in chaîne:  
        if c.isdigit():  
            liste.append(c)  
    return liste
```

SCRIPT

```
>>> chiffres('0livier E5T 2314 fois 5YMP4')
```

SHELL

- ▶ Calculons la liste des chiffres d'une chaîne `s`.
  - ▶ Nous utilisons la méthode `isdigit()` de la classe `str`.
  - ▶ Vous devez être capable d'écrire `isdigit()`! (cf. TP 3)

```
>>> '456'.isdigit() SHELL  
True
```

```
>>> '45a6'.isdigit() SHELL  
False
```

- ▶ Méthode classique avec boucle `for`.

```
def chiffres(chaîne):  
    liste = []  
    for c in chaîne:  
        if c.isdigit():  
            liste.append(c)  
    return liste
```

SCRIPT

```
>>> chiffres('0livier E5T 2314 fois 5YMP4')  
['0', '1', '5', '2', '3', '4', '5', '4']
```

SHELL

- ▶ Méthode Pythonique : compréhension de liste.

```
def chiffres(s):  
    return [c for c in s if c.isdigit()]
```

SCRIPT



- ▶ Méthode Pythonique : compréhension de liste.

```
def chiffres(s):  
    return [c for c in s if c.isdigit()]
```

SCRIPT

- ▶ Notez l'ordre des instructions :
  - ▶ D'abord le `for` puis le `if`
  - ▶ Comme dans le programme précédent.

- ▶ Méthode Pythonesque : compréhension de liste.

```
def chiffres(s):  
    return [c for c in s if c.isdigit()]
```

SCRIPT

- ▶ Notez l'ordre des instructions :
  - ▶ D'abord le `for` puis le `if`
  - ▶ Comme dans le programme précédent.
- ▶ Ne marche pas (forcément) avec les autres langages.

- ▶ Méthode Pythonesque : compréhension de liste.

```
def chiffres(s):  
    return [c for c in s if c.isdigit()]
```

SCRIPT

- ▶ Notez l'ordre des instructions :
  - ▶ D'abord le `for` puis le `if`
  - ▶ Comme dans le programme précédent.
- ▶ Ne marche pas (forcément) avec les autres langages.
- ▶ **Durant cette UE** privilégiez la méthode classique.

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶  $d$  divise  $n$  si et seulement si  $n \% d == 0$ ,

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶  $d$  divise  $n$  si et seulement si  $n \% d == 0$ ,
  - ▶  $n$  non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶  $d$  divise  $n$  si et seulement si  $n\%d == 0$ ,
  - ▶  $n$  non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SCRIPT

>>>

SHELL



- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶ d divise n si et seulement si  $n \% d == 0$ ,
  - ▶ n non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

SCRIPT

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SHELL

```
>>> ppdiv(2003) # premier
```

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶  $d$  divise  $n$  si et seulement si  $n\%d == 0$ ,
  - ▶  $n$  non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

SCRIPT

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SHELL

```
>>> ppdiv(2003) # premier  
2003  
>>>
```

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶  $d$  divise  $n$  si et seulement si  $n\%d == 0$ ,
  - ▶  $n$  non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SCRIPT

```
>>> ppdiv(2003) # premier  
2003  
>>> ppdiv(1003) # non premier
```

SHELL

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶  $d$  divise  $n$  si et seulement si  $n\%d == 0$ ,
  - ▶  $n$  non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SCRIPT

```
>>> ppdiv(2003) # premier  
2003  
>>> ppdiv(1003) # non premier  
17  
>>>
```

SHELL

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶ d divise n si et seulement si  $n\%d == 0$ ,
  - ▶ n non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

SCRIPT

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SHELL

```
>>> ppdiv(2003) # premier  
2003  
>>> ppdiv(1003) # non premier  
17  
>>> [(c,ppdiv(c)) for c in range(15,20)]
```

- ▶ Principes : (Déjà vu dans un cours précédent : Cours 2)
  - ▶ on traite les nombres pairs (divisibles par 2) à part,
  - ▶ d divise n si et seulement si  $n\%d == 0$ ,
  - ▶ n non premier admet nécessairement un diviseur  $d \leq \sqrt{n}$

SCRIPT

```
def ppdiv(n):  
    if n%2==0:  
        return 2  
    racine = int(sqrt(n))  
    for d in range(3,racine+1,2):  
        if n%d==0:  
            return d  
    return n
```

SHELL

```
>>> ppdiv(2003) # premier  
2003  
>>> ppdiv(1003) # non premier  
17  
>>> [(c,ppdiv(c)) for c in range(15,20)]  
[(15, 3), (16, 2), (17, 17), (18, 2), (19, 19)]
```

- ▶ Il est facile de tester si un nombre est premier en utilisant ppdiv...

```
def estPremier(n):  
    if n<2:  
        return False  
    else:  
        return ppdiv(n)==n
```

SCRIPT

- ▶ Il est facile de tester si un nombre est premier en utilisant `ppdiv`...

```
def estPremier(n):  
    if n<2:  
        return False  
    else:  
        return ppdiv(n)==n
```

SCRIPT

- ▶ ... puis de construire la liste des nombres premiers jusqu'à `n` :

```
def premiers(n):  
    liste = []  
    for p in range(n + 1):  
        if estPremier(p):  
            liste.append(p)  
    return liste
```

SCRIPT

&gt;&gt;&gt;

SHELL



- ▶ Il est facile de tester si un nombre est premier en utilisant `ppdiv`...

```
def estPremier(n):  
    if n<2:  
        return False  
    else:  
        return ppdiv(n)==n
```

SCRIPT

- ▶ ... puis de construire la liste des nombres premiers jusqu'à `n` :

```
def premiers(n):  
    liste = []  
    for p in range(n + 1):  
        if estPremier(p):  
            liste.append(p)  
    return liste
```

SCRIPT

```
>>> premiers(50)
```

SHELL

- Il est facile de tester si un nombre est premier en utilisant `ppdiv`...

```
def estPremier(n):  
    if n<2:  
        return False  
    else:  
        return ppdiv(n)==n
```

SCRIPT

- ... puis de construire la liste des nombres premiers jusqu'à `n` :

```
def premiers(n):  
    liste = []  
    for p in range(n + 1):  
        if estPremier(p):  
            liste.append(p)  
    return liste
```

SCRIPT

```
>>> premiers(50)  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

SHELL

- ▶ Il est facile de tester si un nombre est premier en utilisant `ppdiv`...

```
def estPremier(n):  
    if n<2:  
        return False  
    else:  
        return ppdiv(n)==n
```

SCRIPT

- ▶ ... puis de construire la liste des nombres premiers jusqu'à `n` :

```
def premiers(n):  
    liste = []  
    for p in range(n + 1):  
        if estPremier(p):  
            liste.append(p)  
    return liste
```

SCRIPT

```
>>> premiers(50)  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

SHELL

- ▶ Exercice : écrire cette fonction en une ligne (par compréhension)

- ▶ Python possède deux primitives pour trier une liste  $L$  de nombres.

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>>
```

SHELL

```
>>>
```

SHELL

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL
```



- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL  
>>>
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL  
>>>
```

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL  
>>> sorted(L1)
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL  
>>> L2.sort()
```

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>>
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL
>>> L2.sort()
>>>
```

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>> L1
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL
>>> L2.sort()
>>> L2
```

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>> L1
[3, 1, 6, 5, 2]
>>>
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL
>>> L2.sort()
>>> L2
[1, 2, 3, 5, 6]
>>>
```

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>> L1
[3, 1, 6, 5, 2]
>>> L1 == sorted(L1)
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL
>>> L2.sort()
>>> L2
[1, 2, 3, 5, 6]
>>> L2 == sorted(L2)
```

- ▶ Python possède deux primitives pour trier une liste L de nombres.
  - ▶ la fonction `sorted(L)` qui construit une nouvelle copie de L
  - ▶ La méthode `L.sort()` qui trie la liste L en place (sans créer de nouvelle liste, en modifiant le contenu de L).

```
>>> L1 = [3, 1, 6, 5, 2] SHELL
>>> sorted(L1)
[1, 2, 3, 5, 6]
>>> L1
[3, 1, 6, 5, 2]
>>> L1 == sorted(L1)
False
```

```
>>> L2 = [3, 1, 6, 5, 2] SHELL
>>> L2.sort()
>>> L2
[1, 2, 3, 5, 6]
>>> L2 == sorted(L2)
True
```

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ .



- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

&gt;&gt;&gt;

SHELL

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

```
>>> chrono(200000)
```

SHELL

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

```
>>> chrono(200000)
Temps pour 200000 24 ms
>>>
```

SHELL

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

```
>>> chrono(200000)
Temps pour 200000 24 ms
>>> chrono(400000)
```

SHELL

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

```
>>> chrono(200000)
Temps pour 200000 24 ms
>>> chrono(400000)
Temps pour 400000 51 ms
```

SHELL

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

```
>>> chrono(200000)
Temps pour 200000 24 ms
>>> chrono(400000)
Temps pour 400000 51 ms
```

SHELL

- ▶ En doublant  $n$ , on fait un peu plus que doubler le temps de calcul

- ▶ Le temps de calcul de cet algorithme pour trier une liste à  $n$  éléments est proportionnel à  $n \cdot \log(n)$ . Petite expérience :

```
from time import time
from random import randint

def chrono(n):
    # On crée une liste de n nombres aléatoires
    L = [randint(1,1000) for i in range(n)]
    t = time()      # On lance le chrono
    L.sort()       # On fait le tri
    t = time() - t # On arrête le chrono
    print('Temps pour',n,':',int(t*1000),'ms')
```

SCRIPT

```
>>> chrono(200000)
Temps pour 200000 24 ms
>>> chrono(400000)
Temps pour 400000 51 ms
```

SHELL

- ▶ En doublant  $n$ , on fait un peu plus que doubler le temps de calcul
  - ▶ ce qui est cohérent avec «  $n \cdot \log(n)$  »

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.



- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[5,6,7,2,4]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[ 5,6,7,2,4 ]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[ 5,6,7,2,4 ]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[ 2,6,7,5,4 ]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2, 6, 7, 5, 4]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2, 6, 7, 5, 4]



- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2, 4, 7, 5, 6]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4, 7,5,6]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4, 7,5,6]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4, 5,7,6]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4,5, 7,6]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4,5, 7,6]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4,5, 6,7]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4,5,6, 7]



- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4,5,6,7]

- ▶ Il existe de nombreux algorithmes de tri.
  - ▶ Tri rapide, tri fusion, tri par tas, du sleep, etc.
  - ▶ les meilleurs sont en  $n \cdot \log(n)$
  - ▶ Le tri par sélection est proportionnel à  $n^2$
- ▶ Concrètement

[2,4,5,6,7]

[ 5,6,7,2,4]

[ 2,6,7,5,4]

[2, 6,7,5,4]

[2, 4,7,5,6]

[2,4, 7,5,6]

[2,4, 5,7,6]

[2,4,5, 7,6]

[2,4,5, 6,7]

[2,4,5,6, 7]

- ▶ `imin` trouve le minimum à partir de `i` et renvoie son indice

```
def imin(L, i):  
    m = i  
    for j in range(i + 1, len(L)): # i+1 ... len(L)-1  
        if L[j] < L[m]:  
            m = j  
    return m
```

SCRIPT

- ▶ `imin` trouve le minimum à partir de `i` et renvoie son indice

```
def imin(L, i):  
    m = i  
    for j in range(i + 1, len(L)): # i+1 ... len(L)-1  
        if L[j] < L[m]:  
            m = j  
    return m
```

SCRIPT

- ▶ L'algorithme de tri par sélection

```
def tri(L):  
    n = len(L)  
    for i in range(n-1):  
        m = imin(L,i) # minimum de la fin de liste  
        (L[i], L[m]) = (L[m], L[i]) # échange  
        print('L=', L) # pour espionner la boucle
```

SCRIPT

- ▶ À la fin de chaque étape les `i` premiers éléments sont triés.

- ▶ `imin` trouve le minimum à partir de `i` et renvoie son indice

```
def imin(L, i):  
    m = i  
    for j in range(i + 1, len(L)): # i+1 ... len(L)-1  
        if L[j] < L[m]:  
            m = j  
    return m
```

SCRIPT

- ▶ L'algorithme de tri par sélection

```
def tri(L):  
    n = len(L)  
    for i in range(n-1):  
        m = imin(L,i) # minimum de la fin de liste  
        (L[i], L[m]) = (L[m], L[i]) # échange  
    print('L=', L) # pour espionner la boucle
```

SCRIPT

- ▶ À la fin de chaque étape les `i` premiers éléments sont triés.
- ▶ Par super efficace, mais ça marche !

```
>>> L = [4,2,7,11,2,1]
```

SHELL

```
>>> L = [4,2,7,11,2,1]  
>>>
```

SHELL

```
>>> L = [4,2,7,11,2,1]  
>>> tri(L)
```

SHELL



```
>>> L = [4,2,7,11,2,1]
>>> tri(L)
L = [1, 2, 7, 11, 2, 4]
L = [1, 2, 7, 11, 2, 4]
L = [1, 2, 2, 11, 7, 4]
L = [1, 2, 2, 4, 7, 11]
L = [1, 2, 2, 4, 7, 11]
>>>
```

SHELL

SCRIPT

```
def chrono2(f,n):  
    # On crée une liste de n nombres aléatoires  
    L = [randint(1,1000) for i in range(n)]  
    t = time() ; f(L) ; t = time() - t  
    print(f.__name__, ':', int(t*1000), 'ms')
```

SHELL

```
>>> L = [4,2,7,11,2,1]  
>>> tri(L)  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 2, 11, 7, 4]  
L = [1, 2, 2, 4, 7, 11]  
L = [1, 2, 2, 4, 7, 11]  
>>> chrono2(sorted,10000)
```

SCRIPT

```
def chrono2(f,n):  
    # On crée une liste de n nombres aléatoires  
    L = [randint(1,1000) for i in range(n)]  
    t = time() ; f(L) ; t = time() - t  
    print(f.__name__, ':', int(t*1000), 'ms')
```

SHELL

```
>>> L = [4,2,7,11,2,1]  
>>> tri(L)  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 2, 11, 7, 4]  
L = [1, 2, 2, 4, 7, 11]  
L = [1, 2, 2, 4, 7, 11]  
>>> chrono2(sorted,10000)  
sorted : 2 ms  
>>>
```

```
def chrono2(f,n):  
    # On crée une liste de n nombres aléatoires  
    L = [randint(1,1000) for i in range(n)]  
    t = time() ; f(L) ; t = time() - t  
    print(f.__name__, ':', int(t*1000), 'ms')
```

SCRIPT

```
>>> L = [4,2,7,11,2,1]  
>>> tri(L)  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 2, 11, 7, 4]  
L = [1, 2, 2, 4, 7, 11]  
L = [1, 2, 2, 4, 7, 11]  
>>> chrono2(sorted,10000)  
sorted : 2 ms  
>>> chrono2(tri,10000) # Après avoir enlevé les prints
```

SHELL

```
def chrono2(f,n):  
    # On crée une liste de n nombres aléatoires  
    L = [randint(1,1000) for i in range(n)]  
    t = time() ; f(L) ; t = time() - t  
    print(f.__name__, ':', int(t*1000), 'ms')
```

SCRIPT

```
>>> L = [4,2,7,11,2,1]  
>>> tri(L)  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 7, 11, 2, 4]  
L = [1, 2, 2, 11, 7, 4]  
L = [1, 2, 2, 4, 7, 11]  
L = [1, 2, 2, 4, 7, 11]  
>>> chrono2(sorted,10000)  
sorted : 2 ms  
>>> chrono2(tri,10000) # Après avoir enlevé les prints  
tri : 1818 ms
```

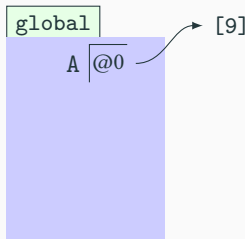
SHELL

- 🍃 Partie I. Séquences
- 🍃 Partie II. Accès et constructeur
- 🍃 Partie III. Mutabilité des listes
- 🍃 Partie IV. Petits algorithmes
- 🍃 **Partie V. Gestion de la mémoire**
- 🍃 Partie VI. Table des matières

- ▶ Une simple affectation (L=M) copie simplement le pointeur.
  - ▶ Avantage : rapide
  - ▶ Inconvénient : dangereux et ne permet pas de copier.
- ▶ Il existe une méthode `M.copy()` pour copier la liste.
- ▶ Il existe aussi une méthode `M.deepcopy()` pour la copier en profondeur.

```
from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant
```

SCRIPT

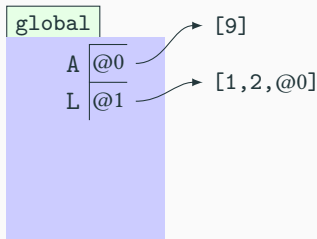


- ▶ À savoir écrire vous même !

- ▶ Une simple affectation (L=M) copie simplement le pointeur.
  - ▶ Avantage : rapide
  - ▶ Inconvénient : dangereux et ne permet pas de copier.
- ▶ Il existe une méthode M.copy() pour copier la liste.
- ▶ Il existe aussi une méthode M.deepcopy() pour la copier en profondeur.

```
from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant
```

SCRIPT



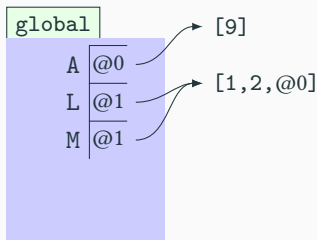
- ▶ À savoir écrire vous même !



- ▶ Une simple affectation ( $L=M$ ) copie simplement le pointeur.
  - ▶ Avantage : rapide
  - ▶ Inconvénient : dangereux et ne permet pas de copier.
- ▶ Il existe une méthode `M.copy()` pour copier la liste.
- ▶ Il existe aussi une méthode `M.deepcopy()` pour la copier en profondeur.

```
from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant
```

SCRIPT



- ▶ À savoir écrire vous même !

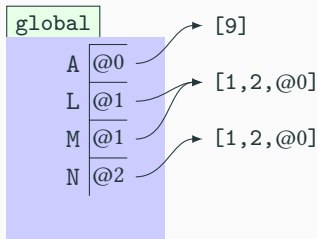
- ▶ Une simple affectation (L=M) copie simplement le pointeur.
  - ▶ Avantage : rapide
  - ▶ Inconvénient : dangereux et ne permet pas de copier.
- ▶ Il existe une méthode M.copy() pour copier la liste.
- ▶ Il existe aussi une méthode M.deepcopy() pour la copier en profondeur.

```

from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant

```

SCRIPT



- ▶ À savoir écrire vous même !

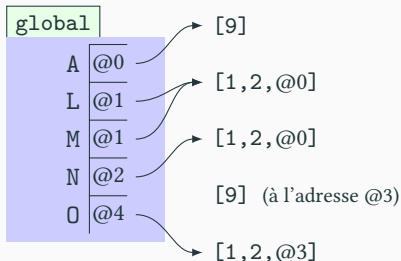
- ▶ Une simple affectation (L=M) copie simplement le pointeur.
  - ▶ Avantage : rapide
  - ▶ Inconvénient : dangereux et ne permet pas de copier.
- ▶ Il existe une méthode M.copy() pour copier la liste.
- ▶ Il existe aussi une méthode M.deepcopy() pour la copier en profondeur.

```

from copy import copy
from copy import deepcopy
A=[9]
L = [1, 2, A]
M = L
N = copy(L)
# Si je modifie A, je modifie
# aussi M et N !
O = deepcopy(L)
# O est vraiment indépendant

```

SCRIPT



- ▶ À savoir écrire vous même !

- ▶ Que signifie L et M sont la même liste ?

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>>
```

```
SHELL
```

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le **même contenu**.
  - ▶ L'identité testée avec `is` : les listes **pointent au même endroit** mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]  
>>>
```

SHELL



- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>>
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
True True
>>>
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
True True
>>> print(M is N , M==N) # Non identiques mais égales
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

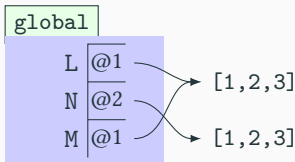
```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
True True
>>> print(M is N , M==N) # Non identiques mais égales
False True
```

SHELL

- ▶ Que signifie L et M sont la même liste ?
- ▶ À ne pas confondre :
  - ▶ L'égalité testée avec `==` : les listes ont le même contenu.
  - ▶ L'identité testée avec `is` : les listes pointent au même endroit mémoire.

```
>>> L = [1,2,3] ; N = [1,2,3]
>>> M = L
>>> print(M is L , M==L) # M et L sont identiques
True True
>>> print(M is N , M==N) # Non identiques mais égales
False True
```

SHELL



- ▶ Exercice : écrire une fonction `copy(L)` qui retourne une liste égale mais distincte de L.

```
L = [1,2]
M = L
L.append(3)
L = L + [4]
```

SCRIPT

global





SCRIPT

```
L = [1,2]  
M = L  
L.append(3)  
L = L + [4]
```

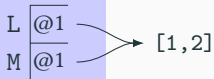
global

L |@1 → [1,2]

```
L = [1,2]
M = L
L.append(3)
L = L + [4]
```

SCRIPT

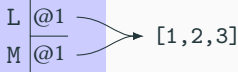
global



```
L = [1,2]
M = L
L.append(3)
L = L + [4]
```

SCRIPT

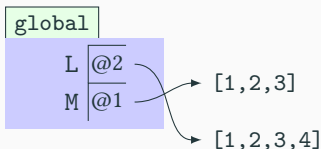
global



- ▶ L'appel de `L.append(3)` **prolonge la liste**
  - ▶ Ne modifie pas L (toujours égale à @1), seulement le contenu de la liste.
  - ▶ est très rapide
  - ▶ a un effet global!

```
L = [1,2]
M = L
L.append(3)
L = L + [4]
```

SCRIPT



- ▶ L'appel de `L.append(3)` **prolonge la liste**
  - ▶ Ne modifie pas L (toujours égale à @1), seulement le contenu de la liste.
  - ▶ est très rapide
  - ▶ a un effet global!
  
- ▶ Le calcul de `L+[4]` **crée une nouvelle liste**
  - ▶ Peut être très long si L contient beaucoup d'éléments
  - ▶ Sécurisé mais souvent inutile
  - ▶ Crée un nouvel objet en mémoire

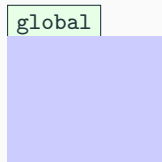
- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```

SCRIPT



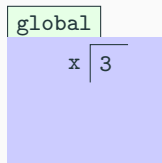
- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```

SCRIPT



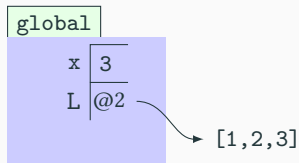
- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```

SCRIPT



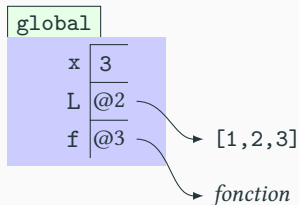
- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```

SCRIPT





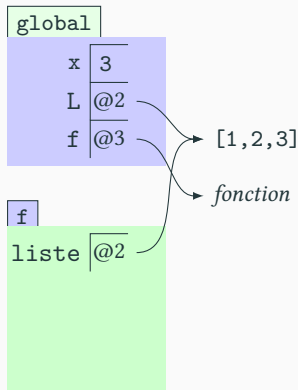
- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```

SCRIPT

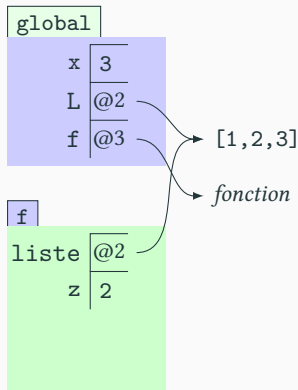


- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
SCRIPT
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```

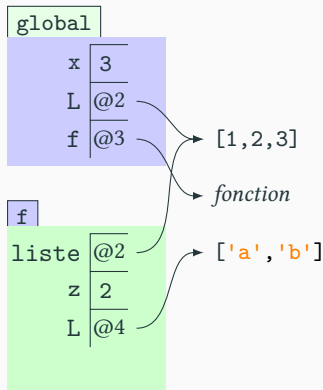


- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```
SCRIPT
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)
```



- ▶ Pour simplifier, il y a trois endroits où Python stocke des informations.
  - ▶ La mémoire des variables globales.
  - ▶ La mémoire des variables locales communément appelée pile
  - ▶ La mémoire des objets (liste, fonction, etc) communément appelée tas
  - ▶ Une variable peut contenir un lien vers un objet du tas.

```

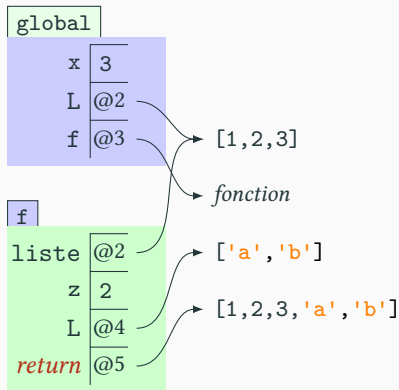
x=3
L=[1,2,3]

def f(liste):
    z=2
    L=['a','b']
    return liste+L

f(L)

```

SCRIPT



```
def f(x):  
    y = 1  
    g(y)  
    print(x)
```

```
def g(y):  
    x = y
```

```
f(3) # affiche 3
```

SCRIPT

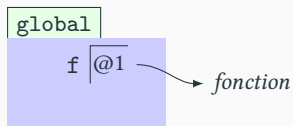
global



```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

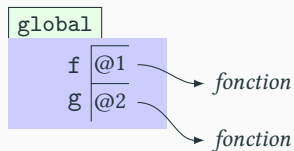
- ▶ On définit **f**



```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

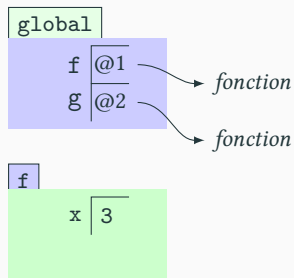
- ▶ On définit **f**
- ▶ On définit **g**



```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

- ▶ On définit **f**
- ▶ On définit **g**
- ▶ On appelle **f(3)**

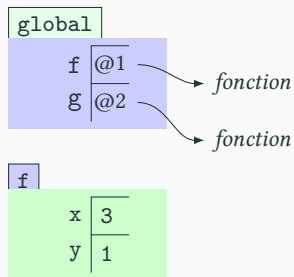




```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

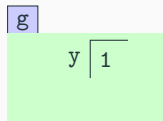
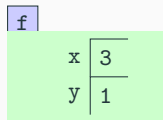
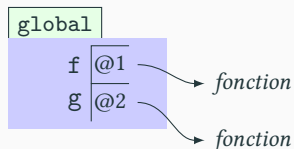
- ▶ On définit **f**
- ▶ On définit **g**
- ▶ On appelle **f(3)**



```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

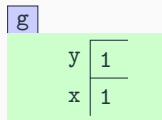
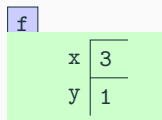
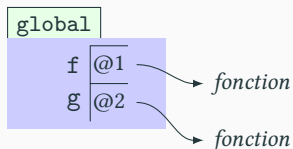
- ▶ On définit **f**
- ▶ On définit **g**
- ▶ On appelle **f(3)**
- ▶ On appelle **g(y)** dans **f**



```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

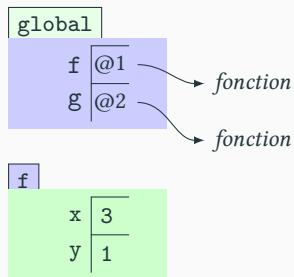
- ▶ On définit **f**
- ▶ On définit **g**
- ▶ On appelle **f(3)**
- ▶ On appelle **g(y)** dans **f**



```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

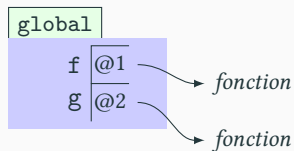
- ▶ On définit **f**
- ▶ On définit **g**
- ▶ On appelle **f(3)**
- ▶ On appelle **g(y)** dans **f**
- ▶ **g** se termine, on libère sa mémoire



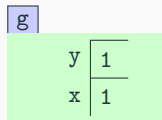
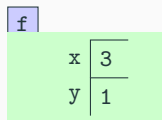
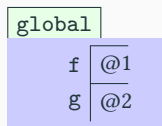
```
def f(x):  
    y = 1  
    g(y)  
    print(x)  
  
def g(y):  
    x = y  
  
f(3) # affiche 3
```

SCRIPT

- ▶ On définit **f**
- ▶ On définit **g**
- ▶ On appelle **f(3)**
- ▶ On appelle **g(y)** dans **f**
- ▶ **g** se termine, on libère sa mémoire
- ▶ **f** se termine, on libère sa mémoire

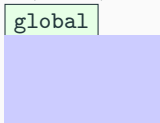


- ▶ Chaque appel de fonction ajoute un cadre sur la pile.
  - ▶ Les cadres s'empilent, ils forment une pile.
  - ▶ Il ne dure que le temps de l'appel.
  - ▶ Ensuite, il disparaît avec ses variables locales.
- ▶ Chaque appel de fonction a son propre **espace de nom**.
  - ▶ Le x de f n'est pas le même que celui de g
  - ▶ Cela permet de ne pas s'embrouiller
- ▶ Si on appelle plusieurs fois une fonction, il y aura **un cadre par appel**.



- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument  $g(z)$
- ▶ par valeur de retour `return x+x`



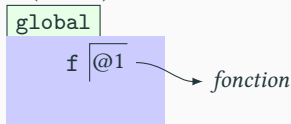
```
def f(x):  
    z=[x+2]  
    y=g(z)  
  
def g(x):  
    return x+x  
  
f(5)
```

SCRIPT

C'est la manière propre

- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument  $g(z)$
- ▶ par valeur de retour `return x+x`



```
def f(x):  
    z=[x+2]  
    y=g(z)  
  
def g(x):  
    return x+x  
  
f(5)
```

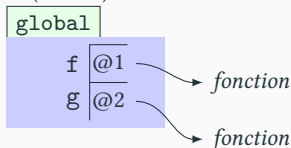
SCRIPT

C'est la manière propre



- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument  $g(z)$
- ▶ par valeur de retour `return x+x`



```
def f(x):  
    z=[x+2]  
    y=g(z)  
  
def g(x):  
    return x+x  
  
f(5)
```

SCRIPT

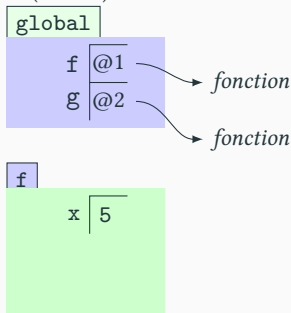
C'est la manière propre

- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument  $g(z)$
- ▶ par valeur de retour `return x+x`

```
def f(x):  
    z=[x+2]  
    y=g(z)  
  
def g(x):  
    return x+x  
  
f(5)
```

SCRIPT



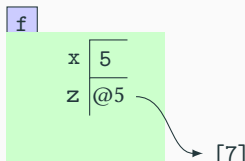
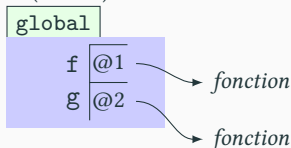
C'est la manière propre

- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument  $g(z)$
- ▶ par valeur de retour `return x+x`

```
def f(x):  
    z=[x+2]  
    y=g(z)  
  
def g(x):  
    return x+x  
  
f(5)
```

SCRIPT



C'est la manière propre

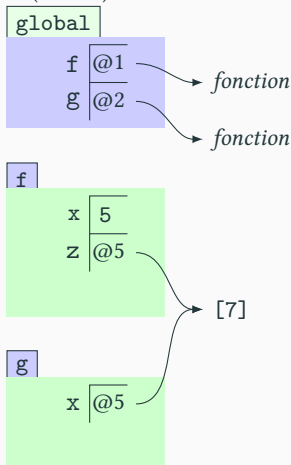
- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument `g(z)`
- ▶ par valeur de retour `return x+x`

```
def f(x):  
    z=[x+2]  
    y=g(z)  
  
def g(x):  
    return x+x  
  
f(5)
```

SCRIPT

C'est la manière propre



- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument `g(z)`
- ▶ par valeur de retour `return x+x`

```

def f(x):
    z=[x+2]
    y=g(z)

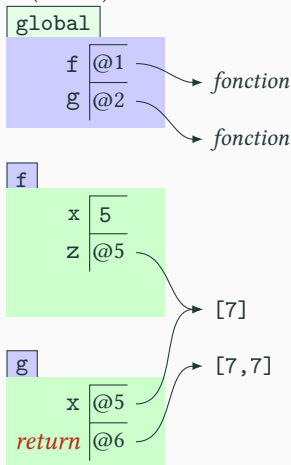
def g(x):
    return x+x

f(5)

```

SCRIPT

C'est la manière propre



- ▶ Il est impossible de partager des variables locales entre fonctions
- ▶ Mais on peut se transmettre des valeurs (du tas) entre fonctions :

- ▶ par passage d'argument `g(z)`
- ▶ par valeur de retour `return x+x`

```

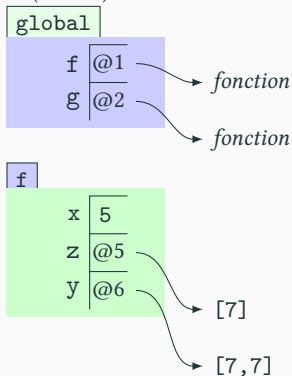
def f(x):
    z=[x+2]
    y=g(z)

def g(x):
    return x+x

f(5)

```

SCRIPT



C'est la manière propre

- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

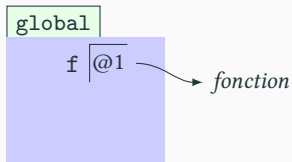
SCRIPT

global

- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

SCRIPT

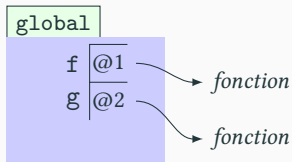




- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

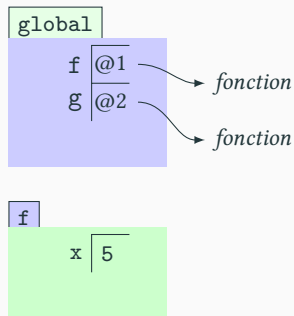
SCRIPT



- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

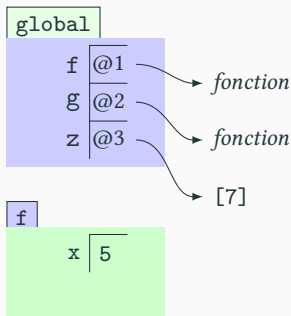
SCRIPT



- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

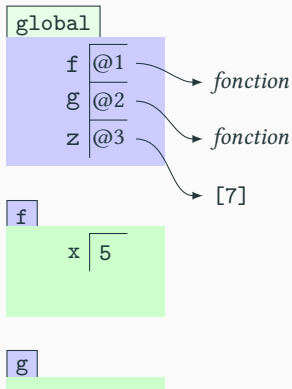
SCRIPT



- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

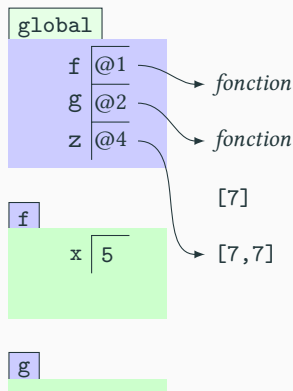
SCRIPT



- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

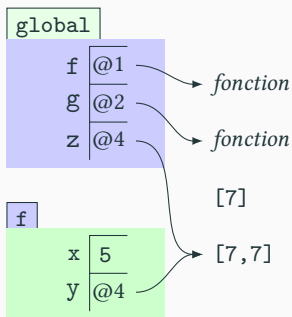
SCRIPT



- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

```
def f(x):  
    global z  
    z=[x+2]  
    g()  
    y=z  
  
def g():  
    global z  
    z=z+z  
  
f(5)
```

SCRIPT



- ▶ On peut aussi utiliser des variables globales (cours 4)
- ▶ Ici, f et g vont modifier la variable globale z.

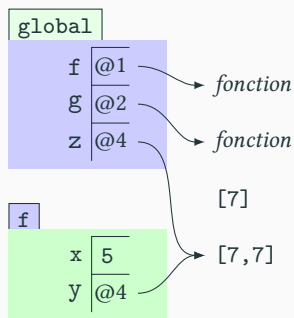
```

def f(x):
    global z
    z=[x+2]
    g()
    y=z

def g():
    global z
    z=z+z

f(5)
  
```

SCRIPT



C'est la manière (très!) sale  
#balancetongoret

- ▶ **Attention** : le contenu d'une liste est **toujours modifiable globalement**.



- ▶ **Attention** : le contenu d'une liste est **toujours modifiable globalement**.
  - ▶ La variable L contient un lien vers la liste [1, 2, 3]

- ▶ **Attention** : le contenu d'une liste est **toujours modifiable globalement**.
  - ▶ La variable L contient un lien vers la liste [1, 2, 3]
  - ▶ **Ce lien n'est pas modifiable** (par défaut) par une fonction (variable locale).

- ▶ **Attention** : le contenu d'une liste est **toujours modifiable globalement**.
  - ▶ La variable L contient un lien vers la liste [1, 2, 3]
  - ▶ **Ce lien n'est pas modifiable** (par défaut) par une fonction (variable locale).
  - ▶ **Le contenu** pointé par le lien est toujours modifiable.

- ▶ **Attention** : le contenu d'une liste est **toujours modifiable globalement**.
  - ▶ La variable L contient un lien vers la liste [1, 2, 3]
  - ▶ **Ce lien n'est pas modifiable** (par défaut) par une fonction (variable locale).
  - ▶ **Le contenu** pointé par le lien est toujours modifiable.

```
def swap(i,j,L):  
    (L[i], L[j]) = (L[j], L[i])  
    i = i + 1  
  
i = 1  
M = [1, 2, 3, 4, 5]  
  
swap(i,3,M)  
print(M) # affiche [1, 4, 3, 2, 5]  
print(i) # affiche 1
```

SCRIPT

- ▶ **Attention** : le contenu d'une liste est **toujours modifiable globalement**.
  - ▶ La variable L contient un lien vers la liste [1, 2, 3]
  - ▶ Ce lien **n'est pas modifiable** (par défaut) par une fonction (variable locale).
  - ▶ Le **contenu** pointé par le lien est toujours modifiable.

```

def swap(i,j,L):
    (L[i], L[j]) = (L[j], L[i])
    i = i + 1

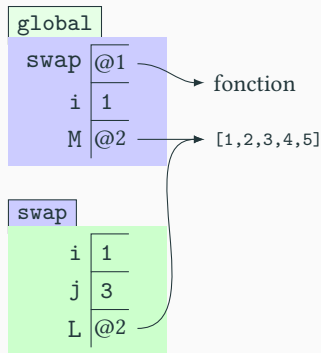
i = 1
M = [1, 2, 3, 4, 5]

swap(i,3,M)
print(M) # affiche [1, 4, 3, 2, 5]
print(i) # affiche 1

```

SCRIPT

- ▶ Pas de **global** M dans la fonction **swap**
- ▶ Donc M n'est pas modifié (pointe toujours vers la même liste)
- ▶ Mais le contenu de cette liste a changé!



```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>>
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)
```

```
>>>
```

SHELL



```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)
```

```
>>> (p.x,p.y)
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)  
>>> (p.x,p.y)  
(0, 0)  
>>>
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)  
>>> (p.x,p.y)  
(0, 0)  
>>> translation(p)
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)  
>>> (p.x,p.y)  
(0, 0)  
>>> translation(p)  
>>>
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)  
>>> (p.x,p.y)  
(0, 0)  
>>> translation(p)  
>>> (p.x,p.y)
```

SHELL

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)  
>>> (p.x,p.y)  
(0, 0)  
>>> translation(p)  
>>> (p.x,p.y)  
(20, 30)
```

SHELL

- ▶ Techniquement, on ne modifie pas la variable p (contenant le pointeur)...

```
class Point:
```

SCRIPT

```
    def __init__(self,x,y):  
        self.x = x  
        self.y = y
```

```
def translation(q): # q n'est pas une variable globale  
    q.x = q.x + 20  
    q.y = q.y + 30
```

```
>>> p = Point(0,0)  
>>> (p.x,p.y)  
(0, 0)  
>>> translation(p)  
>>> (p.x,p.y)  
(20, 30)
```

SHELL

- ▶ Techniquement, on ne modifie pas la variable p (contenant le pointeur)...
  - ▶ ...seulement son contenu.

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

def translation(q): # q n'est pas une variable globale
    q.x = q.x + 20
    q.y = q.y + 30
```

SCRIPT

```
>>> p = Point(0,0)
>>> (p.x,p.y)
(0, 0)
>>> translation(p)
>>> (p.x,p.y)
(20, 30)
```

SHELL

- ▶ Techniquement, on ne modifie pas la variable p (contenant le pointeur)...
  - ▶ ...seulement son contenu.
- ▶ Attention danger !



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1
```

```
x=2  
y=3  
f(y)
```

SCRIPT

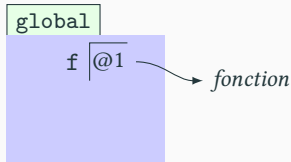
global



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1  
  
x=2  
y=3  
f(y)
```

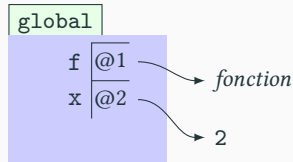
SCRIPT



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1  
  
x=2  
y=3  
f(y)
```

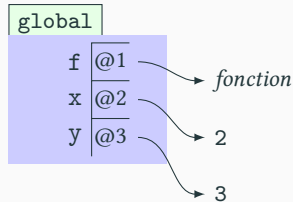
SCRIPT



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1  
  
x=2  
y=3  
f(y)
```

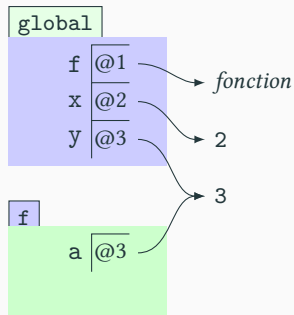
SCRIPT



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1  
  
x=2  
y=3  
f(y)
```

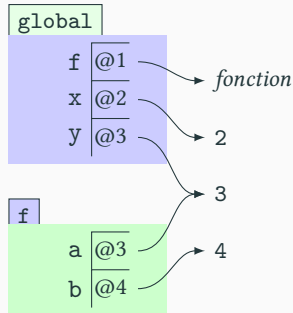
SCRIPT



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1  
  
x=2  
y=3  
f(y)
```

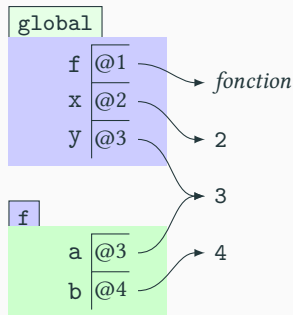
SCRIPT



- ▶ En Python tout est objet
- ▶ En pratique et contrairement à beaucoup d'autres langages, les variables ne contiennent que des pointeurs.

```
def f(a):  
    b=a+1  
  
x=2  
y=3  
f(y)
```

SCRIPT



- ▶ Cela signifie-t-il que `f` peut modifier `y` à travers `a` ?

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>>
```

SHELL



- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>> x=3
```

SHELL

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>> x=3  
>>>
```

SHELL

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>> x=3  
>>> x=4
```

SHELL

- ▶ je modifie `x` mais pas l'entier `3`

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>> x=3
>>> x=4
>>>
```

SHELL

- ▶ je modifie `x` mais pas l'entier `3`

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>> x=3
>>> x=4
>>> L=[1]
```

SHELL

- ▶ je modifie `x` mais pas l'entier `3`

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur `3` (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre `3`!

```
>>> x=3
>>> x=4
>>> L=[1]
>>>
```

SHELL

- ▶ je modifie `x` mais pas l'entier `3`

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur 3 (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre 3!

```
>>> x=3
>>> x=4
>>> L=[1]
>>> L.append(3)
```

SHELL

- ▶ je modifie `x` mais pas l'entier 3
- ▶ Je modifie la liste `[1]` (qui devient `[1,3]`) mais pas `L`!

- ▶ La réponse est non!
  - ▶ `f` ne peut pas modifier le pointeur dans `y` (pas de `global`)
  - ▶ `f` ne peut pas modifier la valeur 3 (qui sera toujours égale à trois).
- ▶ La majorité des types sont heureusement immuables.
  - ▶ `int`, `float`, `str`, `tuple`, `Nonetype`
  - ▶ Il serait gênant de pouvoir modifier la valeur du nombre 3!

```
>>> x=3
>>> x=4
>>> L=[1]
>>> L.append(3)
```

SHELL

- ▶ je modifie `x` mais pas l'entier 3
  - ▶ Je modifie la liste `[1]` (qui devient `[1,3]`) mais pas `L`!
- ▶ Seuls les objets plus complexes (listes, ou objets créés par l'utilisateur) sont mutables.
  - ▶ source de bogues!



Merci pour votre attention

Questions



# Cours 5 — Listes, tuples et gestion mémoire

Examen de mi-semestre

## Partie i. Séquences

Définitions

Tuples

Listes

## Partie ii. Accès et constructeur

Initialisation

Opérations sur les séquences

Accès par indice

Parcours

Écrire une fonction `index`

La méthode `index`

Déstructuration

Extraction de tranche

Construction par compréhension

Exercices

Conversions

Construction d'une matrice

## Partie iii. Mutabilité des listes

Modification de liste

Remarques sur la mémoire

Ajout d'un élément

Ajout d'un élément : explication

Exemple : créer la liste des diviseurs

Méthodes usuelles sur les listes

## Partie iv. Petits algorithmes

Chiffres dans une chaîne

Chiffres dans une chaîne avec compréhension

Plus petit diviseur

Liste des nombres premiers

Tri d'une liste par ordre croissant

Tri : Temps de calcul

Tri par sélection — Qu'est-ce?

Tri par sélection — Comment?

Tri par sélection — L'humiliation

## Partie v. Gestion de la mémoire

Variables et affectations

Être ou ne pas être égal...

Modification et affectation

Zones de mémoire

Mémoire et appels de fonction

Mémoire et variables locales

Partage de variables locales entre fonctions

Partage de variables globales entre fonctions

Listes et appels de fonction

Objets et appels de fonction

Tout est objet

Objets mutables et immuables

## Partie vi. Table des matières