UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Introduction à Python

Enseignant Responsable

Manuele Kirsch Pinheiro

Manuele.Kirsch-Pinheiro@univ-paris1.fr

Exemples et vidéos sur :

http://cours.univ-paris1.fr/fixe/06-M1-SI-Informatique

https://replit.com/@ManueleKirsch/SystemeInformationInformatique

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python

- C'est quoi ??
  - Langage de **programmation**
  - Très populaire dans la **data analyse**

- Plusieurs versions
  - Python2 → créé en 2000, arrêtée en 2020
  - **Python3 → créé en 2008**

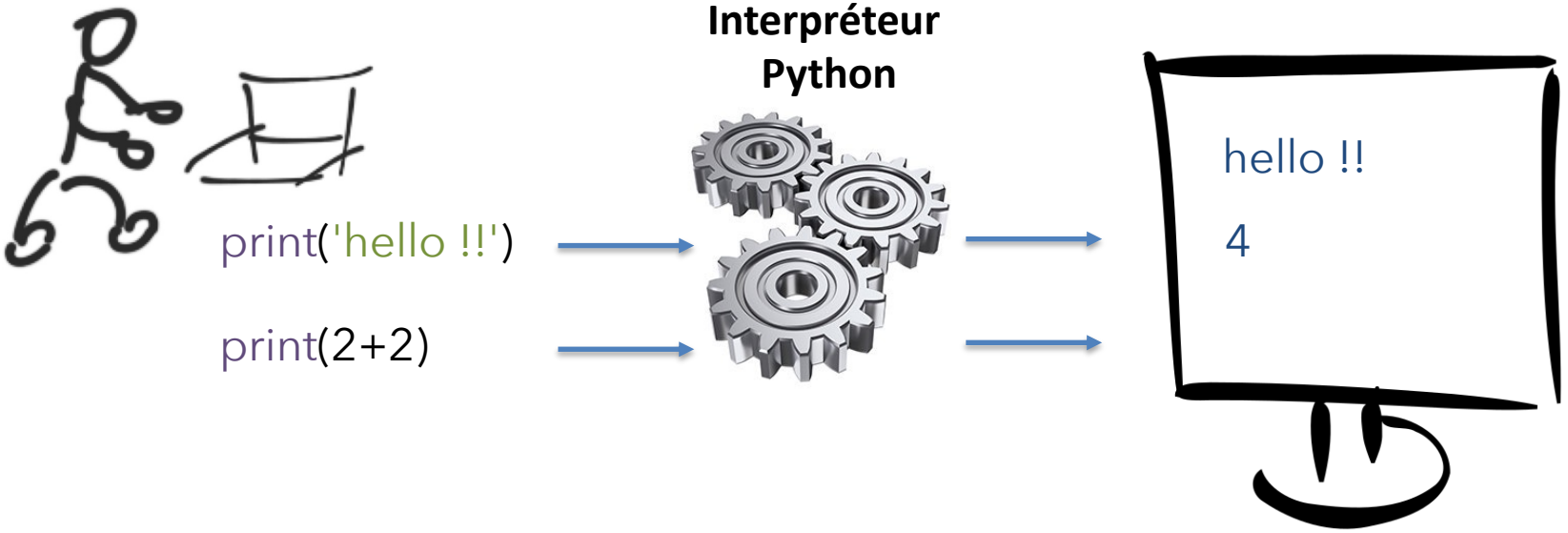
Attention !
Ces versions sont **incompatibles** !

Python

- **Comment ça marche ?**

Mode itératif

C'est lui qui interprète et exécute le code en Python



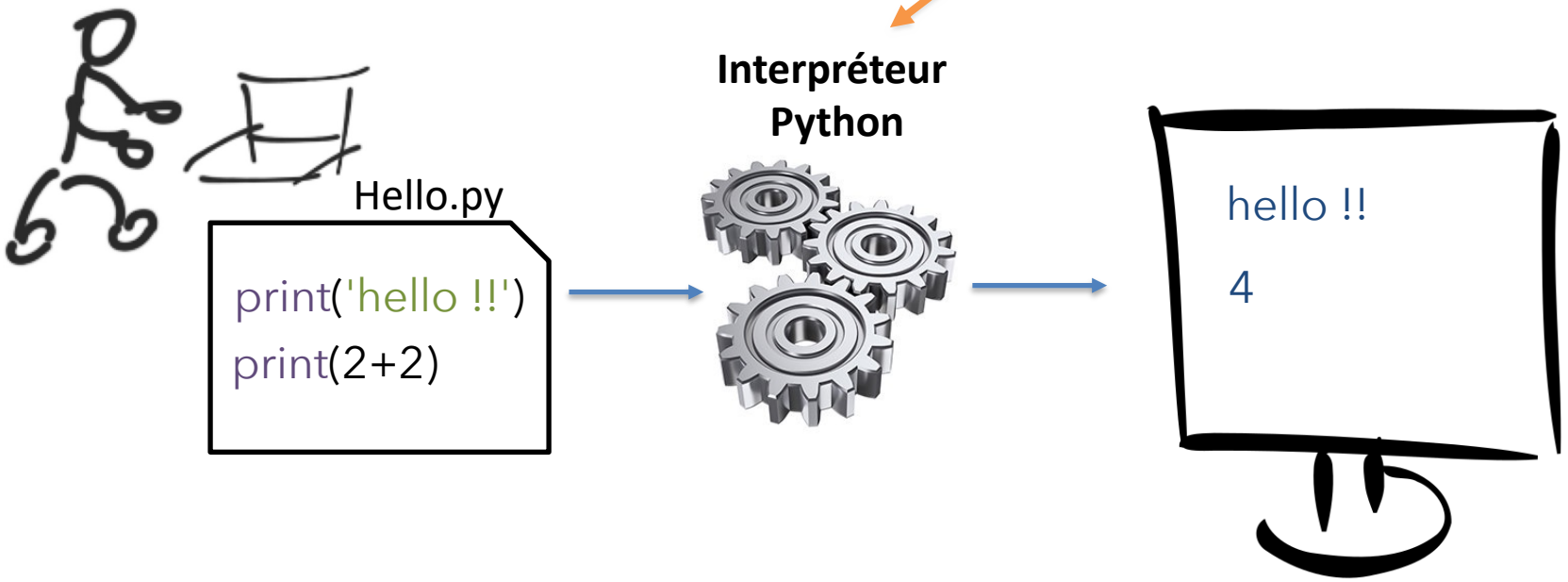
Ligne par ligne

Python

- **Comment ça marche ?**

Mode « batch »

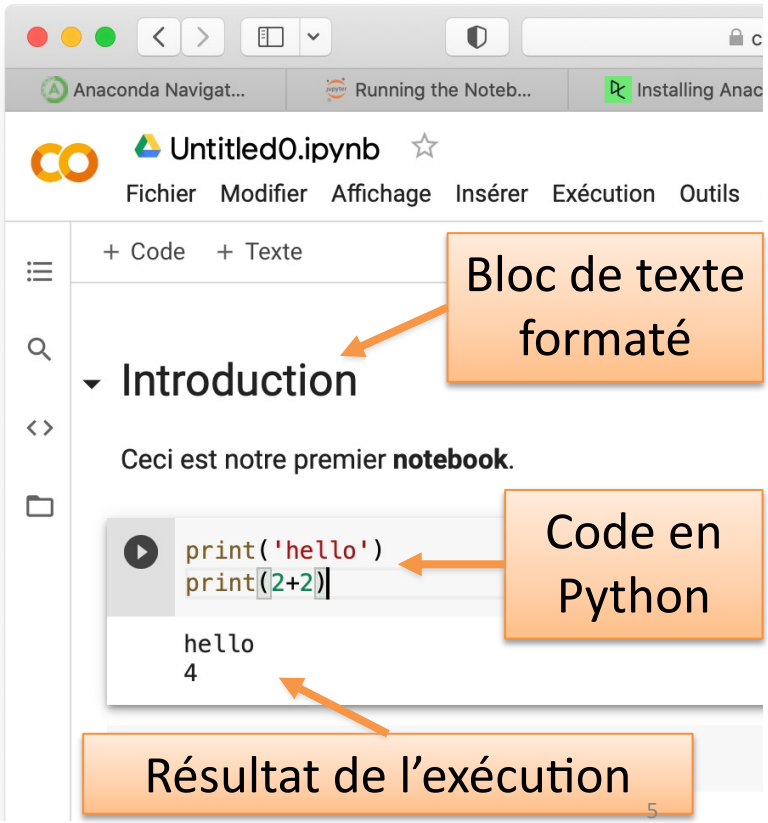
C'est lui qui interprète et exécute le code en Python



Fichier en entrée

Python

- **Et les Notebooks ?**
  - **Documents « actifs »** où on alterne des **blocs de texte** et de **blocs de code Python** (qui peuvent être **exécutés**)
    - **Texte** en format « **markdown** »
    - **Code Python** exécuté en **mode « itératif »**
  - Très utilisés en **Data Analyse**
  - Sur le Web (**navigateur**)



UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python

- Il faut avoir l'interpréteur Python
  - Disponible en certains Mac OS
  - Téléchargeable à partir de https://www.python.org
    - Mac OS & Windows
  - Gestionnaire d'installation **Anaconda** :
    - https://www.anaconda.com/products/individual

- Il faut un environnement de développement (IDE)
  - IDLE : disponible avec la distribution Python
  - **Visual Studio Code** : https://code.visualstudio.com
  - Notebook : **Jupyter Lab**

- Environnement on-line disponible
  - **Repl.It** : https://repl.it ou https://replit.com/

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python & Algorithme

Éléments dans un code

- **Variables**
  - **Informations**

```
a = 2.0
tab = [1.20, 2.0, 3.0 ]
```

- **Expressions**
  - **Actions**

```
print(a)
taille = len(tab)
```

- **Fonctions**
  - **Bibliothèques**

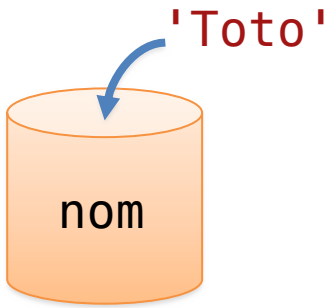
```
from math import sqrt
r = sqrt(2.0)
```

- **Structures de contrôle**
  - **Conditionnelles**
  - **Itératives**

Python : Variables & Expressions

- **Variables**
  - Les variables en Python correspondent à des **objets**
  - Objets créés lors du **premier usage**

```
nom = 'Toto'
```

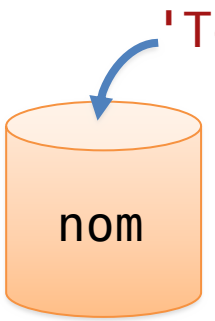


  - Une variable peut **changer de type** pendant exécution, en fonction des valeurs qu'on lui attribue
    - Variable = container
    - Le contenu du container change, le type associé aussi
  - Le **type** d'une variable (**classe**) conditionne les **opérations** possibles

Python : Variables & Expressions

Affectation :
On attribue une valeur à la variable

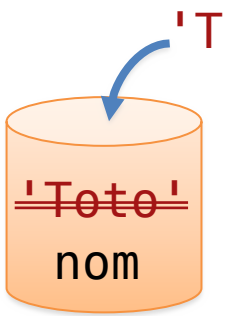
```
nom = 'Toto'
print('Hello ! Je suis', nom)
```

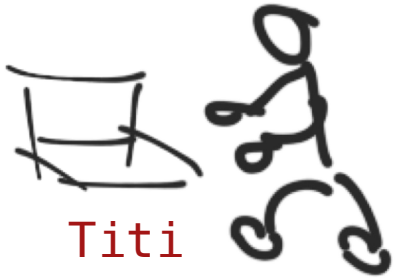


print (*val , val , val …)*
On affiche une valeur (une variable ou un message)


```
nom = input('votre nom ? ')
print('Hello, ', nom)
```

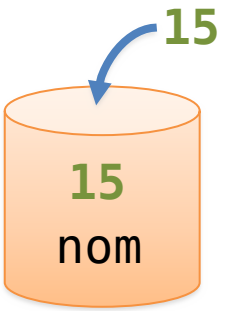
input (*message*)
permet de lire un texte à partir du clavier

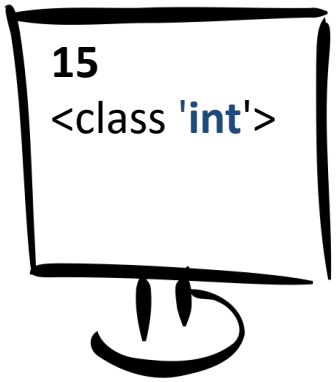




UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

```
a = 15
print (a)
print (type(a))
```

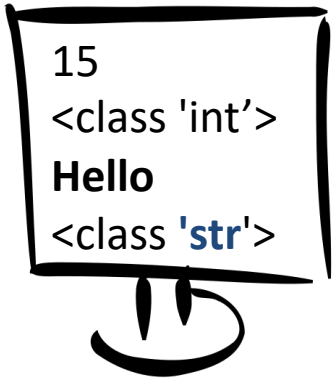




Le **type** associé à la variable « **a** » est « **int** » (**entier**)
a = **15**

```
a = 'Hello '
print (a)
print (type(a))
```

'Hello '



Le **type** de la variable « **a** » passe à « **str** » (String)
a = **'Hello '**

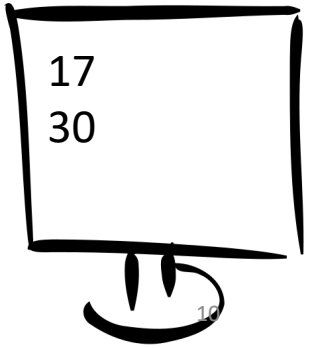
Une variable a un **type**.
Ce type peut changer avec le **contenu de la variable**.

Le **type** conditionne les **opérations** qu'on pourra réaliser avec **la variable**.

```
b = 'World '
print (a + b)
print (a * 2)
```

Pour les **strings** :
+ → concaténation
* → reproduction

```
b = 2
a = 15
print ( a + b )
print ( a * b )
```



Pour les **int** (entiers) :
+ → addition, * → multiplication

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

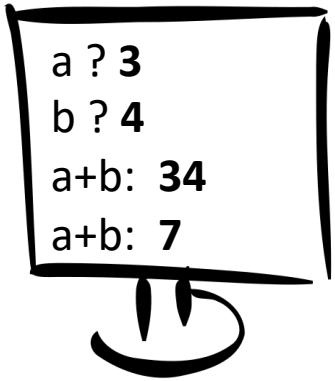
Python : Variables & Expressions

- On peut convertir les valeurs :
  - int(), float(), str()…

```
a = input('a ? ')
b = input('b ? ')

somme = a + b
print("a+b: ", somme)
```

On demande une valeur pour « **a** » et « **b** ».
La fonction « **input** » lit des chaîne de caractères (**String**)

```
inta = int(a)
intb = int(b)

somme = inta + intb
print("a+b: ", somme)
```



La fonction « **int** » permet de convertir la valeur en **entier (int)**

L'opérateur **+** fait alors une **somme** (et pas une **concaténation**)

Opérateurs le plus communs

\+ : addition / concaténation
\- : différence
* : multiplication / reproduction
/ : division
// : division entière (arrondi)
% : reste (modulo)
** : puissance

Opérateurs logiques

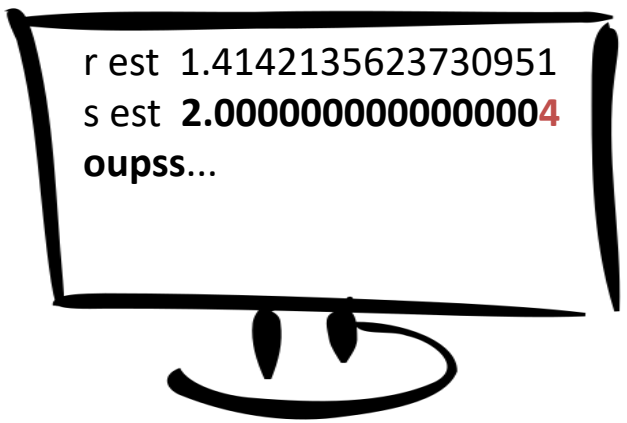
< , <=, >, >= : comparateurs
== : égalité (égale à)
!= : différent
and : ET logique
or : OU logique
not : négation

Python : Variables & Expressions

- **Attention à la précision : le cas du *float***
  - La représentation interne (à l'ordinateur) des numéros réels peut induire à une **perte de précision**
  - Il faut se rappeler de ça lorsqu'on fera des **tests logiques** (et **éviter** les == lorsqu'on parle de *float*)

```
from math import sqrt

r = sqrt(2.0)
s = r * r

print("r est ", r)
print("s est ", s)

if s == 2.0 :
    print('précision ok')
else :
    print('oupss...')
```

Contrairement à ce qu'on attend,
« s » ne vaut **pas exactement 2.0**



Python : Variables & Expressions

- **Manipulation de « strings » (chaînes de caractères)**
  - Python offre la classe « str » qui permet de manipuler les strings

```
nom = 'Toto'
```

Une string peut être vue comme un **tableau de lettres**

T	o	t	o
0	1	2	3

```
taille = len(nom)

premier = nom[ 0 ]
milieu = nom[ taille // 2 ]
dernier = nom[ -1 ]

print('taille :', taille )
print(premier, milieu, dernier)
```

taille : 4
T t p
== ele ? **False**
> Titi ? **True**

[**-1**] → dernière position
Équivaut à **taille - 1**

On peut les **comparer**

```
print('== ele ? ', (nom == 'ele'))
print('> Titi ? ', (nom > 'Titi'))
```

Comparaison (< et >) par rapport à l'ordre alphabétique

Et plein d'autres **opérations**

```
nom.upper()          → TOTO
dernier.isspace()    → False
nom.isalpha()        → True
```

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Tableaux & cie

- **Collections :**
  - Une collection est un **ensemble de valeurs**
  - Python offre différents types de collections : **listes**, tuples, dictionnaires

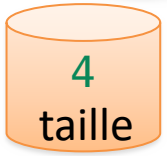
- Les **tableaux** en Python sont, réalité, une **liste** de valeurs

- Chaque collection, son usage :
  - **List** : collection **ordonnée** et **modifiable** de valeurs (tableau)
  - **Tuple** : collection **ordonnée** et **non-modifiable** de valeurs
  - **Dictionary** : collection de type **clé: valeur** (non-ordonnée, mais **indexée**)
  - **Set** : collection de **valeurs uniques**, ≈ ensemble mathématique (**non-ordonnée** et modifiable)

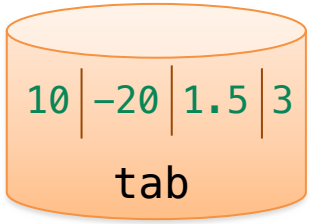
UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Tableaux & cie

*La fonction **len()** permet de connaître la **taille** (**nb d'éléments**)*



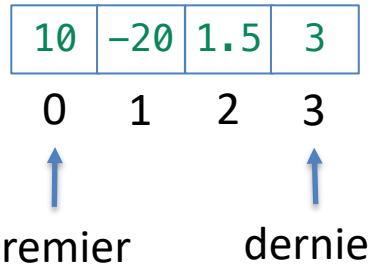
- **List** : Un **tableau** est une **liste de valeurs**
  - Chaque valeur est accessible par sa **position [*i*]**
  - Un tableau peut contenir toute sorte de valeurs




```
tab = [ 10 , -20 , 1.5 , 3 ]
taille = len(tab)
premier = tab[0]
dernier = tab[-1]
print(tab)
```

Première valeur
Dernière valeur

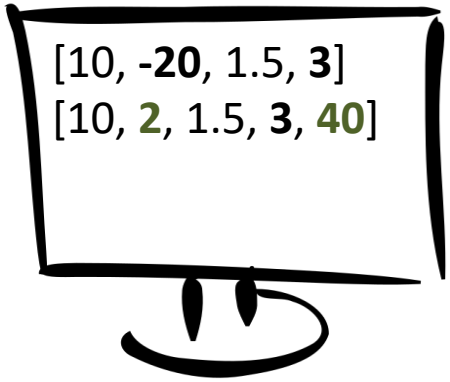
On définit la liste en lui affectant des valeurs

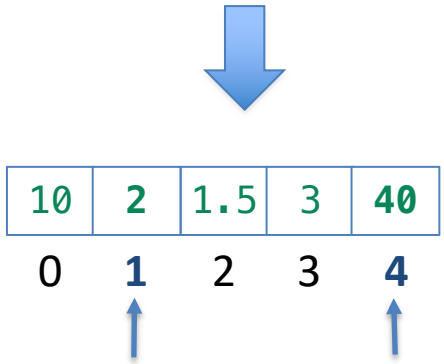


On peut **récupérer** ou **modifier** les valeurs avec **[ ]**

  - On peut **modifier** les valeurs
  - Et en **rajouter** des nouveaux

```
tab[1] = 2
tab.append(40)
print(tab)
```



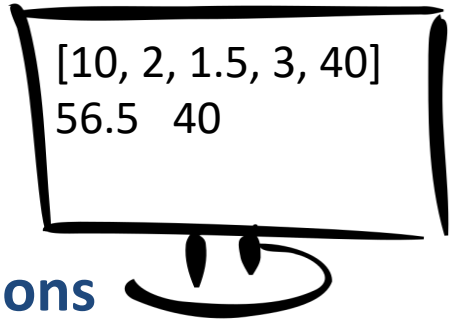


L'opération **append()** permet d'**ajouter** une valeur **à la fin** de la liste

Python : Tableaux & cie

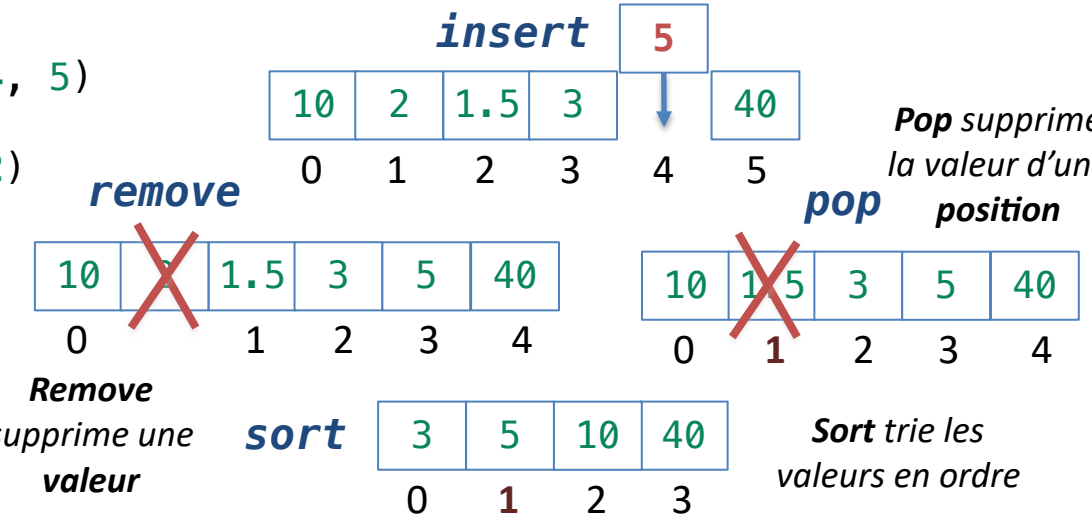
- **List : Manipulation des tableaux**
- Des nombreuses fonctions sont disponibles
  - len, sum, min, max…

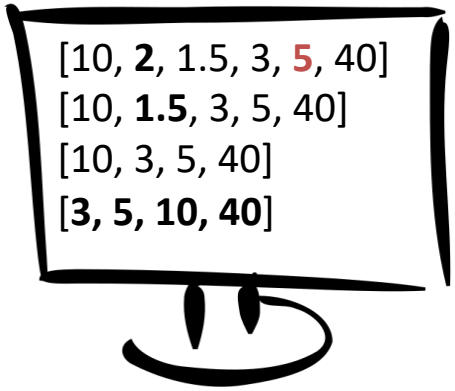

```
print (tab)
somme = sum(tab)
maxi = max(tab)
print (somme, maxi)
```



- La classe **List** offre aussi des nombreuses **opérations**
  - **append**(*valeur*), **insert**(*pos, val*), **remove**(*val*), **pop**(*pos*), **sort**()…
  - Attention à la notation : **objet.*opération*** **(** *paramètres* **)**

```
tab.insert(4, 5)
print(tab)
tab.remove(2)
print(tab)
tab.pop(1)
print(tab)
tab.sort()
print(tab)
```





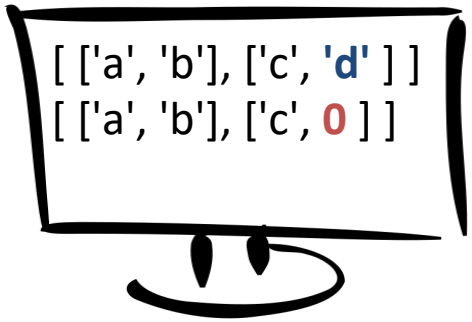
Python : Tableaux & cie

- **List** : Un **tableau** est une **liste de valeurs**
  - Une liste **contient des objets**, peu importe leur classe
  - Une liste peut contenir des **objets de différentes classes** (**types**)

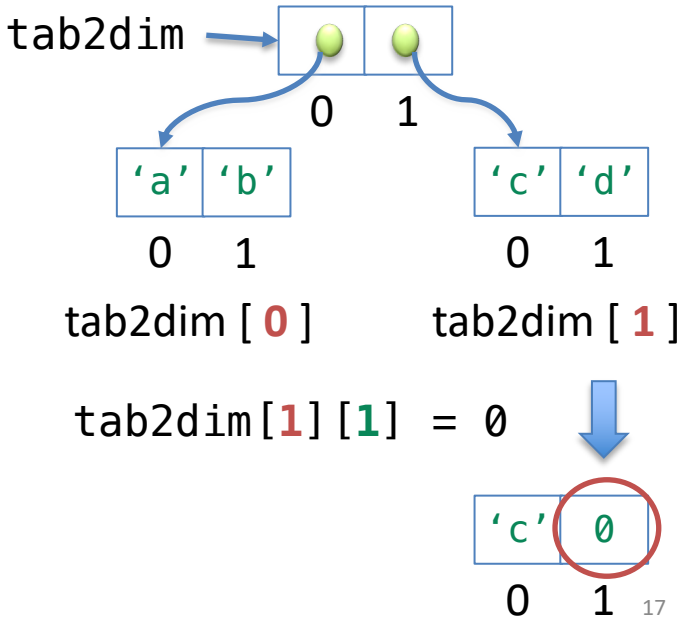
- **Tableau multidimensionnel = liste de listes**
  - Liste dont les **valeurs** correspondent à d'autres **listes**

```
tab2dim = [ ['a' , 'b' ], [ 'c', 'd'] ]
print(tab2dim)

tab2dim[1][1] = 0
print(tab2dim)
```

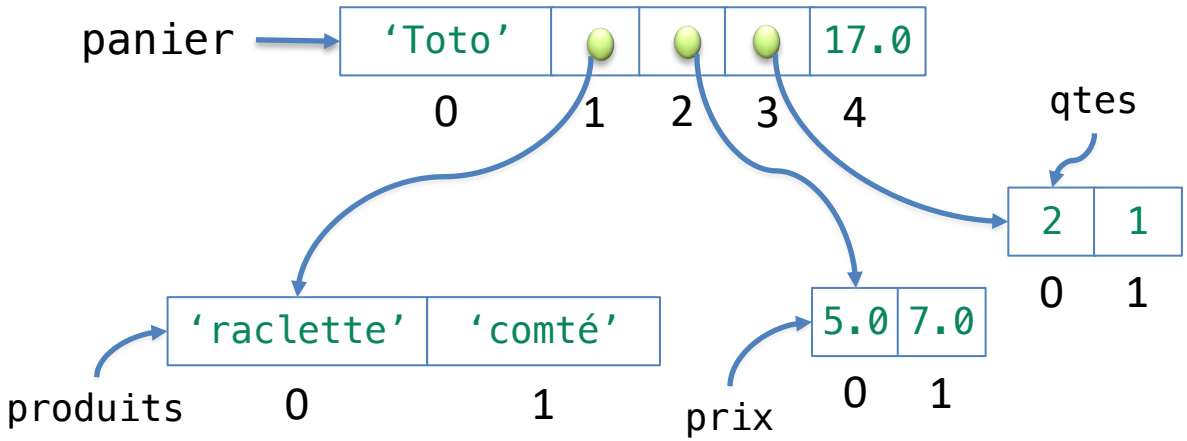
on peut ajouter un **int** (0) à notre tableau de **string**



tab2dim *[0] correspond à* ***une liste***
tab2dim *[1] correspond à* ***une autre liste***

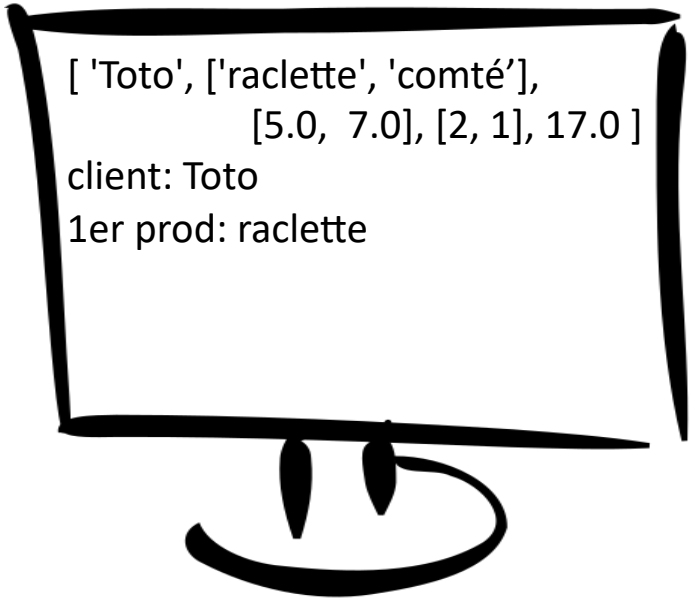


Python : Tableaux & cie



Un tableau « **panier** » est possible !

```
produits = ['raclette', 'comté']
prix = [ 5.0 , 7.0 ]
qtes = [ 2, 1 ]
panier = [ 'Toto', produits, prix, qtes, 17.0]
print(panier)
print('client:',panier[0])
print('1er prod:',panier[1][0])
```



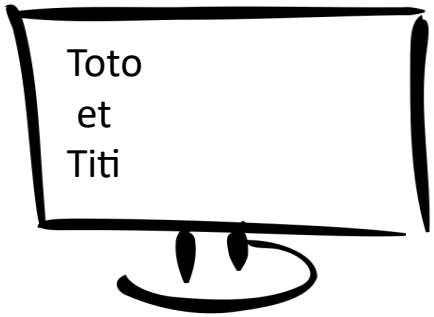
*Notre variable « **panier** » correspond à une **liste** de **5 positions**, **dont 3 positions** sont occupées par d'**autres listes** (correspondant aux variables **produits**, **prix** et **qtes**)*

Python : Tableaux & cie

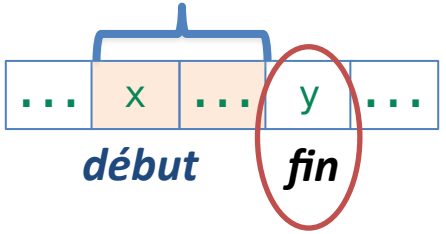
Aussi possible sur les **strings** et les **tuples** (**collections ordonnées**)

- **List : Découpage de listes (*slices*)**
  - L'opérateur « **:** » récupère une partie des éléments d'une liste
  - À partir d'une position ***début*** jusqu'à une position ***fin*** (**sans celle-ci**)

```
nom = 'Toto et Titi'
print(nom[:4])
print(nom[4:8])
print(nom[8:])
```



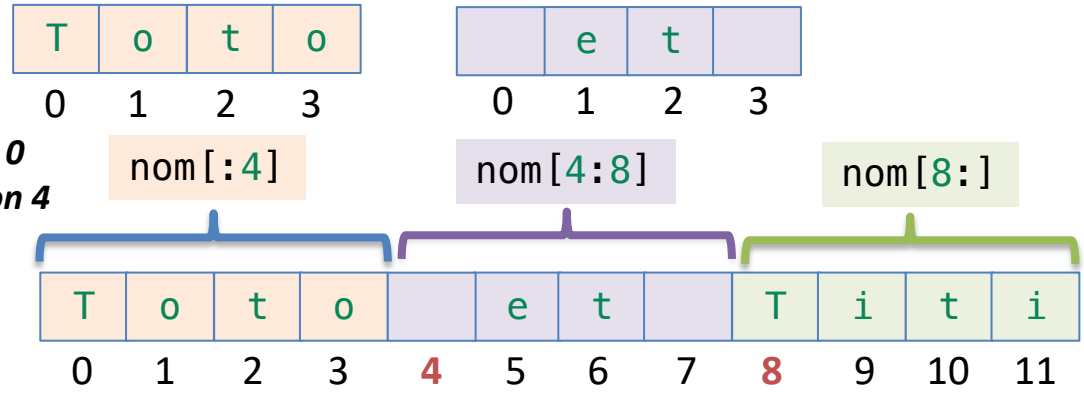
liste [***début*** : ***fin***] → x ...

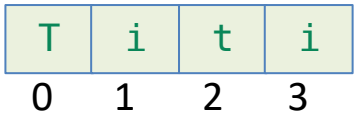


Le contenu de la position ***fin*** n'est pas incluse

[4 : 8]
*Dès la **position 4** jusqu'à la **position 8** (sans celle-ci)*

[:4]
*Dès la **position 0** jusqu'à la **position 4** (sans celle-ci)*





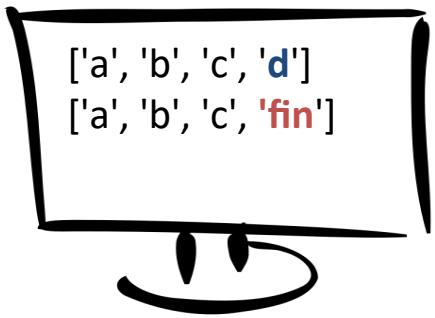
[8 :]
*Dès la **position 8** jusqu'à la **fin***

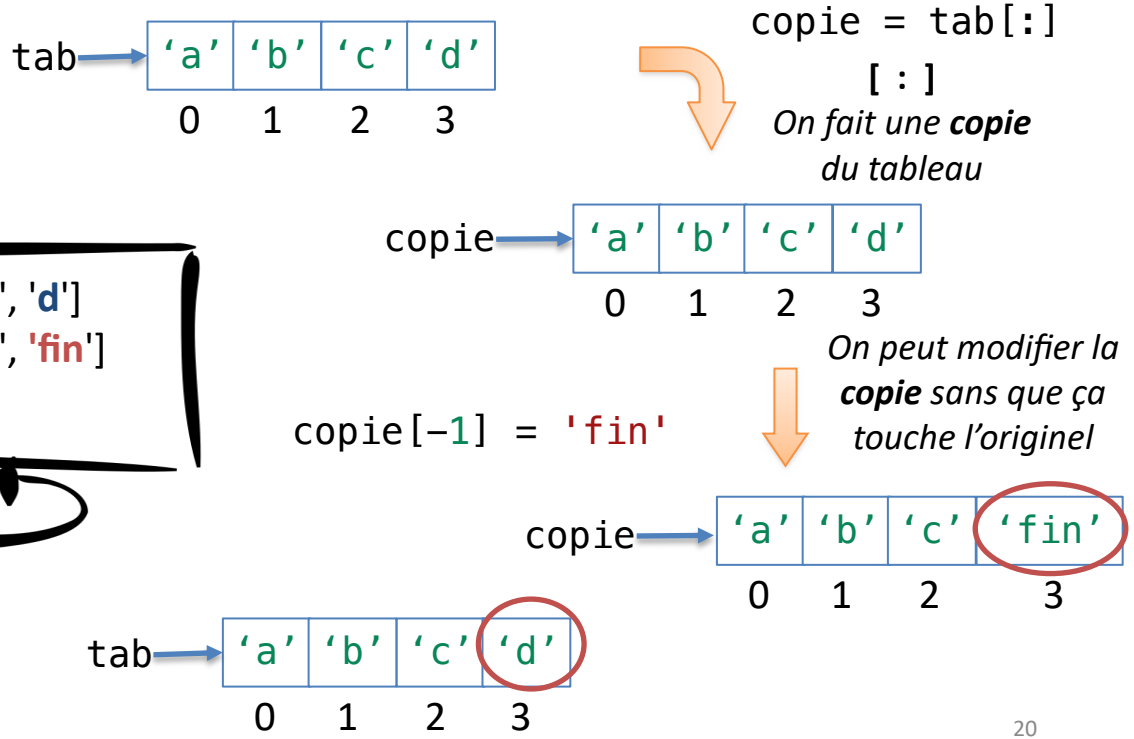
Python : Tableaux & cie

Aussi possible sur les **strings** et les **tuples** (**collections ordonnées**)

- **List : Découpage de listes (*slices*)**
  - L'opérateur « **:** » récupère une partie des éléments d'une liste
  - Chaque découpage est une **copie de l'originel**
  - On peut également faire une **copie de la liste entière**

```
tab = ['a','b','c','d']
copie = tab[:]
copie[-1] = 'fin'
print(tab)
print(copie)
```

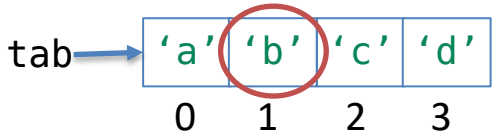




Python : Tableaux & cie

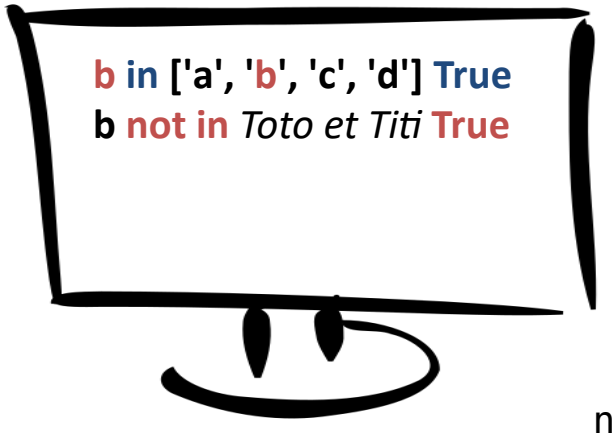
Aussi possible sur les **strings** et les **tuples** (**collections ordonnées**)

- **List : Opérateurs « in » et « not in »**
  - On peut vérifier **si un élément** se trouve **dans une collection** (liste, string, tuple, etc.)




```
element = 'b'
print (element, 'in', tab, ( element in tab ) )
print (element, 'not in', nom, ( element not in nom ) )
```

```
True
'b' in tab
```



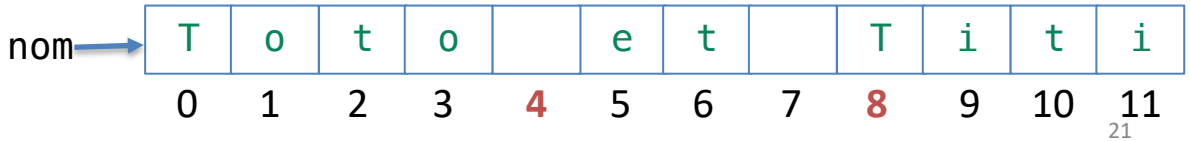
in

*L'opérateur **in** répond **True** (vrai) si la liste **contient** l'élément, **False** (faux) sinon.*

not in

*L'opérateur **not in** répond **True** (vrai) si la liste **ne contient PAS** l'élément, **False** (faux) sinon.*

```
'b' not in nom
    True
```

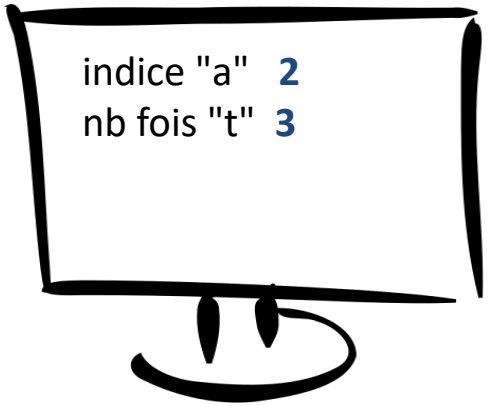


Python : Tableaux & cie

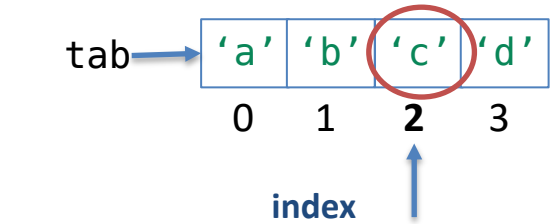
Aussi possible sur les **strings** et les **tuples** (**collections ordonnées**)

- **List : Opérations « index » et « count »**
  - On peut vérifier **si un élément** se trouve **dans une collection** (liste, string, tuple, etc.)

```
indice = tab.index('c')
nbfois = nom.count('t')
print ('indice "a"', indice)
print ('nb fois "t"', nbfois)
```



Il s'agit d'**opérations** implémentées par les **classes** **objet.opération** (…)



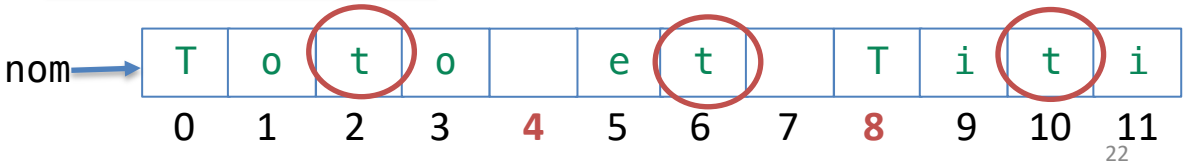
*L'opération **index** indique la **positon** d'un élément dans la liste*

Une erreur (**exception**) si l'élément n'**y est pas**

Majuscules et **minuscules** sont **différentes**

count

*L'opération **count** indique le nombre de fois que l'élément apparaît dans la liste*



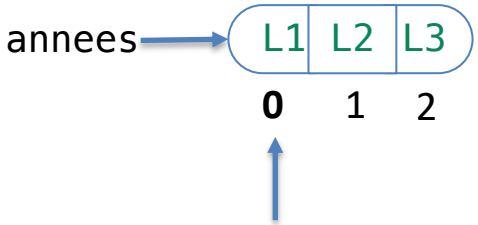
Python : Tableaux & cie

- **Tuple : ensemble ordonné et non-modifiable de valeurs**
  - Création d'un **tuple** avec ses valeurs **(val, val , val)**
  - Chaque **valeur** est accessible par sa **position :** tuple**[x]**
  - On **ne peut pas modifier** une valeur (***tuple [x] = y*** est **interdit**)

Création d'un tuple avec ses valeurs

```
annees = ( 'L1', 'L2', 'L3' )

print(annees)
print ('1er année :', annees[0] )
```

On récupère une valeur par sa position




```
tab = [ 10 , 12 , 14, 16 ]
mention = tuple(tab)
print (mention)
```

On peut également créer un tuple à partir d'une liste

```
print ('années en lic :', len(annees))
print ('M1 inclus ?', ( 'M1' in annees ) )
print ('position L3 :', annees.index('L3'))
```

On manipule un tuple comme n'importe quelle collection

('L1', 'L2', 'L3')
1er année : **L1**
(10, 12, 14, 16)
années en lic : **3**
M1 inclus ? **False**
position L3 : **2**

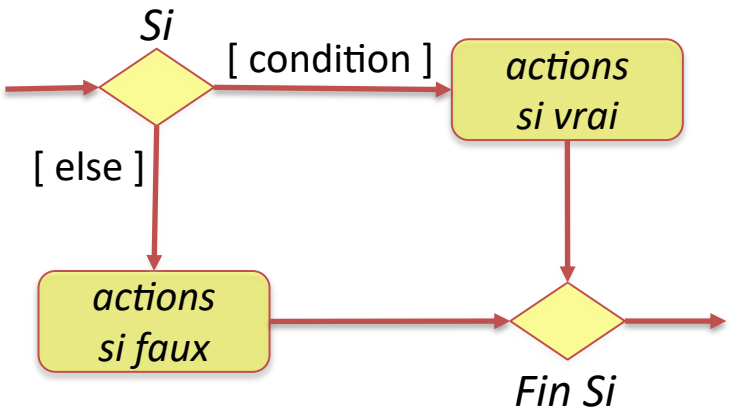
```
annee [ 2 ] = 'M1'
```

On ne peut pas modifier un tuple

Python & Algorithme

Éléments dans un code

- Variables ✓
- Expressions ✓
- Fonctions
  - Bibliothèques
- Structures de contrôle
  - Conditionnelles
  - Itératives



UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Structures conditionnelles

- **Structures conditionnelles « if – else »**
  - Python offre la structure « **if – else** » pour les **tests** « **si – sinon** »

On n'oublie pas le « : » pour démarrer le bloc

Condition logique (type « vrai » ou « faux »)

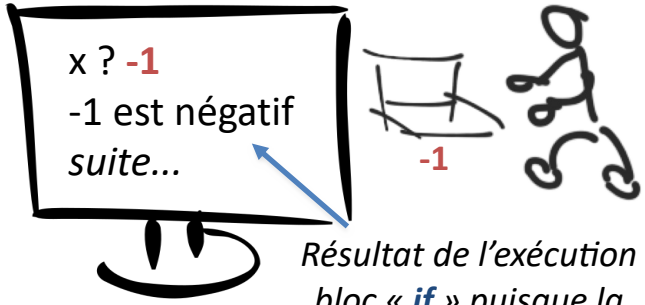
```
if condition :
    instructions si condition vrai
    . . .
else :
    instructions si condition faux
    instructions …
Instructions hors if
```

- **Notion de bloc : Attention aux espaces**
  - Un bloc d'instructions se définit par le nombre d'espaces
  - Il faut **exactement** le même **nombre d'espaces avant**
  - Dès que le **nb d'espaces change**, on **sort** du bloc (« **fin si** »)

```
x = int(input('x ? '))

if x < 0 :
    print (x, 'est négatif')
else :
    print (x, 'est positif')

print('suite...')
```

*Même **nombre d'espaces** pour bien identifier chaque bloc.*

*Dès que le **nombre d'espaces change**, on change de bloc.*



*Résultat de l'exécution bloc « **if** » puisque la condition est **vraie***

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Structures conditionnelles

- **Structures conditionnelles « if – else »**
  - On peut enchainer les « **si – sinon si – sinon** » à l'aide de la structure « **if – elif – else** »

x → 1

```
if x < 0 :
    print (x, '< à 0')
elif x > 0 :
    print (x, '> à 0')
else :
    print ('zéro')
print('suite...')
```

if ***condition 1*** **:**
instructions si condition 1 ***Vrai***
. . .
elif ***condition 2*** **:**
instructions si condition 2 ***Vrai***
. . .
else :
instructions si ***toutes*** *les conditions sont* ***Faux***

Instructions hors if

Toujours faire attention aux **espaces**

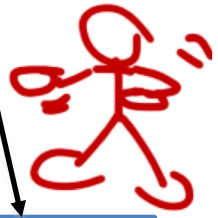
On peut aussi avoir un « **if** » ***sans*** **else**

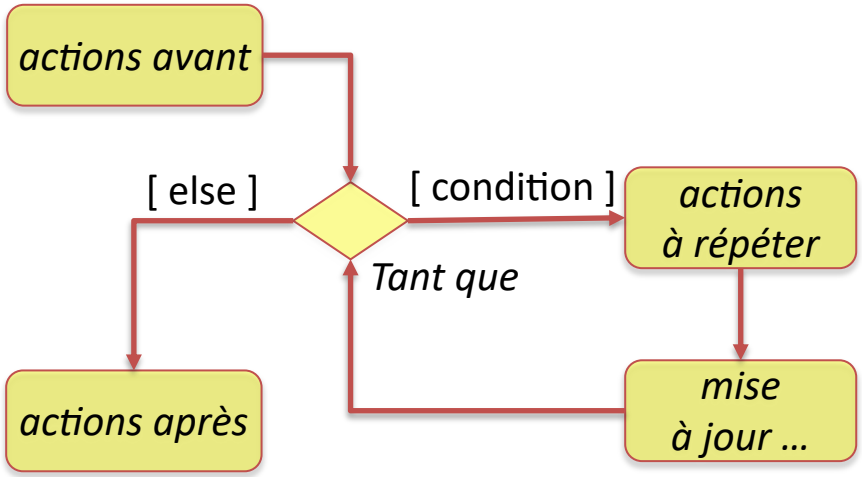
```
if x >= 0 :
    print (x, 'positif ou 0')
print('suite...')
```

Python & Algorithme

Éléments dans un code

- Variables ✓
- Expressions ✓
- Fonctions
  - Bibliothèques
- Structures de contrôle
  - Conditionnelles ✓
  - Itératives





UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Structures itératives

- **Structures itératives : les boucles « while » et « for »**
  - Python offre deux types de boucles
    - La boucle « **while condition** » permet de répéter un bloc d'instructions tant qu'une **condition** est **vraie**
    - La boucle « **for e in collection** » permet de répéter un bloc d'instructions pour **chaque élément e** présent dans une **collection** (*liste, string*…)
  - Il n'y a **pas de boucle « do … while »** dans Python !

instructions avant boucle…

while ***condition*** **:**
___*instructions à* ***répéter***
___*tant que la condition* ***vrai***
. . .
instructions hors while

Toujours le « : » pour démarrer le bloc

Toujours le même nombre de « » pour indiquer la durée du bloc

instructions avant boucle…

for e in ***collection*** **:**
___*instructions à* ***répéter***
___*pour chaque e*
. . .
instructions hors for

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : boucle while

salaire → 3000 somme → 0

Entrée dans la boucle

salaire → 3000 somme → 3000

Mise à jour variable

salaire → 4000 *Nouveau tour boucle* somme → 7000

salaire → -1 *Sortie de la boucle*

On n'oublie pas la « *checklist* »

1. Situation initial **(avant la boucle)**
2. **Condition**
3. **Mise à jour variable de contrôle**

```
#entree
salaire = float(input('Salaire initial ? '))

#etat initial
somme = 0

# condition
while salaire > 0.0 :
    somme = somme + salaire
    # mise à jour variable de contrôle
    salaire = float(input('Nouveau salaire ? (-1 pour sortir) '))

print('Total reçu : %.2f euros ' % ( somme ) )
```

Toujours faire attention aux **espaces**

Salaire initial ? **3000**
Nouveau salaire ? (-1 pour sortir) **4000**
Nouveau salaire ? (-1 pour sortir) **-1**
Total reçu : **7000.00** euros

Print avec de format de sortie :
' **%**.2f %d ' **% (valeurs)**
%.2f → 2 cases décimal
%d → n° entier

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : boucle while

10000
2000
500
200

taille

4

revenus

0	0	0	0
0	1	2	3

compteur

0

Entrée dans la boucle

1000	0	0	0
0			

Mise à jour compteur

1

1000	2000	0	0
0	1		

Mise à jour compteur

2

...

Sortie de la boucle

revenus

1000	2000	500	200
0	1	2	3

compteur

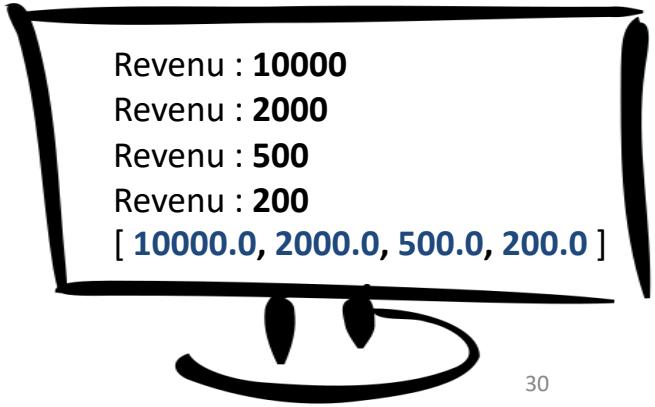
4

On n'oublie pas la « *checklist* »

1. Situation initial **(avant la boucle)**
2. **Condition**
3. **Mise à jour variable de contrôle**

```
revenus = [ 0, 0, 0 , 0 ]

taille = len(revenus)
compteur = 0

while compteur < taille :
    revenus[compteur] = float(input('Revenu : '))
    compteur += 1

print(revenus)
```

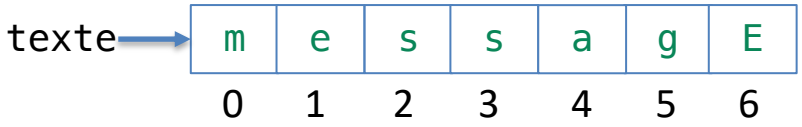
compteur += 1 équivaut à
compteur = compteur + 1



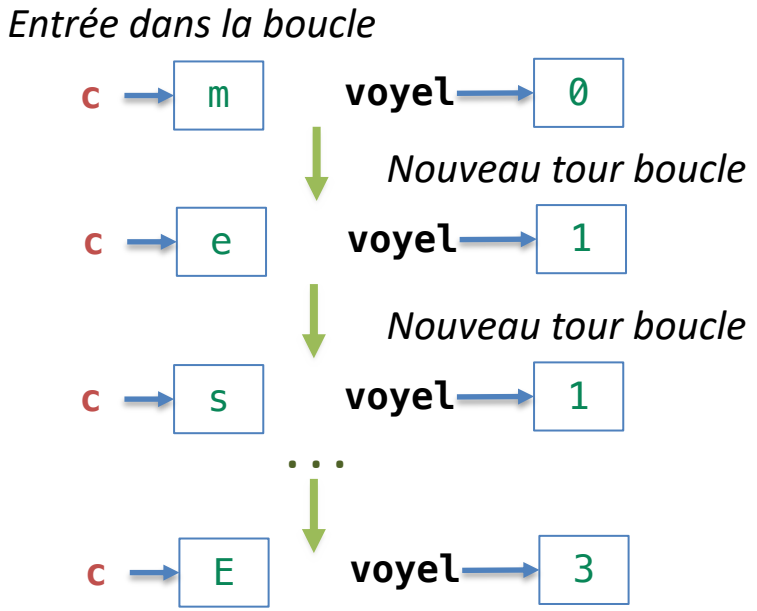
UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : boucle for

- **Boucle « for »** :
  - On répète les actions **chaque élément** d'une **collection**



À **chaque tour**, la variable **c** prend la valeur du **prochain élément**




```
texte = input('Message : ')
voyel = 0

#pour chaque lettre c dans le texte
for c in texte :
    #on vérifie si c'est une voyel
    if c.lower() in 'aeiou' :
        voyel += 1

print(texte, 'a %d voyelles' % (voyel))
```

Toujours faire très attention aux **espaces**

Message : **messagE**
messagE a **3** voyelles

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : boucle for

- **Boucle « for »** :
  - On peut utiliser un **range()** pour générer une **séquence** de **valeurs**

```
for e in range ( fin ) :
    instructions à répéter
    pour valeur e de début à fin
    . . .
instructions hors for
```

*On répète le bloc pour **chaque valeur** dans la **séquence***

range (*début, fin, step*)
*génère une **séquence de valeurs** de **début** jusqu'à **fin** (sans celui-ci), de **step** en **step***

```
taille = 3
for i in range(taille) :
    print('.'*i, i)

print("Booooommmm !! :-)")
```

*On répète pour chaque valeur de **0** à **3** (**taille**) **sans** inclure ce **dernier** (on va jusqu'à **taille - 1**)*

```
0
. 1
.. 2
Booooommmm !! :-)
```

```
print('range(0,5,2)')
for i in range(0,5,2)
    print(i, end=' ')
```

```
range(0,5,2)
0 2 4
```

*Séquence de 0 à 5 (**sans ce dernier**), avançant de 2 en 2*

Python : boucles

- **Création de listes avec « range » et la boucle « for »**
  - Toutes les variables sont créées à **leur premier usage**, les listes aussi
  - On va créer une liste en lui **affectant des valeurs**
  - Mais et si on **ne connaît pas** les **valeurs** (ou la **taille**) ?
    Comment créer une liste « vide » ?

```
taille = int(input('taille ? '))
liste = [ None ] * taille
print (liste, 'taille', len(liste))
```

Créer une liste de 3 positions
*On **reproduit** la valeur*
*« **None** » (null) **3x***

```
liste = list(range(taille))
print (liste, 'taille', len(liste))
```

*On peut se servir d'un **range***
qui va démarrer la liste avec
autant de valeurs

```
liste = [ None for c in range(taille) ]
print (liste, 'taille', len(liste))
```

*On peut aussi se servir d'un **for in range**(taille)*
*pour reproduire la valeur « **None** » autant de fois.*

Python : boucles

- **Création de listes avec « range » et la boucle « for »**
  - **Comment créer une liste multidimensionnelle « vide » ?**

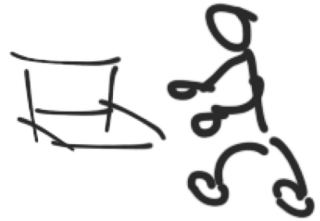
*D'abord on indique la **1ère ligne** (avec un **nb de colonnes**)*

*Puis on **reproduit** ça pour autant de **lignes** qu'on souhaite (**nb lignes**)*

```
liste = [ [ None ] * nbColonnes for l in range(nbLignes) ]
print(liste, len(liste), 'x', len(liste[0]))
```

Nb lignes

Nb colonnes de la ligne [0]



```
Nb de lignes ? 3
Nb de colonnes ? 2
```

[[None, None], [None, None], [None, None]] **3** x **2**

[[None, None], [None, None], [None, None]] **3** x **2**

*D'abord on indique la composition de **1ère ligne** (avec un **nb de colonnes**)*

```
liste = [ [ None for c in range(nbColonnes) ] \
                  for l in range(nbLignes) ]
```

*Puis on **reproduit** ça pour autant de **lignes** (**nb lignes**)*

```
print(liste, len(liste), 'x', len(liste[0]))
```

Python & Algorithme

Éléments dans un code

- Variables ✓
- Expressions ✓
- Fonctions
  - Bibliothèques

```
from math import sqrt
r = sqrt(2.0)
```

- Structures de contrôle ✓
  - Conditionnelles
  - Itératives

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Les bibliothèques en Python

- Une bibliothèque est un **catalogue de fonctions**
  - Ensemble des codes prêts et disponibles pour usage
  - **Solutions préexistantes** pour des problèmes connus
  - On y trouve de **fonctions**, de **classes** et de **constantes**
  - **Réutilisation d'un savoir-faire**

Constante = *variable dont la valeur ne change pas*

On ne réinvente pas la roue !

- Les bibliothèques ont largement contribué à la renommée de Python
  - Des nombreuses ressources supplémentaires sont disponibles
  - Sujets très variés :
    - **Data analyse, Machine Learning**, mais aussi biologie, Big Data…

Bibliothèques

```
from math import sqrt
r = sqrt(2.0)
```

Python :
Fonctions & Bibliothèques

- **Bibliothèque standard** disponible dans l'installation Python offre des nombreuses possibilités
  - Exemples : math, random, decimal, datetime, calendar…

- Pour utiliser une bibliothèque, on doit d'abord **importer** la **bibliothèque**, puis indiquer la(es) **fonction**(s) ou **classe**(s) qu'on veut **utiliser** :

À partir de la bibliothèque ***math*** *on veut la fonction* ***sqrt***

```
from math import sqrt

racine = sqrt(2.0)
```

qu'on utilise ici

À partir de la bibliothèque ***random*** *on veut la fonction* ***randint***

```
from random import randint

tirageAuSort = randint(1,6)
```

qu'on utilise après dans le code

Python :
Fonctions & Bibliothèques

- **Exemples bibliothèque standard :**
  - math, random, decimal, datetime, calendar…

- Exemple : **math**
  - Opérations arithmétique, trigonométrie…
  - Quelques fonctions : ***sqrt*, *exp*, *log*, *sin*, *cos***…
  - Quelques constantes : ***pi*, *e***…

- Exemple : **datetime**
  - Classe ***datetime***
  - Manipulation et mise en forme de dates (date + heure)

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Fonctions & Bibliothèques

Quelle bibliothèque — *Ce qu'on va utiliser*

```
from math import sqrt, pi

print ('pi = ',pi)

numero = float(input('numéro ? '))

racine = sqrt(numero)

print('racine :',racine)
```

*Usage constante **pi***

*Usage fonction **sqrt ()***

pi = **3.141592653589793**
numéro ? **4**
racine : **2.0**

as** : **Alias** pour la **classe** **datetime
*Au lieu d'écrire **datetime**.xxx, on fait **dt**.xxx*

```
from datetime import datetime as dt

auj = dt.today()
print('auj :', auj.date())

# texte vers datetime
unjour = input('un jour (JJ/MM/AAAA) ? ')
jour = dt.strptime( unjour ,'%d/%m/%Y')
print('format ISO:', jour.date())

# datetime vers texte
unjour = auj.strftime('%d/%m/%Y')
print('auj :', unjour)
```

Méthode de classe: **classe**.*op*()

Méthode d'instance **obj**.*op*()

auj : **2021-03-20**
un jour (JJ/MM/AAAA) ? ***1/4/2021***
format ISO: **2021-04-01**
auj : **20/03/2021**

UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Fonctions & Bibliothèques

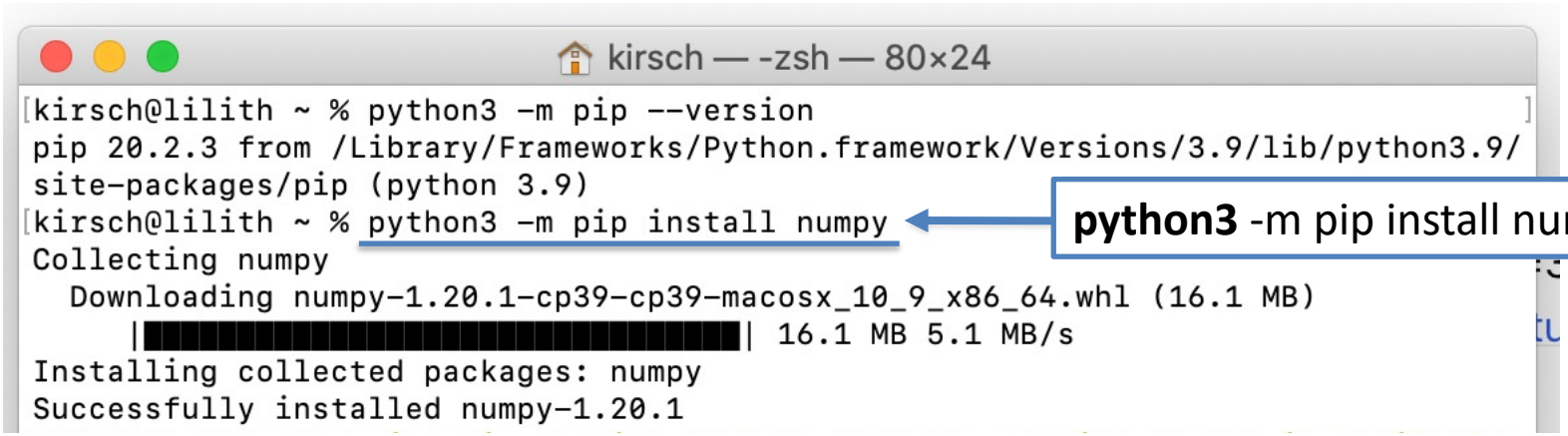
- D'autres bibliothèques sont disponibles
  - Exemple : **numpy**, **pandas**, **scikit-learn**, **matplotlib**, tensorflow…

- On doit **installer** les nouvelles bibliothèques
  - Ajouter la bibliothèque à l'installation Python de base
  - Ce n'est qu'après l'avoir installé, qu'on peut l'utiliser

- Installation d'une nouvelle bibliothèque
  - Outil « **pip** » disponible dans l'installation de base Python
  - Il va falloir ouvrir un **terminal** et entrer la commande
    - **python -m pip install numpy**
    - **python3 -m pip install scikit-learn**
  - On peut aussi installer et utiliser le gestionnaire « **anaconda** »

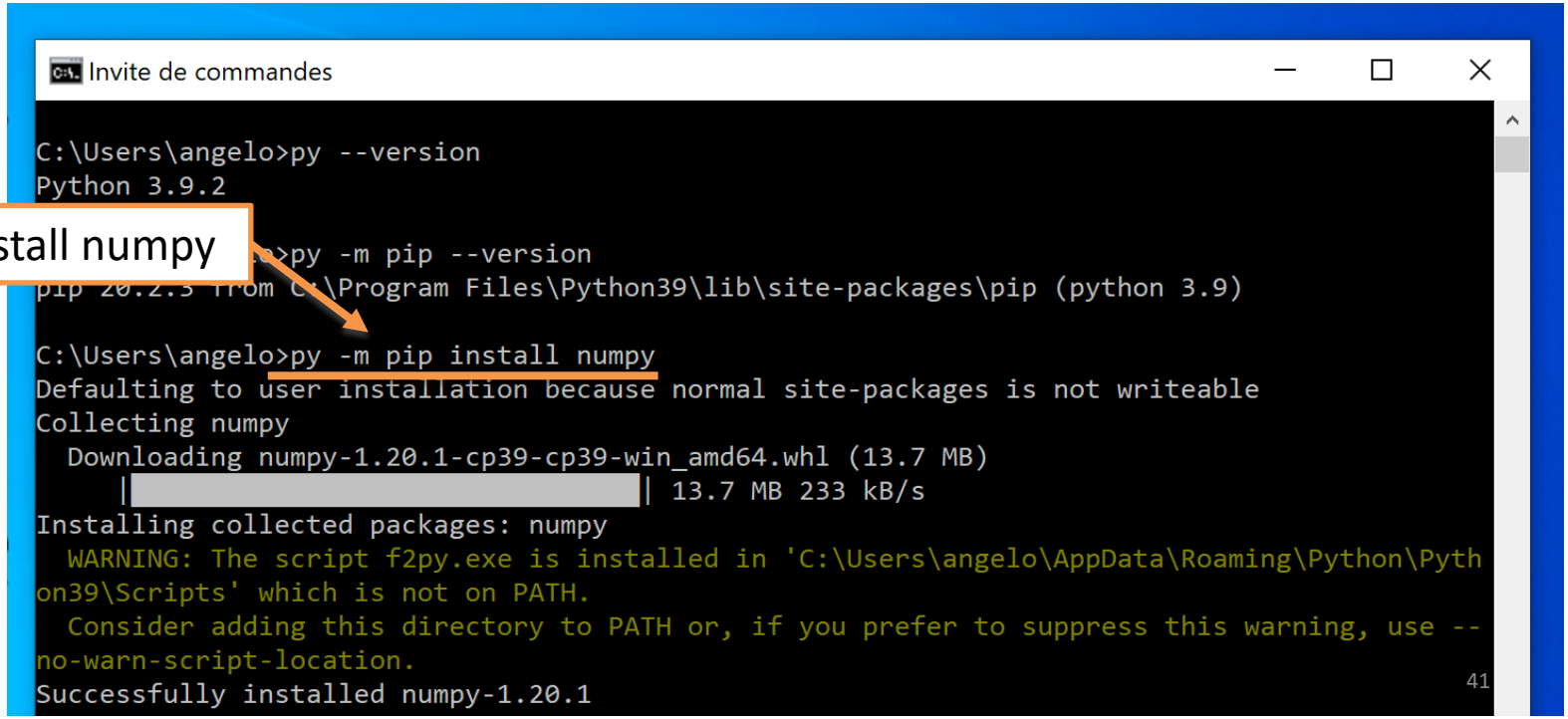
UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python :
Fonctions & Bibliothèques



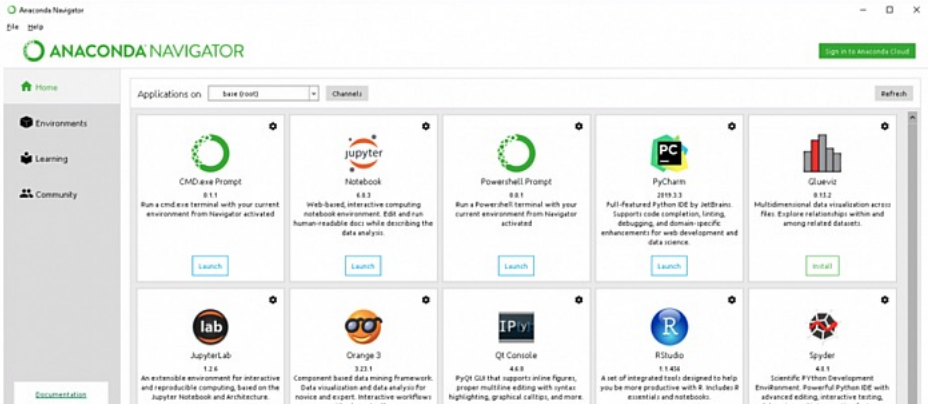
python3 -m pip install numpy

py -m pip install numpy



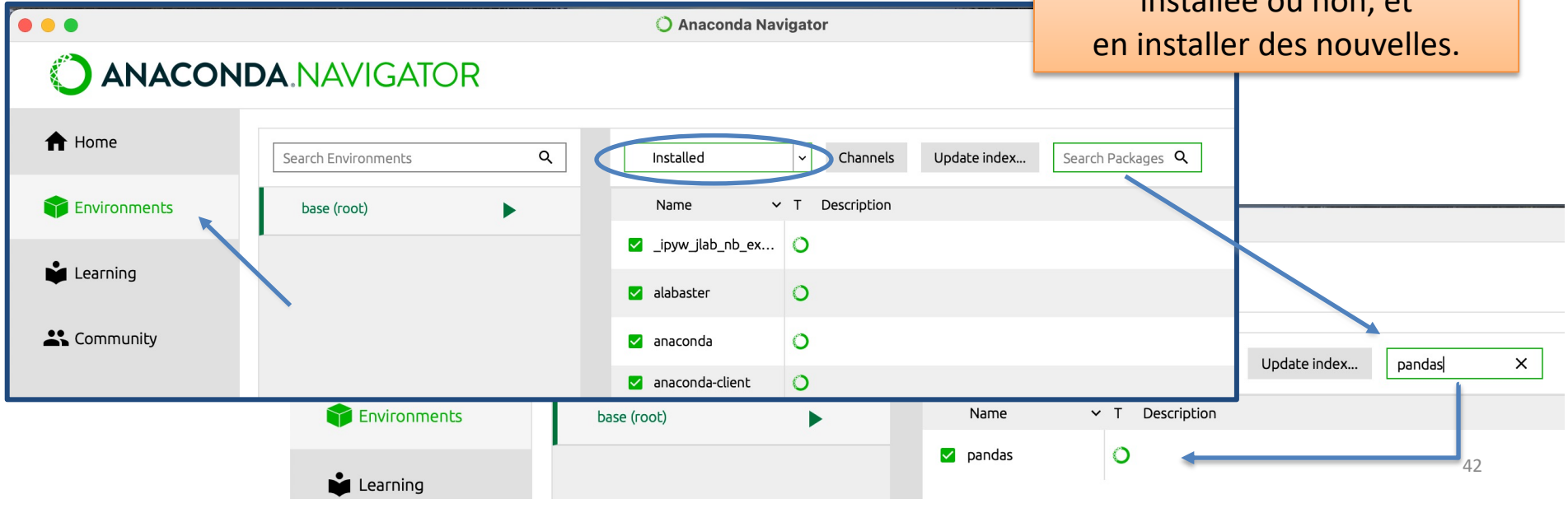
UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Fonctions & Bibliothèques



- **Installation avec Anaconda**
  - Anaconda est un « **gestionnaire de paquets** »
  - Il permet de **gérer son installation** Python et ses **bibliothèques**
  - Interface graphique : **Anaconda.Navigator**

On peut savoir ce qui est installé, savoir si une bibliothèque est installée ou non, et en installer des nouvelles.



UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Bibliothèque NumPy

- **Exemple bibliothèque NumPy**
  - Bibliothèque dédié au **calcul scientifique**
  - Simple et **performante**
  - Propose la structure **array** (classe **ndarray**) pour la manipulation des **matrices multidimensionnelles**
    - Nombreuses opérations de calcul scientifique (algèbre linéaire, vecteurs…)
  - Exemple : ExempleNumpy.py

Quelques liens :

- https://numpy.org/doc/stable/user/absolute_beginners.html
- https://numpy.org/doc/stable/user/basics.html

Python : Bibliothèque NumPy

- Quelques éléments intéressants : **création d'un tableau**
  - Exemple : ExempleNumpy.py

Importer la bibliothèque
as → alias « **np** » (pour faire court)

Création d'une matrice 2x3 remplie de zéros
(X,Y → colonnes x lignes)

0 0 0
0 0 **3**

Position [1][2]

```
import numpy as np

matrice = np.zeros( (3, 2) )
matrice[1][2] = 3

matriceObjets = np.empty( (2, 2) , dtype='object')
matriceObjets[1][0] = 'Toto'

matriceFloat = np.empty ( (2, 3) , dtype='float64')
print(matriceFloat)
```

Création d'une matrice 2x3 capable de contenir n'importe quel type d'objet

None None
'Toto' None

Position [1][0]

Création d'une matrice **3x2** contenant des n° réels (`float64`).
La matrice contiendra une valeur « aléatoire » pour chaque position

```
[ [-2.68156159e+154 -2.68156159e+154]
 [ 1.48219694e-323  0.00000000e+000]
 [ 0.00000000e+000  4.17201348e-309] ]
```

Python : Bibliothèque NumPy

- Quelques éléments intéressants : **quelques opérations**
  - Exemple : ExempleNumpy.py

```
matrice = np.zeros( (2, 3) )
```

```
1.  0.
2.  0.
3.  3.
```

```
mI = matrice.transpose()
```

mI

```
1. 2. 3.
0. 0. 3.
```

Matrice transposée

```
diagonal = matrice.diagonal()
```

→ `[ 1.  0. ]` Diagonal

```
sommeM = matrice.sum()
```

→ 9.0

Somme tous éléments

```
sommeLigne = matrice.sum(axis=1)
```

→ `[ 1.  2.  6. ]` (**axe = 1**)

Somme de chaque **ligne**

```
sommeColonne = matrice.sum(axis=0)
```

→ `[ 6.  3. ]` Somme de chaque **colonne** (**axe x=0**)

```
sup2 = matrice[matrice >= 2]
```

→ `[ 2.  3.  3. ]`

Éléments **>= 2**

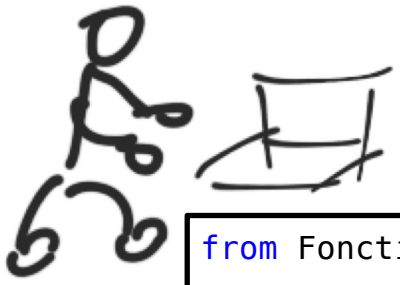
Python : Bibliothèques
Bon à savoir…

- Pour aller plus vite, Python garde le code « compilé »
- **Comment ça marche ?**

Lors de la 1ère usage d'une bibliothèque, l'interpréteur « garde » le code « prêt » (déjà analysé)

Fichier FonctionPair.pyc

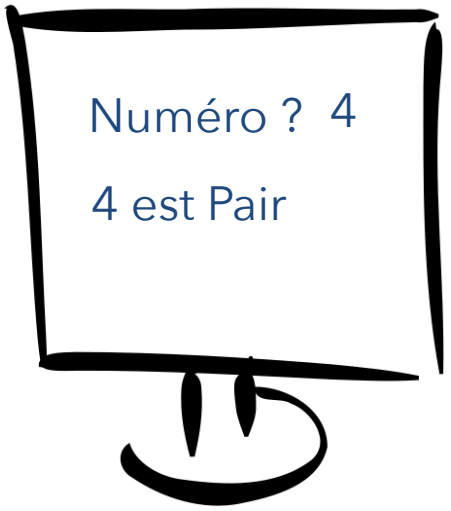
La prochaine fois qu'on utilisera la bibliothèque, il cherchera le code « prêt »



Hello.py

```
from FonctionsPair import …
...
estPair(numero)
```

Interpréteur Python



FonctionPair.py

```
def estPair(numero) :
  reste = numero % 2
```

Python : Fonctions & Bibliothèques

- On peut créer ses **propres fonctions** et ses **propres bibliothèques**
- **Fonction**
  - Un **morceau de code**, avec un **nom**, réalisant une **fonctionnalité** qu'on peut **invoquer** lorsqu'on en a besoin
  - Une fonction englobe alors un **algorithme**
  - Une fonction correspond donc à la notion d'**activité**

On découpe un problème en plusieurs petits problèmes

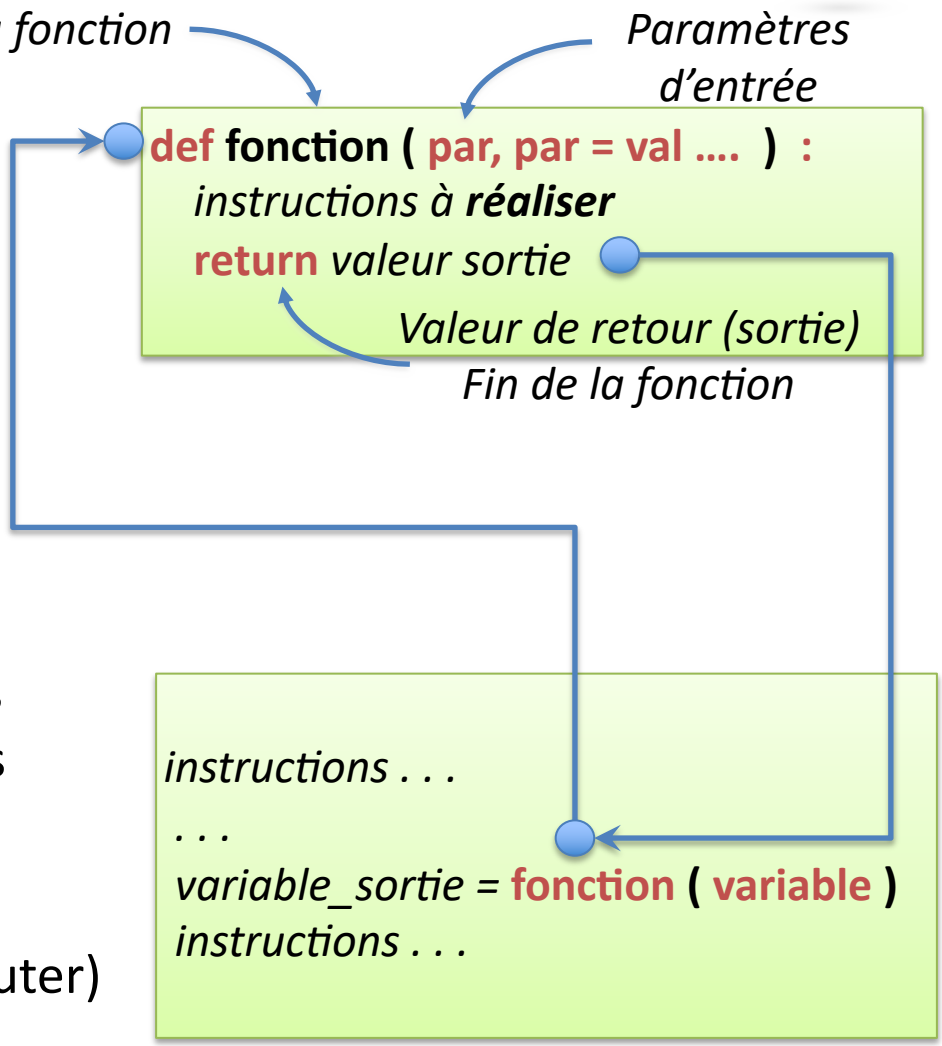
- **Pourquoi faire des fonctions ?**
  - **Réduire la complexité** d'un code en le découpant en plusieurs morceaux
  - Chaque morceau (**fonction**) est **responsable** d'une **fonctionnalité**
  - Plus facile à **lire**, à **maintenir** et à faire **évoluer**
  - Promouvoir la **réutilisation**

On ne réinvente pas la roue

Python : Fonctions

- **Comment créer une fonction ?**
  - Il faut **d'abord définir** la fonction
  - Ensuite on peut **l'utiliser**

- **Comment ça marche ?**
  - Lorsqu'on **appelle** la fonction, on lui transfert le « contrôle »
  - Les **valeurs** des variables indiquées en **entrée** sont « **transférées** » vers les **paramètres d'entrée**
  - La fonction **s'exécute**, puis à sa **fin** (un **return** ou plus de lignes à exécuter) on **revient** à la case de départ
  - Les **valeurs** retournées par le **return** sont **affectées** à *variable_sortie*



Python : Fonctions

• Définition d'une fonction

```
def volumeCube (cote) :
    """ Documentation : fonction
    calculant le volume d'un cube """
    volume = cote ** 3
    return volume
```

Code fonction

```
# usage des fonctions
cot = int(input('cote ? '))
vol = volumeCube(cot)
print ('volume :', vol)
```

Code principal

Plus le même nb d'espace,
Plus dans la fonction…

Début du bloc « : »

```
def fonction ( par, par = val ....  ) :
    instructions à réaliser
    return valeur sortie
```

Attention à toujours respecter le nb d'espace (changement nb d'espaces = fin de la fonction)

On peut avoir plusieurs paramètres d'entrée et de sortie (ou en avoir aucun).

Dans le code principal

cot → 3

appel volumeCube

3

Dans le code fonction

cote → 3

exécution code de la fonction

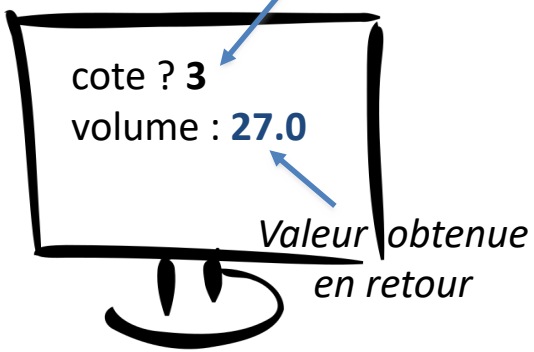
volume → 27

retour

27

vol → 27

Valeur envoyée en paramètre



Python : Fonctions

- **Fonction : valeurs de retour**
  - Un « **return** » permet de **terminer** la fonction et d'envoyer une **valeur en retour** (une **sortie**)

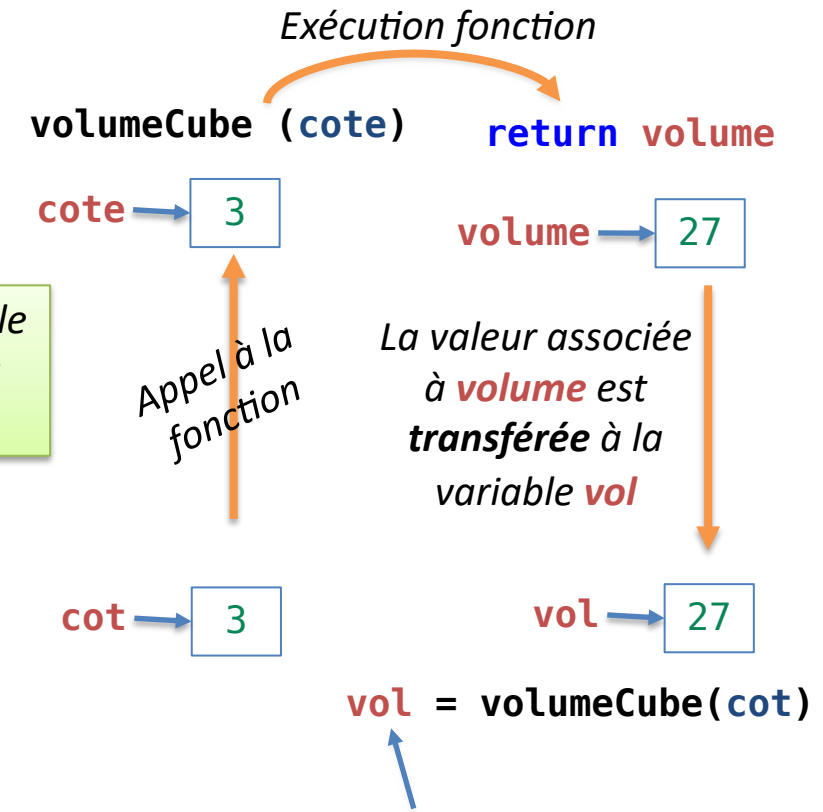
Code fonction

```
def volumeCube (cote) :
    volume = cote ** 3
    return volume

. . .
```

Code principal

```
vol = volumeCube(cot)
print ('volume :', vol)
```

*La valeur indiquée dans le **return** est **affectée** à la variable **vol***



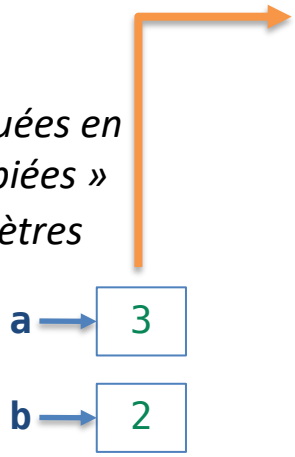
Python : Fonctions

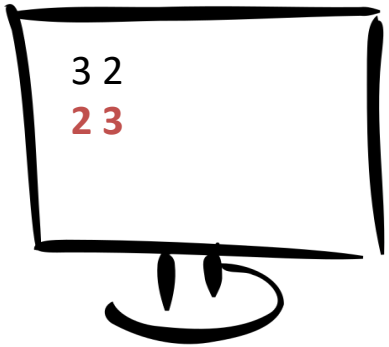
- **Fonction : valeurs de retour**
  - On peut retourner **plusieurs valeurs**
  - C'est comme si on retournait un **tuple** contenant les valeurs

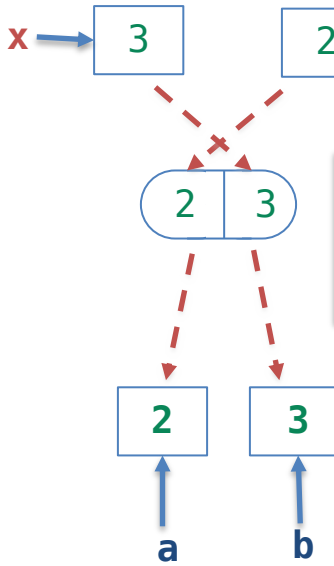
```
def swap (x, y) :
    return y, x
```

Les valeurs indiquées en entrée sont « copiées » vers les paramètres

```
a = 3
b = 2
print (a,b)
a,b = swap (a,b)
print (a,b)
```







*On transfère les **valeurs de retour** aux variable dans le **même ordre***

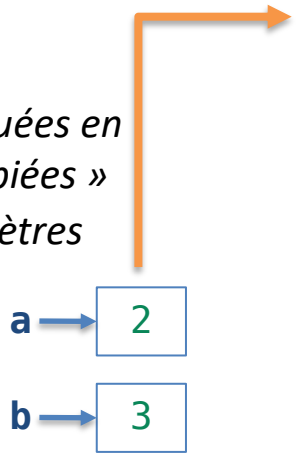
```
a,b = swap (a,b)
```

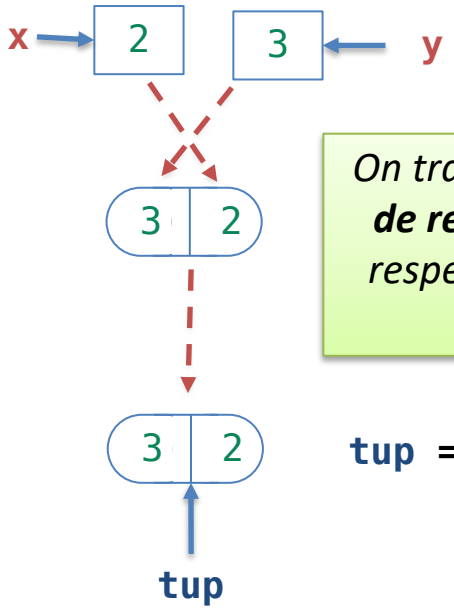
Python : Fonctions

- **Fonction : valeurs de retour**
  - On peut également affecter directement les **valeurs** à un **tuple**

```
def swap (x, y) :
    return y, x
```

Les valeurs indiquées en entrée sont « copiées » vers les paramètres

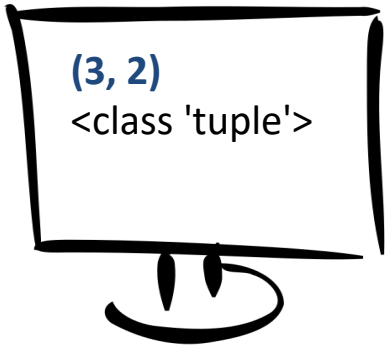




*On transfère les **valeurs de retour** au **tuple** en respectant l'**ordre des valeurs***

```
tup = swap (a,b)
```

```
. . .
tup = swap(a,b)
print (tup)
print (type(tup))
```



Python : Fonctions

- **Fonction :**
  - Un **paramètre** qui a une **valeur par défaut** devient **optionnel** (qu'on n'a **pas besoin de renseigner** à l'appel de la fonction)
  - Si **aucune valeur** n'est renseignée à **l'appel**, la paramètre assume la **valeur par défaut**

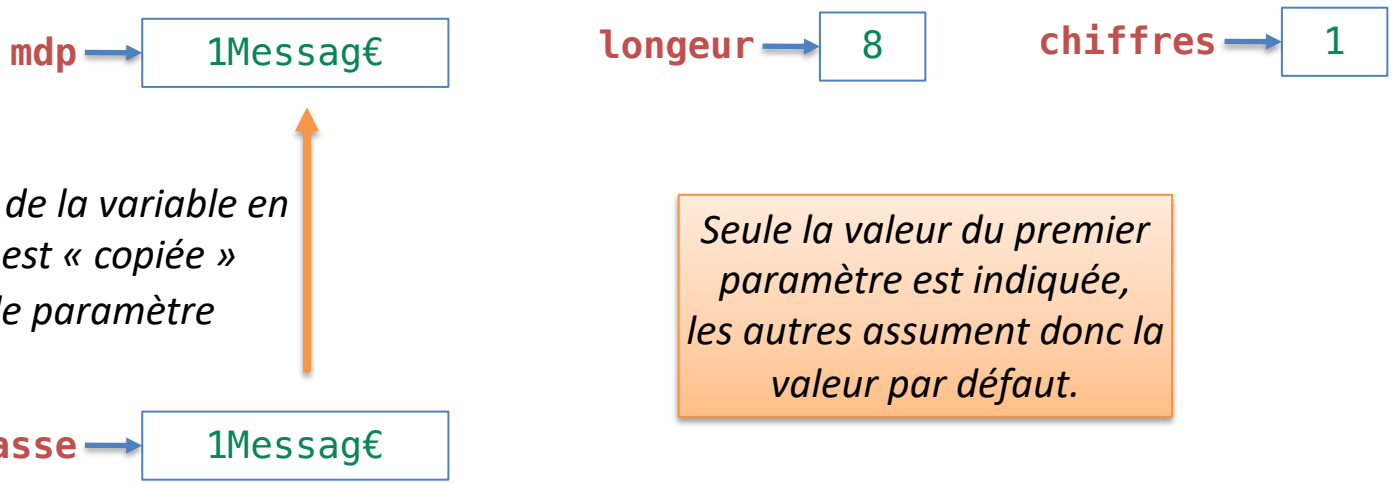
Les paramètres **obligatoires** sont indiqués **au début**

Si **aucune valeur** n'est renseignée, ces paramètres auront les **valeurs par défaut**

```
def validerMdP (mdp, longueur = 8, chiffres = 1 ) :

    nbChiffres = 0

    for lettre in mdp :
        if lettre.isdigit() :
            nbChiffres += 1

    taille = len(mdp)
    if taille >= longueur and nbChiffres >= chiffres :
        estValide = True
    else :
        estValide = False

    return estValide
```

Les paramètres sont des **variables** utilisées dans la **dans la fonction**

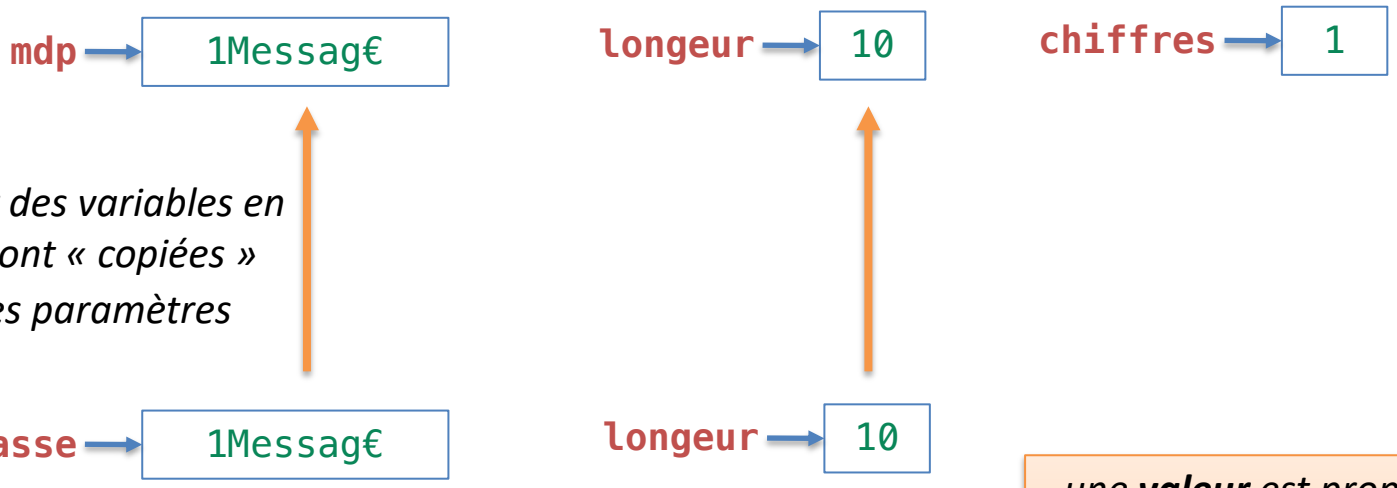
```
def validerMdP (mdp, longueur = 8, chiffres = 1 ) :

    nbChiffres = 0
    ...
    return estValide
```



Seule la valeur du premier paramètre est indiquée, les autres assument donc la valeur par défaut.

```
valide = validerMdP(motdepasse)
print ('valide ?', valide)

. . .
```

```
def validerMdP (mdp, longueur = 8, chiffres = 1 ) :

    nbChiffres = 0
    ...
    return estValide
```




```
valide = validerMdP(motdepasse, longueur=10)
print ('valide ?', valide)
```

*une **valeur** est proposée pour la **longueur**, **celle-ci** est donc **utilisée**, mais **aucune valeur** n'est proposée pour **chiffres**, c'est donc la **valeur par défaut** qui est utilisée.*

Python : Fonctions

- **Fonctions : attention aux paramètres**
  - Les **valeurs** indiquées **en entrée** sont « **copiées** » vers les **paramètres**
  - Si ceux-ci sont **modifiés**, les **variables originelles** (dans le **code principal**) ne sont **pas affectées**…

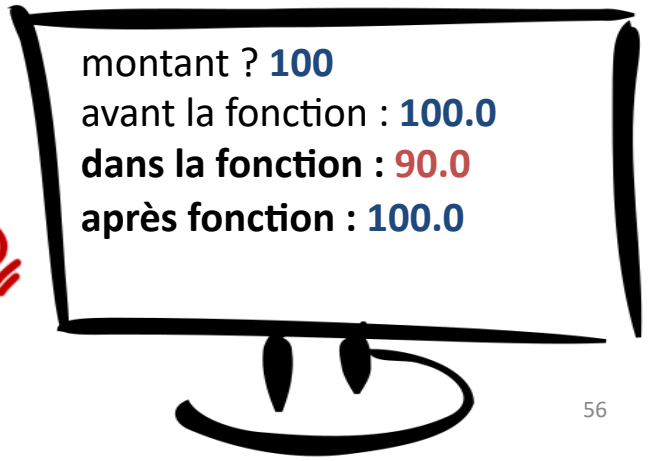
Code fonction

```
def calculerRemise(montant) :
    if 100 <= montant < 150 :
        remise = montant * 0.10
    elif montant >= 150 :
        remise = montant * 0.20
    else :
        remise = 0

    montant -= remise
    print ('dans la fonction :', montant)

    return remise
```

Ça ne **change rien** à la valeur de « **montant** » dans le **code principal**

Code principal

```
montant = float(input('montant ? '))
print ('avant la fonction :', montant)

remise = calculerRemise(montant)
print('après fonction :', montant)
```



UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Fonctions

Que s'est-il passé ?

Suivons les choses dans l'ordre…

1) L'exécution démarre par le code principal…

```
montant = float(input('montant ? '))
print ('avant la fonction :', montant)
```

*2) Grâce à l'exécution de fonction **input**, on va affecter une **valeur** à la **variable montant***

```
remise = calculerRemise(montant)
```

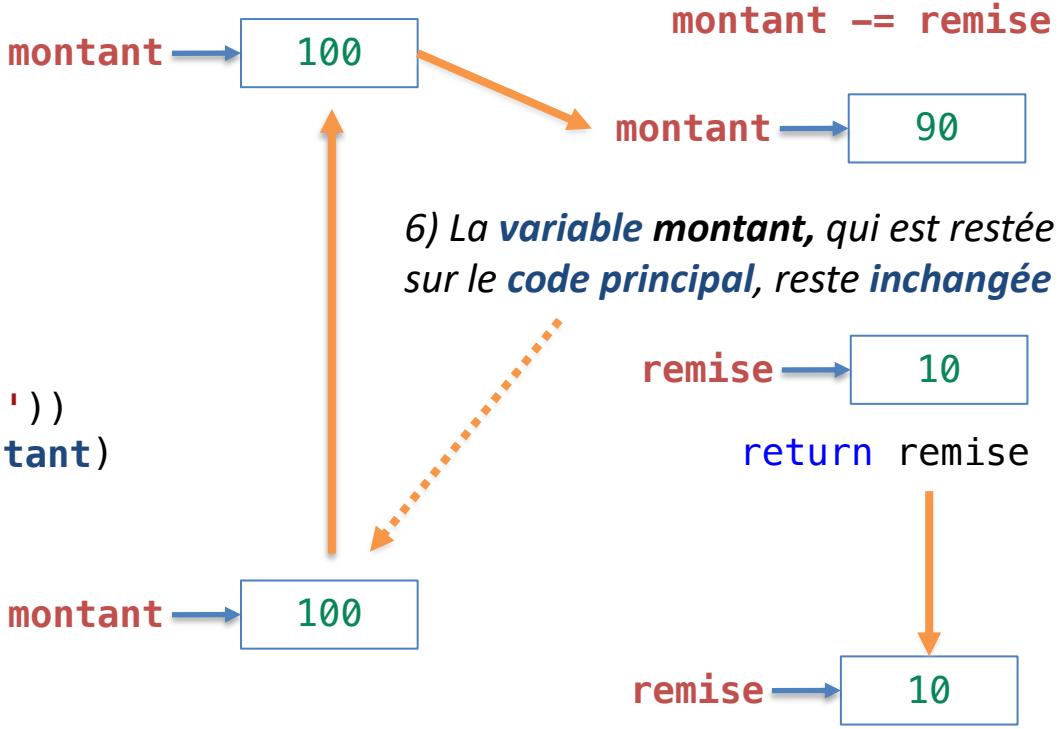
*3) On arrive à **l'appel** à la fonction **calculerRemise**. A ce moment, la **valeur** associée à la **variable montant** est « **copiée** » vers le **paramètre** de la **fonction***

*4) On arrive donc au **code** de la fonction **calculerRemise**. Ce code va utiliser sa propre « **copie** », qui est le **paramètre montant***

```
def calculerRemise(montant) :
    if 100 <= montant < 150 :
```

*5) C'est le **paramètre montant** qui est **modifié***

```
montant -= remise
```



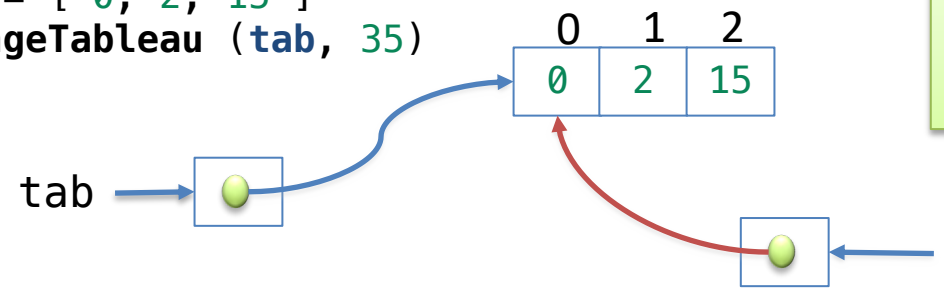
*6) La **variable montant,** qui est restée sur le **code principal**, reste **inchangée***

*7) Au moment du **return**, on revient au **code principal**, et la **variable montant continue** avec la même valeur. Seule la valeur de retour **remise** est **récupérée***

Python : Fonctions

- **Fonctions : attention aux paramètres**
  - Par contre, ça ne marche pas pour les **tableaux** : le **contenu** des tableaux (**listes**) sera **modifié par les fonctions !**
  - **Pourquoi ?**
    - Les **listes** gardent en effet un « ***pointeur*** » (comme une « **adresse** ») vers les valeurs.
    - **L'adresse** est **copiée**, de coup, la fonction connaît l'adresse, elle peut alors **changer la valeur** qui est **à l'adresse**…
    - La **variable originelle** faisant référence à la **même adresse**, elle pourra **observer les changements** réalisés sur les valeurs

```
tab = [ 0, 2, 15 ]
changeTableau (tab, 35)
```

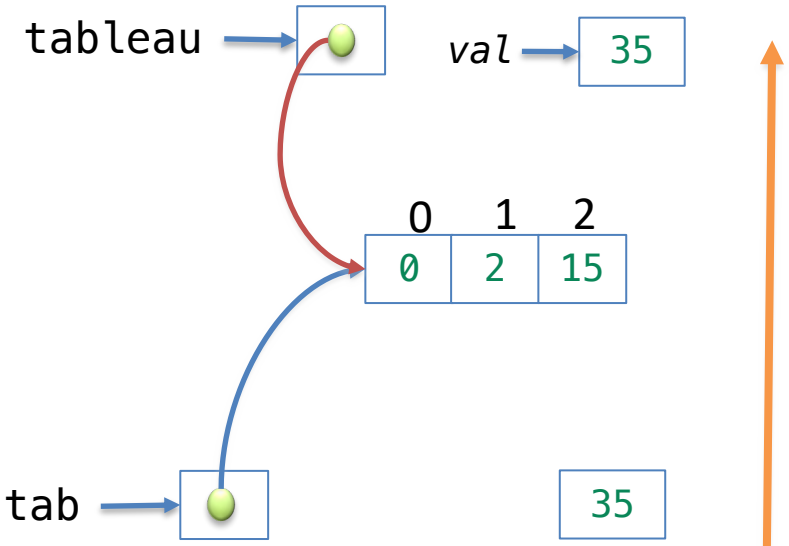


On copie les valeurs. Or une **liste** contient « l'**adresse** » où se trouvent les **valeurs.** On copie donc cette **adresse**, on peut alors aller directement sur les **vraies valeurs.**

```
def changeTableau (tableau, val) :
tableau
```

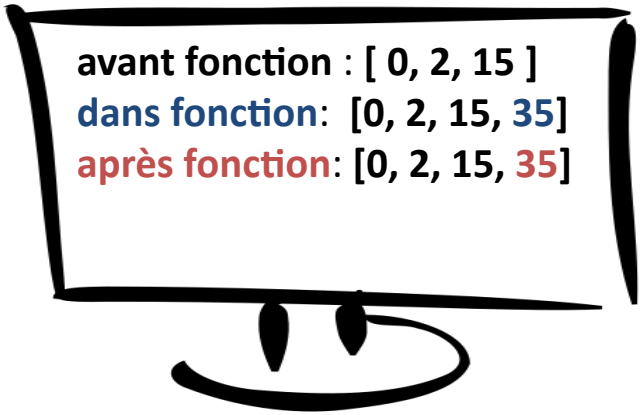
Python : Fonctions

```
def changeTableau (tableau, val) :
    tableau.append(val)
    print('dans fonction: ',tableau)
```



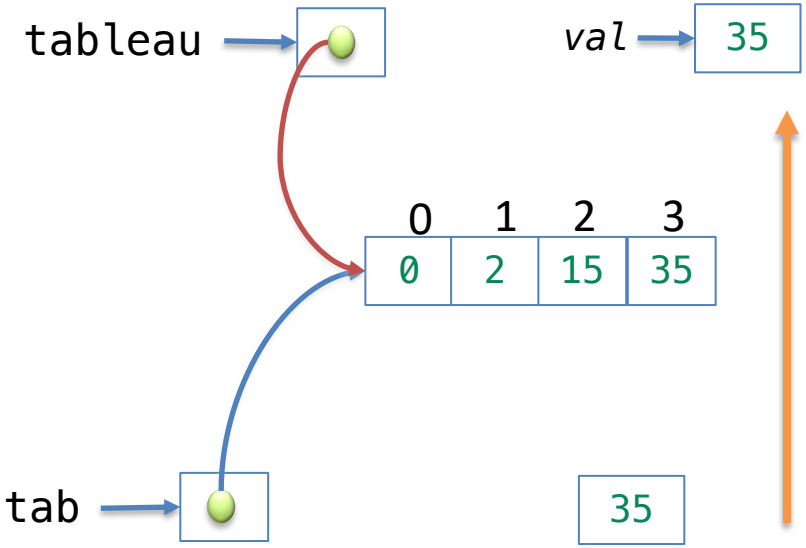
*Les **valeurs** de **l'entrée** « **tab** » sont **transférées** au **paramètre** « **tableau** ».*
*S'agissant d'une **liste**, il s'agit de « l'**adresse** » où se trouvent les vraies **valeurs**.*

```
tab = [ 0, 2, 15 ]
print ('avant fonction :', tab)
changeTableau (tab, 35)
print('après fonction :', tab)
```



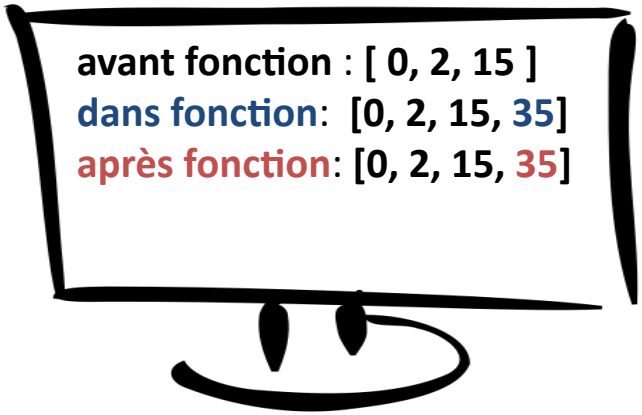
Python : Fonctions

```
def changeTableau (tableau, val) :
    tableau.append(val)
    print('dans fonction: ',tableau)
```



*Lorsqu'on modifie le paramètre « **tableau** » dans la **fonction** (avec l'opération **append**), on **modifie** donc directement les **vraies valeurs**. La variable « **tab** », restée dans le **code principal**, sera donc **affectée**.*

```
tab = [ 0, 2, 15 ]
print ('avant fonction :', tab)
changeTableau (tab, 35)
print('après fonction :', tab)
```



UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE
ÉCOLE DE MANAGEMENT
DE LA SORBONNE

Python : Bibliothèques

- On peut créer nos **propres bibliothèques** avec nos **fonctions**
- **Bibliothèque**
  - **Catalogue de fonctions**
  - Réunir dans un **même fichier** plusieurs **fonctions** ensemble
  - On peut alors **réutiliser** notre bibliothèque avec l'import du ficher

Fichier *MaBibli*.py

```
def volumeCube (cote) :
    """ fonction calculant
        le volume d'un cube """
    volume = cote ** 3
    return volume

def surfaceRectangle (largeur,hauteur) :
    """ calcul de la surface d'un
    rectangle (hauteur x largeur) """

    return (largeur * hauteur)
```

*Import bibliothèque (nom fichier **sans le .py**)*

Fichier *ExempleMaBibliotheque*.py

```
from MaBibli import surfaceRectangle

larg = float (input('Largeur ? '))
haut = float (input('Hauteur ? '))
print (larg, 'x', haut)

surface = surfaceRectangle(larg, haut)
print ('surface :', surface)
```

Usage des fonctions qu'on a fait l'import (comme n'importe quelle autre bibliothèque)

Python : Bibliothèques

```
def volumeCube (cote) :
    """ fonction calculant
        le volume d'un cube """
    volume = cote ** 3
    return volume

def surfaceRectangle (largeur,hauteur) :
    """ calcul de la surface d'un
    rectangle (hauteur x largeur) """

    return (largeur * hauteur)
```

1 *Import de la fonction* ***surfaceRectangle*** *à partir de la bibliothèque* ***MaBibli***

```
from MaBibli import surfaceRectangle

larg = float (input('Largeur ? '))
haut = float (input('Hauteur ? '))
print (larg, 'x', haut)

surface = surfaceRectangle(larg, haut)
print ('surface :', surface)
```

2 *Appel à la fonction* ***surfaceRectangle***