Résumé N°1

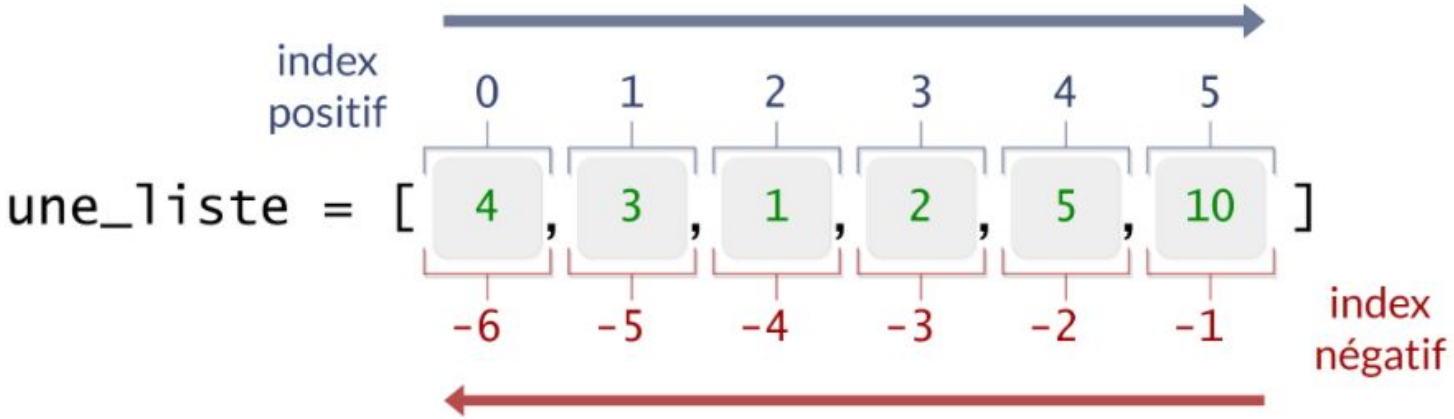
01/03/2021

Tables des matières

Rappels Python

1.1-Les listes

Une **liste** est une structure de données qui peut contenir plusieurs variables de types différents. Elle est **modifiable** et **indexée**. L'indexation commence par **0** dans le sens direct et **-1** dans le sens indirect (l'indice **0** représente le premier élément de la liste et l'indice **-1** représente le dernier).



Pour récupérer une sous-liste d'une liste, on peut procéder par méthode de **découpage**.

- *Pour une liste L :*

  L[i:j] ⇒ La sous-liste de L constituée par les éléments dont les indices sont compris entre i et j-1 .

- *Par exemple* : Pour L=[1,5,3,7,9,6,8] ; L[1 : 5] = [5, 3, 7, 9]

On peut aussi utiliser L[:i] qui renvoi les éléments de L depuis le début jusqu'à l'élément d'indice i-1 ou la commande L[i:] qui renvoi les éléments de L depuis l'élément d'indice i jusqu'à la fin de la liste.

- *Par exemple* : L[:4] = [1, 5, 3, 7] et L[4:] = [9, 6, 8]

Centrale IA

Pour supprimer un élément d'une liste, on peut utiliser la méthode **pop(i)**. Cette dernière supprime l'élément d'**indice i** et permet de le **récupérer** .

- Par exemple

```
>>> L=[1,5,3,7,9,6,8]
>>> a=L.pop(4)
>>> print(L)
[1,5,3,7,6,8]
>>> print(a)
9
```

Pour ajouter un élément, on peut utiliser la méthode **insert(i,x)**. Elle permet d'insérer l'élément x à l'**indice i.**
La méthode **append(x)** permet d'ajouter l'élément x à la **fin de la liste**.

- Par exemple : et L.append('CIA')>>>[1,5,3,7,9,6,8,'CIA'].

```
>>> L.insert(3, "CIA")
[1, 5, 3, "CIA", 7, 9, 6, 8]
```

```
>>> L.append("CIA")
[1, 5, 3, 7, 9, 6, 8, "CIA"]
```

Pour fusionner deux listes, on utilise la méthode **L1.extend(L2)**. Elle permet d'ajouter la liste L2 à la fin de la liste L1.

```
>>> liste_1 = ["Bonjour", "comment", "ça", "va", "?"]
>>> liste_2 = ["Bien", "et", "toi", "?"]
>>> liste_1.extend(liste_2)
>>> print(liste_1)
["Bonjour", "comment", "ça", "va", "?" ,"Bien", "et", "toi", "?"]
```

Cette tâche peut être aussi réalisée en utilisant la syntaxe: **L1=L1+L2**.

1.2-Les tuples:

Les **tuples** sont des structures de données qui ressemblent aux listes et qui utilisent la même indexation.

Ils ne sont pas modifiables et peuvent être définis avec ou sans parenthèses.

un_tuple = (5 , True , "A") ⇔ un_tuple = 5 , True , "A"

Ils permettent aussi d'assigner des valeurs plus facilement.

Exemple : Soit T un tuple contenant 3 éléments . Pour stocker ses valeurs dans des variables x , y , z, il suffit d'écrire **x , y , z = T**.

Les tuples sont aussi utilisés pour interchanger les valeurs. On peut échanger les valeurs de deux variables a et b avec : **a , b = b , a**.

1.3-Les dictionnaires:

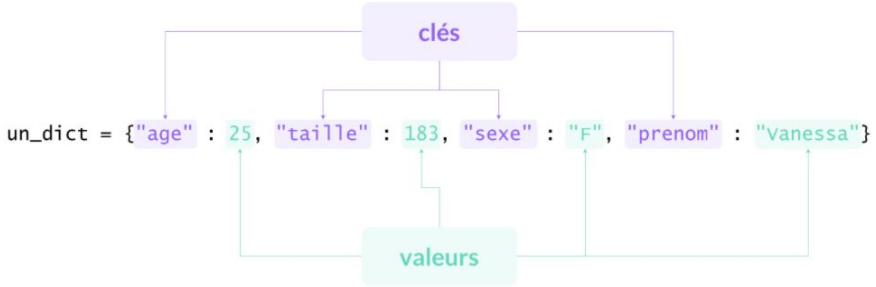
Les **dictionnaires** sont des structures de données particulières définis par des crochets **{ }**.

Comme pour les listes, ils peuvent contenir plusieurs éléments mais leur **indexation** se fait par des **clés**. Ils peuvent contenir **n'importe quel type de variable**.

Pour stocker une valeur var de clé c dans un dictionnaire D, on utilise la syntaxe : **D[c]= var**.

Pour accéder à l'élément indexée par la clé c, on utilise **D[c]**.

Pour supprimer une variable d'un dictionnaire, on utilise la même méthode **pop()** vu pour les listes, mais cette fois-ci, elle prend comme argument la **clé** de la variable.



1.4-Les fonctions *enumerate* et *ZIP*:

La fonction enumerate est une fonction utilisée au niveau de la boucle for. Elle permet d'accéder à la fois à un élément ainsi que son index.

Par exemple, dans la boucle **for i , elt in enumerate(L):** *elt* stocke l'élément, alors que i stocke son indice.

- *Exemple*

```
>>> for i,elt in enumerate([5,6,1]):
>>>       print (( i , elt ))
>>>
(0,5)
(1,6)
(2,1)
```

La fonction **zip** permet quant à elle de parcourir plusieurs séquences de même longueur dans une seule boucle for en utilisant la syntaxe suivante :

for elt1,elt2,... in zip(seq1,seq2,...)

Pour mieux comprendre voici un petit exemple:

```
>>> L1 , L2 , L3 , L = [1,8,3] , [5,7,1] , “CIA” , []
>>> for e1,e2,e3 in zip(L1,L2,L3) :
>>>      L.append((e1+e2,e3))
>>> print(L)
[(6, 'C'), (15, 'I'), (4, 'A')]
```

1.5-Les fonctions:

Les **fonctions** peuvent prendre des **arguments par défaut** au cas ou on ne les fournit pas. Cela se fait au niveau de la définition de la fonction en faisant en attribuant une valeur par défaut à l'argument.

Exemple :

```
>>> def mult(a=1, b=2): return a*b
>>> print( mult(5, ) )
>>> print( mult(, 3) )
>>> print( mult(,) )
10
3
2
```

La **documentation des fonctions** est une étape utile surtout lorsqu'on travaille sur de longs programmes ou quand notre fonction sera utilisée par plusieurs utilisateurs. Il s'agit d'un commentaire explicative de la fonction qui apparaît quand on utilise la fonction **help()** qui est utilisée également pour les fonctions prédéfinis.

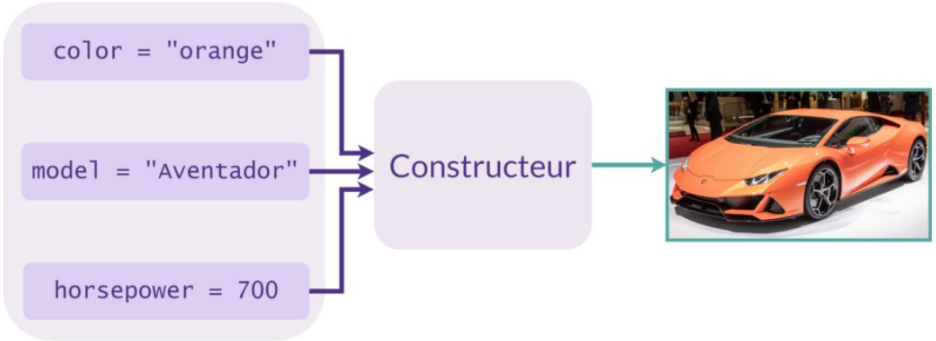
La documentation se fait dans la première ligne du bloc de définition en écrivant le commentaire délimité par des " ".

- *Exemple* :

```
>>> def mult ( a , b ) :
>>>       """ cette fonction permet de multiplier les arguments d'entré """
>>>       return a*b
>>> print( help( mult ) )
Help on function mult in module ___main___:   mult( a , b)

     cette fonction permet de multiplier les arguments d'entré

     None
```

1.6 Les classes:

En programmation orientée objet (elle sera traitée lors d'une autre séance), on crée des classes d'objets qui sont des types d'objets avec leurs propres attributs, fonctions et outils de manipulation. Elles sont définies par 3 types d'éléments qui sont : le **constructeur**, les **attributs** et les **méthodes**.



- Le **constructeur** : fonction qui permet d'initialiser/créer un objet de la classe.
- Les **attributs** : sont les variables propres aux objets de la classe.
- Les **méthodes** : sont des fonctions prédéfinies qui opèrent sur les objets de la classe.

Les classes sont définies par la clause **class** qui permet de débuter un bloc où on va définir notre classe. La définition du constructeur se fait par la méthode **__init__** ce qui va permettre d'initialiser les attributs. Cette méthode prend en argument d'abord **self** et ensuite les **attributs** de l'objet qu'on veut construire . **self** correspond à l'objet qui fait appel aux méthodes de la classe et permet d'accéder aux attributs de l'objet défini.

Toutes les méthodes d'une classe doivent avoir comme premier argument l'argument **self**. On reprend l'exemple de la voiture dont la classe est définie comme suit :

Rappels Python

Centrale IA

```
class Car : #définition de la classe
        def __init__ ( self , color , model , horsepower ) : #définition du constructeur
                #initialisation des attributs de la classe
                self.color = color
                self.model = model
                self.horsepower = horsepower
            #définition d'une méthode qui permet de changer la couleur
        def change_color ( self , new_color ) :
                self.color = new_color

#Création d'un objet de la classe car
        aventador = Car( color = "orange" , model = "Avendator" , horsepower = 700 )
```

On ajoute aussi un autre exemple où on définit une classe pour les nombres complexes :

```
class Complexe :
        def __init__ ( self , a , b ) :
                self.partie_re = a
                self.partie_im = b
        #Definition d'une méthode qui permet d'afficher le complexe sous forme algébrique
        def afficher(self):
                if ( self.partie_im < 0 ) :
                        print( self.partie_re, "-" , -self.partie_im , "i")
                elif ( self.partie_im == 0 ) :
                print( self.partie_re )
        else :
                print( self.partie_re, "+" , self.partie_im , "i")
>>> val = Complexe( 5 , -2 )
>>> val.afficher()
5 - 2 i
```

Comme pour les fonctions, il est important de **documenter** les classes et leurs méthodes.

Pour ajouter un commentaire pour la classe toute entière, on l'écrit dans la ligne qui suit la clause class entre des " ".

Pour ajouter un commentaire pour une méthode de la classe, on fait de même mais juste après la clause def. Pour visualiser ces commentaires, on aura toujours recours à la fonction **help()**.

Pour les classes, cette fonction (**help()**) prend comme argument le nom de la classe. Pour leurs méthodes de classe, elle prend comme argument le nom de la classe suivit d'un point et puis le nom de la méthode.

- Exemple :

```
help( Car )   # Pour afficher la documentation de la classe.
help ( Car.change_color )   # Pour afficher la documentation de la méthode change_color.
```

Les numpy array

2.1 Création des numpy array:

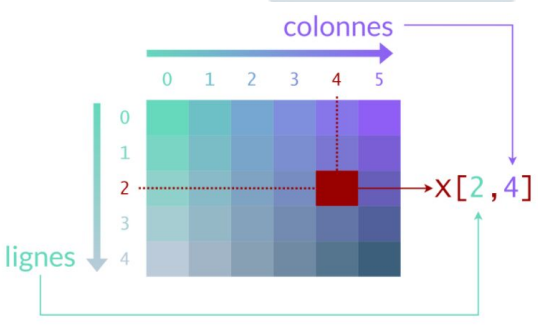
Un array est une classe d'objets du module numpy qui sert à représenter des matrices. Ces objets peuvent être initialisés par plusieurs constructeurs qui prennent en général un tuple de dimension nommé **shape**:

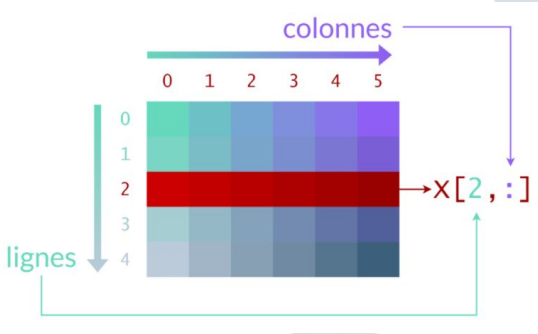
Exemple :

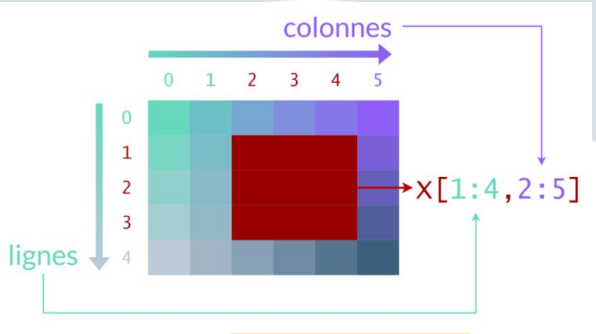
```
import numpy as np

x=np.zeros( shape = ( 8 , 4 ) ) # La méthode .zeros( ) va permettre de créer un array rempli de zéros de dim ( 8 , 4 ).

x=np.ones( shape = ( 8 , 4 ) ) # La méthode .ones( ) va permettre de créer un array remplis de 1 de dim ( 8 , 4 ).

x=np. array( [ [ 4 , 5 , 7] , [ 9 , 3 ,0 ] ] ) # La méthode .array() permet de créer un array à partir de la liste en argument.
```

2.2 Indexation d'un numpy array:

L'indexation d'un numpy array se fait sur toute les dimensions. Comme pour les listes, l'indexation commence de 0. Par exemple, pour accéder à un élément d'un numpy array X, qui se trouve sur la ligne 3 et la colonne 5 on utilise la syntaxe suivante : `X[2:4]`. On peut aussi utiliser le **slicing** comme l'illustre la figure suivante :







Centrale IA

Exemple : On cherche à créer la matrice suivante en utilisant le slicing :

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & -1 & -1 \end{pmatrix}$$

Le code qui permet de la créer et le suivant :

numpy array

```
import numpy as np
mat = np.zeros( shape = ( 6 , 6 ) )
mat [ :3 , :3 ] = 1
mat [ 3: , 3: ] = -1
print( mat )
>>>
[ 1. 1. 1. 0. 0. 0. ]
1. 1. 1. 0. 0. 0. ]
1. 1. 1. 0. 0. 0. ]
0. 0. 0. -1. -1. -1. ]
0. 0. 0. -1. -1. -1. ]
0. 0. 0. -1. -1. -1. ] ]
```

```
[
[
[
[
[
[
```

2.3 Quelques fonctions du module numpy:

Le module **numpy** ne sert pas uniquement à créer des tableaux, mais il permet aussi de faire des **calculs optimisés**. On y retrouve alors plusieurs fonctions applicables à la fois aux tableaux et à d'autres types de variables comme :

Fonction	Fonction Numpy
e^x	`np.exp(x)`
$\log(x)$	`np.log(x)`
$\sin(x)$	`np.sin(x)`
$\cos(x)$	`np.cos(x)`
Arrondi à **n** décimales	`np.round(x, decimals = n)`

2.4 Indexation conditionnelle d'un numpy array:

En plus de l'indexation par slicing, les tableaux numpy peuvent être indexés par des conditions

- *Exemple*

```
>>> X = np.array( [ [ -1 , 0 , 30 ] ,
                    [ -2 , 3 , -5 ] ,
                    [ 5 , -5 , 10 ] ] )
>>> X[X<0] = 0
>>> print( X )
[[ 0  0 30]
 [ 0  3  0]
 [ 5  0 10]]
```

Centrale IA

Il est aussi possible d'indexer un tableau numpy par une condition appliquée sur un tableau comme on le voit sur l'exemple ci-dessous :

```
>>> X = np.array([3, -7, -10, 3, 6, 5, 2, 9])
>>> Y = np.array([0, 1, 1, 1, 0, 1, 0, 0])
>>> X[Y == 1] = -1
>>> print(X)
[3 -1 -1 -1 6 -1 2 9]
>>> print(X[Y == 0])
[3 6 2 9]
```

2.5 Redimensionnement d'un numpy array:

Parfois, il est indispensable de redimensionner un tableau. Cette démarche connue aussi sous le nom de **reshaping** se fait à l'aide de la méthode **reshape** qui prend en argument le shape désiré. Il faut juste s'assurer que le nombre d'élément contenu dans le shape cible soit égal au nombre initial d'éléments dans l'array.

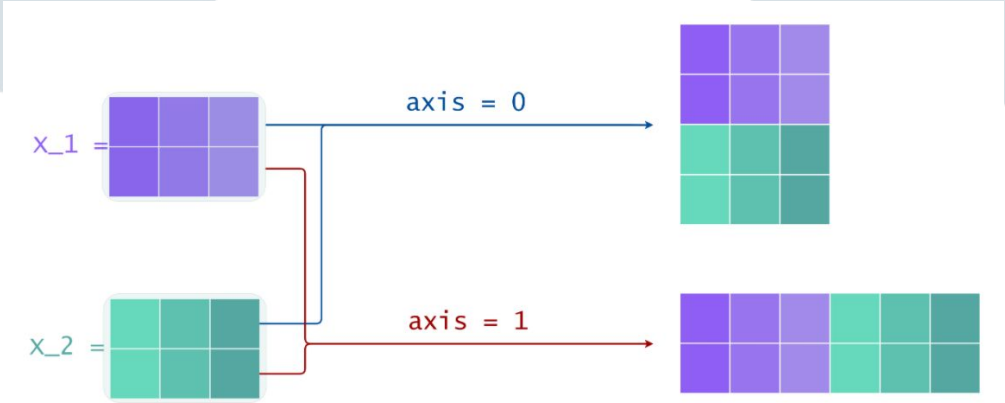
Exemple :

```
>>> X = np.array([i for i in range(1, 11)])
>>> print(X)
[1  2  3  4  5  6  7  8  9 10]
>>> X_reshaped = X.reshape((2, 5))
>>> print(X_reshaped)
[[ 1  2  3  4  5]
[ 6  7  8  9 10]]
```

2.6 Concaténation des tableaux:

Parfois on a besoin de fusionner plusieurs tableaux dans un seul. Pour effectuer cette tâche, on utilise la fonction **concatenate()** du module **numpy**. Elle prend deux arguments : le premier est sous forme d'une liste ou d'un tuple contenant les tableaux à concaténer et le second contient " axis " . Il s'agit de la **dimension de concaténation**.

N.B: Les tableaux doivent être de même taille.



numpy array

Centrale IA

Quelques exemples :

```
>>>X_1 = np.ones(shape = (3, 2))
>>>print(X_1)
[[1. 1.]
[1. 1.]
[1. 1.]]
>>>X_2 = np.zeros(shape = (3, 2))
>>>print(X_2)
[[0. 0.]
 [0. 0.]
 [0. 0.]]
>>>X_3 = np.concatenate([X_1, X_2], axis = 1)
>>>print(X_3)
[[1. 1. 0. 0.]
[1. 1. 0. 0.]
[1. 1. 0. 0.]]
```

2.7 Opérateurs arithmétiques:

En python, il est possible d'appliquer les opérateurs arithmétiques de base (`*` , `/` , `+` , `-` , `**`) entre un **tableau** et une **valeur**. Cela conduit à appliquer l'opération entre chaque élément du tableau et cette valeur.

Il est aussi possible d'appliquer ces opérateurs entre **deux tableaux.** L'opération sera appliquée entre chaque deux éléments de même indice. Pour effectuer un produit matriciel, on fait recours à la méthode **.dot()** des **np.array()**.

- Par exemple :

```
>>>M = np.array([[5, 1],[3, 0]])
>>>N = np.array([[2, 4],[0, 8]])
>>>print(M.dot(N))
[[10 28]
 [ 6 12]]
```

2.8 Broadcasting:

Lorsqu'on effectue une opération entre deux éléments de dimensions différentes, numpy effectue ce qu'on appelle le **broadcasting** pour comprendre l'opération à effectuer et l'appliquer.

Par exemple pour le broadcasting entre une **matrice** et une **valeur c**, numpy crée une matrice remplie de c et de même taille que notre matrice afin d'effectuer l'opération entre ces deux dernières.

Pour le broadcasting entre une **matrice** et un **vecteur**, **numpy** nous permet d'effectuer ce type d'opérations mais sous certaines conditions. Pour vérifier que deux tableaux sont compatibles pour ce genre d'opérations, numpy **compare** les dimensions des deux tableaux et vérifie si **elles sont égales** ou si l'**une des deux est égale à 1**.

Une fois l'une des deux conditions est vérifiée pour chaque dimension, numpy arrive à comprendre l'opération désignée. Dans le cas contraire, on reçoit un message d'erreur.

Exemple : Pour la matrice et le vecteur suivants : $M = \begin{pmatrix} 3 & 1 & 2 \\ -2 & 1 & 5 \end{pmatrix}, v = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$

Le broadcasting pour M*v donne :

$$v = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \xrightarrow{\text{broadcasting}} V = [v \quad v \quad v] = \begin{pmatrix} 2 & 2 & 2 \\ 5 & 5 & 5 \end{pmatrix}$$

$$M * v \xrightarrow{\text{broadcasting}} M * V = \begin{pmatrix} 3*2 & 1*2 & 2*2 \\ -2*5 & 1*5 & 5*5 \end{pmatrix} = \begin{pmatrix} 6 & 2 & 4 \\ -10 & 5 & 25 \end{pmatrix}$$

Centrale IA

Si on prend maintenant un nouveau vecteur u de dim (1, 2) : $u = \begin{pmatrix} 3 & 4 \end{pmatrix}$

Pour l'opération u+vle broadcasting donne :

$$v = \begin{pmatrix} 2 \\ 5 \end{pmatrix} \xrightarrow{\text{broadcasting}} V = \begin{pmatrix} 2 & 2 \\ 5 & 5 \end{pmatrix}$$

$$u = \begin{pmatrix} 3 & 4 \end{pmatrix} \xrightarrow{\text{broadcasting}} U = \begin{pmatrix} 3 & 4 \\ 3 & 4 \end{pmatrix}$$

$$v + u = \begin{pmatrix} 2 \\ 5 \end{pmatrix} + \begin{pmatrix} 3 & 4 \end{pmatrix}$$

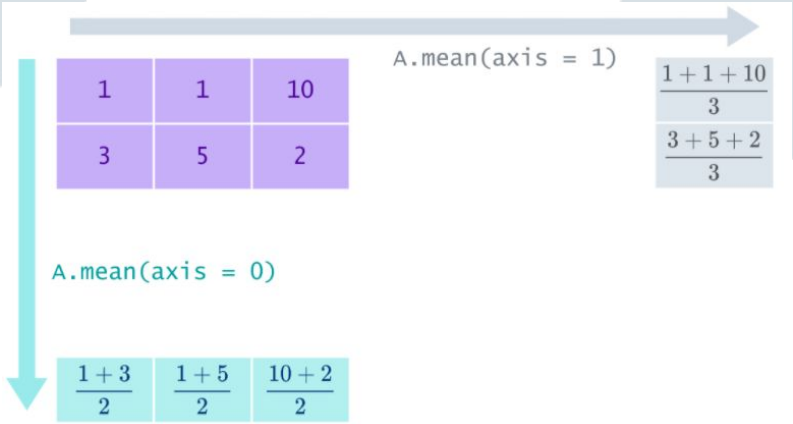
$$\xrightarrow{\text{broadcasting}} V + U = \begin{pmatrix} 2 & 2 \\ 5 & 5 \end{pmatrix} + \begin{pmatrix} 3 & 4 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix}$$

2.9 Les méthodes statistiques:

Les tableaux numpy disposent aussi de certaines méthodes pour les calculs complexes. L'une des méthodes les plus utilisées est la méthode **.mean()**. Cette dernière permet de calculer la **moyenne d'un tableau**.

Par exemple, pour **A = np.array([[1, 1, 10],[3, 5, 2]])** , **print(A.mean())** donne : **3.67.**

Pour calculer la moyenne suivant un **axe i** donné, il suffit de donner à cette méthode ***axis = i*** comme argument (**print(A.mean(axis=0)**). Dans l'exemple suivant, on trouve ***[2. 3. 6.]*** ("calcule des moyennes suivant les colonnes").



Il existe d'autres méthodes statistiques comme :

sum: Calcule la somme des éléments d'un tableau.

std: Calcule l'écart type.

min: Trouve la valeur minimale parmi les éléments d'un tableau.

max: Trouve la valeur maximale parmi les éléments d'un tableau.

argmin: Renvoie l'indice de la valeur minimale.

argmax: Renvoie l'indice de la valeur maximale.

Chargement et exploration des jeux de données

3.1 Le module pandas:

Le module ***pandas*** a été développé pour fournir les outils nécessaires pour analyser et manipuler de gros volumes de données. Ce module introduit la classe DataFrame, une structure de données pour représenter des tableaux et qui propose une manipulation et une exploration de données plus poussée que les numpy arrays.

Ce module permet aussi la récupération des données depuis des fichiers et les manipuler.

3.2 Format d'un DataFrame:

Un **DataFrame** se retrouve sous la forme de matrice dont chaque ligne et chaque colonne porte un indice. En général, les colonnes sont indexées par leurs noms.

Les **DataFrame** servent à stocker des bases de données. Les différents entrées (individus, animaux, objets, etc.) sont représentées par les différentes lignes et leurs caractéristiques sont les différentes colonnes.

	Nom	Sexe	Taille	Age
0	Robert	M	174	23
1	Mark	M	182	40
2	Aline	F	169	56

La colonne contenant les numérotations des lignes est appelée ***index*** et ne se gère pas de la même façon qu'une colonne du dataset. De plus, chaque colonne contient le même type de variables.

On peut également indexer par l'une des colonnes du DataFrame ou par une liste qu'on définit nous même:

Exemple : Indexation par la colonne `'Nom'`:

	Sexe	Taille	Age
Robert	M	174	23
Mark	M	182	40
Aline	F	169	56

Exemple : Indexation par la liste `['personne_1', 'personne_2', 'personne_3']`:

	Nom	Sexe	Taille	Age
personne_1	Robert	M	174	23
personne_2	Mark	M	182	40
personne_3	Aline	F	169	56

3.3 Création d'un DataFrame à partir d'un numpy array:

Avec pandas, on peut créer un DataFrame à partire d'un numpy array à l'aide du constructeur .DataFrame(). Mais l'inconvénient de cette méthode est que le type de données doit être le même pour toutes les colonnes. Ce constructeur prend comme **1er argument** le numpy array et comme **2ème argument** une liste contenant les indices d'entrées. Pour le **3ème argument**, il s'agit d'une liste contenant le nom des colonnes.

- *Exemple* :

```
>>>array = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>>df = pd.DataFrame(data = array, index = ['i_1', 'i_2', 'i_3'], columns = ['A', 'B', 'C', 'D'])
```

Le DataFrame df qu'on a créé est sous la forme :

	A	B	C	D
i_1	1	2	3	4
i_2	5	6	7	8
i_3	9	10	11	12

3.4 Création d'un DataFrame à partir d'un dictionnaire:

Pour créer un DataFrame à partir d'un dictionnaire on utilise encore le constructeur .DataFrame() , mais cette fois, il ne prend que deux arguments. Le **premier** est un dictionnaire contenant les éléments de chaque colonne rangés dans des listes et dont les clés sont les noms des colonnes. Le **deuxième argument** est une liste des indices d'entrées.

- **Exemple** :

```
>>>D= {'A': [1, 5, 9], 'B': [2, 6, 10], 'C': [3, 7, 11],
'D': [4, 8, 12]}
>>>df = pd.DataFrame( D , ['i_1', 'i_2', 'i_3'])
>>>print(df)
A   B   C   D
i_1    1   2   3   4
i_2  5   6   7   8
i_3  9  10  11  12
```

3.5 Création d'un DataFrame à partir d'un fichier de données:

Les **DataFrame** sont souvent créés à partir d'un fichier (CSV ,Excel ,Text ...). Le format le plus souvent utilisé est le ***CSV***. *Exemple* :

```
A, B, C, D,
1, 2, 3, 4,
5, 6, 7, 8,
9, 10, 11, 12
```

Avec ce format, la première ligne contient les noms des colonnes. Mais il se peut que cette données ne soit pas fournies. Chaque ligne correspond à une entrée de la base de données et les valeurs sont séparées par un caractère comme " , " ou " ; " .

Pour créer un DataFrame à partir de ce type de fichiers, on utilise la syntaxe :

```
pd.read_csv(filepath_or_buffer , sep = ',', header = 0, index_col = 0 ... )
```

Les arguments essentiels pour la fonction **pd.read_csv** à connaître sont:

- **filepath_or_buffer**: Le chemin d'accès du fichier **.csv** relativement à l'**environnement d'exécution**. Si le fichier se trouve dans le même dossier que l'environnement Python, il suffit de renseigner le nom du fichier. Ce chemin doit être renseigné sous forme de chaîne de caractères.
- **sep**: Le caractère utilisé dans le fichier .csv pour séparer les différentes colonnes. Cet argument doit être spécifié sous forme de caractères.
- **header**: Le numéro de la ligne qui contient les noms des colonnes. Si par exemple les noms de colonnes sont renseignés dans la **première ligne du fichier .csv**, alors il faut spécifier **header = 0**. Si les noms ne sont pas renseignés, on mettra **header = None**.
- **index_col** : Le nom ou numéro de la colonne contenant les indices de la base de données. Si les entrées de la base sont indexées par la **première colonne**, il faudra renseigner **index_col = 0**. Alternativement, si les entrées sont indexées par une colonne qui porte le nom **"Id"**, on pourra spécifier **index_col = "Id"**.

Cette fonction retournera un objet de type **DataFrame** qui contient toutes les données du fichier.

3.6 Visualisation d'un DataFrame : méthode head, attributs columns et shape:

Il est possible de visualiser que les premières ou les dernières lignes d'un DataFrame. On utilise pour cela les méthodes **.head()** et **.tail()**. Elles prennent en argument le nombre n des lignes à afficher. La méthode **.head()** affiche les premières n lignes le moment où **.tail()** affiche les n dernières. Pour récupérer les noms des colonnes d'un DataFrame, on utilise l'attribut **.columns**. D'autre part, pour récupérer ses dimensions, on utilise l'attribut **.shape** . Si on revient sur l'exemple précédent du DataFrame créé à partir d'un dictionnaire, on a :

```
>>>print(df.columns)
['A', 'B', 'C', 'D']
>>>print(df.shape)
(3,4)
```

3.7 Sélection des colonnes d'un DataFrame:

L'extraction des colonnes d'un DataFrame est presque identique à l'extraction de données d'un dictionnaire. En effet pour extraire une colonne d'un DataFrame, il suffit de renseigner entre crochets le nom de la colonne à extraire.

Pour extraire plusieurs colonnes, on renseigne la liste des noms des colonnes à extraire.

- **Exemple** :

```
>>>Nv_df = df [ [ A , C ] ]
>>>print( Nv_df )
A    C
i_1  1     3
i_2  5     7
i_3  9    11
```

3.8 Sélection de ligne d'un Dataframe: méthodes loc et iloc:

Afin d'extraire une ou plusieurs lignes d'un DataFrame , on fait recours à la méthode **.loc[]**.

NB: Cette méthode prends les arguments entre [] et non des ().

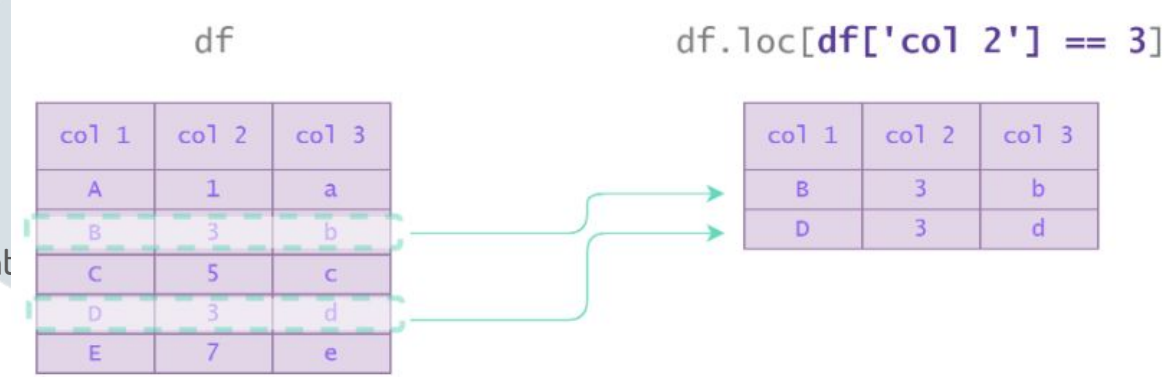
Pour **extraire une ligne**, on renseigne son indice et pour en extraire plusieurs ,on renseigne une liste contenant les indices des lignes.

Afin de récupérer **plusieurs lignes,** on peut aussi utiliser le slicing en précisant les indices de début et de fin. Pour l'utiliser, il faut que les indices soient uniques, ce qui n'est pas toujours le cas.

Enfin, **.loc[]** peut aussi prendre comme deuxième argument une colonne ou liste de colonnes afin d'affiner l'extraction de données. La méthode **.iloc[]** permet d'indexer un DataFrame exactement comme les tableaux numpy, c'est-à-dire en ne renseignant que les indexes numériques des lignes et des colonnes. Ceci permet d'utiliser le slicing sans contraintes.

3.9 Indexation conditionnelle d'un DataFrame :

Comme on l'a vu avant avec les tableaux numpy, on peut utiliser l'indexation conditionnelle sur les DataFrame. Pour mieux comprendre, on prend l'exemple suivant :



n effet, cette indexat

E

NB : Si on veut assigner une nouvelle valeur à ces entrées, il faut absolument utiliser la méthode .**loc**[]

3.10 Rapide étude statistique des données d'un DataFrame:

La méthode **.describe** d'un DataFrame retourne un résumé des statistiques descriptives (**min**, **max**, **moyenne**, **quantiles**,..) de ses variables quantitatives

→ "Une variable quantitative est une variable qui mesure une quantité pouvant prendre une infinité de valeurs" . C'est donc un outil très utile pour une première visualisation du type et de la distribution des variables.

→ "Une variable catégorielle est une variable qui ne prend qu'un nombre fini de modalités"

Pour analyser les variables catégorielles, il est préférable de commencer par utiliser la méthode **.value_counts** qui renvoie le nombre d'occurrences pour chaque modalité de variables. La méthode **.value_counts** ne peut pas s'utiliser directement sur un DataFrame mais que sur ses colonnes qui sont des objets de la classe **pd.Series**.