

# SY09 Printemps 2020

## TP 0 — Introduction à NumPy et Pandas

### 1 Introduction à NumPy

NumPy est une bibliothèque pour le calcul numérique en Python. Pour charger la bibliothèque, il suffit d'exécuter l'instruction suivante :

```
In [1]: import numpy as np
```

#### Tableaux unidimensionnels

##### Création

Des tableaux unidimensionnels peuvent être créés à partir d'une simple liste Python.

```
In [2]: a = np.array([1, 2, 3])
        b = np.array(range(4))
```

Il existe des fonctions prédéfinies pour créer rapidement des séquences.

```
In [3]: np.arange(10)
Out [3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [4]: np.linspace(0, 5, 10)
Out [4]: array([0.          , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
               ↪ 2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.          ])
In [5]: np.logspace(0, 2, 5)
Out [5]: array([ 1.          ,  3.16227766, 10.          , 31.6227766 , 100.
               ↪ ])
In [6]: np.geomspace(0.1, 100, 4)
Out [6]: array([ 0.1,  1. , 10. , 100. ])
```

① Définir les tableaux suivants :

- $t_1 = (1, 10, 100, \dots, 10^{10})$
- $t_2 = (0, 2, 4, \dots, 100)$
- $t_3 = (0, -1, -2, \dots, -10)$

```
In [7]: np.logspace(0, 10, 11)
Out [7]: array([1.e+00, 1.e+01, 1.e+02, 1.e+03, 1.e+04, 1.e+05, 1.e+06, 1.e+07,
               ↪ 1.e+08, 1.e+09, 1.e+10])
In [8]: np.linspace(0, 100, 51)
Out [8]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18., 20.,
               ↪ 22., 24., 26., 28., 30., 32., 34., 36., 38., 40., 42.,
               ↪ 44., 46., 48., 50., 52., 54., 56., 58., 60., 62., 64.,
               ↪ 66., 68., 70., 72., 74., 76., 78., 80., 82., 84., 86.,
               ↪ 88., 90., 92., 94., 96., 98., 100.])
In [9]: np.linspace(0, -10, 11)
```

```
Out [9]: array([ 0., -1., -2., -3., -4., -5., -6., -7., -8., -9.,  
               ↪ -10.])
```

On peut également spécifier la longueur et la valeur d'initialisation.

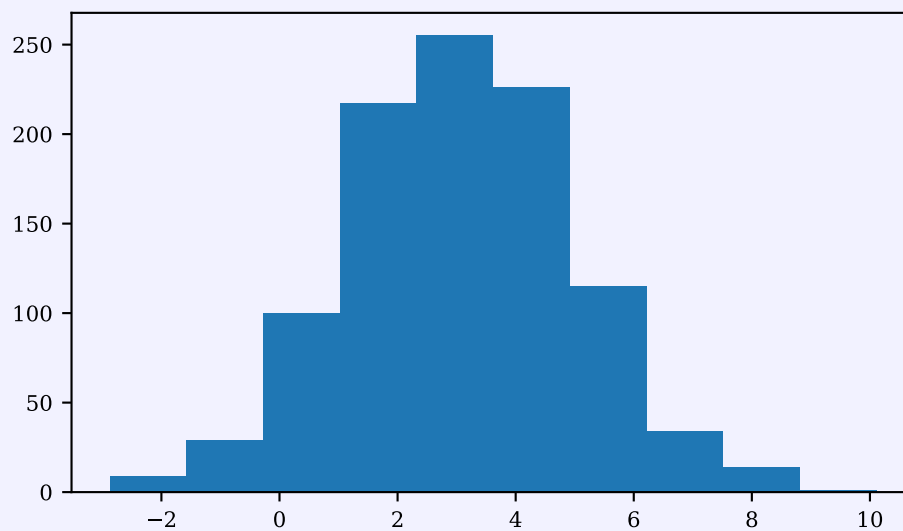
```
In [10]: c = np.zeros(4)  
         d = np.ones(3)  
         e = np.full(5, 3.8)
```

Ou alors spécifier la valeur d'initialisation et la taille d'un autre tableau.

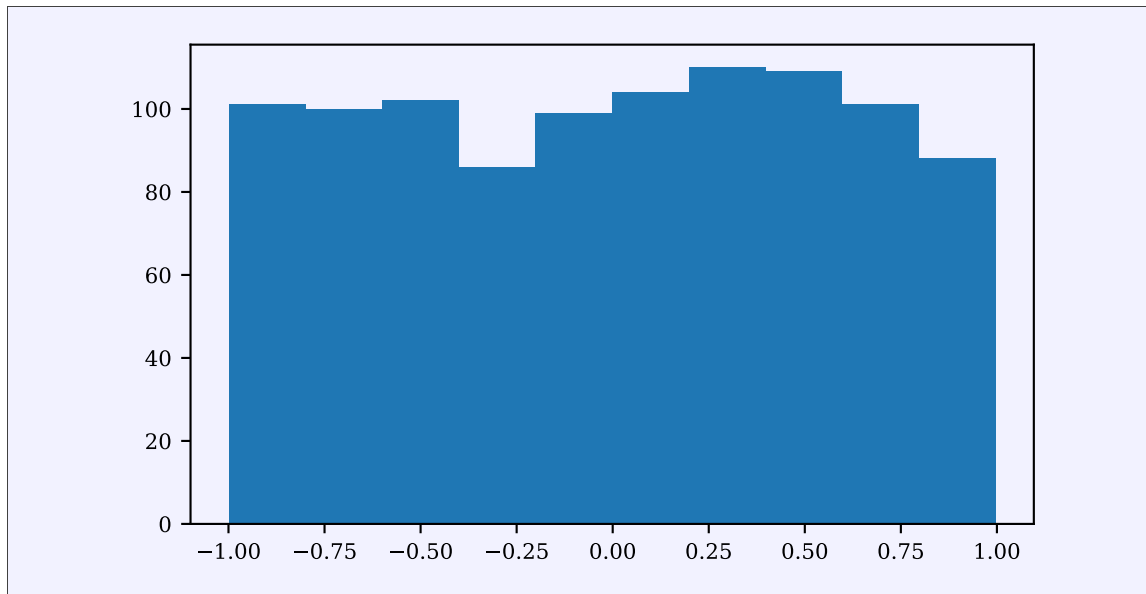
```
In [11]: f = np.zeros_like(d)  
         g = np.ones_like(e)
```

On peut également créer des tableaux suivant une loi

```
In [12]: h = np.random.normal(loc=3, scale=2, size=1000)  
         plt.hist(h)  
         plt.show()
```

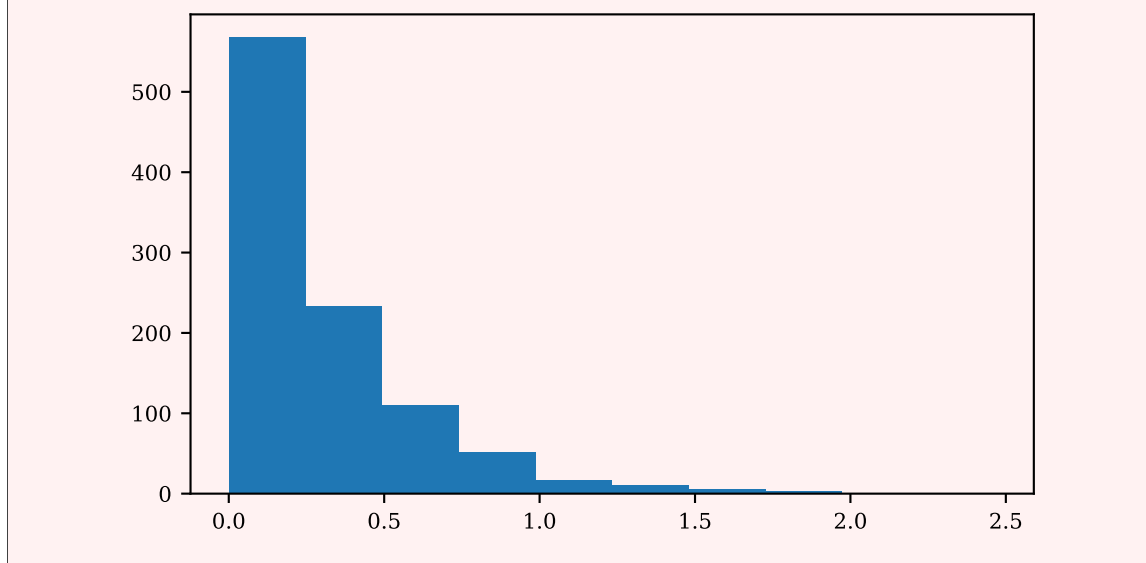


```
In [13]: i = np.random.uniform(low=-1, high=1, size=1000)  
         plt.hist(i)  
         plt.show()
```



② Créer un échantillon de longueur 1000 suivant une loi exponentielle de paramètre 0.3. Tracer son histogramme.

```
In [14]: e = np.random.exponential(scale=.3, size=1000)
plt.hist(e)
plt.show()
```



### Indexation

L'extraction d'éléments ou de sous-tableaux est très similaire à la syntaxe utilisée pour les listes Python. On utilise la notation « [] ».

```
In [15]: b
Out [15]: array([0, 1, 2, 3])
In [16]: b[2]
Out [16]: 2
In [17]: b[1:2]
Out [17]: array([1])
```

*# On extrait le troisième élément*

*# On sélectionne les indices 1 et 2*

```
In [18]: b[1:]           # On coupe avant l'indice 1, on garde la partie droite
Out [18]: array([1, 2, 3])
In [19]: b[:2]           # On coupe avant l'indice 2, on garde la partie gauche
Out [19]: array([0, 1])
In [20]: b[:-1]          # On coupe avant le dernier indice, on garde la partie gauche
Out [20]: array([0, 1, 2])
```

## Opérations élémentaires

Les tableaux Numpy sont utilisés pour du calcul numérique, on dispose donc des opérateurs suivants :

```
In [21]: a, d
Out [21]: (array([1, 2, 3]), array([1., 1., 1.]))
In [22]: a + d
Out [22]: array([2., 3., 4.])
In [23]: a - d
Out [23]: array([0., 1., 2.])
In [24]: a * a
Out [24]: array([1, 4, 9])
In [25]: a / a
Out [25]: array([1., 1., 1.])
In [26]: 3 * a
Out [26]: array([3, 6, 9])
In [27]: a + 10
Out [27]: array([11, 12, 13])
```

## Opérateurs d'agrégation

```
In [28]: a
Out [28]: array([1, 2, 3])
In [29]: a.sum()
Out [29]: 6
In [30]: a.mean()
Out [30]: 2.0
In [31]: a.min()
Out [31]: 1
In [32]: a.max()
Out [32]: 3
In [33]: a.std()
Out [33]: 0.816496580927726
In [34]: a.var()
Out [34]: 0.6666666666666666
```

③ Recalculer `a.mean()` et `a.std()` en utilisant des opérations élémentaires et des opérateurs d'agrégation plus simples. Pour utiliser la fonction racine carrée, il faut la rendre disponible avec l'instruction suivante :

```
from math import sqrt
```

```
In [35]: a.sum() / len(a)
Out [35]: 2.0
In [36]: from math import sqrt
          sqrt(((a - a.mean()) ** 2).mean())
Out [36]: 0.816496580927726
```

## Tableaux multidimensionnels

Plus généralement, on peut créer des tableaux avec un nombre quelconque de dimensions.

### Création

Des tableaux multidimensionnels peuvent être créés à partir d'une liste de listes, (de listes,...).

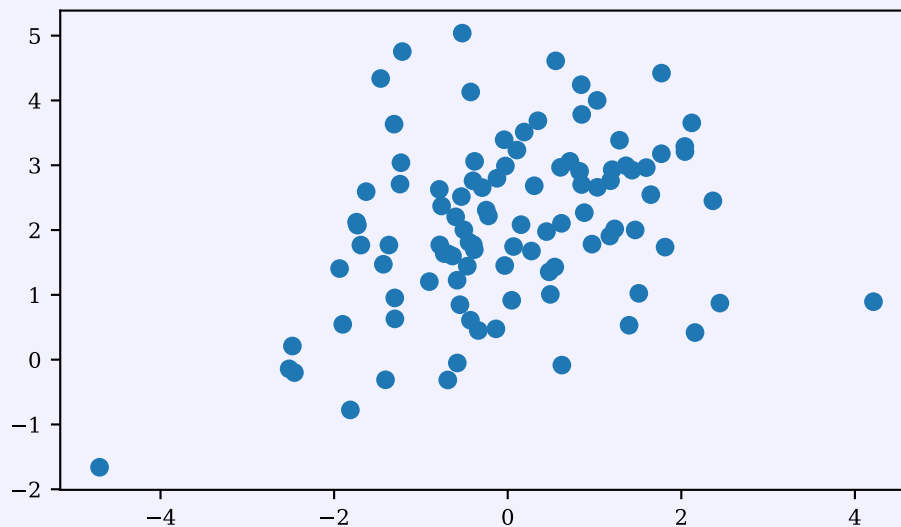
```
In [37]: a = np.array([[4, 5, 6], [7, 8, 9]])
```

On peut également spécifier les dimensions et la valeur d'initialisation.

```
In [38]: b = np.zeros((3, 4, 2))
         c = np.ones((3, 4))
         d = np.full((2, 3), 3)
```

Il existe également

```
In [39]: np.random.normal(loc=0, scale=1, size=(2, 3))
Out [39]: array([[ -0.82824544, -0.13609548, -0.51752851],
                 [ 1.32973951,  1.43341784,  0.12911216]])
In [40]: n = np.random.multivariate_normal(mean=[0, 2], cov=[[2, 1], [1, 2]],
      ↪ size=100)
      plt.scatter(n[:, 0], n[:, 1])
      plt.show()
```



Pour les tableaux bidimensionnels (les matrices), il existe des fonctions issues de l'algèbre linéaire.

```
In [41]: e = np.diag([1, 0, 1])           # Matrice diagonale de diagonale fixée
         f = np.eye(3)                    # Matrice identité d'ordre 3
```

Une fois créés, on peut calculer la taille des tableaux ou le nombre de dimensions

```
In [42]: a.shape, a.ndim, b.shape, b.ndim
Out [42]: ((2, 3), 2, (3, 4, 2), 3)
```

④ Que donne la fonction longueur `len` lorsqu'on l'applique sur un tableau ?

La fonction `len` donne le nombre d'éléments de la *première* dimension. C'est l'équivalent de `a.shape[0]`

⑤ Créer les tableaux suivants :

$$A_1 = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}, \quad A_2 = \begin{pmatrix} -3 & -2 & 1 \end{pmatrix}, \quad A_3 = \begin{pmatrix} 0 & -2 & -2 \\ -2 & 0 & -2 \\ -2 & -2 & 0 \end{pmatrix}.$$

```
In [43]: A1 = np.array([[1], [-1], [0]])
         A2 = np.array([-3, -2, 1])
         A3 = np.full((3, 3), -2) + 2 * np.eye(3)
```

## Transformations

Les tableaux peuvent être combinés entre eux de plusieurs manières.

**Restructuration avec reshape** On peut tout d'abord changer la structure d'un tableau.

```
In [44]: a.reshape((3, 2))
Out [44]: array([[4, 5],
                 [6, 7],
                 [8, 9]])
In [45]: a.reshape((6, 1))
Out [45]: array([[4],
                 [5],
                 [6],
                 [7],
                 [8],
                 [9]])
```

⑥ Utiliser la fonction `reshape` pour créer les matrices suivantes

$$A_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}, \quad A_2 = \begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \end{pmatrix}$$

```
In [46]: (1 + np.arange(8)).reshape((2, -1))
Out [46]: array([[1, 2, 3, 4],
                 [5, 6, 7, 8]])
In [47]: (1 + np.arange(8)).reshape((2, -1), order="F")
Out [47]: array([[1, 3, 5, 7],
                 [2, 4, 6, 8]])
```

**Ajout de fausse dimension** Il est souvent utile d'ajouter une fausse dimension à un tableau existant. L'exemple le plus courant est lorsqu'il s'agit de considérer un tableau unidimensionnel en une matrice ligne ou colonne.

```
In [48]: a = np.array([1, 2, 3])
         a.shape; a.ndim
Out [48]: (3,)
         1
```

On ajoute ensuite une fausse dimension avec l'instruction `np.newaxis` :

```
In [49]: b = a[np.newaxis, :]
         b.ndim; b.shape
```

```

Out [49]: 2
          (1, 3)
In [50]: c = a[:, np.newaxis]
          c.ndim; c.shape
Out [50]: 2
          (3, 1)

```

L'opération inverse est possible avec la fonction `squeeze`.

```

In [51]: b.squeeze()
Out [51]: array([1, 2, 3])
In [52]: c.squeeze()
Out [52]: array([1, 2, 3])

```

⑦ Écrire une fonction qui transforme une matrice colonne en une matrice ligne.

```

In [53]: def transpose_vector(v):
          return v.squeeze()[np.newaxis, :]

          a = np.array([[1], [2], [3]])
          transpose_vector(a)
Out [53]: array([[1, 2, 3]])

```

**Concaténation avec `np.concatenate`** Il s'agit de « coller » deux tableaux selon une dimension. Les tableaux doivent avoir le *même nombre de dimensions*.

```

In [54]: a = np.array([1, 2, 3])
          b = np.array([4, 5, 6])
          np.concatenate((a, b))
Out [54]: array([1, 2, 3, 4, 5, 6])

```

Lorsque les tableaux ont plusieurs dimensions, il faut indiquer sur quelle dimension on souhaite concaténer via l'argument `axis`. Par défaut, la concaténation se fait selon la première dimension. On peut concaténer selon la dernière dimension en spécifiant `axis=-1`.

```

In [55]: a = np.array([[1, 2, 3], [4, 5, 6]])
          b = np.array([[7, 8, 9], [10, 11, 12]])
          np.concatenate((a, b), axis=0)
Out [55]: array([[ 1,  2,  3],
                  [ 4,  5,  6],
                  [ 7,  8,  9],
                  [10, 11, 12]])

In [56]: np.concatenate((a, b))
Out [56]: array([[ 1,  2,  3],
                  [ 4,  5,  6],
                  [ 7,  8,  9],
                  [10, 11, 12]])

In [57]: np.concatenate((a, b), axis=1)
Out [57]: array([[ 1,  2,  3,  7,  8,  9],
                  [ 4,  5,  6, 10, 11, 12]])

In [58]: np.concatenate((a, b), axis=-1)
Out [58]: array([[ 1,  2,  3,  7,  8,  9],
                  [ 4,  5,  6, 10, 11, 12]])

```

- ⑧ Soient  $A_1$  et  $A_2$  deux matrices carrées. Écrire une fonction qui renvoie la matrice suivante

$$\begin{pmatrix} A_1 & -A_2 \\ A_2 & A_1 \end{pmatrix}.$$

```
In [59]: def matrix(A1, A2):
    A = np.concatenate((A1, -A2), axis=1)
    B = np.concatenate((A2, A1), axis=1)
    return np.concatenate((A, B), axis=0)

    matrix(np.diag([1, 2]), np.diag([3, 4]))
Out [59]: array([[ 1,  0, -3,  0],
                 [ 0,  2,  0, -4],
                 [ 3,  0,  1,  0],
                 [ 0,  4,  0,  2]])
```

- ⑨ Soit  $A$  une matrice et  $v$  un vecteur colonne et  $\lambda$  un scalaire. Écrire une fonction qui renvoie la matrice suivante

$$\begin{pmatrix} A & v \\ v^T & \lambda \end{pmatrix}.$$

```
In [60]: def matrix2(A, v, lbd):
    Av = np.concatenate((A, v[:, np.newaxis]), axis=1)
    vlbd = np.concatenate((v, [lbd]))
    return np.concatenate((Av, vlbd[np.newaxis, :]))

    matrix2(np.diag([1, 2]), np.array([3, 4]), 5)
Out [60]: array([[1, 0, 3],
                 [0, 2, 4],
                 [3, 4, 5]])
```



- ⑩ Écrire une fonction qui prend en argument la première ligne de la matrice circulante suivante

$$C = \begin{pmatrix} c_0 & c_1 & c_2 & \dots & c_{n-1} \\ c_{n-1} & c_0 & c_1 & & c_{n-2} \\ c_{n-2} & c_{n-1} & c_0 & & c_{n-3} \\ \vdots & & & \ddots & \vdots \\ c_1 & c_2 & c_3 & \dots & c_0 \end{pmatrix},$$

et renvoie cette matrice. On pourra commencer par générer la matrice  $C$  sans sa dernière colonne, la fonction `np.tile` pourra être utile.

```
In [61]: def circulant(row):
    n = len(row)
    c = np.array(row.tolist() * (n-1))
    c = c.reshape((n, n-1))
    c = np.concatenate((c, row[:-1, np.newaxis]), axis=1)
    return c

    circulant(np.arange(6))
```



```
Out [61]: array([[0, 1, 2, 3, 4, 5],
                [5, 0, 1, 2, 3, 4],
                [4, 5, 0, 1, 2, 3],
                [3, 4, 5, 0, 1, 2],
                [2, 3, 4, 5, 0, 1],
                [1, 2, 3, 4, 5, 0]])
```

**Empilement avec `np.stack`** Lorsqu'on souhaite « empiler » des tableaux, on ajoute une nouvelle dimension. Il faut alors utiliser la fonction `np.stack`.

```
In [62]: a = np.array([1, 2])
        b = np.array([3, 4])
        np.stack((a, b))
Out [62]: array([[1, 2],
                [3, 4]])
```

```
In [63]: a = np.diag([1, 2, 3])
        b = np.diag([4, 5, 6])
        c = np.stack((a, b))
        c
Out [63]: array([[1, 0, 0],
                [0, 2, 0],
                [0, 0, 3]],

                [[4, 0, 0],
                [0, 5, 0],
                [0, 0, 6]])
In [64]: c.shape
Out [64]: (2, 3, 3)
```

Par défaut la dimension rajoutée est en première position. L'argument `axis` permet de changer ce comportement. La valeur `-1` veut dire la dernière dimension.

```
In [65]: c = np.stack((a, b), axis=-1)
        c
Out [65]: array([[1, 4],
                [0, 0],
                [0, 0]],

                [[0, 0],
                [2, 5],
                [0, 0]],

                [[0, 0],
                [0, 0],
                [3, 6]])
In [66]: c.shape
Out [66]: (3, 3, 2)
In [67]: d = np.stack((a, b), axis=1)
        d
```

```

Out [67]: array([[1, 0, 0],
                [4, 0, 0]],

               [[0, 2, 0],
                [0, 5, 0]],

               [[0, 0, 3],
                [0, 0, 6]])
In [68]: d.shape
Out [68]: (3, 2, 3)

```

## Opérations algébriques

Lorsque les tableaux sont unidimensionnels, on peut calculer le produit scalaire.

```

In [69]: a = np.array([1, 2, 3]); b = np.array([4, 5, 6])
         np.dot(a, b)
Out [69]: 32

```

Pour des tableaux bidimensionnels

```

In [70]: c = np.array([[1, 2], [3, 4]])
         d = np.array([[0, 1], [1, 0]])
         np.matmul(c, d)
Out [70]: array([[2, 1],
                [4, 3]])
In [71]: np.transpose(c)
Out [71]: array([[1, 3],
                [2, 4]])
In [72]: d.transpose()
Out [72]: array([[0, 1],
                [1, 0]])
In [73]: d.T
Out [73]: array([[0, 1],
                [1, 0]])
In [74]: c ** 2
Out [74]: array([[ 1,  4],
                [ 9, 16]])

```

*# Produit case-à-case !*

- 11) Écrire une fonction permettant de créer une matrice  $A$  de taille  $n \times p$  telle que  $A_{ij} = 1/(ij)$ . On remarquera que si  $u$  et  $v$  sont deux vecteurs de taille  $n$  et  $p$  alors  $uv^T$  est une matrice de taille  $n \times p$  telle que  $[uv^T]_{ij} = u_i v_j$

```

def matrix(n, p):
    # Définir la fonction ici

```

```

In [75]: def matrix(n, p):
         u = (1 + np.arange(n)).reshape(n, -1)
         v = (1 + np.arange(p)).reshape(p, -1)
         return 1 / (u @ v.T)

```

## 2 Découverte de Pandas

**Pandas** est une bibliothèque Python qui permet de manipuler et analyser des structures de données. Pour commencer, il faut charger la bibliothèque **Pandas** avec l'instruction suivante.

```
In [76]: import pandas as pd
```

Une convention largement utilisée est d'importer **Pandas** sous le raccourci **pd**. Toutes les fonctionnalités de **Pandas** seront donc accessibles à travers ce raccourci.

### 2.1 Structure de données Pandas

#### 2.1.1 Les séries

La première structure de données fournie par **Pandas** est une collection d'objets de même type appelée une *série*.

**Création** On peut les définir à partir d'une liste Python ou d'un tableau unidimensionnel NumPy :

```
In [77]: a = pd.Series([1, 2, 3])
         b = pd.Series(["foo", "bar", "baz"])
         c = np.array([1, 2, 3])
         d = pd.Series(c)
```

Si jamais on souhaite convertir une série en un tableau unidimensionnel NumPy :

```
In [78]: a = pd.Series([1, 2, 3])
         a.to_numpy()
Out [78]: array([1, 2, 3])
```

On peut donner un nom à la série

```
In [79]: b = pd.Series(["foo", "bar", "baz"], name="Nom")
```

**L'index** Une différence majeure avec un tableau unidimensionnel est qu'une série **Pandas** dispose d'un *index* : une collection d'étiquettes qui repère chaque élément de la série. Lorsque ces étiquettes ne sont pas fournies, **Pandas** utilise les nombres entiers à partir de zéro.

```
In [80]: pd.Series(["foo", "bar", "baz"])
Out [80]: 0    foo
         1    bar
         2    baz
         dtype: object
```

Pour fournir cet index, il suffit d'utiliser l'argument **index**.

```
In [81]: age = pd.Series(
         [23, 24, 24, 25],
         index=["Agathe", "Béatrice", "Cédric", "Donna"]
         )
         age
Out [81]: Agathe      23
         Béatrice    24
         Cédric      24
         Donna       25
         dtype: int64
```

Pour retourner à l'index par défaut (avec des entiers à partir de 0), on utilise

```
In [82]: age.reset_index(drop=True)
Out [82]: 0    23
          1    24
          2    24
          3    25
          dtype: int64
```

Pour changer l'index, on affecte simplement l'attribut `index`.

```
In [83]: age.index = ["Ag", "Bé", "Cé", "Do"]
          age
Out [83]: Ag    23
          Bé    24
          Cé    24
          Do    25
          dtype: int64
```

**Accès** On peut accéder aux éléments d'une série de deux manières différentes :

1. Grâce aux étiquettes formant l'index. On utilise alors la méthode `loc`.
2. En utilisant la position absolue des éléments dans la série. On utilise alors la méthode `iloc`.

```
In [84]: a = pd.Series(["Un", "Deux", "Trois"], index=[1, 2, 3])
          a.loc[1]
Out [84]: 'Un'
In [85]: a.iloc[1]
Out [85]: 'Deux'
```

En utilisant `loc` ou `iloc`, on peut extraire de plusieurs manières :

```
In [86]: age = pd.Series(
          [23, 24, 24, 25],
          index=["Agathe", "Béatrice", "Cédric", "Donna"],
          name="Age"
          )
```

- Directement en donnant un élément de l'index pour extraire l'élément correspondant

```
In [87]: age.loc["Agathe"]
Out [87]: 23
In [88]: age.loc["Cédric"]
Out [88]: 24
In [89]: age.iloc[0]
Out [89]: 23
In [90]: age.iloc[-1]
Out [90]: 25
```

- En fournissant une liste d'éléments de l'index pour extraire une sous-série

```
In [91]: age.loc[["Agathe", "Donna"]]
Out [91]: Agathe    23
          Donna     25
          Name: Age, dtype: int64
In [92]: age.iloc[[1, 3]]
Out [92]: Béatrice    24
          Donna       25
          Name: Age, dtype: int64
```

- En utilisant la notation *slice*

```
In [93]: age.iloc[: -1]
Out [93]: Agathe      23
          Béatrice    24
          Cédric      24
          Name: Age, dtype: int64

In [94]: age.iloc[2:]
Out [94]: Cédric      24
          Donna      25
          Name: Age, dtype: int64

In [95]: age.loc["Cédric":]
Out [95]: Cédric      24
          Donna      25
          Name: Age, dtype: int64

In [96]: age.loc["Béatrice":"Donna"]
Out [96]: Béatrice    24
          Cédric      24
          Donna      25
          Name: Age, dtype: int64

In [97]: age.loc["Agathe":"Donna":2]
Out [97]: Agathe      23
          Cédric      24
          Name: Age, dtype: int64
```

- En fournissant un masque : un tableau de booléen de même taille que la série.

```
In [98]: masque = [True, False, False, True]
          age.loc[masque]
Out [98]: Agathe      23
          Donna      25
          Name: Age, dtype: int64
```

Le masque peut également être une série de booléens de même index.

```
In [99]: masque = pd.Series([True, False, False, True])
          masque.index = age.index
          masque
Out [99]: Agathe      True
          Béatrice   False
          Cédric     False
          Donna      True
          dtype: bool

In [100]: age.loc[masque]
Out [100]: Agathe      23
          Donna      25
          Name: Age, dtype: int64
```

À noter que les masques sont rarement fournis directement sous forme d'un tableau ou d'une série de booléens. Ils sont plutôt construits directement à partir d'une autre série voire de la série elle-même.

```
In [101]: masque = age > 23
          masque
Out [101]: Agathe     False
          Béatrice    True
          Cédric      True
          Donna       True
          Name: Age, dtype: bool

In [102]: age.loc[masque]
```

```
Out [102]: Béatrice    24
           Cédric      24
           Donna       25
           Name: Age, dtype: int64
```

- ⑫ Charger la série avec la commande suivante.

```
s1 = pd.read_csv("data/s1.csv", index_col=0, squeeze=True)
```

Construire les séries suivantes :

- La liste des prénoms donnés plus de 100000 fois
- L'effectif total des prénoms précédant SACHA

```
In [103]: s1 = pd.read_csv("data/s1.csv", index_col=0, squeeze=True)
           s1.loc[s1 > 100000].index
Out [103]: Index(['LUCAS', 'EMMA'], dtype='object', name='preusuel')
In [104]: s1.loc[:"SACHA"].sum()
Out [104]: 3231384
```

### 2.1.2 Les *DataFrame*

La structure de données la plus utilisée est le tableau individu-variable. **Pandas** les modélise avec la classe `pd.DataFrame`. Il s'agit d'une collection d'objets de type `pd.Series` représentant les colonnes d'un tableau individu-variable.

**Création** On peut construire des *DataFrame* de plusieurs manières :

- en fournissant un dictionnaire dont les clés sont les noms des caractéristiques et les valeurs sont des listes Python, des tableaux unidimensionnels **NumPy** ou des séries **Pandas** représentant les caractéristiques.

```
In [105]: col1 = np.array([23, 24, 24, 25])
           col2 = ["Agathe", "Béatrice", "Cédric", "Donna"]
           pd.DataFrame({
               "Age": col1,
               "Nom": col2
           })
Out [105]:   Age      Nom
0    23    Agathe
1    24  Béatrice
2    24    Cédric
3    25    Donna

In [106]: age = pd.Series([23, 24, 24, 25])
           nom = pd.Series(["Agathe", "Béatrice", "Cédric", "Donna"])
           pd.DataFrame({
               "Age": age,
               "Nom": nom
           })
Out [106]:   Age      Nom
0    23    Agathe
1    24  Béatrice
2    24    Cédric
3    25    Donna
```

Attention, l'index de chaque série est utilisé pour construire un index commun

```
In [107]: age = pd.Series([23, 24, 24, 25], index=[0, 1, 2, 3])
          nom = pd.Series(["Agathe", "Béatrice", "Cédric", "Donna"],
                           ↪ index=[2, 3, 4, 5])
          pd.DataFrame({
              "Age": age,
              "Nom": nom
          })
```

```
Out [107]:
```

	Age	Nom
0	23.0	NaN
1	24.0	NaN
2	24.0	Agathe
3	25.0	Béatrice
4	NaN	Cédric
5	NaN	Donna

— à partir d'un fichier csv qu'on charge à l'aide la fonction `pd.read_csv`.

```
In [108]: pd.read_csv("data/s1.csv")
```

```
Out [108]:
```

	preusuel	nombre
0	LUCAS	117001
1	EMMA	105209
2	ENZO	97980
3	LÉA	96491
4	HUGO	91960
..	...	...
995	KELIAN	1984
996	JAD	1983
997	THELMA	1982
998	MANELLE	1981
999	KESSY	1972

[1000 rows x 2 columns]

Pour convertir un *DataFrame* en un tableau bidimensionnel NumPy, on peut faire appel à la fonction `to_numpy()` :

```
In [109]: col1 = np.array([23, 24, 24, 25])
          col2 = ["Agathe", "Béatrice", "Cédric", "Donna"]
          df = pd.DataFrame({
              "Age": col1,
              "Nom": col2
          })
          df.to_numpy()
```

```
Out [109]: array([[23, 'Agathe'],
                  [24, 'Béatrice'],
                  [24, 'Cédric'],
                  [25, 'Donna']], dtype=object)
```

Attention cependant au type utilisé pour le tableau NumPy, ici `object` pour tous les éléments.

## Informations utiles

```

In [110]: col1 = np.array([23, 24, 24, 25])
          col2 = ["Agathe", "Béatrice", "Cédric", "Donna"]
          df = pd.DataFrame({
              "Age": col1,
              "Nom": col2
          })
          df.columns
Out [110]: Index(['Age', 'Nom'], dtype='object')
In [111]: df.shape
Out [111]: (4, 2)
In [112]: len(df)
Out [112]: 4
In [113]: df.dtypes
Out [113]: Age      int64
          Nom      object
          dtype: object

```

La plupart des informations ci-dessus sont synthétisées lors de l'appel à `info()`

```

In [114]: df.info()
Out [114]: <class 'pandas.core.frame.DataFrame'>
          RangeIndex: 4 entries, 0 to 3
          Data columns (total 2 columns):
          Age      4 non-null int64
          Nom      4 non-null object
          dtypes: int64(1), object(1)
          memory usage: 192.0+ bytes

```

**L'index** Les *DataFrame* dispose également d'un index commun à toutes les colonnes pour repérer tout individu (une ligne) dans le tableaux individus-variables.

On peut changer l'index à partir d'une colonne existante.

```

In [115]: df
Out [115]:
   Age  Nom
0   23  Agathe
1   24  Béatrice
2   24  Cédric
3   25  Donna

In [116]: df.set_index("Nom")
Out [116]:
   Age
Nom
Agathe    23
Béatrice  24
Cédric    24
Donna     25

```

**Accès** On peut extraire les colonnes sous forme de séries de plusieurs manières :

- Si le nom de colonne est un nom d'attribut valide (pas d'espace ou caractères spéciaux...)

```

In [117]: df.Age
Out [117]:
0    23
1    24
2    24
3    25
Name: Age, dtype: int64

```



— De manière plus générale

```
In [118]: df["Age"]
Out [118]: 0    23
           1    24
           2    24
           3    25
           Name: Age, dtype: int64
```

Pour extraire plusieurs colonnes, il suffit de donner la liste

```
In [119]: df[["Age", "Nom"]]
Out [119]:   Age    Nom
           0    23  Agathe
           1    24 Béatrice
           2    24  Cédric
           3    25   Donna
```

Pour extraire des enregistrements (lignes), on utilise `loc` et `iloc`

```
In [120]: df.loc[df.Age >= 24]
Out [120]:   Age    Nom
           1    24 Béatrice
           2    24  Cédric
           3    25   Donna

In [121]: df.iloc[[2, 3]]
Out [121]:   Age    Nom
           2    24  Cédric
           3    25   Donna
```

On peut également combiner les deux types d'extraction

```
In [122]: df.loc[df.Age >= 24, ["Age"]]
Out [122]:   Age
           1    24
           2    24
           3    25
```

**Modification** Ajout/suppression de colonnes

```
In [123]: df.drop(columns=["Age"])
Out [123]:   Nom
           0  Agathe
           1 Béatrice
           2  Cédric
           3   Donna
```