

# SY09

## TP — Introduction à NumPy et Pandas

numpy=1.26.4; seaborn=0.13.2; matplotlib=3.8.3; pandas=2.2.0

### 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
```

Les fonctionnalités NumPy sont maintenant disponibles sous l'alias `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(7))
```

Les éléments d'un tableau NumPy doivent avoir le même type (contrairement à une liste Python). On peut contrôler le type *a posteriori* avec l'attribut `dtype` :

```
In [3]: a.dtype  
Out [3]: dtype('int64')
```

Si les éléments du tableau sont de types différents, les éléments sont convertis.

```
In [4]: a = np.array([1, 2, 3, 3.14])  
        a.dtype  
Out [4]: dtype('float64')  
In [5]: a = np.array([1, "blah", 1.4])  
        a.dtype  
Out [5]: dtype('<U32')
```

On peut également spécifier le type à la création.

```
In [6]: a = np.array([1, 2, 3], float)  
        a.dtype  
Out [6]: dtype('float64')
```

Il existe des fonctions prédéfinies pour créer rapidement des séquences.

1. Des séquences de nombres en spécifiant un début, une fin (exclue) et un pas :

```
In [7]: np.arange(10)
Out [7]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [8]: np.arange(10, 11, .1)
Out [8]: array([10. , 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9])
```

2. Des séquences de nombres en spécifiant un début, une fin (inclue) et une longueur :

```
In [9]: np.linspace(0, 5, 10)
Out [9]: array([0.          , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
               ↪ 2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.          ])
```

Ou en changeant l'échelle

```
In [10]: np.logspace(0, 2, 5)
Out [10]: array([ 1.          ,  3.16227766, 10.          , 31.6227766 ,
               ↪ 100.          ])
In [11]: np.geomspace(0.1, 100, 4)
Out [11]: array([ 0.1,  1. , 10. , 100. ])
```

- 1] Définir les tableaux suivants :

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

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

```
In [12]: c = np.zeros(4)
         d = np.ones(3)
         e = np.full(5, 3.8)
```

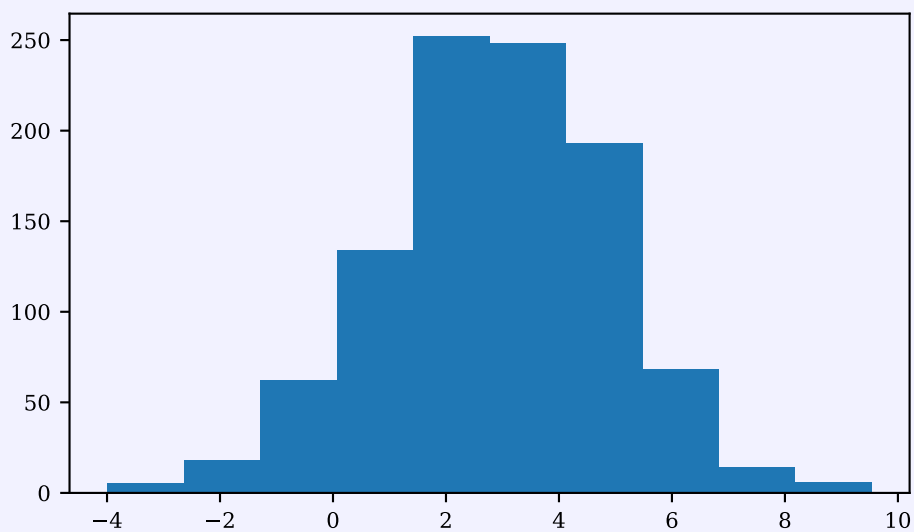
Ou alors spécifier la valeur d'initialisation et la taille d'un autre tableau.

```
In [13]: f = np.zeros_like(d)
         g = np.ones_like(e)
         h = np.full_like(e, 2)
```

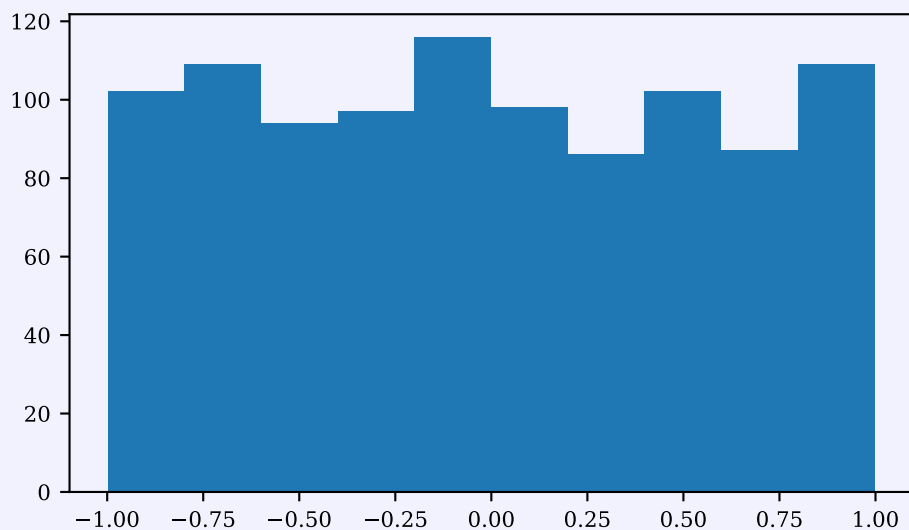
On peut également créer des tableaux suivant une loi

```
In [14]: rng = np.random.default_rng()

         i = rng.normal(loc=3, scale=2, size=1000)
         plt.hist(i)
         plt.show()
```



```
In [15]: j = rng.uniform(low=-1, high=1, size=1000)
plt.hist(j)
plt.show()
```



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

### 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 [16]: b
Out [16]: array([0, 1, 2, 3, 4, 5, 6])
In [17]: b[2]
Out [17]: 2
```

*# On extrait le troisième élément*

```
In [18]: b[1:3]          # On sélectionne les indices 1 et 2, on exclut l'indice 3
Out [18]: array([1, 2])

In [19]: b[2:]          # On coupe juste avant l'indice 2, on garde la partie droite
Out [19]: array([2, 3, 4, 5, 6])

In [20]: b[:2]          # On coupe avant l'indice 2, on garde la partie gauche
Out [20]: array([0, 1])

In [21]: b[:-1]         # On coupe avant le dernier indice, on garde la partie gauche
Out [21]: array([0, 1, 2, 3, 4, 5])

In [22]: b[::2]          # On prend un élément sur deux
Out [22]: array([0, 2, 4, 6])
```

Les expressions précédentes peuvent également être utilisées avec des listes Python. La différence fondamentale est qu'une liste est recréée alors qu'avec NumPy, une « vue » sur le tableau est créée ; les données ne sont pas dupliées. Il est alors possible d'écrire les instructions suivantes (qui produisent une erreur avec une liste Python) :

```
In [23]: b[1:2] = 0
          b
Out [23]: array([0, 0, 2, 3, 4, 5, 6])
```

## Opérations élémentaires

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

```
In [24]: a, d
Out [24]: (array([1., 2., 3.]), array([1., 1., 1.]))

In [25]: a + d
Out [25]: array([2., 3., 4.])

In [26]: a - d
Out [26]: array([0., 1., 2.])

In [27]: a * a
Out [27]: array([1., 4., 9.])

In [28]: a / a
Out [28]: array([1., 1., 1.])

In [29]: 3 * a
Out [29]: array([3., 6., 9.])

In [30]: a + 10
Out [30]: array([11., 12., 13.])
```

## Opérateurs d'agrégation

```
In [31]: a
Out [31]: array([1., 2., 3.])
In [32]: a.sum()
Out [32]: 6.0
In [33]: a.mean()
Out [33]: 2.0
In [34]: a.min()
Out [34]: 1.0
In [35]: a.max()
Out [35]: 3.0
In [36]: a.std()
Out [36]: 0.816496580927726
In [37]: a.var()
Out [37]: 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
```

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

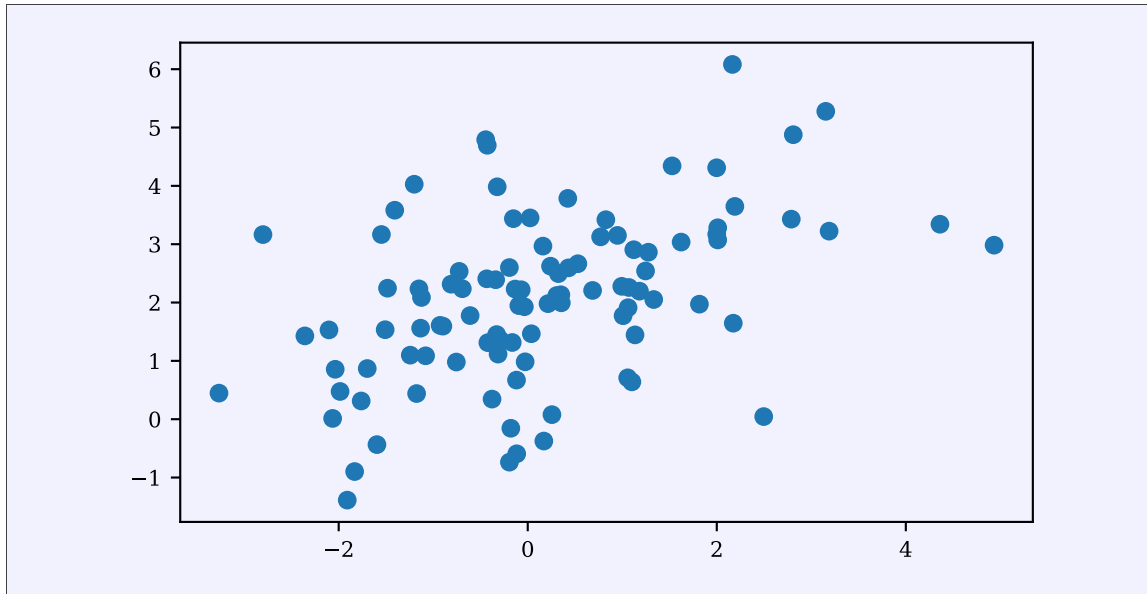
On peut également spécifier les dimensions et la valeur d'initialisation.

```
In [39]: b = np.zeros((3, 4, 2))
         c = np.ones((3, 4))
         d = np.full((2, 3), 3)
```

Il existe également

```
In [40]: rng.normal(loc=0, scale=1, size=(2, 3))
Out [40]: array([[ -0.7914499 ,  0.70066027, -0.07340597],
                 [ 2.07507981, -0.62190846,  2.57061727]])

In [41]: n = rng.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 [42]: 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 [43]: a.shape, a.ndim, b.shape, b.ndim
Out [43]: ((2, 3), 2, (3, 4, 2), 3)
```

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

5 Créer les tableaux suivants :

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

## 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

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

**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 [46]: a = np.array([1, 2, 3])
         a.shape; a.ndim
```

```
Out [46]: (3,)
          1
```

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

```
In [47]: b = a[np.newaxis, :]
         b.ndim; b.shape
```

```
Out [47]: 2
          (1, 3)
```

```
In [48]: c = a[:, np.newaxis]
         c.ndim; c.shape
```

```
Out [48]: 2
          (3, 1)
```

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

```
In [49]: b.squeeze()
```

```
Out [49]: array([1, 2, 3])
```

```
In [50]: c.squeeze()
```

```
Out [50]: array([1, 2, 3])
```

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

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

```
In [51]: a = np.array([1, 2, 3])
         b = np.array([4, 5, 6])
         np.concatenate((a, b))
```

```
Out [51]: 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 [52]: 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 [52]: array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [10, 11, 12]])

In [53]: np.concatenate((a, b))

Out [53]: array([[ 1,  2,  3],
                 [ 4,  5,  6],
                 [ 7,  8,  9],
                 [10, 11, 12]])

In [54]: np.concatenate((a, b), axis=1)

Out [54]: array([[ 1,  2,  3,  7,  8,  9],
                 [ 4,  5,  6, 10, 11, 12]])

In [55]: np.concatenate((a, b), axis=-1)

Out [55]: 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

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

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

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



- 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.

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

```

In [56]: a = np.array([1, 2])
         b = np.array([3, 4])
         np.stack((a, b))

Out [56]: array([[1, 2],
                 [3, 4]])

```

```

In [57]: a = np.diag([1, 2, 3])
         b = np.diag([4, 5, 6])
         c = np.stack((a, b))
         c

```



```
Out [57]: array([[1, 0, 0],
                [0, 2, 0],
                [0, 0, 3]],

               [[4, 0, 0],
                [0, 5, 0],
                [0, 0, 6]])
```

```
In [58]: c.shape
```

```
Out [58]: (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 [59]: c = np.stack((a, b), axis=-1)
        c
```

```
Out [59]: array([[1, 4],
                [0, 0],
                [0, 0]],

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

               [[0, 0],
                [0, 0],
                [3, 6]])
```

```
In [60]: c.shape
```

```
Out [60]: (3, 3, 2)
```

```
In [61]: d = np.stack((a, b), axis=1)
        d
```

```
Out [61]: array([[1, 0, 0],
                [4, 0, 0]],

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

               [[0, 0, 3],
                [0, 0, 6]])
```

```
In [62]: d.shape
```

```
Out [62]: (3, 2, 3)
```

## Opérations algébriques

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

```
In [63]: a = np.array([1, 2, 3]); b = np.array([4, 5, 6])
        np.dot(a, b)
```

```
Out [63]: 32
```

Pour des tableaux bidimensionnels

```

In [64]: c = np.array([[1, 2], [3, 4]])
          d = np.array([[0, 1], [1, 0]])
          c @ d                                     # Produit matriciel

Out [64]: array([[2, 1],
                 [4, 3]])

In [65]: np.matmul(c, d)

Out [65]: array([[2, 1],
                 [4, 3]])

In [66]: np.transpose(c)

Out [66]: array([[1, 3],
                 [2, 4]])

In [67]: d.transpose()

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

In [68]: d.T

Out [68]: array([[0, 1],
                 [1, 0]])

In [69]: c ** 2                                     # Produit case-à-case !

Out [69]: array([[ 1,  4],
                 [ 9, 16]])

In [70]: np.trace(c)                                # La trace

Out [70]: 5

```

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

```

## 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 [71]: 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 [72]: 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 [73]: a = pd.Series([1, 2, 3])
         a.to_numpy()

Out [73]: array([1, 2, 3])
```

On peut donner un nom à la série

```
In [74]: 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 [75]: pd.Series(["foo", "bar", "baz"])

Out [75]: 0    foo
         1    bar
         2    baz
         dtype: object
```

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

```
In [76]: age = pd.Series(
         [23, 24, 24, 25],
         index=["Agathe", "Béatrice", "Cédric", "Donna"]
         )
         age

Out [76]: 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 [77]: age.reset_index(drop=True)

Out [77]: 0    23
         1    24
         2    24
         3    25
         dtype: int64
```

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

```
In [78]: age.index = ["Ag", "Bé", "Cé", "Do"]
         age
```

```
Out [78]: 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 [79]: a = pd.Series(["Un", "Deux", "Trois"], index=[1, 2, 3])
         a.loc[1]

Out [79]: 'Un'

In [80]: a.iloc[1]

Out [80]: 'Deux'
```

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

```
In [81]: 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 [82]: age.loc["Agathe"]

Out [82]: 23

In [83]: age.loc["Cédric"]

Out [83]: 24

In [84]: age.iloc[0]

Out [84]: 23

In [85]: age.iloc[-1]

Out [85]: 25
```

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

```
In [86]: age.loc[["Agathe", "Donna"]]

Out [86]: Agathe    23
         Donna     25
         Name: Age, dtype: int64

In [87]: age.iloc[[1, 3]]

Out [87]: Béatrice    24
         Donna       25
         Name: Age, dtype: int64
```

- En utilisant la notation *slice*

```
In [88]: age.iloc[: -1]
```

```

Out [88]: Agathe      23
          Béatrice    24
          Cédric      24
          Name: Age, dtype: int64

In [89]: age.iloc[2:]

Out [89]: Cédric      24
          Donna      25
          Name: Age, dtype: int64

In [90]: age.loc["Cédric":]

Out [90]: Cédric      24
          Donna      25
          Name: Age, dtype: int64

In [91]: age.loc["Béatrice":"Donna"]

Out [91]: Béatrice    24
          Cédric      24
          Donna      25
          Name: Age, dtype: int64

In [92]: age.loc["Agathe":"Donna":2]

Out [92]: Agathe      23
          Cédric      24
          Name: Age, dtype: int64

```

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

```

In [93]: masque = [True, False, False, True]
          age.loc[masque]

Out [93]: Agathe      23
          Donna      25
          Name: Age, dtype: int64

```

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

```

In [94]: masque = pd.Series([True, False, False, True])
          masque.index = age.index
          masque

Out [94]: Agathe      True
          Béatrice   False
          Cédric     False
          Donna      True
          dtype: bool

In [95]: age.loc[masque]

Out [95]: 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 [96]: masque = age > 23
          masque

```

```
Out [96]: Agathe      False
          Béatrice    True
          Cédric      True
          Donna       True
          Name: Age, dtype: bool

In [97]: age.loc[masque]

Out [97]: 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

### 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 [98]: col1 = np.array([23, 24, 24, 25])
          col2 = ["Agathe", "Béatrice", "Cédric", "Donna"]
          pd.DataFrame({
              "Age": col1,
              "Nom": col2
          })

Out [98]:   Age      Nom
0    23    Agathe
1    24  Béatrice
2    24    Cédric
3    25    Donna

In [99]: age = pd.Series([23, 24, 24, 25])
          nom = pd.Series(["Agathe", "Béatrice", "Cédric", "Donna"])
          pd.DataFrame({
              "Age": age,
              "Nom": nom
          })

Out [99]:   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 [100]: 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 [100]:
```

	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 générateur. Pandas s'attend à ce que chaque élément généré soit une ligne du tableau individu-variable.

```
In [101]: def gen():
          yield "Agathe", 23
          yield "Béatrice", 24
          yield "Cédric", 24
          yield "Donna", 25

          pd.DataFrame(gen())
```

```
Out [101]:
```

	0	1
0	Agathe	23
1	Béatrice	24
2	Cédric	24
3	Donna	25

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

```
In [102]: pd.read_csv("data/s1.csv")
```

```
Out [102]:
```

	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	KESY	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 [103]: col1 = np.array([23, 24, 24, 25])
          col2 = ["Agathe", "Béatrice", "Cédric", "Donna"]
          df = pd.DataFrame({
              "Age": col1,
              "Nom": col2
          })
          df.to_numpy()

Out [103]: 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 [104]: col1 = np.array([23, 24, 24, 25])
          col2 = ["Agathe", "Béatrice", "Cédric", "Donna"]
          df = pd.DataFrame({
              "Age": col1,
              "Nom": col2
          })
          df.columns

Out [104]: Index(['Age', 'Nom'], dtype='object')

In [105]: df.shape

Out [105]: (4, 2)

In [106]: len(df)

Out [106]: 4

In [107]: df.dtypes

Out [107]: Age      int64
          Nom      object
          dtype: object
```

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

```
In [108]: df.info()

Out [108]: <class 'pandas.core.frame.DataFrame'>
          RangeIndex: 4 entries, 0 to 3
          Data columns (total 2 columns):
           #   Column  Non-Null Count  Dtype
          ---  ---
           0   Age      4 non-null      int64
           1   Nom      4 non-null      object
          dtypes: int64(1), object(1)
          memory usage: 196.0+ 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 [109]: df
```



```
Out [109]:
```

	Age	Nom
0	23	Agathe
1	24	Béatrice
2	24	Cédric
3	25	Donna

```
In [110]: df.set_index("Nom")
```

```
Out [110]:
```

	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 [111]: df.Age
```

```
Out [111]:
```

0	23
1	24
2	24
3	25

Name: Age, dtype: int64

- De manière plus générale

```
In [112]: df["Age"]
```

```
Out [112]:
```

0	23
1	24
2	24
3	25

Name: Age, dtype: int64

Pour extraire plusieurs colonnes, il suffit de donner la liste

```
In [113]: df[["Age", "Nom"]]
```

```
Out [113]:
```

	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 [114]: df.loc[df.Age >= 24]
```

```
Out [114]:
```

	Age	Nom
1	24	Béatrice
2	24	Cédric
3	25	Donna

```
In [115]: df.iloc[[2, 3]]
```

```
Out [115]:
```

	Age	Nom
2	24	Cédric
3	25	Donna

On peut également combiner les deux types d'extraction

```
In [116]: df.loc[df.Age >= 24, ["Age"]]
```

```
Out [116]:    Age
1      24
2      24
3      25
```

**Modification** Ajout/suppression de colonnes

```
In [117]: df.drop(columns=["Age"])
```

```
Out [117]:    Nom
0    Agathe
1  Béatrice
2    Cédric
3    Donna
```