

# ANALYSE DE DONNÉES AVEC PYTHON

Enseignant : Louis RAYNAL

Contact : [l-raynal@ices.fr](mailto:l-raynal@ices.fr)

Public : L1 Maths














Ressources : <https://github.com/LouisRaynal/pythonData>



Établissement : ICES

Année : 2024-2025

## TP 1 : Introduction

### Mémos pour Mac :

{ =  +       } =  +       | =  +  +   
[ =  +  +       ] =  +  + 

Selon la version du clavier  peut être à remplacer par 

### Au cas où vous avez un problème avec les fenêtres de Spyder :

View > Window Layouts > Spyder Default Layout

Lors de ce TP d'introduction, vous découvrirez l'interface que nous utiliserons pour programmer en Python. Vous (re)verrez des structures qu'il est essentiel de savoir créer et manipuler pour pouvoir analyser des données.

## 1. Outils du cours

### 1.1. Distribution de Python : Anaconda

Nous utiliserons **Anaconda**, qui est une distribution de Python : c'est-à-dire qu'en installant Anaconda, vous installerez également la dernière version de Python (Python 3), ses librairies les plus courantes, ainsi que d'autres outils/logiciels utiles à l'analyse de données.

Anaconda devrait déjà être installé sur les postes de l'ICES.

Pour son installation sur vos ordinateurs personnels, vous pouvez vous rendre à l'adresse :

<https://www.anaconda.com/products/distribution>

### Pratique

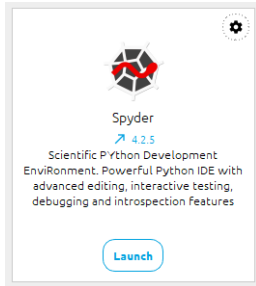
- Ouvrir le navigateur Anaconda (*Anaconda-Navigator*) sur vos postes.

## 1.2. Spyder

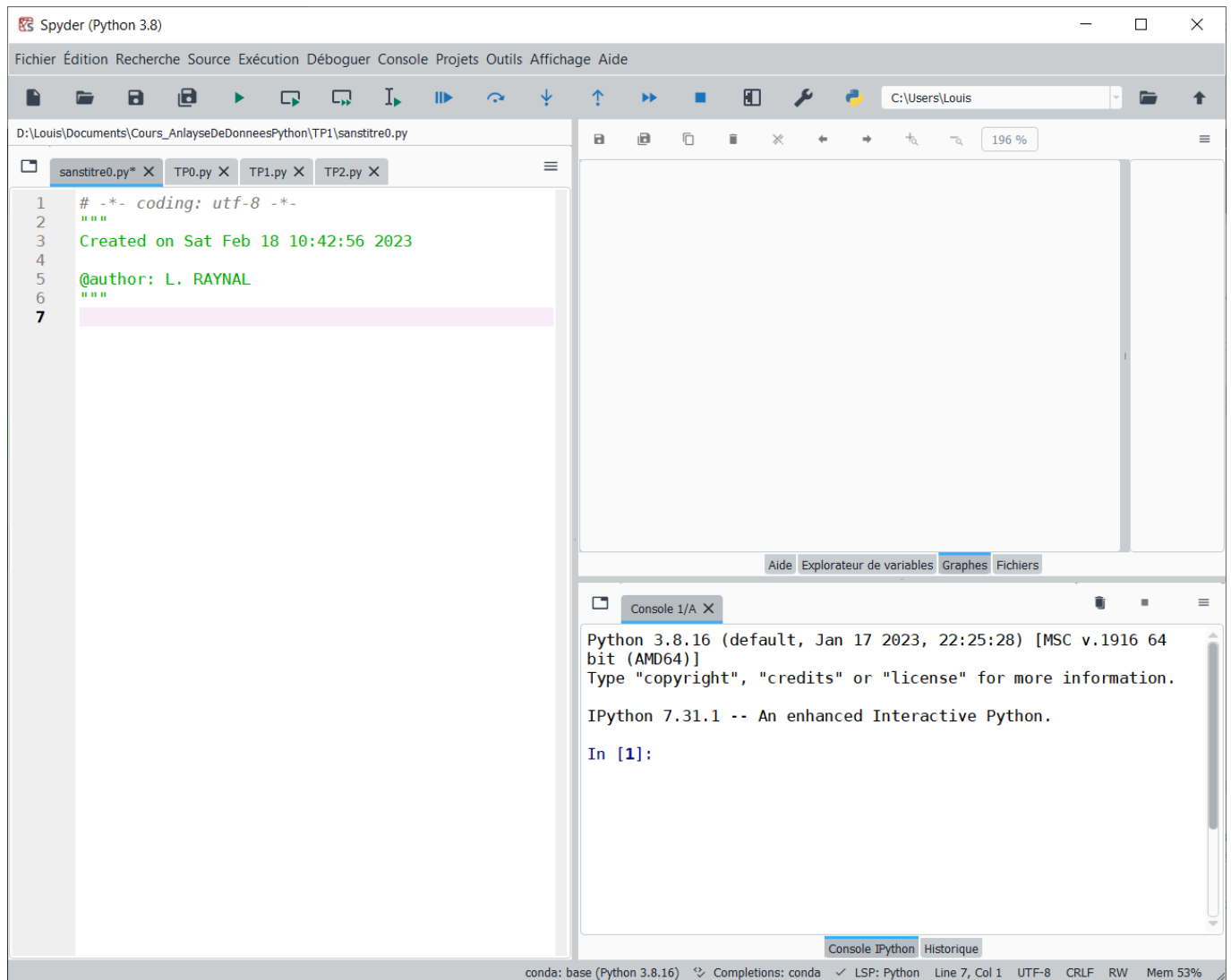
Anaconda contient de nombreux outils pour la programmation Python, notamment des interfaces de développement telles que Spyder et Jupyter notebook. Pour ces TP, nous utiliserons l'interface **Spyder**.


### Pratique

Lancer Spyder depuis le navigateur Anaconda.



L'espace de travail se présente de la manière suivante :



- La fenêtre de gauche est l'espace où vous rédigerez vos commandes en Python. Le nom du script est indiqué en haut à gauche de cette fenêtre (probablement `temp.py` ou `sanstitre.py`). Une `*` à côté de ce nom indique que ce fichier n'a pas été sauvegardé dans sa version la plus récente. Vous pourrez exécuter vos codes en sélectionnant les lignes souhaitées et en appuyant sur le symbole .
- La fenêtre en bas à droite correspond à la **Console**. Les résultats de vos commandes exécutées (hors graphiques) y seront affichés. Cette fenêtre contient aussi un onglet **Historique** contenant l'historique des codes exécutés.
- La fenêtre en haut à droite contient de nombreux onglets, notamment **Explorateur de variables** (pour visualiser les variables définies) et **Graphes** (pour visualiser les graphiques tracés).

La version de base de Python étant rapidement limitée en termes de fonctionnalités, nous utiliserons des bibliothèques complémentaires. Parmi elles vous rencontrerez :

- **numpy** – permet de travailler avec des vecteurs et matrices ;
- **pandas** – ajoute des structures adaptées pour l'analyse statistique et la manipulation de données ;
- **scipy** – contient de nombreux outils statistiques et lois de probabilité élémentaires ;
- **matplotlib** – la bibliothèque standard pour la visualisation de données ;
- **seaborn** – une bibliothèque plus évoluée que **matplotlib** pour la visualisation de données.

### Pratique

Ouvrir un nouveau script Python : *Fichier > Nouveau fichier*.

Sauvegarder ce fichier sur votre bureau avec le nom `TP1.py`.

A la fin de chaque séance de TP, pensez à conserver vos scripts sur le moyen de stockage de votre choix (email, google drive, clé USB, ...).

Rappel : Votre script Python est le fichier qui contiendra vos différentes commandes que vous pourrez exécuter. Veillez à commenter vos codes (symbole `#`) et à ordonner vos codes dans l'ordre logique d'exécution (de haut en bas).

## 2. Rappels sur les structures de base de Python

Nous allons ici revoir des structures de base de Python qu'il est essentiel de connaître pour l'analyse de données.

### Exercices

■ Dans la suite, exécutez **chaque ligne** de code afin de comprendre les structures de données présentées.

### 2.1. Les tuples

Un **tuple** est une collection d'objets, créé grâce à des `( )` ou bien la fonction `tuple`. Une fois défini, un tuple est **immutable**, c'est-à-dire qu'il ne peut pas être modifié après sa création.

#### Code 1 – Exemples de tuples

```
1 tuple_1 = (1, 'a', True)
2 tuple_1
3
4 tuple_1[2]
5
6 tuple_1[2] = 'b' # Ne fonctionne pas car les tuples sont immutables
7
```

```
8 tuple_2 = tuple([1,2,3])
9 tuple_2
10
11 tuple_2[1]
```

On rappelle qu'en Python, les indices commencent à 0. Le  $i$ -ième élément d'un tuple porte donc l'indice  $i - 1$ .

## 2.2. Les listes

Une **liste** est une collection d'objets, créée grâce à des `[ ]` ou bien la fonction `list`. Contrairement aux tuples, les listes sont **mutables**, c'est-à-dire qu'elles peuvent être modifiées après création.

### Code 2 – Exemples de listes

```
1 liste_1 = ['abc', 'def', 'ghi']
2 liste_1[1]
3
4 liste_1[2] = 0
5
6 liste_2 = list(tuple_1)
7 liste_2
```

## 2.3. Les dictionnaires

Un **dictionnaire** est une collection de **clés-valeurs non ordonnées**. Chaque clé est associée à une valeur, qui est un objet Python. Ce sont les clés qui vont permettre l'accès aux valeurs du dictionnaire. Un dictionnaire est habituellement créé grâce à des `{ }` ou avec la fonction `dict`.

### Code 3 – Exemples de dictionnaires

```
1 dico_1 = dict(a=1, b=2, info='des informations')
2 dico_1['a']
3
4 dico_2 = {'a':1, 'b':[1,2,3,4,5], 'c':"plus d'informations"}
5 # Remarque : Pour la dernière valeur on déclare la chaîne entre " " car elle contient un '
6
7 dico_2['b']
8
9 dico_2['b'] = 5
10 dico_2['b']
11
12 dico_2.keys()
13 dico_2.values()
```

Maintenant que vous avez créé quelques variables, allez inspecter l'onglet **Explorateur de variables**. Pour chaque variable, vous retrouverez son nom, son type, sa taille et son contenu. Vous pouvez double-cliquer sur ces variables pour voir plus clairement le contenu de chacune.

### 3. Présentation de NumPy

numpy est une librairie Python qui facilite grandement le travail avec des nombres, et va nous permettre de créer des **vecteurs** et **matrices**.

L'import de la librairie se fait habituellement de la manière suivante (à placer au début de votre script, avec vos autres imports de librairies) :

```
import numpy as np
```

Notez que l'on utilisera **np** comme alias pour **numpy**.

#### 3.1. Création d'arrays

Cette librairie permet de créer des vecteurs et matrices, que l'on appellera **arrays**. Pour créer un vecteur ou une matrice, on peut utiliser la fonction **array** de numpy.

##### Code 4 – Création d'un vecteur

```
1 data_1 = [1.5,2,3.4,4,5,6,7,8] # à partir d'une liste
2 array_1 = np.array(data_1)
3 array_1
```

##### Code 5 – Création d'une matrice

```
1 data_2 = [[1,2,3,4], [5,6,7,8]] # à partir d'une liste de listes
2 array_2 = np.array(data_2)
3 array_2
```

Un array a une **taille**, ainsi qu'un **type** automatiquement déterminé par ce qu'il contient. Les méthodes **shape** et **dtype** d'un array permettent d'accéder à ces deux informations pour cet array.

##### Code 6 – Dimensions et type d'un array

```
1 # Obtenir les dimensions de ces deux arrays
2 array_1.shape
3 array_2.shape
4
5 # Obtenir le type de ces deux arrays
6 array_1.dtype
7 array_2.dtype
```

#### Remarque

On rappelle que :

- une **fonction** est définie avec le mot clé **def**, et retourne les objets définis par le mot clé **return** ;
- une **méthode** est une fonction qui est associée à un objet Python.

Par exemple, **shape** est une méthode associée à un objet de type array, qui retourne les dimensions de cet array.

`numpy` possède de nombreuses autres fonctions pour créer des arrays, en voici quelques-unes. Notez que pour certaines de ces fonctions, donner un tuple au lieu d'une valeur entière changera l'array créé.

#### Code 7 – Autres fonctions pour créer des arrays

```
1 np.zeros(5)
2 np.zeros((5,3))
3
4 np.ones(5)
5 np.ones((5,3))
6
7 np.eye(5)
8
9 np.arange(20)
```

Enfin, bien que le type d'un array soit défini automatiquement par son contenu, il est possible d'indiquer son type lors de sa création avec l'argument `dtype`.

#### Code 8 – Spécification du type d'un array

```
1 array_f = np.array([1,2,3], dtype=np.float64) # nombre flottant à 64 bits
2 array_i = np.array([1,2,3], dtype=np.int32)   # nombre entier à 32 bits
3
4 array_f.dtype
5 array_i.dtype
6
7 array_f
8 array_i
```

### 3.2. Calculs sur des arrays

L'un des avantages des arrays introduits par `numpy` est la facilité avec laquelle nous pouvons effectuer des calculs et comparaisons entre/sur eux.

#### Code 9 – Opérations sur les arrays

```
1 # Calculs
2 array_2 + array_2
3 array_2 * array_2
4 array_2 + 42
5
6 # Comparaisons
7 array_1 > 5
8 array_1 == 2
9 array_1 != array_1
```

`numpy` contient de nombreuses fonctions ou méthodes applicables aux arrays, afin d'effectuer des calculs pour l'analyse de données. Par exemple la méthode `mean()` pour calculer la moyenne, ou bien `var()` pour calculer la variance. Nous reviendrons sur ce genre de méthodes lors du prochain TP.

### 3.3. Extraction de données d'arrays

Pour la suite, il est important de savoir extraire des éléments constituant des arrays : **selon des positions** et **selon des conditions**. La manière de faire varie selon qu'il s'agit d'un vecteur ou d'une matrice.

Pour un vecteur, cela fonctionne de manière similaire à une liste. Voici des exemples à expérimenter.

#### Code 10 – Extractions sur des arrays en une dimension

```
1 # Création d'un vecteur contenant les entiers de 5 à 9
2 array1D = np.arange(5,10)
3 array1D
4
5 # Différentes manières d'extraire des données d'un vecteur
6 array1D[:]      # tous les indices
7 array1D[1]      # indice 1
8 array1D[1:3]    # indice 1 à 3 exclu
9 array1D[1:]     # tout à partir de l'indice 1
10 array1D[:4]    # tout avant l'indice 4 exclu
11 array1D[-1]    # dernier indice
12 array1D[-3:]   # tout à partir de l'avant avant dernier indice
13 array1D[[0,4,2]] # via une liste d'indice
14
15 # Modifications
16 array1D[0:2] = 17
17 array1D
```

Pour une matrice cela se complexifie. La logique est que l'on peut spécifier entre [ , ] les indices des lignes puis des colonnes à extraire.

#### Code 11 – Extractions sur des arrays en deux dimensions

```
1 array2D = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
2 array2D
3
4 # Extraction de lignes
5 array2D[1,:]      # équivalent à array2D[1]
6 array2D[[1,2],:]  # équivalent à array2D[[1,2]]
7 array2D[:,2:]     # équivalent à array2D[:,2]
8 # Le : pour dire "toutes les colonnes" n'est pas obligatoire
9
10 # Extraction de colonnes
11 array2D[:,1]
12 array2D[:,[0,1]]
13 array2D[:,1:]
14 # Le : pour dire "toutes les lignes" est obligatoire
15
16 # Extraction de lignes et colonnes
17 array2D[1][2] # soit en deux temps
18 array2D[1,2] # soit d'un coup
19
20 # Attention !
21 array2D[[0,1],[0,3]]
```

```
22 # donne les éléments avec indices (0,0) et (1,3)
23 # et non, tous les éléments aux lignes d'indice 0 et 1,
24 # et colonnes d'indice 0 et 3
25 # Pour ce faire, vous pouvez utiliser la fonction np.ix_ de numpy
26 # de la manière suivante :
27 array2D[np.ix_([0,1],[0,3])]
```

Une extraction sur un array peut permettre de modifier ses éléments souhaités :

#### Code 12 – Modifications d'un array grâce à une extraction

```
1 array2D[[0,1],[0,3]] = 0
2 array2D
```

Des **comparaisons logiques** sur les éléments d'un array peuvent grandement faciliter le remplacement de certaines de ses valeurs.

#### Code 13 – Comparaisons logiques dans un array

```
1 array2D > 5 # Identifie par True les valeurs > 5, False sinon
2 array2D[ array2D > 5 ] = -1 # On remplace ces valeurs > 5 par -1
3 array2D
```

Cela n'est qu'un aperçu des possibilités offertes par la librairie **numpy**, et nous découvrirons d'autres fonctionnalités dans les prochains TP.

### Exercices

1. Créer un array avec 6 lignes et 4 colonnes avec uniquement des 0.
2. Assigner à la troisième et cinquième ligne les valeurs 1, 2, 3, 4.
3. Assigner 10 à tous les éléments de cet array égaux à 0.
4. Assigner 1.5 à tous les éléments de cet array se trouvant à l'intersection des lignes 2 et 3, et des colonnes 1, 3 et 4. (Utiliser la fonction `np.ix_`, voir fin du Code 11)
5. Recréer votre array de départ en spécifiant `dtype=np.int32` et relancez vos codes des questions 2 à 4. Que constatez-vous ?

### Attention

Faites très attention lorsque vous voulez copier un array dans une nouvelle variable, puis modifier cette copie.

Exécuter le code ci-dessous :

#### Code 14 – La mauvaise copie

```
1 array_1 = np.array([1,2,3,4])
2 array_copie = array_1[0:2]
3 array_copie
4
5 array_copie[1] = 42
6 array_copie
7 array_1
```



Vous voyez que l'array initial a aussi été modifié.

Il faut utiliser la méthode `copy()` lorsque vous faites une copie d'un array.

#### Code 15 – La bonne copie

```
1 array_1 = np.array([1,2,3,4])
2 array_copie = array_1[0:2].copy()
3 array_copie
4
5 array_copie[1] = 42
6 array_copie
7 array_1
```

## 4. Présentation de Pandas et de ses DataFrames

`pandas` est une librairie qui contient des structures, ainsi que des outils de manipulation de données, qui faciliteront grandement les analyses en Python.

L'import de la librairie se fait habituellement de la manière suivante :

```
import pandas as pd
```

L'analyse de données se base sur l'analyse de **données structurées**, prenant le plus souvent la forme d'un **tableau**. Chaque colonne du tableau porte un nom désignant l'information stockée dans celle-ci, et chaque ligne désigne un individu.

Pour reproduire ce genre de structures, `pandas` introduit un objet appelé **DataFrame**. Un DataFrame est une structure en deux dimensions, où chaque colonne porte un nom et est potentiellement de type différent (numérique, chaîne de caractères, booléen, ...), et chaque ligne correspond à un individu.

### 4.1. Création de DataFrames à partir de dictionnaires

Un DataFrame peut se construire de différentes manières via la fonction `DataFrame` de `pandas`. L'une des plus communes est l'utilisation d'un dictionnaire, où toutes ses valeurs seront des listes de tailles égales, et ses clés seront nos noms de colonnes.

Créons par exemple un DataFrame avec trois colonnes, une première nommée **Produit** avec des noms de céréales, une deuxième avec le nombre de **Calories** de chaque céréales, et une troisième avec la quantité de **Protéines**.

#### Code 16 – Création d'un DataFrame à partir d'un dictionnaire

```
1 produit = ['Corn Flakes','Crispix','Golden Grahams','Muesli Raisins','Smacks','Special K']
2 calories = [100,110,110,150,110,110]
3 proteines = [2,2,1,4,2,6]
4
5 data = {'Produit' : produit,
6         'Calories' : calories,
7         'Proteines' : proteines}
8 # Exécutez bien les trois lignes ci-dessus en une fois
9
```

```

10 df = pd.DataFrame(data)
11 df

```

## 4.2. Création de DataFrames à partir d'une liste de listes

Une deuxième manière pour créer un DataFrame consiste à créer une liste de listes. Il faudra par contre spécifier les noms de colonnes avec l'argument `columns` de `pd.DataFrame`.

### Code 17 – Création d'un DataFrame à partir d'une liste de listes

```

1 data2 = [['Corn Flakes',100,2],
2          ['Crispix',110,2],
3          ['Golden Grahams',110,1],
4          ['Muesli Raisins',150,4],
5          ['Smacks',110,2],
6          ['Special K',110,6]]
7
8 df2 = pd.DataFrame(data2, columns=['Produit','Calories','Proteines'])
9 df2

```

Le DataFrame créé est affiché ci-dessous. Il s'agit d'un jeu de données avec 6 individus (ici des céréales), et pour chacun nous avons comme informations le nom du produit, le nombre de calories et la teneur en protéines.

	Produit	Calories	Proteines
0	Corn Flakes	100	2
1	Crispix	110	2
2	Golden Grahams	110	1
3	Muesli Raisins	150	4
4	Smacks	110	2
5	Special K	110	6

On retrouve les **valeurs** du DataFrame. Remarquez qu'il contient bien 3 colonnes nommées **Produit**, **Calories** et **Proteines**, mais aussi que chaque ligne est numérotée, il s'agit du nom porté par chaque ligne. Un DataFrame possède donc des **noms de colonnes**, ainsi que des **noms de lignes**. Si ces noms ne sont pas spécifiés lors de la création du DataFrame ils seront des entiers commençant à 0.

Pour afficher les valeurs, les noms de colonnes et les noms de lignes d'un DataFrame, nous pouvons utiliser les méthodes `values`, `columns` et `index` de celui-ci. Chaque colonne possède aussi un type, récupérable via la méthode `dtypes`.

### Code 18 – Accès aux valeurs / noms de colonnes / noms de lignes

```

1 df.values # remarquez de c'est un objet de type numpy array qui est retourné
2
3 df.columns
4
5 df.index
6
7 df.dtypes

```

Nous verrons qu'il est important de faire la différence entre **noms** de lignes/colonnes et **indices** de lignes/colonnes (c'est-à-dire leurs numéros). Commençons par renommer les colonnes, et surtout les lignes pour pouvoir faire cette distinction.

#### Code 19 – Renommer les colonnes et lignes

```
1 df.columns = ['Produit', 'Calories', 'Protéines']
2
3 df.index = ['un', 'deux', 'trois', 'quatre', 'cinq', 'six']
4
5 df # vérification
```

### 4.3. Extraction de données de DataFrames

Une fois qu'un DataFrame est construit, nous voudrions inspecter les informations qui s'y trouvent. Voici différentes manières de faire.

Nous pouvons utiliser ses méthodes **head** et **tail** pour afficher un certain nombre des premières et dernières lignes du DataFrame.

#### Code 20 – Extractions des premières et dernières lignes

```
1 df.head()
2
3 df.tail()
```

Je vous invite à spécifier un nombre entier entre les parenthèses pour avoir le nombre de lignes souhaité.

Nous pouvons extraire des colonnes ou lignes de différentes manières. Voici quelques manières **simplifiées** de faire :

#### Code 21 – Extractions de colonnes et de lignes

```
1 # Extraction de colonnes (via leurs noms)
2 df.Calories
3 df['Calories']
4 df[['Produit', 'Calories']]
5
6 # Extraction de lignes (via leurs indices)
7 df[2:5]
8 df[:5]
9 df[3:]
10
11 # Extraction de colonnes et lignes
12 df.Produit[1:]
13 df['Calories'][:4]
14 df[['Produit', 'Calories']][2:5]
```

Il existe cependant deux méthodes **spécifiques** pour l'extraction de données d'un DataFrame : **iloc** et **loc**.

La méthode `iloc` est conçue pour **extraire des lignes et colonnes d'un DataFrame en se basant sur les indices des lignes et colonnes**. Cette manière de faire est similaire à des extractions sur des arrays `numpy`.

#### Code 22 – Extractions grâce à `iloc`

```
1 df.iloc[2:5, :]    # équivalent à df.iloc[2:5]
2 df.iloc[:, [0,2]]
3 df.iloc[2:5, [0,2]]
4 df.iloc[2:5, 2:]
5 df.iloc[[0,4], [0,2]]
```

Il ne faut pas confondre `iloc` basée sur les indices (d'où le `i`), avec la méthode `loc` qui permet d'**extraire des lignes et colonnes en se basant sur leurs noms**.

#### Code 23 – Extractions grâce à `loc`

```
1 df.loc[['deux','trois','un'],:]    # équivalent à df.loc[['deux','trois','un']]
2 df.loc[:,['Produit','Protéines']]
3 df.loc[['deux','trois','un'],['Produit','Protéines']]
```

### 4.4. Modification d'un DataFrame

Vous pouvez vous baser sur les extractions précédentes avec `iloc` ou `loc`, pour modifier le contenu d'un DataFrame.

#### Code 24 – Modification de valeurs d'un DataFrame

```
1 df.loc[:, 'Calories'] = 110
2 df
3
4 df.Protéines == 2 # permet d'identifier dans la colonne Protéines, les éléments égaux à 2
5 df.loc[df.Protéines == 2, 'Protéines'] = 1
6 df
7
8 df.iloc[2:, 1] = 120
9 df
```

### 4.5. Quelques méthodes utiles

Voici quelques méthodes utiles d'ordonnancement et de classement d'un DataFrame. A vous d'expérimenter !

#### Code 25 – Trier un DataFrame selon les noms de lignes ou de colonnes

```
1 df.sort_index(axis=0)
2
3 df.sort_index(axis=1)
4
5 df.sort_index(axis=0, ascending=False)
```

**Code 26 – Trier un DataFrame selon ses valeurs**

```
1 df.sort_values(by='Protéines')
2
3 df.sort_values(by=['Protéines', 'Produit'])
```

**Code 27 – Rang dans le classement des valeurs**

```
1 df.rank(method='min')
```

Enfin, il est bon de connaître les méthodes **unique**, **count** et **value\_counts** d'un DataFrame. La première permet de récupérer les modalités/valeurs différentes d'une colonne du DataFrame. La deuxième permet de compter l'effectif total, alors que la troisième permet d'obtenir pour une (ou plusieurs) colonne(s), les effectifs en fonction de ses modalités.

**Code 28 – Modalités et effectifs**

```
1 df.Protéines.unique()
2
3 df.Protéines.count()
4
5 df.Protéines.value_counts()
```

**Exercices**

Pour ces exercices, utiliser le DataFrame **df2** créé dans le Code 17 qui n'est pas censé avoir été modifié, contrairement à **df**.

1. Trier **df2** par ordre alphabétique inverse des noms de céréales.
2. Trier **df2** dans l'ordre décroissant du nombre de calories, et utiliser **head** pour récupérer la ligne du DataFrame avec le plus grand nombre de calories.
3. Afficher pour chaque valeur de la colonne **Calories**, l'effectif correspondant.
4. En utilisant la réponse à la question précédente et la méthode **count()**, afficher la distribution en fréquence de la colonne **Calories**.