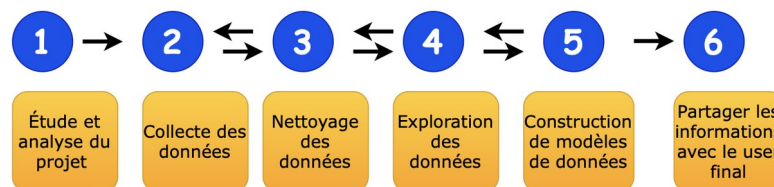


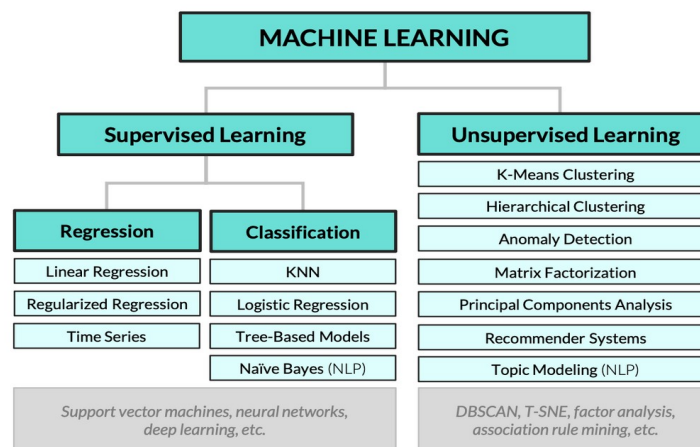
Différentes étapes du processus de la science de données

Le processus de la science des données implique la définition d'un projet, la collecte, le nettoyage, l'exploration des données, la construction de modèles, et enfin, la communication des résultats aux utilisateurs finaux.



→ **1-Étude et analyse du projet** : on commence par définir les objectifs, les techniques et les sources de données qu'on prévoit à utiliser dans notre analyse. Les étapes à entreprendre sont les suivantes :

- Penser comme un utilisateur final (End User -EU- ou partie prenante) qui bénéficiera des résultats de l'analyse.
- Générer des problèmes avec EU afin d'identifier des améliorations et des opportunités.
- Proposer des solutions potentielles. La science des données peut être l'une des solutions possibles, mais ce n'est pas la seule.
- Déterminer les techniques : si on décide d'adopter une approche de science des données pour résoudre le problème, la prochaine étape consiste à déterminer les techniques les plus appropriées : l'apprentissage supervisé (*Supervised Learning*) axé sur l'utilisation de données historiques pour prédire des événements futurs, et l'apprentissage non supervisé (*Unsupervised Learning*) qui se concentre sur la recherche de modèles ou de relations dans les données. La Figure montre les différentes approches de chaque technique.



- Identifier les besoins en données en collectant les bonnes données pour établir une base adéquate pour notre analyse.

→ **2-Collecte des données** : implique la recherche de données, leur lecture dans Python, leur transformation si nécessaire, et le stockage des données dans un *DataFrame Pandas*. Les données peuvent provenir de diverses sources : fichiers (.csv, .xlsx, .tsv, .txt), bases de données, Web (scraping : .html, api : .json, .xml), etc. Les données peuvent être **structurées** (déjà stockées sous forme de tables, telles que les fichiers .xlsx et .db), **semi-structurées** (facilement convertibles en tables, telles que les fichiers .json et .csv), et **non structurées** (non facilement convertibles en tables, telles que .pdf et .jpeg).

→ **3-Nettoyage de données** : pour mettre les données brutes dans un format prêt pour l'analyse. Même s'il existe des outils automatisés disponibles, effectuer un certain nettoyage manuel des données offre une excellente opportunité de commencer à comprendre et de se familiariser avec les données. Cela comprend : conversion des colonnes dans les types de données corrects pour l'analyse, gestion des problèmes de données qui pourraient influencer les résultats de l'analyse, et la création de nouvelles colonnes à partir de colonnes existantes qui sont utiles pour l'analyse.

→ **4-Exploration de données** : comprend une variété de techniques utilisées pour mieux comprendre un ensemble de données et découvrir des motifs et des informations cachées. On mobilise des techniques Python pour explorer un nouvel ensemble de données, telles que le filtrage, le tri, le regroupement et la visualisation (histogrammes, diagrammes de dispersion, etc).

Remarque : on estime qu'environ 50 à 80% du temps d'un *data scientist* est consacré au nettoyage et à l'exploration des données.

→ **5 et 6 : Modélisation de données et partage des observations avec EU** : implique de structurer et de préparer les données pour des techniques de modélisation spécifiques, ainsi que d'appliquer des algorithmes pour faire des prédictions (dans la pratique, la simplicité est la meilleure approche). Finalement, la dernière étape du processus est de résumer les principales observations (*insights*) et de partager des informations avec les utilisateurs finaux ou les parties prenantes. Cela implique de réitérer le problème, d'interpréter les résultats de l'analyse, de partager des recommandations et de se concentrer sur l'impact potentiel des résultats, plutôt que sur les détails techniques.

Dans cette SAE, nous allons découvrir toutes étapes à travers des TPs et un projet les mettant en œuvre à la fin.

Boîte à outils d'un data scientist pour nettoyer et préparer les données

Partie 1- Bibliothèque numpy

« Des données ! Des données ! Donnez-moi des données ! » s'écria-t-il avec impatience. « Je ne peux pas faire de briques sans argile ! » -
Arthur Conan Doyle

A- Le module numpy

L'un des inconvénients d'une liste est que c'est un objet à une dimension, alors qu'on peut avoir besoin de plusieurs dimensions. Une solution naturelle consiste à travailler avec des listes de listes : chaque élément de la liste est alors une liste, et l'accès au j^e élément de la liste `L[i]` se fait par `L[i][j]`. Il est en fait plus simple de définir une structure adaptée à ce problème, le tableau, qui peut être de plusieurs dimensions. On disposera alors des fonctions et méthodes adaptées à cette nouvelle structure de données.

Afin de travailler avec des tableaux en Python, on a besoin d'importer le module **numpy**, ce que nous ferons de la façon suivante : `import numpy as np`

On disposera ainsi de nombreux outils additionnels. Rappelons qu'une fonction **fct** présente dans **numpy** doit alors être appelée par la commande `np.fct`.

I.Définition, création et affichage d'un tableau

Un tableau est une structure ordonnée de données, modifiable et homogène (les données doivent **toutes être de même type**). Chaque élément du tableau est caractérisé par un ou plusieurs index (un pour un tableau à une dimension, deux pour un tableau à deux dimensions...). Ces index sont des entiers naturels. En Python, les tableaux manipulés avec **numpy** sont de classe **numpy.ndarray**.

→ Création et affichage d'un tableau

De manière générale, pour créer un tableau bidimensionnel, il faut créer une liste contenant des listes, ce qui permet de représenter un tableau à deux entrées : les lignes et les colonnes. Pour cela, il suffit de considérer que la liste qu'on donne à la fonction **array()** est une liste de lignes et que les listes contenues dans cette liste représentent les colonnes. Ainsi, chaque ligne contient une liste qui correspond aux colonnes de cette ligne. Voici la syntaxe :

```
tab_a=np.array ( [[L1C1, L1C2, L1C3],[L2C1, L2C2, L2C3 ], ...])
```

À vous de jouer :

1- En utilisant `np.array(...)`, créer un vecteur (1, 2, 3) et l'affecter à la variable `vect`, et une matrice ((4,5,6),(7,8,9)) et l'affecter à la variable `mat`. Afficher le contenu des deux variables et leur type.

2- `vect` et `mat` sont de type `ndarray`. Un objet de ce type ne peut contenir qu'un seul type de données. Pour connaître le type des données qu'il contient, il suffit d'utiliser l'attribut `dtype`. Par exemple : `nom_variable.dtype`. Afficher le type de données contenu dans `vect` et `mat`.

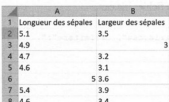
3- `mat` est un tableau d'entiers. Qu'affiche l'instruction `mat[0,0] = "chaine" ?`

Que peut-on déduire vis à vis le changement d'une valeur par un type caractère ?

→ Création d'un ndarray grâce à des fonctions numpy :

Pour créer un tableau, on peut également utiliser des tableaux prédéfinis, comme le montrent les exemples suivants :

Code Python	Explication	Résultat
<code>B= np.arange(5)</code> <code>print(B)</code>	<code>arange([start,] stop[,step,], dtype=None)</code> crée un tableau contenant une séquence d'éléments dont les valeurs sont comprises entre <code>start</code> et <code>stop</code> (avec un pas : <code>step</code>). <code>dtype</code> spécifie le type de données (par défaut <code>None</code>).	<code>[0 1 2 3 4]</code>
<code>C=np.zeros ((2,3))</code> <code>print(C)</code>	<code>np.zeros(shape, dtype=None, order = 'C')</code> Cette fonction crée un tableau rempli de 0. Les paramètres : <code>shape</code> spécifie la forme (dimension) du tableau (Lignes, Colonnes), <code>dtype</code> spécifie le type de données des éléments du tableau (par défaut, <code>float64</code>), et <code>order</code> spécifie l'ordre de stockage des éléments dans la mémoire (par défaut, 'C' pour l'ordre de stockage en ligne et 'F' pour colonnes).	<code>[[0. 0. 0.]</code> <code>[0. 0. 0.]]</code>
<code>E= np.ones ((3,2))</code> <code>print(E)</code>	<code>np.ones(shape, dtype=int, order = 'C')</code> similaire à <code>np.zeros()</code> . Elle remplit le tableau par des 1.	<code>[[1 1]</code> <code>[1 1]</code> <code>[1 1]</code> <code>]</code>
<code>F= np.diag ([13,42])</code> <code>print(F)</code>	<code>np.diag(v, k=0)</code> crée une matrice diagonale à partir d'un vecteur <code>v</code> . Les éléments de <code>v</code> sont placés sur la	<code>[[13 0]</code> <code>[0 42]]</code>

	diagonale principale. Le paramètre k spécifie l'index de décalage par rapport à la diagonale principale (par défaut, k=0 pour la diagonale principale).	
<pre>G= np.eye(3) print(G)</pre>	np.eye(N,M=None,k=0, dtype=float, order='C') crée une matrice identité de taille NxM . Les éléments de la diagonale principale sont remplis de 1 et le reste des éléments sont remplis de 0. Le paramètre k spécifie l'index de décalage par rapport à la diagonale principale	<pre>[[1. 0. 0.] [0. 1. 0.] [0. 0. 1.]]</pre>
<pre>H=np.arange(12).reshape(4,3) print(H)</pre>	np.reshape(shape) permet de modifier la forme d'un tableau multidimensionnel. Elle renvoie une nouvelle vue du tableau avec une nouvelle forme spécifiée shape .	<pre>[[0 1 2] [3 4 5] [6 7 8] [9 10 11]]</pre>
<pre>I = np.random.rand(3,1) print(I)</pre>	np.random.rand(n, p) qui permet de créer un tableau à n lignes et p colonnes formé de réels choisis aléatoirement dans [0 ; 1[<pre>[[0.85381735] [0.81753528] [0.18706386]]</pre>
<pre>J= np.genfromtxt ("mon fichier.txt", delimiter=";", skip_header=True)</pre>	Pour créer un ndarray à travers un fichier, on utilise la fonction genfromtxt(...) . L'option delimiter permet de spécifier à la fonction genfromtxt(...) quel est le séparateur de colonne, et skip_header permet d'ignorer l'entête du fichier.	 <pre>[[5.1 3.5] [4.9 3.] [4.7 3.2] ... [4.6 3.4]]</pre>

À vous de jouer :

- 4-Créer deux vecteurs contenant respectivement 10 valeurs de zéro et 10 valeurs de 8.
- 5- Créer un vecteur de contenant les nombres paires allant de 10 à 62.
- 6- Construire en Python, sans rentrer les valeurs une à une, les matrices suivantes de la manière la plus simple possible. Affecter les matrices créées aux variables **mat_a**, **mat_b** et **mat_c** respectivement puis les afficher.

```
[[ 2  1  1  1]
 [ 1  3  1  1]
 [ 1  1  4  1]
 [ 1  1  1  5]]
```

```
[[13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]
 [25 26 27 28]]
```

```
[[2  -1  0  0]
 [-1  2  -1  0]
 [0  -1  2  -1]
 [0  0  -1  2]]
```

II. Opérations sur les tableaux

→ Indexation (accès à un ou plusieurs éléments)

L'accès et la modification d'éléments fonctionnent exactement comme pour les listes, la seule différence est que cette fois un élément peut être repéré par plusieurs index. Un tableau à une dimension est indexable comme une liste, pour un tableau à deux dimensions les éléments sont repérés par un index de ligne et un index de colonne.

Code Python	Explication	Résultat
<pre>tab[2]</pre> <pre>tab[2,1]</pre> <pre>ou</pre> <pre>tab[2][1]</pre> <pre>tab[2,1] = 10</pre>	<p>Indexation simple :</p> <p>→ Pour un tableau à une seule dimension : on accède à un élément par son indice ou placé entre crochets [].</p> <p>→ Pour un tableau à 2 dimensions : on accède à un élément par ses indices ligne, colonne.</p> <p>→ Pour modifier, on sélectionne la position où se trouve la valeur à modifier et lui assigne la nouvelle valeur</p>	<p>→ On accède à la 3^e valeur du tableau.</p> <p>→ on accède à la valeur située à la 3^e ligne, 2^e colonne du tableau</p>
<pre>tab=</pre> <pre>np.array([[1.5,2,3]</pre> <pre>,[4,5.2,6],</pre> <pre>[7,8,9.1]])</pre> <pre>tab[tab > 5.3]</pre>	<p>Indexation booléenne :</p> <p>On utilise des masques booléens pour sélectionner une portion du tableau. On utilise les opérateurs relationnels qui comparent les valeurs : Rappels: les opérateurs relationnels, permettent de comparer des valeurs : <code>==</code> , <code>!=</code> , <code>></code> , <code><</code> , <code>≥</code> , <code>≤</code></p> <p>La condition <code>tab > 5.3</code> va générer un ndarray de même dimension que le tableau qu'on cherche à indexer, contenant des valeurs booléennes. Ce tableau de booléen permettra à Python de savoir quelles valeurs de <code>tab</code> à garder: celles aux positions où la valeur est à True dans le ndarray généré par <code>tab > 5.3</code>. On obtient comme résultat un ndarray à une seule dimension.</p>	<p>Avec <code>tab > 5.3</code>, on a le tableau de booléens :</p> <pre>[</pre> <pre>False, False, False],</pre> <pre>False, False, True],</pre> <pre>True, True, True]</pre> <pre>]</pre> <p>et <code>tab[tab > 5.3]</code> donnera :</p> <pre>[6. , 7. , 8. , 9.1]</pre>
<pre>tab[[5,2,8,4]]</pre>	<p>Fancy indexing :</p> <p>On peut sélectionner plusieurs éléments non consécutifs par indices. Ainsi on donne un tableau d'indices (ou une liste d'indices ou plusieurs) entre crochets au ndarray.</p>	<p>Ici, on obtient un tableau à une seule dimension contenant les éléments sélectionnés</p>

<pre>tab_a=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])</pre>	<p>Accéder aux éléments par tranche (slicing) :</p> <p>Le slicing permet de sélectionner des tranches de tableaux (slices, en anglais) et donc de sélectionner plusieurs éléments du tableau.</p> <p>→ Tableau à 1D</p> <p>Sa syntaxe est : tab[deb:fin:pas]. deb spécifie le début (0 si ce n'est pas indiqué), fin spécifie la valeur d'index où on veut s'arrêter (nom compris, c'est le dernier indice du tableau si ce n'est pas indiqué), et le pas (1 par défaut).</p> <p>→ Tableau à 2D</p> <p>Sa syntaxe est : tab[deb:fin:pas, deb:fin:pas]</p>	<pre>print(tab_a[:3])</pre> <p>Affiche : [1,2,3]</p> <pre>print(tab_a[16:])</pre> <p>Affiche : [17,18,19,20]</p> <pre>print(tab_a[3:7])</pre> <p>Affiche : [4,5,6,7]</p> <pre>print(tab_a[::2])</pre> <p>Affiche : [1,3,5,7,9,11,13,15,17,19]</p>
---	---	---

À vous de jouer :

7- Créer une matrice de 5x5 contenant des valeurs allant de 1 à 25 et l'affecter à la variable **mat_d**. Afficher la matrice.

8- Écrire les instructions permettant d'obtenir :

```
→ array([[12, 13, 14, 15], [17, 18, 19, 20], [22, 23, 24, 25]])
```

→ la valeur 15 puis la valeur 22.

```
→ array([[ 2], [ 7], [12]])
```

```
→ array([21, 22, 23, 24, 25])
```

```
→ array([[16, 17, 18, 19, 20], [21, 22, 23, 24, 25]])
```

```
→ array([[21 22 23 24 25], [ 6  7  8  9 10], [16 17 18 19 20]])
```

```
→ array([[False False False False False], [False False False False False], [False False True True True], [ True True True True True]])
```

```
→ array([18 19 20 21 22 23 24 25])
```

→ Inspecter un tableau grâce aux attributs de NumPy

Les objets de classe **ndarray** possèdent un ensemble d'attributs qui permettent de les inspecter. La liste de ces attributs est disponible dans la documentation de NumPy:

<https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>

Voici quelques uns :

- **dtype** : pour connaître le type des éléments d'un tableau.
- **size** : pour connaître le nombre d'éléments contenus dans un tableau.
- **T** : pour faire la transposée d'un tableau (permuter les dimensions : les lignes devenant les colonnes et les colonnes des lignes).
- **ndim** : pour connaître la dimension d'un tableau (1D, 2D, etc).
- **shape** : pour connaître la taille des dimensions d'un tableau, c'est-à-dire le nombre de lignes et de colonnes.
- **reshape(n,p)** : transforme le tableau à une dimension et n x p valeurs en un tableau à n lignes et p colonnes contenant ces mêmes valeurs.

Pour utiliser l'un des attributs, il suffit d'utiliser la syntaxe : **array.nom_attribut**.

A vous de jouer :

9- Écrire les instructions permettant d'afficher le nombre d'éléments, la dimension, la taille et la transposée de **mat_d**.

→ Opérateurs mathématiques avec Numpy et fonctions d'agrégation

-Les opérations arithmétiques :

Il est possible d'ajouter, de soustraire, de multiplier ou de diviser des valeurs sur un **ndarray**. Il y a des fonctions NumPy dédiées à cette tâche, ce sont les fonctions **add()** (ajouter), **subtract()** (soustraire), **multiply()** (multiplier) et **divide()** (diviser).

Par exemple, si on a : **mon_tab=np.array ([[10, 11, 12], [13, 14, 15], [16, 17, 18]])**

l'instruction : **np.add (mon tableau, 10)** va afficher : **array ([[20, 21, 22], [23, 24, 25], [26, 27, 28]])**.

Il est possible de faire des calculs entre tableaux.

Par exemple, si on a : **mon_tab2=np.array([10,20,30])**, l'instruction : **np.subtract (mon_tab, mon_tab2)** va afficher : **array ([[0, -9, -18], [3, -6, -15], [6, -3, -12]])** (on enlève 10 à la 1^{re} colonne, 20 à la 2^e colonne et 30 à la 3^e colonne de **mon_tab**). Ici NumPy utilise le Broadcasting en

diffusant `mon_tab2` sur chaque ligne de `mon_tab`, cela revient à soustraire `mon_tab2` à chaque ligne de `mon_tab`.

-Fonctions d'agrégation :

Lorsqu'on a un grand tableau de données, il est important de pouvoir explorer un résumé de ce tableau. La classe `ndarray` de `NumPy` possède des méthodes d'agrégation : `sum()` (somme), `median()` (médiane), `mean()` (moyenne), `min()` (minimum), `max()` (maximum), `corrcoef()` (coefficient de corrélation), `cumsum()` (somme cumulative) et `std()` (écart-type). Ces méthodes peuvent être appliquées soit sur l'ensemble des éléments du tableau, soit par colonne ou par ligne, dans le cas de tableaux à deux dimensions. Pour cela, il suffit de préciser l'axe qu'on veut traiter avec l'option `axis` : `axis=0` (colonne) et `axis=1` (ligne).

Par exemple, soit : `tab = np.array([[1.5,2,3], [4,5.2,6], [7,8,9.1]])`, l'instruction : `np.sum(tab, axis=0)` affiche : `array([12.5, 15.2, 18.1])` (la somme sera effectuée par colonne et on aura une valeur/colonne), et l'instruction : `np.sum(tab, axis=1)` affiche : `array([6.5, 15.2, 24.1])`.

À vous de jouer :

10- Afficher les éléments de `mat_d` élevés à la puissance 3.

11- Afficher la somme et l'écart-type de tous les éléments de `mat_d`. Afficher la somme de toutes les colonnes puis celle de toutes les lignes. Afficher les moyennes des colonnes sous forme de valeurs entières en affectant à l'argument `dtype` la valeur `np.int`.

12-Afficher la valeur minimale et la valeur maximale de `mat_d`.

→ Ajouter et supprimer des éléments dans un tableau

➤ Pour ajouter des éléments, il y a différentes manières de faire. Soit on ajoute les éléments à la fin du tableau avec la fonction `append()` de `NumPy`, soit on les insère à une position donnée avec la fonction `insert()`.

→ Syntaxe d'utilisation de `append()` :

```
np.append(mon_array, elements_aajouter)
```

Exemples :

```
tab = np.array([1,2,3,4,5,6,7,8,9])
```

```
np.append(tab, [21,22,23])    affiche :    array([1,2,3,4,5,6,7,8,9,
21,22, 23])
```

```
tab_bis = np.array([[1.5,2,3], [4,5.2,6], [7,8,9.1]])
```

`np.append(tab_bis, [[10,10,10]],axis=0)` affiche :

```
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5.2,  6. ],
       [ 7. ,  8. ,  9.1],
       [10. , 10. , 10. ]])
```

→ **Syntaxe d'utilisation de insert()** : `np.insert (mon array, index, elt)`

L'élément **elt** sera inséré à la position **index** qu'on spécifie à la fonction **insert()**. Comme pour la fonction **append()** , il est possible de spécifier un axe sur lequel ajouter le ou les éléments.

Exemples :

```
tab = np.array([1,2,3,4,5,6,7,8,9])
```

```
np.insert(tab, 1, [21,22,23]) affiche : array([1,21,22,
23,2,3,4,5,6,7,8,9])
```

```
tab_bis = np.array([[1.5,2,3], [4,5.2,6], [7,8,9.1]])
```

`np.append(tab_bis, 1, [[10,10,10]],axis=0)` affiche :

```
array([[ 1.5,  2. ,  3. ],
       [10. , 10. , 10. ],
       [ 4. ,  5.2,  6. ],
       [ 7. ,  8. ,  9.1]])
```

➤ Pour supprimer des éléments, il suffit d'utiliser la fonction **delete()** de NumPy. Sa syntaxe est : `np.delete (mon array, index, axe)`. Si on ne précise pas l'axe, les éléments aux index donnés seront supprimés. Si on précise l'axe, par exemple **axis=1**, ce sont les colonnes entières, aux numéros d'index spécifiés, qui sont supprimées. Par exemple :

```
tab_bis = np.array([[1.5,2,3], [4,5.2,6], [7,8,9.1]])
```

```
np.delete(tab_bis, 1,axis=1) affiche : array([[1.5,3.], [4.,6.],
[7.,9.1]])
```

À vous de jouer : Manipulation de fichiers sous NumPy

Soit le fichier « **consommation_alcool.csv** » qui contient les données sur la consommation moyenne d'alcool d'un habitant par année, par pays et par type d'alcool. Les descripteurs de ce fichier sont :

→ **Year(Année)** : spécifie l'année à laquelle les données de consommation d'alcool se réfèrent.

- **WHO region (Région OMS)** : spécifie la région de l'organisation mondiale de la santé à laquelle le pays appartient. Par exemple : Western Pacific, Americas, etc.
- **Country (Pays)** : indique le nom du pays concerné par les données.
- **Beverage Types (Types de boissons)** : indique le type de boisson alcoolisée concernée par les données. Par exemple Wine(vin), Other (boisson non spécifiée), etc.
- **Display Value (Valeur affichée)** : représente la quantité moyenne de consommation d'alcool pour une combinaison spécifique d'année, de région, de pays et de type de boisson. Par exemple : une valeur 0 indique une absence de consommation.

Partie 1- Lecture de fichiers et affichage correcte des données :

1- Lire le fichier « **consommation_alcool.csv** » à l'aide de la fonction **genfromtxt(...)**, en affectant son contenu à une variable appelée **donnees**. Ne pas oublier de préciser le délimiteur (séparateur) qui est la virgule.

2- Afficher les 3 premières lignes. Que remarquez-vous ?

On obtient des valeurs **NaN** (Not a Number) dans certaines cellules. En fait, NumPy essaie de deviner le type de chaque colonne. Par défaut, il essaie de convertir les valeurs en nombres. Lorsqu'il rencontre une valeur qu'il ne peut pas convertir en nombre (comme les chaînes de caractères contenant des nombres), il place **NaN** à cette position.

Pour résoudre ce problème, il faut spécifier le type de données dans la fonction **genfromtxt(...)** en utilisant le l'argument **dtype="U75"** (ceci veut dire : lire les données comme des chaînes de caractères Unicode de longueur maximale 75 caractères et donc NumPy ne tente pas de convertir les valeurs en nombres).

3- Ajouter le paramètre **dtype="U75"** à **genfromtxt(...)**, puis ré-afficher les trois premières lignes de **donnees**.

4- Modifier l'appel de la fonction **genfromtxt(...)** de telle sorte de ne pas garder les descripteurs du fichier CSV (soit la première ligne de la variable **donnees**).

Partie 2- Indexation des données

Maintenant que les données sont en bon format. Nous allons essayer d'extraire certaines informations :

- 1- Créer la variable **nigeria_84** et lui assigner le nombre de litres de vin bu par un Nigérien.
- 2- Créer la variable **dernier_pays** et lui assigner le nom du dernier pays qui apparaît dans le fichier CSV.
- 3-Créer une variable **noms_pays** et lui assigner les noms des pays du fichier CSV.

- 4-Créer la variable **consommations_pays** et lui assigner la consommation de tous les pays.
- 5- Créer une variable **annees_regions** et lui assigner toutes les lignes des colonnes année et région du fichier CSV.
- 6- Créer une variables **trente_prem_annees** et lui assigner les 30 premières années du fichier CSV.
- 7- Créer une variable **vingt_prem_pays_conso** et lui assigner les vingts premières lignes des colonnes : nom de pays et consommation.

Partie 3- Analyser les données

Les comparaisons sont le concept clés de Numpy. On peut effectuer des comparaisons avec plusieurs conditions et sélectionner des éléments.

- 1- On dispose de la colonne **noms_pays** qui contient les noms des pays. Comparer le vecteur résultat au pays «Finland». Assigner le résultat de comparaison à la variable **rech_finland**. Sélectionner seulement les lignes du fichier CSV pour lesquelles **rech_finland** vaut **True** et afficher le résultat.

- 2- Faire le même travail que la question précédente et récupérer toutes les lignes correspondantes à l'année 1987. Assigner le résultat à la variables **annees_1987**. Afficher le résultat.

- 3-On peut effectuer des comparaisons en utilisant plusieurs critères en utilisant les opérateurs & et |.

Sélectionner les lignes dont le pays est « Finland » et l'année est « 1987 », puis assigner le résultat à la variable **lignes_finland_en_87**. Afficher le résultat.

- 4-Créer une copie du tableau **donnees** et l'assigner à la variables **donnees_bis**.

- 5- Dans le tableau **donnees_bis**, remplacer toutes les années 1987 par 2024 et tous les alcools de type « Wine » par « Beer ».

Aide :

Soit **tab_orig** est un tableau numpy :

```
tab_orig = np.array([1, 2, 3, 4, 5])
```

Pour créer une copie superficielle, on utilise la fonction **view()** :

```
copie_superf = tab_orig.view()
```

Attention : En changeant le contenu de copie_superf, vous changerez aussi le contenu de tab_orig.

Pour créer une copie profonde, on utilise la fonction **copy()** :

```
copie_prof = tab_orig.copy()
```

Avec la copie profonde, si vous changez le contenu de copie_prof, vous ne modifierez pas le contenu de tab_orig.

Les données stockées dans la variable **donnees** sont des chaînes de caractères. Pour pouvoir réaliser des calculs, il faut convertir ces données en float. Il est à noter que certaines valeurs sont manquantes : on trouve à la place des chaînes vides, et convertir ces dernières en valeur réelles lèvera une erreur de type **ValueError (could not convert string to float: '')**. Pour résoudre ce problème il faut remplacer les valeurs manquantes par « 0 ».

6- Sélectionner les valeurs de la dernière colonne du fichier CSV qui sont sous forme de chaîne de caractères vide puis leur affecter la chaîne de caractères '0'.

Pour convertir vers un type donné, on utilise la fonction **astype(type)**. Par exemple pour convertir en float : **vecteur = np.array(["2", "3", "4"])**, on a l'instruction suivante : **vecteur.astype(float)**.

7-Convertir en décimal (float), la dernière colonne du fichier CSV relative à la consommation moyenne (en litres) d'une personne par rapport à un pays et une année donnée. Assigner le résultat à la variable **moy_consommation**.

8- Utiliser les méthodes **sum()** et **mean()** pour calculer respectivement la somme et la moyenne des valeurs de la variable **moy_consommation**. Affecter ces résultats aux variables **total_conso** et **moyenne_conso**. Afficher les résultats.

9- On souhaite calculer la consommation totale annuelle par habitant pour un pays donné. Pour ce faire :

- Créer une variable **france_86** et lui affecter toutes lignes du fichier CSV correspondant à l'année 1986 et au pays France.
- Calculer le total de consommation d'alcool d'un français durant l'année 86, et l'affecter à la variable **francais_conso_86**. Afficher le résultat.

10- On veut calculer la consommation pour chaque pays. Pour ce faire :

- Créer un dictionnaire **dico_total**, dont les clés sont les noms des pays et les valeurs sont le total de consommation d'alcool pour une année donnée.
- Sélectionner les lignes du fichier CSV correspondant à l'année 1987 et affecter le résultat à **annee_87**.
- Créer une liste **pays**, contenant les noms des pays extraits à partir de **annee_87**.
- Parcourir la liste des pays. Pour chaque pays donné, il faut sélectionner les lignes relatives au pays, extraire la colonne de consommation et remplacer les valeurs

manquantes par '0', ensuite calculer le total de consommation et mettre à jour le dictionnaire.

→ Afficher le dictionnaire.

11- À partir du dictionnaire **dico_total**, écrire les instructions nécessaires pour trouver le pays qui a consommé le plus d'alcool durant l'année 1987.

Boîte à outils d'un data scientist pour nettoyer et préparer les données

Partie 2- Bibliothèque pandas

Pandas est une librairie Python dédiée à la Data Science. Il s'agit d'ailleurs de la librairie Python la plus populaire et la plus performante pour faire de l'analyse de données.

Cette librairie amène avec elle deux nouvelles structures essentielles pour l'analyse de données, appelées **Series** et **DataFrame**. Un objet de type **DataFrame**, peut être assimilé à un tableau à deux dimensions ou encore une feuille Excel, ce qui correspond à la structure de données la plus utilisée en Data Science. Cette structure est composée de lignes représentant des observations ou individus et de colonnes correspondant aux variables décrivant des observations/individus.

The diagram illustrates a DataFrame structure. It shows a table with 4 columns and 4 rows. The columns are labeled 'Variable1', 'Variable2', 'Variable3', and 'Variable4', with specific values 'Taille', 'Poids', 'Age', and 'Sexe' below them. The rows are labeled 'Individu1', 'Individu2', 'Individu3', and 'Individu4'. Brackets on the left group the rows under the label 'Lignes (observations, individus, ...)' and a bracket on top groups the columns under 'Colonnes (variables)'.

	Variable1 Taille	Variable2 Poids	Variable3 Age	Variable4 Sexe
Individu1	183	80	20	M
Individu2	165	63	53	F
Individu3	180	70	19	M
Individu4	154	49	34	F

Un objet de type "**Series**" peut être assimilé à un vecteur, c'est-à-dire à une suite de valeurs. La série correspond aussi à une colonne d'un **DataFrame**. En réalité, le **DataFrame** est constitué d'autant de séries qu'il a de colonnes. Ci-dessous, voici comment des séries peuvent être représentées visuellement.

The diagram shows two separate Series objects. Each has a header 'Variable' and a title 'Taille' for the first and 'Poids' for the second. They share the same row labels: 'Individu1', 'Individu2', 'Individu3', and 'Individu4'. Brackets on the left group the rows under 'Lignes (observations, individus, ...)'. Brackets on top group each column under 'Colonne (variable)'. The first series is labeled 'Series 1' and the second 'Series 2'.

	Variable Taille	Variable Poids
Individu1	183	80
Individu2	165	63
Individu3	180	70
Individu4	154	49

C'est seulement avec l'arrivée de Pandas et ses structures de données que Python a réellement commencé à être utilisé pour faire de l'analyse de données. Cette librairie apporte non seulement de nouvelles structures ultra performantes pour stocker des données, mais elle apporte aussi un ensemble de méthodes et fonctions associées, permettant d'explorer les **DataFrames** et **Series**, de les nettoyer, les transformer, les manipuler ou encore de les visualiser de manière très efficace et rapide.

Pandas s'appuie fortement sur NumPy, puisque ses structures de données sont basées sur les tableaux NumPy, et elle profite aussi des performances de calcul de NumPy. L'avantage de Pandas par rapport à NumPy est que l'objet de type **DataFrame** permet de stocker des données de types différents (on pourra avoir une colonne contenant des chiffres, une autre contenant du texte, une autre des booléens, etc). Cette possibilité de stocker des données hétérogènes vient du fait qu'un **DataFrame** est constitué d'un ensemble de **Series** qui sont indépendantes entre elles et peuvent donc contenir des types de données différents de l'une à l'autre. En revanche, au sein d'une colonne, les données doivent être de même type. De plus, les **DataFrames** et les **séries** sont fournis avec la possibilité d'assigner des étiquettes aux données, plutôt que de travailler avec des index numériques, comme c'était le cas avec les **ndarrays** de NumPy. Avec Pandas, les lignes et les colonnes peuvent être identifiées avec des étiquettes plutôt que des nombres.

Pour utiliser la librairie **pandas**, on utilise la convention d'importation suivante :

```
import pandas as pd
```

Pour commencer à travailler avec pandas, il faut se familiariser avec les deux structures de données de base : **Series** et **DataFrames**. Bien qu'elles ne constituent pas une solution universelle pour tous les problèmes, elles fournissent une base solide et facile d'emploi pour la plupart des applications.

I. Lire et écrire des fichiers avec pandas

Instructions Python	Explication
<pre>data = pd.read_csv(file_path, sep, header)</pre>	<p><code>pd.read_csv(...)</code> permet de lire un fichier CSV. Cette fonction possède un argument obligatoire, qui est le chemin complet vers le fichier ou juste son nom si celui-ci se trouve dans le dossier de travail, sep qui est le séparateur entre les données (par défaut égal à virgule), et header qui spécifie les noms de colonnes (1^{re} ligne) et par défaut header prend la valeur de infer sinon None).</p>

<code>data = pd.read_excel(file_path, sheet_name)</code>	<code>pd.read_excel(...)</code> permet de lire un fichier EXCEL. Cette fonction possède un argument obligatoire , qui est le chemin complet vers le fichier. On peut utiliser <code>sheet_name</code> (par défaut 0-1 ^{re} feuille) pour lire le contenu d'une feuille donnée.
<code>dataframe.to_csv("resultats.csv", sep=";", columns=["col1", "col3"],)</code>	Pour écrire les DataFrames résultats dans un fichier CSV, TXT ou XLSX, il y a les méthodes : <code>dataframe.to_csv(...)</code> ou <code>dataframe.to_excel(...)</code> . Ces fonctions peuvent prendre plusieurs options telles que : le séparateur, la liste des colonnes qu'on souhaite écrire (columns), header pour l'en-tête.
<code>data.head()/data.tail()</code>	<code>head()</code> s'applique à une série ou à un dataframe . Elle permet d'avoir un aperçu des 5 premières lignes de l'objet data/5 dernières lignes. Si on veut afficher les n premières lignes, on écrit : <code>data.head(n)</code> .
<code>noms_colonnes = data.columns</code> <code>dimensions = data.shape</code> <code>data.dtypes</code> <code>data.count()</code> <code>data.info()</code> <code>data.describe()</code>	Voici quelques attributs pour manipuler un DataFrame : → columns : affiche les noms de colonnes → shape : renvoie en tuple (nombre de lignes, nombre de colonnes) → dtypes : affiche les noms des colonnes et leurs types. → count() : affiche le nombre de valeurs dans chaque colonnes. → describe() : affiche les statistiques min , max , mean , etc. → info() : affiche le nombre de valeurs non nulles et le type de données de chaque colonne.

II. Structure de données Pandas : les Series

On peut décrire les séries de Pandas comme étant une colonne d'un **DataFrame**, un tableau à une dimension (vecteur) ou encore une suite de valeurs, numériques ou non. On peut aussi décrire une série comme un ensemble de valeurs représentant une variable pour un ensemble d'observations. Une série peut être créée à partir d'une liste, d'un tableau NumPy, ou encore d'un dictionnaire, grâce à la méthode **Series()**. Elle peut aussi être créée à partir d'un fichier grâce aux fonctions de lecture de fichiers que nous avons vues précédemment, telles que `read_excel()` ou `read_csv()`. La structure de données Series possède deux composants principaux :

- les valeurs, auxquelles on peut accéder avec la méthode **values()**,
- les étiquettes, plus couramment appelées les index, auxquelles on peut accéder avec la méthode **index()**. Les index sont des nombres entiers: ils commencent à 0 et s'incrémentent. Il est possible de les remplacer par des étiquettes de type que l'on souhaite (nombre, caractères).

Instructions Python	Explication
<pre>serie = pd.Series([25,45, 62,12,11]) print(serie)</pre>	<p>pd.Series(...) crée une série à partir d'une liste.</p> <p>On affiche :</p> <pre>0 25 1 45 2 62 3 12 4 11 dtype: int64</pre> <p>Avec Pandas, les différents types sont les suivants :</p> <ul style="list-style-type: none"> → object= chaines de caractères, listes, dico → int64/int32= valeurs entières numériques (int en Python) → float64 = valeurs réelles, à virgules (float en Python) → bool = booléens, valeurs VRAIE oU FAUX (bool en Python)
<pre>serie=pd.read_csv("monfich.csv").squeeze()</pre>	<p>On crée un objet de type Series à partir d'un fichier, en utilisant la méthode squeeze() qui convertit une colonne d'un dataframe en Series.</p>
<pre>ma_serie[1] ma_serie[[1, 5, 9]] ma_serie["val"] ma_serie[["val1", "val2", "val3"]] ma_serie=pd.Series([10,11,12,13], index= [1,3,2,0]) print(ma_serie) ma_serie.loc[0] #affiche 13</pre>	<p>Indexation :</p> <ul style="list-style-type: none"> → via la position : similaire au tableau NumPy, on commence les positions à partir de 0). On peut récupérer des valeurs à partir d'une liste de positions. → via l'étiquette : en récupérant la valeur de l'étiquette "val". On peut également récupérer les valeurs d'une liste des étiquettes. → indexeurs loc et iloc : <p>L'affichage de ma_serie donne :</p> <pre>1 10 3 11 2 12 0 13 dtype: int64</pre> <p>Problème :</p> <p>On constate que le nom de l'index à la position 0</p>

<pre><code>ma_serie.loc[0] #affiche 10</code></pre> <pre><code>sdata = {'Ohio': 35000, 'Texas': 71000, 'Californie': 16000, 'Utah': 5000} obj = pd.Series(sdata) obj[obj >= 27000]</code></pre> <pre><code>obj[1:3] ou obj.iloc[1:3]</code></pre>	<p>est 1 et que le nom d'index à la position 4 de la série est 0. Pour récupérer la valeur 10, on écrit ma_serie[0] ce qui est FAUX car on va récupérer la valeur 13.</p> <p>Solution :</p> <p>Pandas résout ce problème et propose deux solutions pour spécifier si on veut effectuer l'indexation sur les étiquettes d'index ou sur les positions des valeurs. La syntaxe est la suivante :</p> <p>serie.loc[etiquette] serie.iloc [position]</p> <p>L'attribut loc va demander d'utiliser les noms des index, les étiquettes pour sélectionner les valeurs (on renvoie la valeur dont le nom de l'index est 0 dans l'exemple)</p> <p>L'attribut iloc utilise la position des valeurs pour les sélectionner (dans l'exemple, on renvoie la valeur à la position 0)</p> <p>→indexation via une expression booléenne :</p> <p>Similaire au tableau NumPy, on peut utiliser opérateurs de comparaisons (<, ≤, >, ≥, ==, !=).</p> <p>Dans l'exemple, on affiche :</p> <pre><code>Ohio 35000 Texas 71000 dtype: int64</code></pre> <p>→Slicing : découpage de valeurs successives</p> <p>Similaire au tableau NumPy. Pour rappel voici la formule : serie[start: stop: step]</p> <p>Dans l'exemple, on affiche :</p> <pre><code>Texas 71000 Californie 16000 dtype: int64</code></pre>
<p>Les attributs des objets de classe Series :</p> <ul style="list-style-type: none"> → dtype : retourne le type des données contenues dans la série. → iloc et loc : permettent d'accéder aux valeurs de la série, par position ou par étiquette. → index : retourne les index de la série. → values : retourne les valeurs de la série. → name : retourne le nom de la série. → size : retourne le nombre de valeurs contenues dans la série, c'est-à-dire sa taille. 	
<pre><code>legumes = pd.Series(['pomme', 'abricot', np.nan, 'avocat'])</code></pre> <pre><code>print(legumes.isnull())</code></pre>	<p>Valeurs manquantes :</p> <p>pour détecter les valeurs manquantes :</p> <p>→ pd.isnull(serie) ou serie.isnull() : renvoie True pour une valeur manquante dans serie et False sinon. Dans l'exemple, on affiche :</p> <pre><code>0 False 1 False</code></pre>

<pre>legumes[0] = None print(legumes.isnull())</pre>	<pre>2 True 3 False</pre> <p>Dans l'exemple, on affiche :</p> <pre>0 True 1 False 2 True 3 False</pre>
<pre>sdata = {'Ohio': 35000, 'Texas': 71000, 'Californie': 16000, 'Utah': 5000} obj = pd.Series(sdata) obj.value_counts()</pre>	<p>Les méthodes des objets de classe Series :</p> <p>Chaque classe a ses propres méthodes, qui peuvent être comparées à des fonctions capables de manipuler cette classe d'objets précisément. La différence entre les méthodes, spécifiques à une classe, et les fonctions, applicables à plusieurs classes, est la manière de les appeler. Une méthode s'appellera de la manière suivante : serie.methode() alors que la fonction s'appellera de la manière suivante : fonction(serie). Voici quelques exemples :</p> <ul style="list-style-type: none"> → describe() : permet d'afficher un résumé statistique sur les valeurs de la série. → value_counts() : permet de visualiser les valeurs uniques ainsi que leur nombre au sein de la série. → replace() : permet de remplacer une ou plusieurs valeurs par une autre au sein d'une série. → set_index() : permet de redéfinir les index de la série. <p>L'exemple affiche :</p> <pre>35000 1 71000 1 16000 1 5000 1 dtype: int64</pre>
<pre>sdata = {'Ohio': 35000, 'Texas': 71000, 'Californie': 16000, 'Utah': 5000} obj = pd.Series(sdata) obj.append(pd.Series([90500, 120000], index=["NW", "Miami"]))</pre>	<p>Ajout, suppression et modification :</p> <ul style="list-style-type: none"> → Ajout des valeurs à une série : avec la méthode append() qui permet de concaténer un ou plusieurs séries : <pre>serie.append(pd.Series([valeur], index=["label"]))</pre> <p>L'exemple affiche :</p> <pre>Ohio 35000 Texas 71000 Californie 16000 Utah 5000 NW 90500 Miami 120000 dtype: int64</pre>

Trier maintenant dans l'ordre croissant puis dans l'ordre décroissant en affichant le résultat à chaque fois.

5- Utiliser la chaîne de méthodes pour obtenir les cinq dernières lignes des valeurs de `region_data` triées par ordre décroissant.

Rappel : La chaîne de méthodes (ou *method chaining* en anglais) fait référence à la pratique consistant à enchaîner plusieurs méthodes d'un objet les unes après les autres dans une seule ligne de code.

6- Remplacer le contenu de la série `region_data` par le même mais trié dans l'ordre décroissant. Observer l'index. Est-il trié correctement ?

En utilisant la méthode `Series.sort_index()`, trier dans l'ordre croissant selon l'index du nouvel objet `region_data` puis afficher les résultats. Faire la même chose mais dans l'ordre décroissant et afficher les résultats.

7- Exécuter le code suivant permettant de créer un objet Series appelé `emp_date_naiss` qui contient les numéros de 4 employés et leur date de naissance :

```
emp_date_naiss = pd.Series({'No_0001': '1963-08-02',  
                             'No_0002': '1964-06-13', 'No_0003': '1989-12-04', 'No_0004': '1996-  
04-08'})  
print(emp_date_naiss)
```

Donner l'instruction permettant d'afficher séparément les index et les valeurs de `emp_date_naiss`.

III. Structure de données Pandas : DataFrame

On peut décrire les **Dataframes** comme étant des tableaux à deux dimensions, qu'on pourrait aussi comparer à une feuille Excel. Les tableaux à deux dimensions sont les structures de données les plus utilisées en analyse de données, car cela permet de représenter en ligne les individus/observations et en colonne les variables représentant ces individus/observations. Ainsi, ce qui **caractérise** le **dataframe**, ce sont **son index pour les lignes et ses noms de colonnes**. Le **dataframe** peut contenir des données hétérogènes de type `int`, `float`, `bool`, etc., puisque les colonnes sont indépendantes entre elles, chacune correspondant à une série. Le **dataframe** possédant un **index** et des **noms de colonnes**, on dit aussi qu'il possède deux **axes**, appelés **axe0** (pour l'axe des index, ou lignes) et **axe1** (pour l'axe des colonnes). Cette notion d'axe est importante, car certaines fonctions prennent en paramètre l'axe qui est à traiter, il faut donc se rappeler que 0 signifie les lignes et 1 les colonnes. Par exemple, si on souhaite effectuer la somme de chaque colonne d'un **dataframe**, on utilise la fonction `sum()` sur l'axe 0. En revanche, si on souhaite avoir une somme par ligne du **dataframe**, on utilise la fonction `sum()` sur l'axe 1.

The diagram shows a DataFrame table with the following structure:

- Noms de colonnes** (Column names) pointing to the header row.
- Colonnes (axe 1)** (Columns, axis 1) pointing to the header row.
- Etiquettes d'index** (Index labels) pointing to the row indices 0, 1, and 2.
- Index (axe 0)** (Index, axis 0) pointing to the row indices.
- Valeurs** (Values) pointing to the data cells.
- Valeurs manquantes** (Missing values) pointing to the 'NaN' entries in the 'Height' and 'Weight' columns for the third row.

	Name	Sex	Age	Height	Weight	Team	Games	Year	Season	City
0	A Djiang	M	24.0	180.0	80.0	China	1992 Summer	1992	Summer	Barcelona
1	A Lamusi	M	23.0	170.0	60.0	China	2012 Summer	2012	Summer	London
2	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	1920 Summer	1920	Summer	Antwerpen

1. Principaux attributs d'un dataframe

Certains attributs sont très utilisés et permettent de prendre connaissance de la structure et de la dimension du dataframe étudié.

→ **index** : donne la liste des index des lignes (les étiquettes des lignes) du dataframe.

→ **columns** : donne la liste des noms des colonnes du dataframe.

→ **values** : génère un **ndarray Numpy** contenant les valeurs du dataframe.

→ **shape** : donne des informations sur la dimension du dataframe. La sortie est un tuple : (nombre de lignes, nombre de colonnes).

→ **dtypes** : retourne le type des données contenues dans chaque colonne du dataframe.

À vous de jouer : Lecture de fichiers avec pandas-Partie B

Nos partenaires en sondage viennent de terminer la collecte de données de sondage sur les niveaux de bonheur dans chaque pays (fichier : **happiness_data.csv**).

1- En tant que scientifique de données, charger les données de ce fichier dans un dataframe nommé **df_bonheur**. Afficher les cinq premières lignes et le nombre total ainsi que la plage des scores de bonheur.

2- Afficher les descripteurs du **df_bonheur**.

3- Créer un nouveau DataFrame, appelé **df_bonheur_bis**, dont les données sont obtenues à partir des noms des colonnes du DataFrame **df_bonheur**, tandis que les noms des colonnes sont obtenus à partir de la liste **nv_noms_cols** :

```
nv_noms_cols= ['Pays', 'Annee', 'Score_Bonheur', 'Soutien_Social', 'Liberté_Choix_Vie', 'Espérance_Vie_Santé']
```

Aide: il existe deux façons, l'une consiste à utiliser le **paramètre columns**, et l'autre utilise l'**attribut columns**. Essayer les deux.

4- Utiliser un attribut pour obtenir des métadonnées sur les types de données stockés dans les différentes colonnes du DataFrame **df_bonheur**.

5- Bien que cela n'est pas recommandé, afficher les noms des pays sans utiliser l'opérateur d'indexation [].

2. Indexation : sélectionner des valeurs d'un dataframe

Pour accéder aux valeurs d'un **dataframe**, la manière de faire est similaire à celle des séries, sauf qu'ici on aura deux ensembles de valeurs à donner, un ensemble pour les lignes et un pour les colonnes. Ici, l'indexation permettra d'accéder aux valeurs à des lignes et colonnes spécifiques en utilisant les attributs **loc** et **iloc** comme vu précédemment pour les séries.

→ **Indexation et slicing avec l'attribut loc** : **loc** permet de sélectionner des données selon leurs étiquettes, c'est-à-dire selon les étiquettes des lignes et les noms des colonnes. La sélection se fait toujours entre crochets, mais cette fois-ci, deux ensembles de valeurs seront attendus, séparés par une virgule. Pour rappel, l'indexation permet de sélectionner une ou plusieurs valeurs, consécutives ou non.

- ◆ Pour sélectionner une valeur unique :

```
dataframe.loc["nom_ligne", "nom_colonne"]
```

- ◆ Pour sélectionner plusieurs valeurs :

```
dataframe.loc[["nom_L1", "nom_L2",...], ["nom_C1", "nom_C2",...]]
```

- ◆ Pour sélectionner une ligne spécifique et toutes les colonnes :

```
dataframe.loc["nom_ligne",:]
```

- ◆ Pour sélectionner plusieurs lignes et toutes les colonnes :

```
dataframe.loc[["nom_L1", "nom_L2",...],:]
```

- ◆ pour sélectionner une colonne précise et toutes les lignes :

```
dataframe.loc[:, "nom_colonne"]
```

- ◆ Pour récupérer des tranches de valeurs consécutives avec l'attribut **loc**, la syntaxe du *slicing* est la suivante :

```
dataframe.loc['lig_1': 'lig_20':1, 'col_1': 'col_10':2]
```


Ici, on sélectionne l'ensemble des lignes comprises entre `lig_1` et `lig_20` comprises. On sélectionne aussi l'ensemble des colonnes entre `col_1` et `col_10` comprise, avec un pas de 2.

Exemple pratique : Soit le **dataframe** suivant :

	titre	genre	année	réalisateur
0	Titanic	Romance	1997	James Cameron
1	Avatar	Science-fiction	2009	James Cameron
2	Inception	Science-fiction	2010	Christopher Nolan
3	Avengers	Action	2012	Joss Whedon
4	Interstellar	Science-fiction	2014	Christopher Nolan
5	The Dark Knight	Action	2008	Christopher Nolan
6	The Avengers	Action	2012	Joss Whedon
7	Dunkirk	Drame	2017	Christopher Nolan
8	The Social Network	Drame	2010	David Fincher
9	The Dark Knight Rises	Action	2012	Christopher Nolan
10	Interstellar	Science-fiction	2014	Christopher Nolan
11	Inception	Science-fiction	2010	Christopher Nolan

Sachant la colonne « réalisateur » est définie comme **index** (on utilise l'identité du réalisateur comme étiquette pour indexer les lignes), voici ce qu'affichent les instructions suivantes :

→ `print(df.loc['James Cameron'])` affiche :

	titre	genre	année
réalisateur			
James Cameron	Titanic	Romance	1997
James Cameron	Avatar	Science-fiction	2009

→ `print(df.loc['James Cameron', ['titre', 'genre']])` affiche :

	titre	genre
réalisateur		
James Cameron	Titanic	Romance
James Cameron	Avatar	Science-fiction

→ `print(df.loc[['James Cameron', 'Joss Whedon'], ['titre', 'genre']])` affiche :

	titre	genre
réalisateur		
James Cameron	Titanic	Romance
James Cameron	Avatar	Science-fiction
Joss Whedon	Avengers	Action
Joss Whedon	The Avengers	Action

→ **Indexation et slicing avec l'attribut `iloc`** : `iloc` permet de sélectionner des valeurs selon leurs positions dans les lignes et les colonnes.

◆ Pour sélectionner une valeur à une ligne et une colonne précise :

`dataframe.iloc[26,7]`

- ◆ Pour sélectionner des valeurs à plusieurs lignes et colonnes, il faudra donner une liste des positions de lignes et une liste de positions de colonnes :

```
dataframe.iloc[[26, 65], [7,12]]
```

- ◆ Pour sélectionner une ligne et toutes les colonnes :

```
dataframe.iloc[26,:]
```

- ◆ Pour sélectionner plusieurs lignes et toutes les colonnes, il faut donner une liste de positions de lignes : `dataframe.iloc[[26, 65],:]`

- ◆ Pour sélectionner toutes les lignes et une colonne précise :

```
dataframe.iloc[:,7]
```

- ◆ Il est possible de faire du slicing pour sélectionner des tranches de lignes et de colonnes : `dataframe.iloc[0:1, 0:10]`. Ici, on sélectionne la première ligne et les dix premières colonnes du dataframe. Contrairement au slicing avec `loc`, ici le nombre qu'on donne à l'option **end** n'est pas compris dans la sélection.

→ **Indexation avec une expression booléenne** : Pour faire de l'indexation avec une expression booléenne, tout comme pour les séries, il faut utiliser les opérateurs de comparaison (>, <, !=, >=, <=) et les opérateurs bitwise &, | et ~ (de négation). Avec les **dataframes**, on spécifiera la colonne sur laquelle on souhaite appliquer l'expression booléenne. Par exemple :

```
→ df[df['col1'] >= 10]
```

```
→ df[(df ['col1'] == 10) & (df['col3'] == 25)]
```

```
→ df[~(df['col1'] >= 10)]
```

Par exemple, si on veut sélectionner les films du genre « Action » et dont l'année de sortie est postérieure à 2010, on a :

```
df[(df['genre'] == "Action" ) & (df['année'] > 2010 )]
```

On aura comme résultat :

	titre	genre	année
réalisateur			
Joss Whedon	Avengers	Action	2012
Joss Whedon	The Avengers	Action	2012
Christopher Nolan	The Dark Knight Rises	Action	2012

À vous de jouer : Indexation de données-Partie C

- 6- Afficher les noms des 5 derniers pays en utilisant l'opérateur d'indexation [].
- 7- Utiliser la chaîne d'attributs pour obtenir des informations sur le type de données stockées dans la colonne **happiness_score**.
- 8- Afficher les noms des 5 premiers pays, leur score de bonheur et l'année (dans cet ordre). Penser à une manière plus élégante pouvant améliorer la lisibilité du code.
- 9- Afficher le résultat après avoir sélectionné les données de :
- la première ligne du DataFrame **df_bonheur**.
 - l'avant dernière ligne.
 - de la 2^e ligne et la 6^e colonne.
 - la 1^{re} valeur de la colonne du score du bonheur.
 - le nom du quatrième pays.

10- **set_index(...)** dans pandas est utilisée pour définir une ou plusieurs colonnes d'un DataFrame comme nouvel index. Elle prend en entrée le ou les noms de colonnes qu'on souhaite utiliser comme index, et modifie le **DataFrame** en remplaçant l'index existant (par défaut, l'index numérique de 0 à n-1) par les valeurs de la ou des colonnes spécifiées. Voici deux manières pour l'appliquer :

```
df = df.set_index(nom_col) ou df.set_index(nom_col, inplace=True)
```

Définir comme nouvel index pour **df_bonheur**, la colonne « **country_name** ». Afficher les 5 premières lignes puis toutes les lignes relatives aux pays « France » et « Allemagne ».

Ajout, suppression et modification sur un dataframe

→ Pour ajouter une colonne à un dataframe, on a deux méthodes :

- ◆ La première méthode permet d'ajouter une colonne à partir d'une liste ou d'une série, cette colonne s'ajoutant à la fin du **dataframe** :

```
df['nouvelle_col'] = [val1, val2, ...]
```

```
df['nouvelle colonne'] = ma_serie
```

Attention, il faut que le nombre de valeurs dans la liste ou dans la série soit de même taille que le nombre de lignes dans **df**.

Si on veut ajouter une colonne contenant la même valeur à chaque ligne, il est alors possible de donner une liste contenant une valeur unique et cette valeur sera alors la même dans toute la colonne :

```
df['nouvelle_col']=[0]
```

♦ La deuxième méthode d'ajout de colonne consiste à utiliser la méthode **insert** de Pandas. L'avantage de cette méthode est qu'on peut choisir la position à laquelle insérer la nouvelle colonne, plutôt que de l'ajouter à la fin du **dataframe** :

```
df.insert (0, 'Nom_col', [1, 2, 3])
```

```
df.insert (0, 'Nom_col', ma_serie)
```

Cette commande ajoute une colonne nommée **Nom_col**, comme première colonne du **dataframe** (à la position 0) Cette colonne contient les valeurs 1, 2, 3 ou, dans la deuxième syntaxe, les valeurs contenues dans l'objet **ma_serie** de type Series.

→ Pour ajouter une ligne à un dataframe :

Pour ajouter une ou plusieurs lignes, on utilise la méthode **append()** rencontrée sur les objets de classe **Series**. Cette méthode n'effectue pas une modification sur le **dataframe** mais crée une copie de ce dernier contenant les nouvelles données ajoutées et renvoie un nouveau. Voici la syntaxe d'ajout d'une ligne :

```
nv_df=df.append(pd.Series([val1,val2,...], index=df.columns),  
ignore_index=True)
```

L'objet de type **Series** contient le même nombre de valeurs que le nombre de colonnes dans le **dataframe** auquel on souhaite ajouter une ligne. La fonction **pd.Series()** permet de créer la série à la volée, en lui donnant une liste de valeurs ainsi que les index de chacune de ces valeurs. Il faut que les index de ces valeurs correspondent aux noms de colonnes du **dataframe**; on donne donc à l'option **index** les noms de colonnes grâce à l'attribut **columns** qu'on applique sur le dataframe nommé **df**.

Par défaut, l'option **ignore_index** est à **False**, et cela précise à **append()** de garder les index de lignes du **dataframe** intacts et d'utiliser le nom de la série comme index de la nouvelle ligne ajoutée dans le **dataframe**. Le choix de cette option est plutôt simple : si les index du dataframe correspondent à des nombres incrémentés, il faut définir une série sans nom au sein de la méthode **append()** et utiliser l'option **ignore_index=True**. En revanche, si on donne un nom à la série créée dans **append()**, grâce à l'option **Name**, et on souhaite que ce nom précisément

apparaisse comme étiquette de la nouvelle ligne ajoutée, alors on met l'option `ignore_index=False`.

Pour **ajouter plusieurs lignes**, il suffit de donner une liste de séries à la méthode `append()`.

Exemple pratique : Soit le **dataframe** suivant :

	titre	genre	année	réalisateur
0	Titanic	Romance	1997	James Cameron
1	Avatar	Science-fiction	2009	James Cameron
2	Inception	Science-fiction	2010	Christopher Nolan
3	Avengers	Action	2012	Joss Whedon
4	Interstellar	Science-fiction	2014	Christopher Nolan
5	The Dark Knight	Action	2008	Christopher Nolan
6	The Avengers	Action	2012	Joss Whedon
7	Dunkirk	Drame	2017	Christopher Nolan
8	The Social Network	Drame	2010	David Fincher
9	The Dark Knight Rises	Action	2012	Christopher Nolan
10	Interstellar	Science-fiction	2014	Christopher Nolan
11	Inception	Science-fiction	2010	Christopher Nolan

On va ajouter deux nouvelles lignes :

```
# Nouvelles lignes sous forme de pd.Series
nv_l1 = pd.Series(['Jurassic Park', 'Action', 1993, 'Steven Spielberg'], index=df.columns)
nv_l2 = pd.Series(['Pulp Fiction', 'Drame', 1994, 'Quentin Tarantino'], index=df.columns)

# Ajouter les nouvelles lignes au DataFrame
nv_df = df.append([nv_l1, nv_l2], ignore_index=True)

#ou
nv_df2 = pd.concat([df, nv_l1, nv_l2], ignore_index=True)
```

On obtient :

	titre	genre	année	réalisateur
0	Titanic	Romance	1997	James Cameron
1	Avatar	Science-fiction	2009	James Cameron
2	Inception	Science-fiction	2010	Christopher Nolan
3	Avengers	Action	2012	Joss Whedon
4	Interstellar	Science-fiction	2014	Christopher Nolan
5	The Dark Knight	Action	2008	Christopher Nolan
6	The Avengers	Action	2012	Joss Whedon
7	Dunkirk	Drame	2017	Christopher Nolan
8	The Social Network	Drame	2010	David Fincher
9	The Dark Knight Rises	Action	2012	Christopher Nolan
10	Interstellar	Science-fiction	2014	Christopher Nolan
11	Inception	Science-fiction	2010	Christopher Nolan
12	Jurassic Park	Action	1993	Steven Spielberg
13	Pulp Fiction	Drame	1994	Quentin Tarantino

3. Supprimer des lignes ou colonnes d'un dataframe

Pour la suppression d'éléments dans un **dataframe**, que ce soit des lignes ou des colonnes, il faut utiliser la méthode **drop()**.

→ Pour les colonnes, il faut utiliser le ou les noms de colonnes à supprimer en spécifier l'axe d'intérêt, ici l'axe des colonnes, donc 1. Voici la syntaxe :

```
nv_df=df.drop(["col_1","col_2", "col_3"], axis=1)
```

→ Pour supprimer des lignes, on utilise soit leur position, soit leur étiquette d'index.

➤ Pour supprimer des lignes via leur étiquette, on donnera ces étiquettes sous forme de liste en spécifiant l'axe des lignes, ici 0. Voici la syntaxe :

```
nv_df=df.drop(["lig_1","lig_2", "lig_3"], axis=0)
```

➤ Pour supprimer des lignes via leurs positions, on utilise la méthode **index()** au sein de la méthode **drop()**, avec la syntaxe suivante :

```
nv_df=df.drop(df.index[[2,3]], axis=0)
```

La méthode **index()** permet de récupérer les étiquettes des index en donnant leur position. Ainsi, dans cet exemple, on supprime les lignes aux positions 2 et 3 du dataframe (on récupère leurs étiquettes qu'on donne à la méthode **drop()**). La méthode **drop()** ainsi que son option **inplace** vue sur les séries permet de renvoyer une copie du **dataframe** avec les éléments supprimés si **inplace=True**.

Exemple pratique : Pour supprimer le « genre » des films du **dataframe** précédent, voici la syntaxe :

```
nv_df2 = nv_df.drop("genre", axis=1)
```

Pour supprimer les lignes d'index 3 et 8, on a la syntaxe :

```
nv_df2 = nv_df.drop(df.index[[3,8]], axis=0)
```

Le **dataframe** résultat après cette dernière suppression est :

	titre	genre	année	réalisateur
0	Titanic	Romance	1997	James Cameron
1	Avatar	Science-fiction	2009	James Cameron
2	Inception	Science-fiction	2010	Christopher Nolan
4	Interstellar	Science-fiction	2014	Christopher Nolan
5	The Dark Knight	Action	2008	Christopher Nolan
6	The Avengers	Action	2012	Joss Whedon
7	Dunkirk	Drame	2017	Christopher Nolan
9	The Dark Knight Rises	Action	2012	Christopher Nolan
10	Interstellar	Science-fiction	2014	Christopher Nolan
11	Inception	Science-fiction	2010	Christopher Nolan
12	Jurassic Park	Action	1993	Steven Spielberg
13	Pulp Fiction	Drame	1994	Quentin Tarantino

4. Modifier des valeurs dans un dataframe

Comme pour les séries, on peut utiliser les attributs `loc` (en donnant les étiquettes) et `iloc` (en donnant les positions) ainsi que le symbole d'affectation `=` pour modifier les données. La syntaxe est la suivante :

```
df.loc("nom_lig", "nom_col"] = valeur
df.iloc(position_lig, position_col] = valeur
```

Il est aussi possible de modifier les valeurs en donnant une expression booléenne entre crochets à l'attribut `loc`. La syntaxe est la suivante :

```
df.loc[df ["ma_col"] > 1000, ["ma_col"]] = 1000
```

Dans cette syntaxe d'exemple, on récupère l'ensemble des lignes pour lesquelles la valeur de `ma_col > 1000`, puis on sélectionne `ma_col` et on remplace donc les valeurs strictement supérieures à 1000 dans `ma_col` par 1000.

Enfin, une méthode Pandas est dédiée au remplacement de valeurs. Cette méthode s'appelle `replace()` et elle est vraiment puissante, notamment lorsqu'on veut remplacer un ensemble de mêmes valeurs par une autre. Sa syntaxe est la suivante:

```
df.replace(to_replace="mon mot", value="mon mot_2")
```

5. Nettoyage et préparation des données avec Pandas

Avant de pouvoir explorer les données contenues dans un tableau, il faut le nettoyer, le préparer. Il y a deux étapes importantes, avant toute chose: prendre en compte les valeurs manquantes, et les traiter au besoin, mais aussi les valeurs dupliquées, et les supprimer si besoin.

→ **ÉTAPE 1 : Gestion des données manquantes** : Il existe différentes façons de représenter les données manquantes en Python : `np.NaN`, `pd.NA` et `None` (ne permet pas de calcul numérique). Il est important de prendre connaissance de la proportion des valeurs manquantes, voire de les traiter.

La manière la plus simple d'identifier les données manquantes est d'utiliser la méthode `df.isna()`. La méthode `isna()` va remplacer l'ensemble des valeurs du dataframe par `True` (si la valeur est `NaN`) ou `False` (si la valeur est différente de `NaN`). Puis on utilise `df.isna().sum()` où la méthode `sum()` va compter le nombre de valeurs à `True` par colonne du dataframe, ce qui permet de quantifier la présence des valeurs manquantes, ou utiliser `df[df.isna().any(axis=1)]` où `any(axis=1)` sélectionne les lignes contenant des valeurs `NaN`.

On peut également utiliser `.info()` ou `.value_counts(dropna=False)` (compter le nombre de valeurs dans une colonne incluant les données manquantes)

Il existe plusieurs façons de gérer les données manquantes :

- **Conserver les données manquantes telles quelles** (certaines métriques : `mean()`, `max()`, ... les ignorent).
- **Supprimer toute une ligne ou colonne contenant des données manquantes avec `.dropna()`** :

-`df.dropna()` : supprime toute ligne ayant au moins une valeurs **NaN**.

-`df.dropna(how="any")`, l'option `how="any"` permet de spécifier à la méthode `dropna()` que si une seule valeur de la ligne est égale à **NaN**, alors il faut supprimer cette ligne. Il est aussi possible de supprimer les lignes si l'ensemble des valeurs sont à **NaN**, c'est-à-dire si la valeur **NaN** est présente dans l'ensemble des colonnes, grâce à l'option `how="all"`.

-`df.dropna(thresh=n)`, `thresh` permet de spécifier un nombre minimal de valeurs non manquantes requises pour qu'une ligne soit conservée.

-`df.dropna(subset=["nom_col"])` supprime les lignes avec des valeurs **NaN** dans une colonne spécifiée.

-utiliser `df.notna()` plutôt que `df.dropna()` pour conserver les données non manquantes au lieu de les supprimer. Sa syntaxe est : `df[df['ma_col'].notna()]`. En appliquant la méthode `notna()` sur la colonne qui nous intéresse, celle-ci retourne un objet de type **Series** contenant un ensemble de booléens, avec **True** (si la valeur n'est pas manquante) et **False** (si la valeur est manquante). En donnant cette série de booléens entre crochets à l'objet `df`, cela permet de ne sélectionner que les lignes dont les valeurs sont différentes de **NaN** dans la colonne `ma_col`.

Exemple pratique : Soit le **dataframe** `df_legumes` suivant :

	aliment	type	couleur	prix
0	carotte	légume	orange	1.5
1	pomme	fruit	rouge	0.5
2	brocoli	légume	vert	1.0
3	banane	fruit	NaN	NaN
4	NaN	NaN	jaune	0.3
5	courgette	légume	vert	0.8

Le **dataframe** obtenu après l'instruction `df_legumes.dropna(how="any")` est :

	aliment	type	couleur	prix
0	carotte	légume	orange	1.5
1	pomme	fruit	rouge	0.5
2	brocoli	légume	vert	1.0
5	courgette	légume	vert	0.8

Remarque importante : Il faut noter que l'utilisation de `.dropna()` ou `.notna()` pour supprimer les lignes avec des données manquantes ne modifie pas de manière permanente le DataFrame d'origine. Il faut soit sauvegarder la sortie dans un nouveau DataFrame (ou dans le même), soit définir l'argument `inplace=True` pour que les modifications prennent effet directement dans le DataFrame d'origine.

- **Imputer les données numériques manquantes par un 0 ou une valeur de substitution comme la moyenne, le mode, etc :** pour ne pas supprimer les lignes contenant des valeurs manquantes, sous peine de perdre de l'information, mais qu'on souhaite toutefois ne plus avoir de valeurs manquantes dans le tableau. Il est alors possible d'utiliser une méthode appelée `fillna()`, qui va remplacer les valeurs **NaN** parce qu'on lui demande. Par exemple, si on souhaite remplacer l'ensemble des **NaN** par 0, la syntaxe est la suivante:

- `dataframe.fillna(value=0, inplace=True)`

On peut aussi remplacer les valeurs **NaN** par la moyenne de la colonne dans laquelle la valeur se trouve, avec la syntaxe suivante :

```
df[['col_1']] = df [['col_1']].fillna(df[['col_1']].median())
```

- **Résoudre les données manquantes en fonction de l'expertise dans le domaine :** Une alternative à l'imputation des données manquantes est de les résoudre en utilisant une expertise dans le domaine. On peut utiliser `.loc[]` pour mettre à jour une valeur spécifique dans le DataFrame en fonction de la connaissance du domaine. Cela permet de remplir les valeurs manquantes de manière plus contextuelle et adaptée à la nature des données.

→ **ÉTAPE 2 : Gestion des données dupliquées:** Lorsqu'on commence à travailler avec un tableau contenant beaucoup de données, il ne faut pas seulement prendre en compte la gestion des valeurs manquantes, mais aussi celle des informations dupliquées. En effet, cela peut fausser les conclusions que l'on fait sur un jeu de données si certaines informations sont dupliquées de nombreuses fois. Des informations dupliquées peuvent être des lignes entières dupliquées.

- ♦ **Détection/récupération des lignes dupliquées :** La syntaxe générale pour détecter les lignes entièrement dupliquées dans un tableau est la suivante: `df[df.duplicated(keep=False)]`. Ici, la méthode `duplicated()` retourne une série contenant des booléens permettant de dire si la ligne est dupliquée (**True**) ou non (**False**). Ainsi, la syntaxe présentée retournera l'ensemble des lignes dupliquées, sauf la première occurrence. Pour compter le nombre de

lignes dupliquées dans le tableau, il suffit d'utiliser l'attribut **shape** qui renvoie la dimension d'un **dataframe** : (nombre de lignes, nombre de colonnes) :

```
df[df.duplicated() ].shape
```

ou **df.duplicated().sum()** pour calculer le nombre de lignes dupliquées.

♦ **Suppression des lignes dupliquées** : Pour nettoyer le **dataframe**, on supprime l'ensemble des lignes dupliquées, en gardant toujours une occurrence de celles-ci. La syntaxe générale pour faire cette étape de nettoyage est la suivante : **df.drop_duplicates(inplace=True)**

La méthode **drop_duplicates()** va supprimer les lignes dupliquées en prenant en compte l'ensemble des colonnes et en gardant la première occurrence de chaque ligne dupliquée dans le **dataframe** qu'elle retourne.

À vous de jouer : Nettoyage de données-Partie D

L'objectif du nettoyage des données est de transformer les données brutes en un format prêt pour l'analyse. Même s'il existe des outils automatisés disponibles, effectuer certaines étapes de nettoyage des données manuellement offre une bonne opportunité de commencer à comprendre et de bien appréhender vos données. Cela comprend :

- La conversion des colonnes aux types de données corrects pour l'analyse.
- La gestion des problèmes de données qui pourraient affecter les résultats de l'analyse.
- La création de nouvelles colonnes à partir de colonnes existantes qui sont utiles pour l'analyse.

L'ordre dans lequel ces tâches de nettoyage des données sont effectuées, peut varier en fonction de l'ensemble de données.

Étape N°1 : Types de données

En utilisant Pandas pour lire des données, les colonnes se voient automatiquement attribuer un type de données.

Conseils :

- Utiliser l'attribut **.dtypes** pour afficher le type de données de chaque colonne du **DataFrame**. Il faut noter que parfois, les colonnes numériques (**int**, **float**) et les colonnes de date et d'heure (**datetime**) ne sont pas reconnues correctement par Pandas et sont importées en tant que colonnes de texte (**object**).

- Utiliser la méthode `.info()` en tant qu'alternative pour afficher des informations supplémentaires en plus des types de données.
- Utiliser `pd.to_datetime()` pour convertir une colonne de type `object` en une colonne de type `datetime`.
- Utiliser `pd.to_numeric()` pour convertir une colonne de type `object` en colonne numérique. Pour supprimer les caractères non numériques (\$, %, etc.), utiliser `str.replace()`.

Une alternative consiste à utiliser `Series.astype()` pour convertir en types de données plus spécifiques tels que `int`, `float`, `object` et `bool`, mais `pd.to_numeric()` peut gérer les valeurs manquantes (`NaN`), contrairement à `Series.astype()`.

11- Le fichier « `donnees_sur_reveils.xlsx` », contient les données d'une enquête effectuée auprès de quelques milliers de clients ayant acheté des réveils.

Lire les données sous Python et les stocker dans le dataframe `df_reveils`. Afficher le type de données de chaque colonne. Repérer la ou les colonnes qui ont un type de données inapproprié et effectuer les conversions nécessaires pour la/les ajuster.

Étape N°2 : Valeurs manquantes

12- Trouver les données manquantes et les traiter dans `df_reveils`.

Étape N°3 : Textes incohérents et Typos(fautes de frappe)

Les textes incohérents et les fautes de frappe (typos) dans un ensemble de données sont représentés par des valeurs qui sont soit incorrectes de quelques chiffres ou caractères, ou incompatibles avec le reste d'une colonne. Bien qu'il n'existe pas de méthode spécifique pour les identifier, on peut suivre les deux approches suivantes pour vérifier une colonne en fonction de son type de données :

- pour les données catégorielles : examiner les valeurs uniques dans la colonne (par exemple : RG, WI, Wimbledon, ici WI et Wimbledon désigne le même tournoi)
- pour les données numériques : consulter les statistiques descriptives (par exemple, l'intervalle [`min()` , `max()`]).

On peut utiliser les méthodes suivantes pour résoudre ces problèmes :

- `.loc[]` pour mettre à jour une valeur à un emplacement spécifique selon le domaine d'expertise.
- `np.where(condition, if_true, if_false)` pour mettre à jour des valeurs dans une colonne en fonction d'une condition logique (par

exemple changer les valeurs de Wimbledon par WI : `df["col"] = np.where(df["col"] == "Wimbledon", "WI", df["col"])`.

→ `.map(ensemble)` pour mapper un ensemble de valeurs vers un autre ensemble de valeurs (par exemple : `df["col"].map({"Wimbledon" : "WI", "WI", "WI"})`).

→ Méthodes de chaîne de caractères telles que `str.lower()`, `str.strip()` et `str.replace()` pour nettoyer les données textuelles.

13- Trouver les données incohérentes et les typos et les traiter dans `df_reveils`.

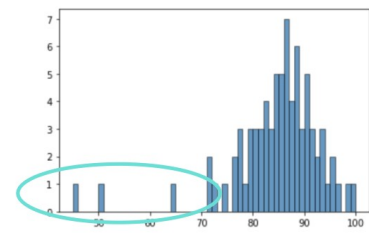
Étape N°4 : Données dupliquées

14- Trouver les données dupliquées et les traiter dans `df_reveils`.

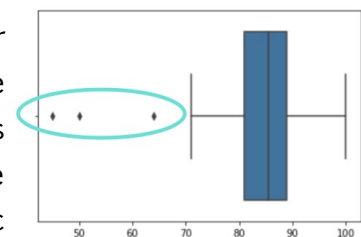
Étape N°5 : Données aberrantes

Une valeur aberrante est une valeur dans un ensemble de données qui est beaucoup plus grande ou plus petite que les autres. On peut identifier ce type de valeurs de différentes manières en utilisant des graphiques et des statistiques :

→ Les histogrammes sont utilisés pour visualiser la distribution (ou la forme) d'une colonne numérique. Ils aident à identifier les valeurs aberrantes en montrant quelles valeurs se situent en dehors de la plage normale. Pour tracer des histogrammes, on importe la bibliothèque `seaborn` et on utilise `seaborn.histplot(df)`.



→ Les boîtes à moustaches (boxplots) sont utilisées pour visualiser les statistiques descriptives d'une colonne numérique. Elles représentent automatiquement les valeurs aberrantes sous forme de points en dehors de la plage de données minimale et maximale. On les trace avec `seaborn.boxplot(x=df)`.



→ L'écart-type (std) est une mesure de la dispersion d'un ensemble de données par rapport à la moyenne. Les valeurs situées à au moins 3 écarts-types de la moyenne ($\text{valeur} < \text{moyenne} - 3 \times \text{std}$ ou $\text{val} > \text{moyenne} + 3 \times \text{std}$) sont considérées comme des valeurs aberrantes.

Il existe plusieurs façons de gérer les valeurs aberrantes : les conserver, supprimer la ligne ou la colonne la contenant, les remplacer par NaN ou la moyenne ou par des valeurs relatives au domaine d'expertise.

15- Trouver les valeurs aberrantes et les traiter dans **df_reveils**. Explorer rapidement le DataFrame mis à jour. Comment se présentent les choses après avoir traité les problèmes de données par rapport au DataFrame d'origine ?

Étape N°6 : Création de nouvelles colonnes

16-Après avoir nettoyé les types de données et résolu les problèmes, on peut toujours ne pas disposer exactement des données dont on a besoin. Dans ce cas, il faut créer de nouvelles colonnes à partir de données existantes pour faciliter l'analyse. Cela peut être fait de plusieurs manières, en fonction du type de données qu'on a :

→ Pour les colonnes numériques, on peut calculer des pourcentages, appliquer des calculs conditionnels, etc.

→ Pour les colonnes de date et d'heure, on peut extraire des composants de date et d'heure, appliquer des calculs de date et d'heure, etc.

Par exemple, utiliser **df.composante**, telle que **composante** est égale à : **Date** pour la date sans heure, **year** pour l'année, **month** pour le mois (1-12), **day** pour le jour, **dayofweek** (0 pour lundi et 6 pour dimanche), **time** (heure sans date), **hour** pour l'heure (0-23) et **minute** et **second** pour minute et seconde (0-59).

Pour effectuer des calculs de date et d'heure entre colonnes, on utilise des opérations arithmétiques de base. On peut également **pd.to_timedelta(valeur, unit)** pour ajouter ou soustraire une période de temps spécifique. Par exemple : **valeur =3 et unit="W"** veut dire qu'on veut ajouter deux semaines, **unit** peut aussi prendre les valeurs suivantes : **D** pour jour, **H** pour heure, **T** pour minute et **S** pour seconde).

→ Pour les colonnes de texte, on peut extraire du texte, le diviser en plusieurs colonnes, rechercher des motifs, etc. Par exemple : **.str[deb : fin]** extrait un texte dont les positions de ses lettres vont de **deb** à **fin-1**, **.str.split(delimiteur)** pour séparer une colonne en plusieurs colonnes en utilisant un délimiteur, **.str.contains(pattern)** pour chercher un motif dans une chaîne.

16- Lire le fichier « **donnees_ventes_stylos.xlsx** » et les sauvegarder dans un dataframe nommé **df_stylos**. Afficher le type de données de chaque colonne.

Créer les colonnes suivantes :

→ « **Depense Totale** » qui inclut à la fois le coût du stylo et les frais d'expédition pour chaque vente.

- « **Livraison gratuite** » qui indique "Oui" si la vente inclut la livraison gratuite, et "Non" sinon."
- « **Delai Livraison** » qui contient le nombre de jours entre la date d'achat et la date de livraison pour chaque vente. Calculer et afficher la moyenne des jours entre l'achat et la livraison.
- « **Nom Utilisateur** », « **Contenu Son Avis** » et « **Fuite** » qui contiennent respectivement le nom de l'utilisateur, le contenu de son avis et oui ou non pour les avis mentionnant fuir.

6. Quelques méthodes de manipulation d'un dataframe (exploration)

→ **Les méthodes de tri** : Il peut être utile de trier les valeurs contenues dans une ou plusieurs colonnes d'un **dataframe**, ou encore de trier les index ou les noms de colonnes de celui-ci. Ainsi, deux méthodes existent pour trier un **dataframe**: soit par ses valeurs, soit par ses index.

- ◆ Syntaxe pour trier les valeurs d'un **dataframe**, avec la méthode **sort_values()** :

```
df.sort_values ("col_1", axis=0, na_position="last", ascending=False)
```

Ici, on précise le nom de la colonne sur laquelle on souhaite trier les valeurs. On pourrait aussi donner une liste de noms de colonnes si on souhaitait trier sur plusieurs colonnes. Ensuite, on précise l'axe sur lequel on veut trier, ici l'axe des index (les lignes), puis on précise qu'on souhaite que les valeurs manquantes (**NaN**) soient mises à la fin du tableau trié grâce à l'option **na_position="last"**.

- ◆ Syntaxe pour trier les index d'un dataframe (étiquettes de lignes ou noms de colonnes) avec la méthode **sort_index()** :

```
df.sort_index(axis=0)
```

```
df.sort_index(axis=1)
```

La méthode **sort_index()** permet de trier soit sur l'axe 0 (trier les étiquettes des lignes), soit sur l'axe 1 (trier les noms de colonnes).

→ **Les opérations groupby** : Il est possible de grouper un **dataframe** sur une ou plusieurs colonnes puis d'appliquer des fonctions dessus, de manière très simple, grâce à la méthode **groupby()** de Pandas, qui signifie littéralement en français "grouper par".

- ◆ **Grouper par colonne** : la syntaxe est la suivante :

```
df.groupby("nom_colonne")
```

- ◆ Il est aussi possible d'appliquer directement une fonction de calcul après la méthode `groupby()`. Par exemple, on pourrait les valeurs de la colonne `nom_colonne` comme suit : `df.groupby("nom_colonne").sum()`.

- ◆ **Grouper sur plusieurs colonnes** : la syntaxe est la suivante :

```
df.groupby(["col_1", "col_2"]).sum()
```

→ **Appliquer plusieurs fonctions avec la méthode `groupby` et `aggregate`** :
Il est possible d'appliquer plusieurs fonctions de calcul plutôt qu'une seule, grâce à la méthode `aggregate`, dont l'alias est : `agg()`. Voici la syntaxe générale :

```
dataframe.groupby('col_1').agg({'col_2': ['median', 'mean'],  
                                'col_3': ['sum', 'mean']})
```

ou

```
df.groupby(col)[cols].aggregation()
```

`col` utilisée pour le groupement (on obtient des valeurs uniques), `cols` sont les colonnes sur lesquelles on applique les calculs, `aggregation()` représente les calculs à appliquer pour chaque groupe (ceux-ci deviennent les nouvelles valeurs dans la sortie) (`mean()`, `sum()`, `min()`, `max()`, `count()`, `nunique()`)

Ainsi, il suffit de donner la liste des fonctions qu'on souhaite appliquer au `dataframe` groupé par modalité de la colonne. Par exemple, on applique la moyenne et la médiane sur la seconde colonne, et la somme et la moyenne sur la troisième colonne.

Exemple pratique : Soit le `dataframe df_villes` suivant :

	ville	région	population
0	Paris	Île-de-France	2140526
1	Marseille	Provence-Alpes-Côte d'Azur	863310
2	Lyon	Auvergne-Rhône-Alpes	515695
3	Toulouse	Occitanie	471941
4	Bordeaux	Nouvelle-Aquitaine	257804
5	Nantes	Pays de la Loire	306694
6	Nice	Provence-Alpes-Côte d'Azur	342637
7	Strasbourg	Grand Est	280966
8	Montpellier	Occitanie	277639
9	Rennes	Bretagne	216815

Après avoir groupé par région et effectué une agrégation sur la population :

```
df_villes.groupby('région').agg({'population': ['sum', 'mean',  
                                                'min', 'max']})
```

On obtient :

Agrégation par région :				
région	population sum	mean	min	max
Auvergne-Rhône-Alpes	515695	515695.0	515695	515695
Bretagne	216815	216815.0	216815	216815
Grand Est	280966	280966.0	280966	280966
Nouvelle-Aquitaine	257804	257804.0	257804	257804
Occitanie	749580	374790.0	277639	471941
Pays de la Loire	306694	306694.0	306694	306694
Provence-Alpes-Côte d'Azur	1205947	602973.5	342637	863310
Île-de-France	2140526	2140526.0	2140526	2140526

Après avoir groupé par région et par ville et effectué une agrégation sur la population et la taille des villes :

```
df_villes.groupby(['région', 'ville']).agg({'population': ['sum', 'mean'], 'ville': 'count'})
```

On obtient :

Agrégation par région et ville :				
région	ville	population sum	ville mean	count
Auvergne-Rhône-Alpes	Lyon	515695	515695.0	1
Bretagne	Rennes	216815	216815.0	1
Grand Est	Strasbourg	280966	280966.0	1
Nouvelle-Aquitaine	Bordeaux	257804	257804.0	1
Occitanie	Montpellier	277639	277639.0	1
	Toulouse	471941	471941.0	1
Pays de la Loire	Nantes	306694	306694.0	1
Provence-Alpes-Côte d'Azur	Marseille	863310	863310.0	1
	Nice	342637	342637.0	1
Île-de-France	Paris	2140526	2140526.0	1

→ Appliquer une fonction à un dataframe avec la méthode **apply** :

La méthode **apply()** permet d'appliquer (comment son nom l'indique) une fonction à un axe du **dataframe**, c'est-à-dire qu'on peut appliquer la fonction sur chaque colonne (**axis = 0**) ou sur chaque ligne (**axis = 1**) du **dataframe**. Par défaut, la méthode **apply()** appliquera la fonction par colonne. Sa syntaxe générale est : **df.apply (fonction, axis=0)**.

Exemple pratique : Soit le **dataframe** **df_ventes** suivant :

	mois	ventes
0	janvier	5000
1	février	6000
2	mars	4500
3	avril	7000
4	mai	5500

On veut ajouter une colonne « bonus » à ce **dataframe**. Si les ventes sont supérieures à 5000, le bonus est calculé étant 10 % des ventes, sinon, il est égal à 0. On définit la fonction suivante :

```
def calculer_bonus(ventes):  
    if ventes > 5000:  
        return 0.1 * ventes  
    else:  
        return 0
```

L'instruction permettant d'ajouter la colonne « **bonus** » est :

```
df_ventes['bonus'] = df_ventes['ventes'].apply(calculer_bonus)
```

On obtient :

	mois	ventes	bonus
0	janvier	5000	0.0
1	février	6000	600.0
2	mars	4500	0.0
3	avril	7000	700.0
4	mai	5500	550.0

À vous de jouer : Exploration de données-Partie E

L'équipe marketing souhaite partager les cinq pays les plus heureux des années 2010 à 2019 sur les réseaux sociaux.

- 17- Regrouper les données par pays et calculer le score de bonheur maximum pour chaque pays.
- 18- Trier les pays regroupés par score de bonheur et retourner les cinq premiers.
- 19- Regrouper les données par pays et calculer le score de bonheur moyen pour chaque pays.
- 20- Trier les pays regroupés par score de bonheur et retourner les cinq premiers.
- 21- Comparer les deux listes.

À vous de jouer : Nettoyage et Exploration de données-Partie F(Projet d'application)

Vous disposez du fichier « **Films_RottenTomatoes.csv** ». Explorer les données en filtrant, en triant, en regroupant les données et en créant de nouvelles colonnes permettant d'aider dans l'analyse selon les étapes suivantes :

→ lire les données et les sauvegarder dans le dataframe **df_films**. Afficher les 3 premiers films.

→ faire en sorte de ne laisser dans `df_films` que ces colonnes : `['titre_film', 'note', 'genre', 'date_sortie_cinema', 'duree_minutes', 'note_tomato-meter', 'nombre_notes_tomato-meter', 'note_public', 'nombre_avis_public']`

(Vous pouvez écraser le contenu du dataframe)

→ Afficher le nombre de films dans le jeu de données.

→ Filtrer les données pour inclure uniquement les films sortis en 2010 ou après. Combien de films y a-t-il dans ce nouveau jeu de données ?

→ Trouver les films les mieux notés selon à la fois les critiques (représentés par la note `note_tomato-meter`) et le grand public (représentés par la note `note_public`). Pour ce faire, vous devrez trier les films en fonction de ces deux critères.

→ Ces films les mieux notés semblent avoir très peu de critiques et de membres du public rédigeant les avis. Nous voulons nous concentrer uniquement sur les films les plus populaires. Filtrer le jeu de données des films pour n'inclure que ceux qui ont plus de 100 000 avis de la part du public (`f_populaires`). Combien de films y a-t-il dans ce jeu de données ?

Trouver les films populaires les mieux notés selon à la fois les critiques (représentés par la note `note_tomato-meter`) et le grand public (représentés par la note `note_public`).

→ Beaucoup de ces films populaires semblent avoir une classification PG ou PG-13. Combien de films populaires relèvent de chaque type de classification ? Afficher ces valeurs.

(PG : Parental Guidance Suggested, PG-13 : Parents Strongly Cautioned, R : Restricted, G : General Audience)

→ Afficher la note moyenne du public pour chaque type de classification de film. Quelle classification est la mieux notée en moyenne ?

→ Créer une colonne dans le DataFrame appelée 'Animation' et renvoyer 1 si c'est un d'animation et 0 sinon. Faire de même pour les films d'Action & Aventure et Comédie.

Astuce : utiliser `np.where` et `str.contains`.

→ Créer un tableau où chaque ligne correspond à une classification, chaque colonne à un genre et chaque valeur au **nombre de films** de cette classification et de ce genre particuliers. Quelles observations pouvez-vous en tirer ?

→ Trouver la note moyenne des critiques et du public pour un film d'animation par rapport à un film non animé. Faire de même pour les films d'Action & Aventure et les Comédies. Quelles observations pouvez-vous en tirer ?