

Python pour les statistiques

Fiche 3 : Pandas et Seaborn

M2 Modélisation statistique, 2023–2024

Table des matières

1	Ce qu’il faut savoir	1
1.1	Lecture de données et opérations basiques	1
1.1.1	Lire et écrire des données	1
1.1.2	Afficher un tableau	3
1.1.3	Modifications basiques	4
1.2	Filtrage de données	5
1.3	Opérations avancées sur les séries, str, datetime, valeurs manquantes	7
1.4	Restructurer les données	8
1.4.1	Fusion	9
1.4.2	Pivot	11
1.4.3	Agrégation	13
1.5	Graphiques	15
2	Exercices	19

1 Ce qu’il faut savoir

On trouvera beaucoup plus d’informations dans le [user guide](#) officiel et la documentation (utiliser l’outil recherche). Je vous donne quelques pistes très rapides pour bien démarrer avec Pandas.

Tout d’abord, pour importer Pandas et Seaborn, les alias standards sont :

```
import pandas as pd
import seaborn as sns
```

Note : Pandas est construit par-dessus Numpy, et Seaborn par-dessus Matplotlib, donc on pourra importer ces deux autres librairies si besoin (Numpy en particulier pourra être très utile).

1.1 Lecture de données et opérations basiques

1.1.1 Lire et écrire des données

Lire des données au format CSV avec `df = pd.read_csv(nom_fichier)`.

La variable `nom_fichier` est une chaîne de caractère, contenant soit le nom d’un fichier sur votre ordinateur, soit une URL vers un fichier en ligne. Généralement, il vaut mieux sauvegarder le fichier sur votre ordinateur pour le charger localement, sinon le script va re-télécharger le fichier en boucle, ce qui serait du gâchis.

Le nom CSV signifie *comma-separated values*, soit “valeurs séparées par des virgules” : des virgules séparent les colonnes du tableau. *Mais attention*, parfois le caractère de séparation peut être le point-virgule (c’est le cas quand on veut mettre des chaînes de caractères avec des virgules dans les données), auquel cas on aura peut-être besoin de préciser `df = pd.read_csv(nom_fichier, sep=';')`.

Pour illustrer cette fonction, on suppose que l'on dispose d'un fichier `data1.csv` contenant :

```
id,animal,taille,poids,vitesse
1,Lièvre,30.,4.4,60.
2,Tortue,19,2.5,.25
3,Lion,180,190,57.
4,Ours,270,400,40
5,Poule,30,3.5,17
```

Alors l'appel `df = pd.read_csv('data1.csv')` puis `print(df)` produira

	id	animal	taille	poids	vitesse
0	1	Lièvre	30.0	4.4	60.00
1	2	Tortue	19.0	2.5	0.25
2	3	Lion	180.0	190.0	57.00
3	4	Ours	270.0	400.0	40.00
4	5	Poule	30.0	3.5	17.00

La fonction `pd.read_csv` renvoie un objet `DataFrame` : un **tableau** contenant plusieurs **colonnes** (objets `Series`). Les colonnes ont un nom, récupéré dans la première ligne du fichier. Une première colonne est mise à part et appelée l'*index* (objet `Index`) : c'est la liste qui identifie les lignes par des étiquettes (numéros ou chaînes de caractères). Par défaut avec la fonction `pd.read_csv`, l'index n'a pas de nom et l'étiquette d'une ligne est son numéro : 0 pour la première, 1 pour la seconde, etc.

Si le fichier contient une colonne qui remplit déjà le rôle d'index (ici, la première colonne `id`), on peut le préciser avec l'argument optionnel `index_col` qui prend soit l'indice de la colonne, soit son nom. Exemple : `pd.read_csv('data1.csv', index_col='id')` renvoie le tableau suivant.

	animal	taille	poids	vitesse
id				
1	Lièvre	30.0	4.4	60.00
2	Tortue	19.0	2.5	0.25
3	Lion	180.0	190.0	57.00
4	Ours	270.0	400.0	40.00
5	Poule	30.0	3.5	17.00

et `pd.read_csv('data1.csv', index_col='animal')` renvoie

	id	taille	poids	vitesse
animal				
Lièvre	1	30.0	4.4	60.00
Tortue	2	19.0	2.5	0.25
Lion	3	180.0	190.0	57.00
Ours	4	270.0	400.0	40.00
Poule	5	30.0	3.5	17.00

Changer l'index d'un tableau existant. Si l'on veut **récupérer un index numérique standard** à partir de l'un de ces deux derniers tableaux, on peut utiliser la méthode `df = df.reset_index()`. La colonne utilisée pour l'index redeviendra une colonne "normale".

À partir de n'importe quel tableau `df`, on peut **définir une colonne comme étant l'index** en utilisant `df.set_index('nom_de_la_colonne')`. **Attention**, l'index existant sera supprimé. Un exemple avec

le tableau df suivant, où l'index est nommé animal :

	id	taille	poids	vitesse
animal				
Lièvre	1	30.0	4.4	60.00
Tortue	2	19.0	2.5	0.25

(a) `df.set_index('id')` renvoie :

	taille	poids	vitesse
id			
1	30.0	4.4	60.00
2	19.0	2.5	0.25

(b) `df.reset_index().set_index('id')` renvoie :

	animal	taille	poids	vitesse
id				
1	Lièvre	30.0	4.4	60.00
2	Tortue	19.0	2.5	0.25

Enregistrer dans un fichier. Enfin pour enregistrer un tableau dans un fichier, on utilise la méthode `df.to_csv('mon_fichier.csv')`. Pour ignorer l'index, si celui-ci n'a pas d'intérêt, on rajoute l'argument optionnel `index=False`.

1.1.2 Afficher un tableau

- Pour les grands tableaux, il peut être intéressant d'afficher seulement les **premières lignes** du tableau avec `df.head()` ou les **dernières lignes** avec `df.tail()`.
- On peut afficher la **liste des colonnes** avec `df.columns`.
- Connaître les **types des données** stockées dans chaque colonnes : `df.dtypes`.
- On peut, si nécessaire, récupérer la matrice de toutes les valeurs du tableau (sans les noms des colonnes et de l'index) avec `df.values`. Note : ceci renvoie un array Numpy.
- Comme les arrays Numpy, `df.shape` renvoie les **dimensions**, c'est-à-dire le tuple (nb de lignes, nb de colonnes) du tableau (l'index n'est pas compris dans les colonnes).
- Accès à une colonne du tableau grâce à `df[nom_colonne]`. Le type d'une seule colonne est Series. Une série est à un tableau ce qu'un vecteur colonne Numpy est à une matrice. Mais dans une série, *l'index est conservé*. C'est très important car Pandas "aligne" et manipule les différentes séries en se basant sur leurs index (on en parlera plus tard).

Exemple avec le tableau `df = pd.read_csv('data1.csv', index_col='animal')` : le code `print(df['taille'])` affiche :

```
animal
Lièvre    30.0
Tortue    19.0
Lion      180.0
Ours      270.0
Poule     30.0
Name: taille, dtype: float64
```

- Les opérations Numpy s'appliquent généralement sur les séries Pandas de type numérique : si `s` est une telle série, par exemple définie par `s = df['taille']`, on peut alors :
 - prendre sa moyenne : `np.mean(s)` ou `s.mean()`
 - faire sa somme : `np.sum(s)` ou `s.sum()`
 - renvoyer son minimum : `np.min(s)` ou `s.min()`
 - prendre les sommes cumulées : `np.cumsum(s)` ou `s.cumsum()`*Note : cette fonction renvoie une série Pandas, et l'ordre des sommes se fait dans l'ordre des lignes.*

- On peut connaître l'ensemble des **valeurs prises par une colonne (sans répétitions)** en utilisant `df['col'].unique()`, où `col` est le nom de la colonne.
- Accès à plusieurs colonnes du tableau en même temps grâce à `df[liste]`, avec `liste = ['col1', 'col2', ...]`. Cette fois, le type obtenu est bien un `DataFrame`.

1.1.3 Modifications basiques

De la même manière que Numpy, **les opérations dans un dataframe se font de manière vectorielle**, le long des colonnes. On n'a donc **pas besoin de boucle `for`** pour créer une série à partir de plusieurs séries existantes. Exemple avec le tableau ci-dessus : l'expression de l'indice de masse corporel est

$$\frac{\text{poids en kg}}{(\text{taille en m})^2}$$

ainsi on calcule l'IMC de nos animaux avec `df['poids'] / (df['taille'] / 100)**2`, ce qui renvoie :

```
animal
Lièvre    48.888889
Tortue    69.252078
Lion      58.641975
Ours      54.869684
Poule     38.888889
dtype: float64
```

C'est une série Pandas, toujours indexée par nos animaux, et le calcul a été effectué pour chacune des lignes. On peut maintenant assigner cette série à une **nouvelle colonne de notre tableau** en exécutant simplement `df['imc'] = df['poids'] / (df['taille'] / 100)**2`. Le tableau est modifié par cette commande : il y a une nouvelle colonne.

On peut aussi **modifier une colonne existante** de manière similaire, en assignant une valeur à `df[col]`. À noter que les séries fonctionnent de paire avec les objets Numpy, on peut les additionner, les multiplier entre elles, etc. Par exemple, si l'on veut perturber les données de la colonne `taille` par une loi normale : `df['taille'] += np.random.normal(scale=5, size=df.shape[0])`.

Pour **supprimer une colonne**, trois méthodes :

```
df = df.drop(columns='imc')
del df['imc']
df.pop('imc')
```

Notons que la première méthode permet de supprimer plusieurs colonnes en fournissant une liste de noms.

Pour **renommer une colonne** :

```
df = df.rename(columns={'taille': 'Taille (cm)', 'poids': 'Poids (kg)'})
```

Au lieu d'un dictionnaire, on peut aussi fournir une fonction, qui sera appliqué à chaque nom de colonne, par exemple : `df.rename(columns=str.title)`, qui ajoute des majuscules au début des mots.

Si vous voulez renommer la colonne numéro `i`, et que son nom est trop compliqué pour être proprement écrit, vous pouvez écrire `df = df.rename(columns={df.columns[i]: 'nouveau_nom'})`.

Si votre tableau a peu de colonnes, vous pouvez **les renommer toutes d'un coup** en exécutant : `df.columns = ['id', 'size', 'weight', 'speed']` (ici, on veut traduire en anglais le nom des colonnes).

Changer les types des données : comme avec Numpy, il est important de manipuler les données avec des types appropriés. Par exemple, si l'on veut interpréter les tailles comme des entiers (on suppose que le nom de la colonne n'a pas changé) : `df['taille'] = df['taille'].astype(int)`.

Le cas des dates : si l'on importe un fichier CSV qui contient des dates, celles-ci seront interprétées comme du texte par défaut, par exemple : '2022-11-22' au lieu de la date "22 novembre 2022". Si l'on a une colonne date de type object (type des chaînes de caractères dans un tableau Pandas), on peut les changer en objets de type "datetime" en utilisant : `df['date'] = pd.to_datetime(df['date'])`.

Cette transformation est intéressante pour pouvoir trier nos données, tracer nos données en fonction du temps, faire des opérations sur des fenêtres roulantes, etc.

Pour **trier un tableau**, on utilise la méthode `df.sort_values`. Par exemple, on veut trier les animaux par poids décroissant : `df = df.sort_values('poids', ascending=False)`. Trier sur plusieurs colonnes (avec l'ordre "lexicographique") est possible en fournissant une liste.

1.2 Filtrage de données

Rappelons que l'on travaille avec le tableau d'exemple obtenu par la ligne

```
df = pd.read_csv('data1.csv', index_col='animal') :
```

	id	taille	poids	vitesse
animal				
Lièvre	1	30.0	4.4	60.00
Tortue	2	19.0	2.5	0.25
Lion	3	180.0	190.0	57.00
Ours	4	270.0	400.0	40.00
Poule	5	30.0	3.5	17.00

On l'a vu, la syntaxe `df[colonnes]` permet de récupérer une ou plusieurs colonnes du tableau. Mais l'on peut aussi chercher à récupérer des lignes précises.

Pour cela, la principale syntaxe utile est l'**utilisation de masques booléens**. En théorie, un filtrage par masque booléen s'effectue quand on passe une liste de booléen de longueur "nombre de lignes" à l'intérieur de `df[...]`. Seules les lignes correspondantes à une valeur **True** seront renvoyées. En pratique c'est très simple, ce que l'on écrit ressemble à la syntaxe d'une phrase en français : si l'on veut récupérer les lignes correspondant aux animaux de taille inférieure à 50 cm, on écrit :

```
print(df[df['taille'] < 50])
#           id  taille  poids  vitesse
# animal
# Lièvre    1    30.0    4.4    60.00
# Tortue    2    19.0    2.5     0.25
# Poule     5    30.0    3.5    17.00
```

Les opérations booléennes ET, OU et NON, au niveau des arrays/séries, s'obtiennent au moyen des opérateurs `&`, `|` et `~`, ou bien (mais c'est plus lourd à écrire), avec les fonctions `np.logical_and`, `np.logical_or` et `np.logical_not`.

Exemples :

```
print(df[(df['taille'] < 50) & (df['vitesse'] > 10)])
#           id  taille  poids  vitesse
# animal
# Lièvre    1    30.0    4.4    60.0
# Poule     5    30.0    3.5    17.0

print(df[(df['vitesse'] < 10) | (df['poids'] > 300)])
#           id  taille  poids  vitesse
# animal
# Tortue    2    19.0    2.5    0.25
# Ours      4   270.0  400.0   40.00

print(df[~(df['poids'] < 50)])
#           id  taille  poids  vitesse
# animal
# Lion      3   180.0  190.0   57.0
# Ours      4   270.0  400.0   40.0
```

Pour **combiner cette opération de filtrage par masques booléens avec une sélection de colonnes**, on utilise la syntaxe `df.loc[masque_lignes, colonnes]`. Ainsi, pour récupérer seulement la taille et la vitesse des animaux ayant un poids supérieur à 50 kg, on écrit :

```
print(df.loc[df['poids'] > 50, ['taille', 'vitesse']])
#           taille  vitesse
# animal
# Lion      180.0    57.0
# Ours      270.0    40.0
```

Note : bien sûr, on peut ne passer qu'un seul nom de colonne : alors une série est renvoyée.

Indexation numérique : on peut simplement vouloir récupérer les lignes et colonnes données par leur position dans le tableau. On utilise alors une syntaxe de slicing ou d'indexation par liste avec `df.iloc[...]`.

Exemple :

```
# Récupère les deux premières lignes, avec les colonnes 3 et 1.
print(df.iloc[:2, [3, 1]])
#           vitesse  taille
# animal
# Lièvre    60.00    30.0
# Tortue    0.25    19.0
```

Supprimer des lignes identifiées par un masque booléen se fait facilement avec les syntaxes vues ci-dessus : `df2 = df.loc[df['taille'] < 50]` (on garde seulement les animaux ayant une taille inférieure à 50 cm).

Attention (!), on peut penser que le `.loc` est inutile plus haut, mais si vous écrivez à la place `df2 = df[df['taille'] < 50]` et que l'on cherche ensuite à modifier le tableau `df2` obtenu, on obtient cet avertissement :

SettingWithCopyWarning:

A value is trying to be set on a copy of a slice from a DataFrame.

Try using `.loc[row_indexer,col_indexer] = value` instead

Ceci veut dire que dans ce cas, si l'on modifie `df2`, le module Pandas ne garantit pas la modification ou non-modification du tableau original `df`, ce qui résulte en un code imprévisible (cf. [la doc](#)). Même si votre code fait clairement ce que vous voulez, ne laissez pas traîner ces `SettingWithCopyWarning`.

1.3 Opérations avancées sur les séries, str, datetime, valeurs manquantes

Les opérations prévues par Numpys et Pandas pour être **vectérielles** sont **plus efficaces** que la même opération codée “à la main”. De plus, c’est aussi *généralement* plus court à écrire. Exemple : si l’on a deux séries `a` et `b`, le code `c = a + b` tournera bien plus vite, tout en donnant le même résultat, que `c = pd.Series(x + y for x, y in zip(a, b))`, qui est toujours plus rapide que

```
c_list = []
for x, y in zip(a, b):
    c_list.append(x + y)
c = pd.Series(c_list)
```

Ces deux derniers exemples sont des **exemples de choses à ne pas faire**, puisque rappelons-le, il suffit d’écrire `c = a + b`.

Dans le cas (rare !) où l’on ne peut pas utiliser une opération vectorielle toute faite, on préférera l’utilisation de la syntaxe `a.apply(fonction)`, qui, si elle est équivalente à une compréhension de liste en terme de performance, a le mérite d’être encore plus compacte.

Exemple : on définit la série `a = pd.Series(['une', 'suite', 'de', 'mots'])`, et on voudrait obtenir une série `b` qui contient les mêmes mots, mais écrits en majuscule. On sait que la fonction `str.upper` prend en argument une chaîne de caractère et renvoie la même chaîne où toutes les lettres sont en majuscules. On peut donc définir : `b = a.apply(str.upper)`.

Mais il se trouve que dans ce cas, Pandas a prévu le coup : beaucoup de fonctions usuelles des objets `str` sont implémentées comme opérations vectorielles, et la syntaxe est la suivante : `b = a.str.upper()`, pour obtenir un objet `b` équivalent à `pd.Series(['UNE', 'SUITE', 'DE', 'MOTS'])`. Ici, `str` est un objet attaché à la classe `Series`, qui contient un ensemble de fonctions qui s’appliquent ligne par ligne sur la série, de manière efficace.

Plus de détails sur les méthodes de type `a.str.fonction` peuvent être trouvés sur [cette page](#) de la documentation.

Les séries dont les données sont de type `datetime` ont aussi leurs fonctions spécifiques, que l’on peut appeler avec `a.dt.fonction`.

Exemple :

```
a = pd.to_datetime(pd.Series(['2019-02-12', '2020-11-28']))
print(list(a.dt.day))           # [12, 28]
print(list(a.dt.month))         # [2, 11]
print(list(a.dt.month_name()))  # ['February', 'November']
```

Plus de détails en commençant par [cette page](#), puis en cherchant toutes les fonctions de la forme `a.dt.fonction` sur la page de l’objet `Series`.

Chercher une entrée dans une liste, de manière vectorielle. Supposons que l'on dispose d'un tableau `df` avec une colonne catégorielle, par exemple une colonne `animal` qui nous donne le nom de l'animal considéré sur la ligne. Si l'on souhaite garder seulement les lignes concernant les animaux dans une liste spécifique `animaux`, alors on peut appliquer le code suivant (**ne pas prendre comme exemple, un code plus efficace suit**) :

```
# Initialise une série remplie de False, indexée comme le tableau df.
masque = pd.Series(False, index=df.index)
for nom in animaux:
    # Bascule sur True les indices des lignes correspondant à l'animal en question.
    masque |= (df['animal'] == nom)
# Récupère le tableau filtré.
df = df[masque]
```

Mais cela a plusieurs inconvénients : c'est lent si la liste est longue, c'est relativement long à écrire et pas évident à comprendre.

Pandas a prévu le coup pour cette opération : la fonction `pd.Series.isin` remplit exactement cette fonction de construction du masque booléen : `df['animal'].isin(animaux)` renvoie exactement la variable `masque` que l'on construit ci-dessus. Ainsi, on écrira simplement

```
df = df[df['animal'].isin(animaux)].
```

Valeurs manquantes. Ici, on ne va pas voir comment traiter les données manquantes pour des problèmes complexes de statistique. La question est de savoir, étant donnée une série `a`, quels sont les indices des lignes sans valeur. Pour Pandas, une valeur manquante est un objet spécial (`np.nan` ou `pd.NA`), représenté par `NaN` ou `<NA>`.

Les méthodes `a.isna()` et son "inverse" `a.notna()` permettent d'obtenir un **masque booléen disant, pour chaque ligne, si la valeur est manquante ou non**.

Pour un tableau `df`, le code `df = df.dropna()` **élimine les lignes contenant une valeur manquante**.

Cela demanderait tout un TP d'apprendre à travailler véritablement avec des données manquantes avec Pandas, pour ce cours d'introduction, on redirige donc vers [la documentation](#).

1.4 Restructurer les données

Dans cette section, on va voir comment restructurer les tableaux de données. On illustrera cette section avec les tableaux `df` et `df_regions` récupérés par le code

```
df = pd.read_csv('data2.csv')
df_regions = pd.read_csv('data3.csv')
```

où les fichiers `data2.csv` et `data3.csv` contiennent respectivement les lignes suivantes :

id,animal,taille,poids,vitesse	ID,animal,Region
10,Lièvre,30.,4.4,60.	21,Tortue,Bretagne
11,Lièvre,25,3.7,58	20,Tortue,Bretagne
20,Tortue,19,2.5,.25	11,Lièvre,Bretagne
21,Tortue,21,3,.22	10,Lièvre,Creuse
30,Lion,180,190,57.	30,Lion,Doubs
31,Lion,165,150,64.	31,Lion,Doubs
40,Ours,270,400,40	40,Ours,Hautes-Pyrénées
41,Ours,290,460,36	51,Poule,Hautes-Pyrénées
50,Poule,30,3.5,17	41,Ours,Jura
51,Poule,25,3.1,17	50,Poule,Jura

Ainsi les résultats de `print(df)` et `print(df_regions)` donnent :

	id	animal	taille	poids	vitesse		ID	animal	Region
0	10	Lièvre	30.0	4.4	60.00	0	21	Tortue	Bretagne
1	11	Lièvre	25.0	3.7	58.00	1	20	Tortue	Bretagne
2	20	Tortue	19.0	2.5	0.25	2	11	Lièvre	Bretagne
3	21	Tortue	21.0	3.0	0.22	3	10	Lièvre	Creuse
4	30	Lion	180.0	190.0	57.00	4	30	Lion	Doubs
5	31	Lion	165.0	150.0	64.00	5	31	Lion	Doubs
6	40	Ours	270.0	400.0	40.00	6	40	Ours	Hautes-Pyrénées
7	41	Ours	290.0	460.0	36.00	7	51	Poule	Hautes-Pyrénées
8	50	Poule	30.0	3.5	17.00	8	41	Ours	Jura
9	51	Poule	25.0	3.1	17.00	9	50	Poule	Jura

1.4.1 Fusion

Nous allons voir trois méthodes qui permettent de fusionner deux tableaux : `df.join`, `df.merge` et `pd.concat`. Les deux premières permettent d'opérer une **fusion en largeur** (joindre des colonnes), et la dernière permet d'opérer une **fusion en longueur** des tableaux (joindre des lignes). Pour notre exemple, on aurait envie de fusionner les tableaux en largeur pour rajouter une colonne `Region` à `df`.

- Le code `df.join(df_regions)` effectue la **fusion en se basant sur l'index**. Si l'on exécute ce code, il va y avoir deux soucis : d'abord, deux colonnes ont le même nom (`animal`), et ça pose problème pour cette fonction. On résout ce problème en spécifiant un suffixe à rajouter sur les colonnes de même nom, soit sur le tableau de gauche avec l'argument optionnel `lsuffix`, soit sur le tableau de droite avec l'argument optionnel `rsuffix`.

Ainsi, on peut exécuter `df.join(df_regions, rsuffix='_reg')`, ce qui donne

	id	animal	taille	poids	vitesse	ID	animal_reg	Region
0	10	Lièvre	30.0	4.4	60.00	21	Tortue	Bretagne
1	11	Lièvre	25.0	3.7	58.00	20	Tortue	Bretagne
2	20	Tortue	19.0	2.5	0.25	11	Lièvre	Bretagne
3	21	Tortue	21.0	3.0	0.22	10	Lièvre	Creuse
4	30	Lion	180.0	190.0	57.00	30	Lion	Doubs
5	31	Lion	165.0	150.0	64.00	31	Lion	Doubs
6	40	Ours	270.0	400.0	40.00	40	Ours	Hautes-Pyrénées
7	41	Ours	290.0	460.0	36.00	51	Poule	Hautes-Pyrénées
8	50	Poule	30.0	3.5	17.00	41	Ours	Jura
9	51	Poule	25.0	3.1	17.00	50	Poule	Jura

C'est là qu'on voit le second problème : en faisant cette fusion sur les index, avec des index "anonyme", on n'a pas tenu compte de la véritable colonne d'intérêt qui identifie nos lignes dans ces jeux de données : id ou ID.

Pour tenir compte de ça, il suffit de dire à Pandas de prendre ces colonnes comme les indices de nos tableaux.

Ainsi, le code `df.set_index('id').join(df_regions.set_index('ID'), rsuffix='_reg')` renvoie :

	animal	taille	poids	vitesse	animal_reg	Region
id						
10	Lièvre	30.0	4.4	60.00	Lièvre	Creuse
11	Lièvre	25.0	3.7	58.00	Lièvre	Bretagne
20	Tortue	19.0	2.5	0.25	Tortue	Bretagne
21	Tortue	21.0	3.0	0.22	Tortue	Bretagne
30	Lion	180.0	190.0	57.00	Lion	Doubs
31	Lion	165.0	150.0	64.00	Lion	Doubs
40	Ours	270.0	400.0	40.00	Ours	Hautes-Pyrénées
41	Ours	290.0	460.0	36.00	Ours	Jura
50	Poule	30.0	3.5	17.00	Poule	Jura
51	Poule	25.0	3.1	17.00	Poule	Hautes-Pyrénées

C'est cohérent avec ce qu'on voulait avoir. On peut maintenant récupérer ce tableau dans une variable `df2`, supprimer la colonne inutile `animal_reg`, remplacer `id` en tant que colonne "normale" avec `df2 = df2.reset_index()`.

- On peut faire la même chose que ci-dessus avec `df.merge`, mais les options par défaut ne sont pas les mêmes que pour `df.join`. **La fusion se fait non pas sur l'index, mais sur les colonnes en commun.** Ainsi `df.merge(df_regions)` renverra un tableau avec 20 lignes : quatre par animal ! Pourquoi ? La seule colonne en commun est `animal`, et pour chaque valeur de celle-ci, on a deux lignes dans chaque tableau. Pandas ne sait pas laquelle des deux lignes de `df` va avec laquelle des deux lignes de `df_regions` (et d'ailleurs Pandas ne suppose pas qu'il y a une correspondance un à un), donc le nouveau tableau renvoie, pour chaque valeur, les $2 \times 2 = 4$ combinaisons possibles.

Encore une fois, la solution consiste à préciser que la fusion doit faire correspondre la colonne `id` de `df` avec `ID` de `df_regions`.

Ceci peut s'obtenir avec `df.merge(df_regions, left_on='id', right_on='ID')`, ce qui renvoie :

	id	animal_x	taille	poids	vitesse	ID	animal_y	Region
0	10	Lièvre	30.0	4.4	60.00	10	Lièvre	Creuse
1	11	Lièvre	25.0	3.7	58.00	11	Lièvre	Bretagne
2	20	Tortue	19.0	2.5	0.25	20	Tortue	Bretagne
3	21	Tortue	21.0	3.0	0.22	21	Tortue	Bretagne
4	30	Lion	180.0	190.0	57.00	30	Lion	Doubs
5	31	Lion	165.0	150.0	64.00	31	Lion	Doubs
6	40	Ours	270.0	400.0	40.00	40	Ours	Hautes-Pyrénées
7	41	Ours	290.0	460.0	36.00	41	Ours	Jura
8	50	Poule	30.0	3.5	17.00	50	Poule	Jura
9	51	Poule	25.0	3.1	17.00	51	Poule	Hautes-Pyrénées

Encore plus simple, pour avoir moins de colonnes à gérer à la fin, faire correspondre le nom de toutes les colonnes pouvant fusionner : `df.merge(df_regions.rename(columns={'ID': 'id'}))`,

ce qui renvoie simplement :

	id	animal	taille	poids	vitesse	Region
0	10	Lièvre	30.0	4.4	60.00	Creuse
1	11	Lièvre	25.0	3.7	58.00	Bretagne
2	20	Tortue	19.0	2.5	0.25	Bretagne
3	21	Tortue	21.0	3.0	0.22	Bretagne
4	30	Lion	180.0	190.0	57.00	Doubs
5	31	Lion	165.0	150.0	64.00	Doubs
6	40	Ours	270.0	400.0	40.00	Hautes-Pyrénées
7	41	Ours	290.0	460.0	36.00	Jura
8	50	Poule	30.0	3.5	17.00	Jura
9	51	Poule	25.0	3.1	17.00	Hautes-Pyrénées

- Enfin pour **ajouter des lignes à la fin d'un tableau** `df`, si l'on dispose d'un tableau `df2` avec les mêmes colonnes, utiliser `df = pd.concat([df, df2])`. Concrètement, on n'a pas besoin d'avoir les mêmes colonnes, auquel cas on aura des valeurs manquantes dans le résultat de la concaténation. Aussi, la fonction `pd.concat` prend en argument une liste de tableaux (donc potentiellement plus de deux tableaux) qui seront concaténés les uns à la suite des autres.

Si à la suite de vos opérations de fusion on a des index qui n'ont pas de sens (des numéros répétés, qui n'ont pas de sens pour les données), on peut

- Indexer le tableau obtenu `dfres` avec une colonne existante : `dfres = dfres.set_index(colonne)`.
- Renommer le tableau avec un index "anonyme" : `dfres = dfres.reset_index(drop=True)`.

1.4.2 Pivot

Pour illustrer cette section, prenons un fichier `data4.csv` composé des lignes suivantes :

```
id,animal,taille,poids,vitesse,Region,sexe
10,Lièvre,30.0,4.4,60.0,Creuse,M
11,Lièvre,25.0,3.7,58.0,Bretagne,F
20,Tortue,19.0,2.5,0.25,Bretagne,M
21,Tortue,21.0,3.0,0.22,Bretagne,F
30,Lion,180.0,190.0,57.0,Doubs,M
31,Lion,165.0,150.0,64.0,Doubs,F
40,Ours,270.0,400.0,40.0,Hautes-Pyrénées,M
41,Ours,290.0,460.0,36.0,Jura,F
50,Poule,30.0,3.5,17.0,Jura,M
51,Poule,25.0,3.1,17.0,Hautes-Pyrénées,F
```

Ainsi le tableau `df = pd.read_csv('data4.csv')` contient

	id	animal	taille	poids	vitesse	Region	sexe
0	10	Lièvre	30.0	4.4	60.00	Creuse	M
1	11	Lièvre	25.0	3.7	58.00	Bretagne	F
2	20	Tortue	19.0	2.5	0.25	Bretagne	M
3	21	Tortue	21.0	3.0	0.22	Bretagne	F
4	30	Lion	180.0	190.0	57.00	Doubs	M
5	31	Lion	165.0	150.0	64.00	Doubs	F
6	40	Ours	270.0	400.0	40.00	Hautes-Pyrénées	M

7	41	Ours	290.0	460.0	36.00	Jura	F
8	50	Poule	30.0	3.5	17.00	Jura	M
9	51	Poule	25.0	3.1	17.00	Hautes-Pyrénées	F

Si l'on souhaite récupérer, dans un nouveau tableau `df_poids`, le poids des différents individus en fonction de leur sexe, il peut être plus clair d'afficher l'information avec une ligne par sexe, une colonne par animal. Ceci s'obtient **en pivotant le tableau** grâce à la fonction `df.pivot`. Exemple : `df_poids = df.pivot(index='sexe', columns='animal', values='poids')`, ce qui produit le tableau suivant :

animal	Lion	Lièvre	Ours	Poule	Tortue
sexe					
F	150.0	3.7	460.0	3.1	3.0
M	190.0	4.4	400.0	3.5	2.5

Cela bascule la colonne `animal` en tant qu'index : pour la remettre en tant que colonne normale, utiliser `df_poids = df_poids.reset_index()`.

Multi-index : si l'on souhaitait obtenir, au lieu d'une ligne par sexe, une ligne par combinaison de valeurs de deux colonnes différentes (par exemple, même si ce n'a pas trop de sens car on n'a pas assez de données ici, une ligne par combinaison sexe / région), alors on peut le préciser : `df_poids = df.pivot(index=['sexe', 'Region'], columns='animal', values='poids')`.

Alors l'index est ici une liste de tuples : `[('F', 'Bretagne'), ('F', 'Doubs'), ...]`. Ce n'est pas du tout la seule manière de se retrouver avec des multi-index et l'on ne va pas s'attarder dessus (cf. [la documentation](#) pour plus d'infos), mais ça peut être pratique dans certains cas. Pour **se débarasser de tout multi-index** et se retrouver avec des colonnes normales, on utilise `df.reset_index()`.

Melting. Pour **regrouper des colonnes dans une seule**, on utilise la fonction `df.melt`. Reprenons le tableau simple `df = pd.read_csv('data1.csv')`, qui contient 5 lignes et dont la liste des colonnes est `['id', 'animal', 'taille', 'poids', 'vitesse']`. Si l'on applique `df.melt(['id', 'animal'])`, les trois autres colonnes sont regroupées dans une colonne catégorielle `variable` et une colonne numérique `value`, qui donne la valeur correspondante. Ainsi, on obtient :

	id	animal	variable	value
0	1	Lièvre	taille	30.00
1	2	Tortue	taille	19.00
2	3	Lion	taille	180.00
3	4	Ours	taille	270.00
4	5	Poule	taille	30.00
5	1	Lièvre	poids	4.40
6	2	Tortue	poids	2.50
7	3	Lion	poids	190.00
8	4	Ours	poids	400.00
9	5	Poule	poids	3.50
10	1	Lièvre	vitesse	60.00
11	2	Tortue	vitesse	0.25
12	3	Lion	vitesse	57.00
13	4	Ours	vitesse	40.00
14	5	Poule	vitesse	17.00

Pour changer les noms variable et value, on utilise les arguments optionnels `var_name` et `value_name`.

1.4.3 Agrégation

Avec le pivot, ci-dessus, on a transformé toutes les catégories d'une colonne catégorielle (`animal`) en nouvelles colonnes (Lion, Lièvre, etc.). Si l'on ne souhaite pas faire ceci, c'est sans doute la méthode `df.groupby` que l'on veut utiliser. Exemple avec, comme plus haut, `df = pd.read_csv('data4.csv')` : le code `df.groupby('animal').mean()` ou `df.groupby('animal').agg('mean')` fait la moyenne des variables *numériques* concernant chacune catégorie de la colonne `animal`. Cela renvoie donc :

	id	taille	poids	vitesse
animal				
Lion	30.5	172.5	170.00	60.500
Lièvre	10.5	27.5	4.05	59.000
Ours	40.5	280.0	430.00	38.000
Poule	50.5	27.5	3.30	17.000
Tortue	20.5	20.0	2.75	0.235

Remarque : les colonnes de type object (chaînes de caractères) ont disparu, puisqu'on ne sait pas en faire la moyenne. Les lignes obtenues font la **moyenne des deux lignes concernant chaque animal**.

Enfin, on peut complexifier l'agrégation : par exemple, si l'on veut (dans l'exemple précédent) :

- ignorer la colonne `id` ;
- récupérer la moyenne et le maximum de la taille,
- la somme des poids,
- et le maximum des vitesses.

Pour cette opération on utilise la fonction `aggregate` ou son alias `agg` : la ligne

```
df.groupby('animal').agg({'taille': ['mean', 'max'], 'poids': 'sum', 'vitesse': 'max'})
```

renvoie le tableau :

	taille		poids	vitesse
	mean	max	sum	max
animal				
Lion	172.5	180.0	340.0	64.00
Lièvre	27.5	30.0	8.1	60.00
Ours	280.0	290.0	860.0	40.00
Poule	27.5	30.0	6.6	17.00
Tortue	20.0	21.0	5.5	0.25

Ici, on se retrouve par contre avec des **colonnes multi-indexées**. On peut travailler avec, mais pour faire simple et s'en débarrasser : la ligne `df.columns = ['_'].join(t) for t in df.columns]` change le tableau en :

	taille_mean	taille_max	poids_sum	vitesse_max
animal				
Lion	172.5	180.0	340.0	64.00
Lièvre	27.5	30.0	8.1	60.00
Ours	280.0	290.0	860.0	40.00
Poule	27.5	30.0	6.6	17.00

Tortue	20.0	21.0	5.5	0.25
--------	------	------	-----	------

Fenêtres roulantes. Un autre type d'agrégation est l'agrégation par fenêtres roulantes en utilisant `df.rolling`. On considère le tableau `df`, obtenu avec :

`df = pd.DataFrame(np.arange(12).reshape((6,2)), columns=['A', 'B'])`, ce qui produit :

	A	B
0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11

Pour faire la moyenne des valeurs, le long de chaque colonne, sur une fenêtre de 3 lignes, on utilise `df.rolling(3).mean()`, ce qui renvoie :

	A	B
0	NaN	NaN
1	NaN	NaN
2	2.0	3.0
3	4.0	5.0
4	6.0	7.0
5	8.0	9.0

Comme pour `groupby`, on peut utiliser plusieurs fonctions d'agrégation : `df.rolling(3).agg(['min', 'sum'])` renvoie

	A		B	
	min	sum	min	sum
0	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN
2	0.0	6.0	1.0	9.0
3	2.0	12.0	3.0	15.0
4	4.0	18.0	5.0	21.0
5	6.0	24.0	7.0	27.0

Remarque : ici, l'agrégation est faite sur les trois lignes qui précèdent la ligne en question (d'où les valeurs manquantes pour les deux premières lignes). Pour centrer la fenêtre roulante, utiliser `df.rolling(3, center=True).mean()` :

	A	B
0	NaN	NaN
1	2.0	3.0
2	4.0	5.0
3	6.0	7.0
4	8.0	9.0
5	NaN	NaN

Pour faire une agrégation par fenêtre roulante sur des dates, dans le cas où l'index est donné par des dates, on peut passer une durée à la fonction `df.rolling`. Par exemple, on modifie d'abord `df` :

```
df.index = pd.to_datetime(['2022-11-01', '2022-11-02', '2022-11-04',
                           '2022-11-05', '2022-11-08', '2022-11-10'])
```

Ceci fait que le tableau df est :

	A	B
2022-11-01	0	1
2022-11-02	2	3
2022-11-04	4	5
2022-11-05	6	7
2022-11-08	8	9
2022-11-10	10	11

Ici, comme notre index est une date, on peut faire une agrégation par somme (pour que ce soit clair) sur une durée de trois jours : `df.rolling('3d').sum()` renvoie :

	A	B
2022-11-01	0.0	1.0
2022-11-02	2.0	4.0
2022-11-04	6.0	8.0
2022-11-05	10.0	12.0
2022-11-08	8.0	9.0
2022-11-10	18.0	20.0

Noter que la durée prise en compte par défaut est par défaut la durée qui précède la date de la ligne. Pour centrer autour de la date de la ligne, on utilise à nouveau l'argument optionnel `center=True`. Enfin, l'argument `'3d'` désigne trois jours, mais on peut aussi écrire `'3m'` pour trois minutes, `'3s'` pour trois secondes, etc. Toute chaîne qui est acceptée par le constructeur `pd.Timedelta` (objet qui contient un différentiel de temps) est acceptée par cette fonction `df.rolling`.

1.5 Graphiques

On peut utiliser Matplotlib pour tracer ce que l'on veut à partir des colonnes de nos tableaux. Cependant, les fonctions intégrées de Pandas, et la librairie Seaborn, peuvent s'avérer plus pratiques. Le [tutoriel de Seaborn](#) est assez clair, je vais simplement donner quelques pistes, et les nombreux exemples en lignes vous permettront en général de trouver comment faire ce que vous cherchez à faire.

L'utilisation basique est la suivante :

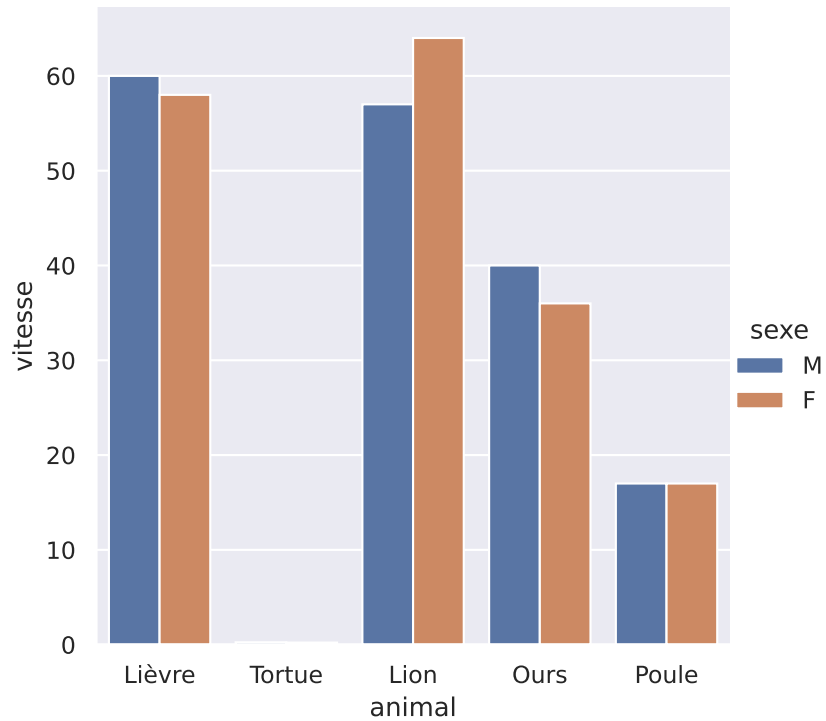
- (Optionnel) on charge le thème par défaut de Seaborn avec `sns.set_theme()`.
- On utilise :
 - `sns.relplot` pour des graphiques “relationnels” (tracer une variable en fonction d'une autre).
 - `sns.displot` pour des graphiques “de distributions” (tracer un histogramme, une estimation de densité, etc.).
 - `sns.catplot` pour des graphiques “catégoriels” (tracer de l'information suivant les catégories d'une variable).
- Toutes les précisions sur comment doit se tracer le graphique sont déterminées par des arguments optionnels :

- L'argument optionnel (mais nécessaire) `data=df` précise le tableau utilisé pour produire le graphe.
- Si l'on doit mettre des variables en abscisses et en ordonnées : `x='var'` ou `y='var'`, ou `var` est le nom de la colonne dans `df`.
- Pour déterminer quelle sous-catégorie de graphique tracer : `kind='...'`. Les trois points sont à remplacer par
 - pour `sns.relplot` : `'line'` pour avoir des courbes, `'scatter'` pour avoir un scatterplot,
 - pour `sns.displot` : `'hist'` pour un histogramme, `'kde'` pour l'estimation de densité, etc.
 - pour `sns.catplot` : `'bar'` pour un diagramme à barres, `'box'` pour des boîtes à moustache, etc.
- Pour distinguer différentes courbes ou points en changeant de couleur en fonction d'une variable catégorielle : `hue='var'`.
- Pour distinguer différentes courbes ou points en changeant de style (pointillés, ronds ou croix, etc.) en fonction d'une variable catégorielle : `style='var'`.
- Pour créer des sous-figures en fonction des valeurs d'une variable catégorielle :
 - `row='var'` pour créer une ligne par valeur distincte de `var`.
 - `col='var'` pour créer une colonne par valeur distincte de `var`.
- (Optionnel) Enfin (en tout cas c'est simple sur une figure sans sous-figure), on peut changer les nom des axes, et le titre de la figure avec les commandes Matplotlib. Avec Seaborn, la légende est mise automatiquement par défaut. Vous pouvez toujours utiliser `plt.show()` pour afficher la figure (si le script est lancé dans Spyder, la fenêtre graphique sera ouverte même sans `plt.show()` mais ce n'est pas le cas en général).

Les pages qui suivent montrent deux exemples de graphiques très simples.

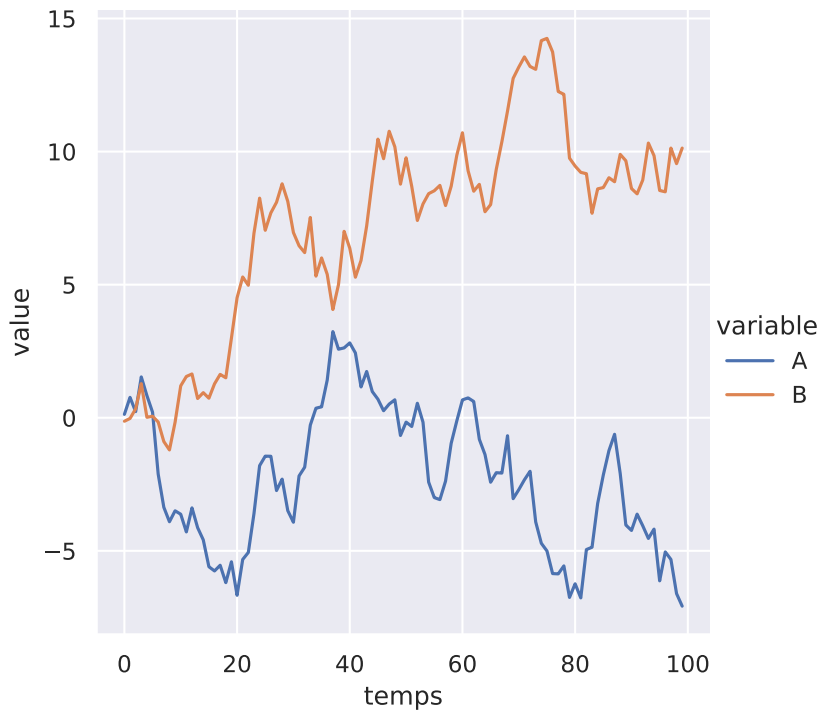
Premier exemple :

```
df = pd.read_csv('data4.csv')
sns.catplot(data=df, kind='bar', x='animal', y='vitesse', hue='sexe')
plt.show()
```



Second exemple :

```
# Construction du tableau
rng = np.random.default_rng(0)
df = pd.DataFrame(rng.normal(size=(100,2)), columns=['A', 'B']).cumsum()
df['temps'] = df.index
df = df.melt('temps')
# Le graphique
sns.relplot(data=df, x='temps', y='value', hue='variable', kind='line')
plt.show()
```



Pour voir la syntaxe Seaborn en application dans plus d'exemples, allez voir la [galerie](#) de Seaborn. Il faudra s'en inspirer pour comprendre comment tracer vos figures.

2 Exercices

Exercice 1 : Sur [cette page](#) du site `statistique.quebec.ca` se trouve une estimation de l'effectif des 1000 noms de familles les plus courants au Québec, dans un tableau HTML. Le but de cet exercice est d'importer ce tableau avec Pandas, de manière utilisable. Évidemment, il n'y a pas de fichier `csv` fourni sur le site, sinon ce serait trop simple !

- a) Effectuer la requête `tables = pd.read_html(url)`, où `url` est une chaîne de caractère contenant l'URL de la page qui nous intéresse.

Il est possible que vous ayez besoin d'installer le module `lxml` avec Anaconda, puis de redémarrer Spyder pour que cette commande fonctionne.

- b) Que récupère-t-on dans la variable `tables` ? Récupérer le dataframe qui nous intéresse dans une variable `df`, puis exporter-le dès à présent dans un fichier `brut.csv`.

Maintenant, vous pouvez commenter tout le début de votre script (en particulier l'appel à `pd.read_html`) et le remplacer par `df = pd.read_csv('brut.csv')`.

- c) Afficher les premières lignes de `df` et ses dernières lignes. Effacer celles qui sont sans intérêt.
- d) Séparer le tableau judicieusement en quatre tableaux dans le sens de la largeur.
- e) Pour chacun de ces tableaux, supprimer la colonne inutile et renommer les autres en `'rang'`, `'nom'` et `'nombre'`.
- f) Empiler les quatre tableaux en un seul en utilisant `pd.concat`. Assigner le tableau résultant à la variable `df`.
- g) Si tout va bien, votre tableau a 1000 lignes et 3 colonnes. Si ce n'est pas le cas, réparer le script jusqu'à obtenir ce que l'on veut. Enfin, exporter le tableau dans un fichier `noms0.csv`.

Exercice 2 : On reprend l'exercice précédent (on suppose donc l'existence du fichier `noms0.csv` comme obtenu à la fin de l'exercice).

- a) On voudrait classer le tableau par ordre de rang croissant, plutôt que par ordre alphabétique. Afficher les premières lignes de `df.sort_values('rang')`. Quel est le souci ?
- b) On voudrait donc convertir les types des colonnes `rang` et `nombre` en `int`. Quel autre problème rencontre-t-on, si l'on essaie la méthode la plus directe ?
- c) Écrire une fonction `asint(text)` qui enlève les espaces de la chaîne de caractère `text` puis utilise `int()` pour renvoyer l'entier correspondant.
Pour enlever les espaces, on pourra penser par exemple aux fonctions `split` et `join`, ou au module `re`.
- d) Grâce à la question précédente, changer les deux colonnes problématiques de `df` en colonnes d'entiers, puis trier le tableau par ordre croissant de `rang` et le ré-indexer.
- e) Quels sont les 35 noms les plus courants au Québec ? Votre nom fait-il partie de la liste des 1000 ?
- f) (Optionnel) Enregistrer le tableau dans un fichier `noms.csv` pour pouvoir l'admirer plus tard.

Le site [data.gouv.fr](#) vise à rassembler de nombreux jeux de données concernant la France : si vous cherchez des données pour un projet, vous pouvez notamment chercher là-dedans (ou dans le site *our world in data*, dont on parle plus bas). L'exercice suivant propose de travailler avec un jeu de données de enregistré sur ce site.

Exercice 3 : Dans cet exercice, on s'intéressera aux températures quotidiennes par département.

- a) Importer les [données](#) dans un tableau `temp`. Changer la colonne `date_obs` en type "datetime".
- b) Se fixer une liste de trois départements et restreindre le jeu de données à ceux-ci.
- c) Tracer les courbes de températures moyennes au cours du temps, pour vos départements. Qu'est-ce qui gêne la lisibilité ?
- d) On voudrait utiliser Pandas pour lisser les données sur une fenêtre roulante. Écrire une fonction `lissage(df, colonne, methode, fenetre)` qui :
 - Pivote `df` pour avoir les dates en index, les départements en colonne et la colonne initiale colonne en valeurs.
 - Utilise `rolling` puis `agg` pour faire l'opération décrite par `methode`, sur une fenêtre donnée par `fenetre`.
 - Utilise `melt` pour remettre les données en forme avec les trois colonnes `date_obs`, `departement` et "colonne" (avec son nom initial).

Le tableau modifié par ces opérations successives est finalement renvoyé.

Concrètement, on doit pouvoir utiliser `df = lissage(temp, 'tmoy', 'mean', '60d')` pour obtenir les données moyennées sur une fenêtre de 60 jours.

Note : On pourrait lisser les données de minimum, moyenne et maximum en même temps mais ça reste complexe. Exo bonus : chercher à faire cela en ne manipulant qu'un seul tableau.

- e) Utiliser la fonction précédente pour tracer le graphe des températures maximales, puis le graphe des températures moyennes, sur une fenêtre roulante de 60 jours, pour les départements considérés.
- f) On veut tracer le profil de température d'une année typique, pour chaque département. Pour cela :
 - Ajouter une colonne `semaine` au tableau, qui contient le numéro de la semaine dans l'année (de 1 à 52). On pourra utiliser la méthode `pandas.Series.dt.isocalendar`.
 - Ajouter une colonne `année`, de manière similaire.
 - En utilisant la méthode `temp.groupby`, puis la méthode `agg`, définir un nouveau tableau `temp_groupe` dont chaque ligne correspond à un département, une année et une semaine, et dont les colonnes `tmin` et `tmax` sont définies comme les minimums et maximums des températures observées sur la semaine en question.
- g) Afficher les profils de vos départements avec

```
sns.relplot(data=temp_groupe, x='tmin', y='tmax', hue='departement')
```

Le 15 novembre 2022, la population mondiale a atteint **8 milliards d'êtres humains**, alors que vous vous rappelez peut-être avoir appris enfant que l'on était 6 milliards (c'était le cas autour de l'an 1999). Les deux exercices suivants ont pour but de "fêter" cet événement et de nous permettre de (re)visualiser la croissance de la population humaine au cours de son histoire.

Exercice 4 : Sur [cette page](#) du site [our world in data \(OWID\)](#), vous trouverez un graphe interactif montrant l'évolution de la population dans différentes régions du monde, depuis l'an -10 000. Les données utilisées sur ce graphe proviennent de plusieurs sources : on peut cliquer sur l'onglet du graphe "Sources" pour les voir, (on pourrait donc télécharger les données brutes et faire une analyse détaillée). Dans cet exercice, on va directement télécharger les données agrégées et nettoyées par les scientifiques de OWID.

- a) Cliquer sur l'onglet "Download" et télécharger le fichier CSV proposé. Mettez-le dans le même dossier que votre script.
- b) Utiliser Pandas pour charger le jeu de données dans un tableau `dfpop1`.
- c) Explorer le tableau avec Pandas : afficher le nom des colonnes, leurs types.
Si vous trouvez que des colonnes ont un nom trop long, vous pouvez les renommer.
- d) Utiliser Seaborn pour tracer l'évolution de la population mondiale (attention, on parle de tracer une seule courbe) au cours du temps.
- e) On voudrait maintenant comparer l'évolution de la population de nos pays préférés au cours du temps.
Écrire une fonction `regions_par_pop(df, annee)` qui renvoie un tableau dont les lignes sont les pays et l'unique colonne donne le nombre d'habitants pour l'année passée en argument. L'argument `df` est bien sûr censé recevoir la variable `dfpop1`.
- f) En ordonnant les régions du monde par nombre d'habitants en l'an 2000, identifier votre pays favori et trois autres pays qui ont un nombre d'habitants comparable. Enregistrer leurs noms dans une liste.
- g) Écrire une fonction `plot_pays(df, liste_pays, depart=None)` qui trace sur un même graphe l'évolution des populations des pays listés, depuis l'année `depart` (dans le cas `depart=None`, ne pas mettre de restriction d'années).
Tester sur vos pays favoris. Améliorer l'affichage si nécessaire.

Exercice 5 : Cet exercice fait la paire avec l'exercice précédent : sur [cette page](#) se trouve un graphe qui montre le taux de croissance (en naissance / an) de différentes régions du monde, avec des projections jusqu'à l'an 2100. Encore une fois vous trouverez la source : ce sont les prévisions de l'ONU.

- a) Utiliser l'onglet "Download" pour télécharger les données au format CSV, et les importer dans un tableau `dfpop2`.
- b) Remarquons que les prédictions et les données connues (plus exactement les estimations du passé) sont dans deux colonnes différentes. On va changer ça :
 - Rajouter au tableau une colonne '**Prédiction**' de valeurs booléennes, qui indique si la ligne correspond à une prédiction ou non.
 - Réunir les deux colonnes des taux de croissances dans une seule colonne appelée '**Croissance**'. Les deux anciennes colonnes seront supprimées.
- c) Sur la même figure, tracer les deux courbes du taux de croissance mondial (réel pour les années 1951–2021 et prédit pour les années 2022–2100). Faire en sorte que les deux courbes s'affichent avec la même couleur, la deuxième en pointillés.
- d) On veut maintenant prédire les populations des différents pays pour les années 2021–2100. Grâce à la méthode `dfpop2.pivot`, construire un tableau `dfpop2_pivot` ayant pour lignes les années, pour colonnes les régions et pour valeurs les taux de croissance par année.
- e) Construire similairement `dfpop1_pivot` à partir du tableau `dfpop1` de l'exercice précédent.
- f) Construire un tableau `dfpop3` de la manière suivante :
 - Utiliser la fonction `pd.concat` pour construire la concaténation de la ligne de `dfpop1_pivot` correspondant à l'année 2021 avec les lignes de taux de croissance prédit (donc à partir de 2022) du tableau `dfpop2_pivot`. On pourra ignorer les colonnes non présentes dans les deux tableaux grâce à l'argument optionnel `join='inner'`.

- Effectuer la somme cumulée le long des colonnes pour obtenir les prévisions pour les années 2022–2100.
- g) Enfin, modifier `dfpop3` pour intégrer les lignes correspondant aux années passées (normalement, elles sont dans `dfpop1_pivot`).
- h) Modifier la fonction `plot_pays(df, liste_pays, depart=None)` de l'exercice précédent pour qu'elle intègre les prévisions, affichées en style "tirets". Cette fois, l'argument `df` sera destiné à recevoir la variable `dfpop3`.
- Tester sur vos pays favoris. Améliorer l'affichage si nécessaire.

Exercice 6 : Reproduire du mieux possible les graphiques des pages suivantes :

- Performance scolaire des enfants de 15 ans.
- Genre des fumeurs.
- Morts dues à la pollution de l'air.
- Part des véhicules électriques et hybrides.
- Consommation de viande par personne.
- Démocraties et dictatures.

Exercice 7 : Trouvez vos propres jeux de données à manipuler avec Pandas et Seaborn sur data.gouv.fr, [our world in data](https://ourworldindata.org) ou un autre site de votre choix.