M55 - Data Science & Scientific Computing 5

Gloria Faccanoni

11 novembre 2026

Table des matières

1. M55 - Data Science and Scientific Computing 5 1
Introduction à la manipulation de données avec Pandas . . . 1

I Introduction 3

2. Syllabus du cours “Introduction à la manipulation de données avec Pandas” 5
2.1. Parcours “Data Science and Scientific Computing” . . . 5
2.2. Déroulement du cours DSSC-5 : Introduction à la manipulation de données avec Pandas . . . 5
2.2.1. Plan des séances . . . 6
2.2.2. Prérequis . . . 6
2.2.3. Évaluation . . . 6
2.2.4. Ressources pour trouver des jeux de données . . . 7

II Cours 9

3. Premiers pas avec Pandas 11
3.1. Importation de pandas . . . 11
3.2. Une analyse complète, en quinze lignes . . . 12
3.2.1. 1. Lire le fichier . . . 12
3.2.2. 2. Regarder la forme du jeu de données . . . 13
3.2.3. 3. Créer une colonne exploitable . . . 13
3.2.4. 4. Agréger : la consommation dépend-elle du nombre de cylindres ? . . . 14
3.2.5. 5. Visualiser la relation complète . . . 14
3.2.6. 6. Conclure . . . 15
3.2.7. Ce que nous venons de faire, et où vous l’apprendrez . . . 15

4. Introduction aux structures de données pandas : les Series (tableaux unidimensionnels) 17

5. Création d’une Series 19
5.1. À partir d’une liste/array de valeurs . . . 19
5.2. À partir de deux listes/array : une liste de valeurs et une liste d’index associés . . . 19
5.3. À partir d’un dictionnaire `{ label1: valeur1, label2: valeur2, ...}` . . . 21

6. Slicing ⇝ sélection avec `loc` et `iloc` 23

7. Ajouter des éléments à une Series / Fusionner deux Series 27

8. Supprimer des éléments à une Series 29

9. Tableau comparatif entre `list`, `dict`, `numpy.ndarray` et `pandas.Series` 31

10. Représentation graphique d’une Series 33

11. Introduction aux structures de données pandas : les DataFrames (tableaux bidimensionnels) 37

12. Création d'un DataFrame 39

12.1. À partir d'un tableau NumPy à deux dimensions . . . 39
12.2. Colonne par colonne à partir … . . . 40
12.2.1. … d'une seule Series . . . 40
12.2.2. … à partir d'un dictionnaire de Series `{label_1: serie_1, label_2: serie_2, etc}` ou de listes de valeurs `{label_1: [val_1, val_2, etc], label_2: [val_1, val_2, etc], etc}` 42
12.3. Ligne par ligne à partir … . . . 43
12.3.1. … d'une liste de Series `[serie_1, serie_2, etc]` ou d'une liste de dictionnaires `[dico_1, dico_2, etc]` . . . 43
12.4. À partir d'un fichier : `read_csv` . . . 44
12.4.1. Les options à connaître . . . 46
12.4.2. Un cas réel : l'espace insécable . . . 47
12.4.3. Exporter : `to_csv` . . . 47
12.4.4. Bonus : manipuler des dates . . . 48

13. Accés aux éléments 51

13.1. Selectionner une colonne : `df['nom_colonne']` ou `df.nom_colonne` . . . 51
13.2. Selectionner une ligne : `loc` et `iloc` . . . 53
13.2.1. Tableau comparatif entre `numpy.ndarray` et `pandas.DataFrame` . . . 55
13.2.2. Résumé des principales syntaxes d'accès aux éléments d'un DataFrame : . . . 56
13.3. Ajout de lignes/colonnes . . . 56
13.3.1. Ajout d'une colonne . . . 56
13.3.2. Ajout d'une ligne . . . 57
13.4. Réordonner les observations . . . 57
13.5. Suppression de lignes/colonnes . . . 58

14. Aperçu des données 61

14.1. Afficher les premières/dernières lignes . . . 61
14.2. Attributs d'un DataFrame . . . 62
14.3. Les valeurs manquantes . . . 64
14.3.1. Repérer les valeurs manquantes . . . 64
14.3.2. Comment les calculs les traitent . . . 66
14.3.3. Supprimer : `dropna` . . . 66
14.3.4. Remplacer : `fillna` . . . 67
14.4. Statistiques descriptives . . . 68
14.5. Comptage : valeurs uniques, dénombrement de valeurs et appartenance . . . 71

15. Filtres et opérations 75

15.1. Opérations sur les DataFrames . . . 75
15.2. Filtrer un DataFrame : masque booléen *vs* `query()` . . . 75
15.2.1. Filtre "à la NumPy" (masques booléens) . . . 76
15.2.2. Filtre avec `query()` . . . 77

16. Aggregations 79

16.1. `groupby` : analyse par groupes . . . 79
16.2. `pivot_table` : tableaux croisés . . . 85
16.3. `crosstab` : tableaux croisés . . . 87
16.4. Bonus : `melt`, du format large au format long . . . 89
16.5. Arithmétique et alignement des données : différences avec `Numpy` . . . 90

16.6. L'indexation des DataFrames . . . 91
16.6.1. Bonus : `set_index` et `reset_index` . . . 92
16.7. Copie *vs* vue . . . 94
16.7.1. Pourquoi ? Ce qui se passe réellement . . . 94
16.7.2. Ce qui change avec le *Copy-on-Write* . . . 95

17. Combiner des tables 97
17.1. Empiler des tables : `pd.concat` . . . 97
17.2. Joindre deux tables : `pd.merge` . . . 99
17.2.1. Les quatre types de jointure : l'argument `how` . . . 100
17.2.2. Quand les colonnes clés n'ont pas le même nom . . . 102
17.2.3. Vérifier une jointure . . . 103
17.2.4. `join` : la variante qui travaille sur l'index . . . 104
17.2.5. En résumé . . . 104

18. Analyse statistique et visualisation : le jeu de données du Titanic 105
18.1. Importation des données . . . 106
18.2. Étude globale de l'ensemble de données . . . 107
18.2.1. Données manquantes ? . . . 108
18.2.2. Bilan sur l'ensemble de données . . . 109
18.2.3. Statistiques descriptives . . . 110
18.3. Analyse univariée (chaque colonne individuellement) . . . 111
18.3.1. Colonne `nom` . . . 111
18.3.2. Préambule : valeurs uniques dans chaque colonne . . . 116
18.3.3. Colonne `classe` . . . 117
18.3.4. Colonne `sexe` . . . 121
18.3.5. Colonne `survivant` . . . 124
18.3.6. Colonne `age` . . . 126
18.3.7. Nouvelle Colonne `age_group` . . . 130
18.4. Liens entre deux ou plusieurs colonnes . . . 133
18.4.1. Analyse bivariée, vue d'ensemble . . . 134
18.4.2. Classe / Survie . . . 136
18.4.3. Sexe / Survie . . . 145
18.4.4. Age / Survie . . . 148
18.4.5. Age_group / Survie . . . 149
18.4.6. Port de départ / Survie . . . 151
18.4.7. Bonus : nouvelle colonne femmes / hommes / enfants . . . 153
18.4.8. Classe / Âge / Survie . . . 155
18.4.9. Sexe / Âge / Survie . . . 157
18.4.10. Prix / Âge / Survie . . . 158
18.4.11. Classe / Sexe / Survie . . . 160
18.4.12. Classe / Prix / Survie . . . 163
18.4.13. Prix / Âge / Sexe / Survie . . . 165
18.5. Miscellanea . . . 166
18.5.1. `groupby()` . . . 166
18.5.2. Filtres et `value_counts` . . . 167
18.5.3. Corrélation entre les données : `corr` . . . 167
18.6. Autres ressources . . . 170
18.7. Pour aller plus loin : analyse exploratoire → prédiction . . . 170
18.7.1. Mesures de fiabilité des prédictions . . . 174

III Exercices 177

19. Premières manipulations avec Pandas 179
19.1. Exercice : premières manipulations de Series . . . 179
19.2. Exercice : premières manipulations de DataFrame . . . 181
19.3. Exercice : dict de dict en DataFrame . . . 183
19.4. Exercice : régression linéaire . . . 184
19.5. Le grillon rayé . . . 184
19.6. Bonus : et la cigale ? . . . 188

20. Notes et moyennes pondérées en L3 Mathématiques 191
20.1. Exercice . . . 191
20.1.1. Semestre 1 . . . 191
20.1.2. Semestre 2 . . . 191
20.1.3. Calculs à effectuer à partir des deux DataFrames . . . 192

21. Ventes et pluie : agréger, c'est choisir 197
21.1. Exercice . . . 197
21.1.1. Partie A : Ventes de deux magasins . . . 197
21.1.2. Partie B : Pluviométrie de trois villes . . . 197

22. Partie A : Ventes de deux magasins 199
22.1. 1. Charger le fichier et vérifier les types . . . 199
22.2. 2. Ajouter la colonne `montant` . . . 199
22.3. 3. Quel magasin réalise le plus gros chiffre d'affaires ? . . . 200
22.4. 4. Quel produit porte chaque magasin ? . . . 200
22.5. 5. Évolution trimestrielle . . . 201
22.6. 6. Et en nombre d'articles ? . . . 201

23. Partie B : Pluviométrie de trois villes 203
23.1. 1. Charger le fichier . . . 203
23.2. 2. et 3. Deux classements . . . 203
23.3. 4. Comparaison des deux classements . . . 204
23.4. 5. L'intensité des pluies . . . 204
23.5. 6. Tableau croisé station × mois . . . 204
23.6. 7. Le mois le plus arrosé de chaque ville . . . 205

24. Conclusion 207

25. Population des communes de France 209

26. Exercice 211
26.1. Afficher les données . . . 212
26.1.1. D'un code à un nom : une jointure . . . 213
26.2. Nettoyer les colonnes et les données . . . 218
26.2.1. Suppression de colonnes inutiles . . . 218
26.2.2. Modification des labels des colonnes . . . 219
26.2.3. Modification des valeurs (nombre d'habitants) : nettoyage et typecasting . . . 219
26.3. Aggregations . . . 221
26.3.1. Combien de départements par region ? . . . 221
26.3.2. Combien de communes par région ? . . . 222

26.3.3. Combien de communes par région et département ? . . . 224
26.3.4. Combien de personnes par region en 2021 ? . . . 225
26.4. Étude de la région PACA . . . 228
26.4.1. Combien de départements dans la région PACA ? . . . 228
26.4.2. Combien de communes dans la région PACA ? . . . 228
26.4.3. Combien de commune dans chaque département de la région PACA ? . . . 229
26.4.4. Combien de personnes habitaient en PACA en 2021 ? . . . 230
26.4.5. Combien de personnes habitaient dans chaque département de la région PACA en 2021 ? . . . 230
26.5. Étude du département VAR . . . 232
26.6. Affichages divers . . . 237

27. Émissions de CO des communes françaises 241
27.1. Les données . . . 241
27.2. Exercice . . . 241
27.2.1. 1. Découvrir les données . . . 241
27.2.2. 2. Les valeurs manquantes . . . 241
27.2.3. 3. Joindre à la population . . . 242
27.2.4. 4. Émissions par habitant . . . 242
27.2.5. 5. Cartographier . . . 242

28. 1. Découvrir les données 243

29. 2. Les valeurs manquantes 245

30. 3. Joindre à la population 247

31. 4. Émissions par habitant 249
31.1. Agréger par département . . . 250
31.2. Total ou par habitant ? . . . 251

32. 5. Cartographier 253

33. Conclusion 255

34. Netflix User Data 257
34.1. Exercice . . . 257
34.1.1. Affichages . . . 258
34.1.2. Modification des données . . . 261
34.1.3. Aggregations/Visualisations . . . 262
34.1.4. Répartition selon chaque caractéristique . . . 262
34.1.5. L'âge : une variable numérique . . . 264
34.1.6. Les dates d'inscription et de paiement . . . 265
34.1.7. Peut-on tout corréler ? Un contre-exemple . . . 268
34.1.8. La bonne façon de faire . . . 270

35. Loi de Benford et suspicion de fraudes aux présidentielles des USA en 2016 273
35.1. Exercice . . . 273
35.2. Loi de Benford . . . 274
35.3. Ouverture et séléction des données . . . 276
35.4. Bonus : pourquoi une fraude éventuelle dans ces trois états aurait changé le résultat de l'élection ? . . . 279
35.5. Sélection des données dans les **États** du Michigan, de la **Pennsylvanie** et du **Wisconsin** . . . 285
35.6. Analyse des données . . . 286

35.7. Bonus : Simulation pour la détection d'éventuelles anomalies 288
35.8. Conclusion 290

IV Downloads 291

36. Notebook Originaux et fichiers de données à télécharger 293
36.1. Cours 293
36.2. Exercices corrigés 293

Chapitre 1.

M55 - Data Science and Scientific Computing 5

Introduction à la manipulation de données avec Pandas

Ce polycopié accompagne le cours **DSSC-5** de la licence de mathématiques de l'université de Toulon. Il a un objectif simple : vous rendre capable de prendre un fichier de données que vous n'avez jamais vu, de le charger, de le nettoyer, d'en extraire une information, et de **dire ce que cette information signifie**.

Il se lit dans l'ordre du menu de gauche. Les quatre chapitres de **cours** introduisent les notions, les six **exercices corrigés** les mettent en œuvre sur des données réelles : population des communes françaises, passagers du Titanic, élections américaines. Commencez par le syllabus, qui précise le déroulement des séances, les prérequis et les modalités d'évaluation.

Tous les notebooks et les jeux de données sont téléchargeables depuis la page Fichiers sources, pour que vous puissiez les exécuter et les modifier sur votre propre machine. C'est d'ailleurs la seule façon d'apprendre : lire du code ne suffit pas, il faut essayer par soi-même, se tromper et s'auto-corriger.



Première partie

Introduction

Chapitre 2.

Syllabus du cours “Introduction à la manipulation de données avec Pandas”

2.1. Parcours “Data Science and Scientific Computing”

Depuis septembre 2024, l'université de Toulon propose, dans sa licence de Mathématiques, un parcours nommé *Data Science and Scientific Computing*. Il se situe au croisement des mathématiques et de l'informatique. L'objectif principal est d'apprendre à formuler des questions de manière à ce que le calcul informatique puisse fournir des réponses (exactes ou approchées) satisfaisantes.

Ce parcours sur trois ans fournit une base de connaissances et de compétences pratiques en analyse de données et en calcul scientifique, en utilisant principalement **Python** et les **Notebook Jupyter**. On y apprend à manipuler, analyser et visualiser les données, ainsi qu'à résoudre des problèmes scientifiques et mathématiques complexes en utilisant les outils et bibliothèques appropriés :

- **DSSC-1** *Introduction à la programmation scientifique avec Python* : apprentissage des bases de la programmation en Python, notamment les structures de contrôle, les fonctions et les types de données, ainsi que l'utilisation de Matplotlib pour créer des visualisations.
- **DSSC-2** *Approximations et simulations* : étude des bases de l'analyse numérique, avec des comparaisons aux résultats issus de la bibliothèque SciPy, pour résoudre des problèmes scientifiques et d'ingénierie : interpolations, intégrations numériques, zéros d'une fonction. On apprendra également à utiliser des Notebook Jupyter pour expliquer et documenter les calculs.
- **DSSC-3** *Calcul formel avec SymPy* : familiarisation avec SymPy pour réaliser des calculs mathématiques symboliques comme la manipulation d'expressions et la résolution d'équations et sytèmes. Application en analyse mathématique (calcul de limites, dérivées, intégrales symboliques dans $\mathbb{R}^n$) et en algèbre linéaire (applications linéaires, noyeaux, etc.). On apprendra également à rédiger des rapports en LaTeX.
- **DSSC-4** *Manipulation avancée des tableaux avec NumPy* : approfondissement des connaissances sur le calcul matriciel avec NumPy pour des calculs performants.
- **DSSC-5** *Manipulation de données avec Pandas : une initiation* : utilisation de Pandas pour manipuler efficacement les jeux de données, incluant l'importation, le nettoyage, la transformation, le filtrage, l'agrégation et la gestion des valeurs manquantes.
- **DSSC-6** *Approximations d'équations différentielles* : étude des schémas numériques pour l'approximation de problèmes de Cauchy.

2.2. Déroulement du cours DSSC-5 : Introduction à la manipulation de données avec Pandas

Ce polycopié est associé au cinquième cours, initiant à l'utilisation de la bibliothèque pandas. Les objectifs de ce cours sont

- comprendre les notions de base de Pandas et de la manipulation de données ;
- importer et exporter des données ;
- manipuler des données : filtrer, trier, grouper, agréger, fusionner, joindre, concaténer, etc. ;
- visualiser des données.

2.2.1. Plan des séances

Le cours se déroule en **5 séances de 3 heures** dans le bâtiment U1. Chaque séance commence par le cours ; le temps restant est consacré aux exercices associés, dont le corrigé figure dans ce polycopié.

Séance	Cours	Exercices associés
1	**Premiers pas avec Pandas** et l'objet `Series` : création, indexation par étiquettes, `loc` et `iloc`, visualisations avec Matplotlib, Seaborn et Plotly	Premières manipulations ; Notes et moyennes pondérées en L3
2	**Les `DataFrame`** : création, accès aux éléments, ajout et suppression, filtrage, agrégations avec `groupby`, tableaux croisés	Ventes et pluie : agréger, c'est choisir ; Population des communes de France
3	**Étude de cas complète : le Titanic**, de l'importation du fichier à l'analyse multivariée	
4		Loi de Benford et élections présidentielles USA 2016 ; Émissions de CO des communes ; Netflix User Data
5	**Évaluation** : examen sur machine	

Tous les supports de cours sont fournis sous forme de notebooks Jupyter et téléchargeables depuis la page en annexe (menu a gauche ci-contre).

Pour plus d'informations sur Pandas, vous pouvez également consulter ce livre (un peu daté) : Python Data Science Handbook.

2.2.2. Prérequis

- Bases de Python : types primitifs (int, float, str), structures de données (listes, tuples, dictionnaires), conditions, boucles, fonctions, modules (numpy, matplotlib.pyplot) ; cf. **DSSC-1** ;
- Manipulation de Notebook Jupyter : cellules python *vs* markdown, écriture de formules mathématiques en LaTeX, exécution, sauvegarde, export, etc. ; cf. **DSSC-2** ;
- Connaissances de base de NumPy : manipulation de tableaux et extraction de sous-tableaux par *slices* ; cf. **DSSC-4**.

Pour manipuler les notebooks Jupyter, on utilisera l'éditeur VS Codium (au lieu de Jupyter Notebook dans le navigateur). Dans les salles de TP, Python 3.12 est installé.

2.2.3. Évaluation

Le cours est évalué en **contrôle continu**, sous la forme d'un **examen sur machine** qui se déroule pendant la **cinquième séance**.

2.2.3.1. Déroulement

- Les postes de la salle de TP sont **déconnectés d'Internet**. Seul **Moodle** reste accessible, pour télécharger le sujet et le(s) fichier(s) de données, puis déposer votre travail en fin d'épreuve.
- Vous travaillez dans un **notebook Jupyter** sous VS Codium, avec Python 3.12 et les modules vus en cours (`pandas`, `numpy`, `matplotlib`, `seaborn`, `plotly`).
- Durée : **2 heures**.
- Aucun assistant de code (IA générative) n'est disponible ni autorisé.

> **Documents autorisés**
>
> Moodle restant accessible, vous pouvez consulter pendant l'épreuve :
> - **ce polycopié** et les **notebooks du cours et des exercices**, déposés sur Moodle ;

- **vos propres notebooks** de TP.

Aucune autre ressource en ligne n'est accessible : ni moteur de recherche, ni documentation Pandas en ligne, ni assistant de code.

2.2.3.2. Ce qui est évalué

L'examen porte sur les compétences travaillées lors des quatre premières séances, appliquées à un jeu de données que vous découvrez le jour même :

- importer un fichier de données et en faire un premier examen : dimensions, types des colonnes, valeurs manquantes, statistiques descriptives ;
- sélectionner, filtrer et trier des lignes et des colonnes ;
- créer de nouvelles colonnes à partir des colonnes existantes ;
- agréger et croiser des variables (`groupby`, `pivot_table`, `crosstab`) ;
- produire des visualisations pertinentes et **les commenter** ;
- rédiger en cellules Markdown des conclusions courtes et justifiées par les résultats obtenus.

Il ne s'agit pas d'un examen de programmation Python générale : concentrez-vous sur l'utilisation de Pandas pour extraire et communiquer de l'information pertinente.

2.2.3.3. Rendu

Déposez sur Moodle le notebook `.ipynb`, nommé `NOM_Prenom.ipynb`. Avant de déposer, **exécutez toutes les cellules dans l'ordre** (*Restart Kernel and Run All Cells*) et vérifiez qu'aucune ne produit d'erreur : un notebook qui ne s'exécute pas intégralement est fortement pénalisé.

2.2.3.4. Comment s'y préparer

Les études de cas et les exercices corrigées de ce polycopié constituent le meilleur entraînement possible : refaites-les **sans regarder le corrigé**, puis comparez.

2.2.4. Ressources pour trouver des jeux de données

- UCI Machine Learning Repository
- Kaggle
- Google Dataset Search
- Data.gov
- Open Data EU
- Data World
- Football CSV

Sur vos ordinateurs personnels, je vous conseille d'installer Visual Studio Code et bien sûr une distribution Python.

Si vous ne pouvez pas installer une distribution Python, vous pouvez utiliser Google Colab. Dans ce cas, charger un fichier csv nécessite une étape supplémentaire. Le plus simple est de télécharger directement le fichier dans le notebook en exécutant le code suivant. On vous demandera de sélectionner un fichier. Cliquez sur «Choisir les fichiers» puis sélectionnez et téléchargez le fichier csv de votre choix. Vous devriez voir le nom du fichier affiché une fois que Colab l'aura téléchargé.

```
from google.colab import files
uploaded = files.upload()
```

Deuxième partie

Cours

Chapitre 3.

Premiers pas avec Pandas

Dans le cours DS&SC-4, nous avons étudié `NumPy`, dont l'objet principal, le `ndarray`, permet de stocker et de manipuler efficacement des **tableaux homogènes**. Cet objet, idéal pour les valeurs numériques (et donc la mise en œuvre de schémas d'approximation), montre ses limites dès qu'il s'agit de données variées (catégories, libellés, dates, texte).

En analyse de données, on organise souvent l'information sous forme de lignes (individus) et de colonnes (variables), qui ne sont pas nécessairement numériques. Exemples :

- dans l'automobile, chaque individu est une voiture, avec des caractéristiques telles que la puissance du moteur, les dimensions, la marque, le modèle, la couleur, etc. ;
- en grande distribution, chaque individu est un produit, avec des informations comme le prix et la catégorie ;
- dans le secteur bancaire, chaque individu est une personne, avec des informations comme le salaire moyen, l'âge et les mensualités d'un prêt.

Nous avons donc besoin d'un outil capable de :

- représenter les données au format individus $\times$ variables ;
- manipuler des types hétérogènes ;
- lire et écrire des données depuis diverses sources.

C'est exactement le rôle de la bibliothèque `pandas`.

Objectifs d'apprentissage

- Identifier les objets de base de `pandas` (Series, DataFrame) et leur lien avec `NumPy`.
- Lire et écrire des données (CSV) et comprendre les types hétérogènes.
- Sélectionner, filtrer, trier, agréger et fusionner des données.
- Calculer et interpréter des statistiques descriptives (describe, moyennes, écarts-types, quantiles).
- Visualiser des données avec `Matplotlib`, `Seaborn`, `Plotly` et choisir une représentation adaptée.

3.1. Importation de pandas

Une fois `pandas` installé, vérifions que tout fonctionne correctement. La communauté Python a adopté quelques conventions de nommage pour les modules courants :

```
import pandas as pd
import numpy as np
#
import matplotlib.pyplot as plt
#
import seaborn as sns
# Pour des graphiques interactifs, notamment en 3D
import plotly.express as px

from IPython.display import display, Markdown
```

Définissons quelques options par défaut pour la création des figures :

```
plt.rcParams['font.size'] = 15
plt.rcParams['figure.figsize'] = (15, 7) # Taille par défaut des figures
# plt.style.use('ggplot') # set ggplot style


sns.set_theme(context = "notebook", # paper, notebook, talk, poster
              style = "darkgrid", # ticks, whitegrid, darkgrid, white, dark
              palette = "pastel", # None, Set2, husl, deep, muted, bright, pastel, dark, colorblind
              rc = {"axes.spines.right": False,
                    "axes.spines.top": False}
             )
```

Quelques remarques sur les bibliothèques de visualisation. Trois options populaires se distinguent : `Matplotlib`, `Seaborn` et `Plotly`, chacune avec ses forces et particularités.

- **Matplotlib** est la bibliothèque de base en Python. Elle offre une grande flexibilité et un contrôle fin de tous les aspects des graphiques, des tracés simples aux visualisations complexes. Elle est idéale pour une personnalisation poussée et un large éventail de types de graphiques. Cette flexibilité s'accompagne toutefois d'une complexité accrue et demande souvent davantage de code.

- **Seaborn**, construit sur Matplotlib, simplifie la création de graphiques statistiques élégants. Il propose des thèmes et des palettes intégrés qui rendent les visualisations plus attrayantes et informatives. En contrepartie, les personnalisations très spécifiques peuvent être plus limitées.

Avec Matplotlib et Seaborn, les figures sont statiques : toute l'information doit être visible dans la figure, ce qui peut la surcharger. On peut concevoir des visualisations à plusieurs niveaux : un premier niveau « coup d'œil » pour les messages clés, puis des détails supplémentaires. Les visualisations interactives, devenues la norme en dataviz, facilitent cette approche : au survol de la souris, on obtient des informations complémentaires (par exemple, les valeurs exactes).

- **Plotly**, basé sur JavaScript, permet de créer des visualisations interactives et dynamiques. Il prend en charge les graphiques 3D et les cartes, ce qui le rend très puissant.

3.2. Une analyse complète, en quinze lignes

Ne cherchez pas à comprendre le code de cette section. Il met en œuvre à peu près tout ce que nous verrons pendant les quatre séances : lecture d'un fichier, inspection, création de colonnes, agrégation, visualisation.

L'objectif ici est seulement de voir **à quoi ressemble une analyse de données complète**, du fichier brut à la conclusion : et de constater qu'elle tient en une quinzaine de lignes. Vous saurez tout écrire d'ici la fin du cours.

Reprenons l'exemple de l'automobile évoqué plus haut. Le fichier `voitures.csv` décrit 32 modèles testés par le magazine *Motor Trend* en 1974 : un jeu de données classique en statistique.

3.2.1. 1. Lire le fichier

```
voitures = pd.read_csv('voitures.csv', index_col=0)
voitures.head()
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2

Premier constat, et il est général : **les données réelles n'arrivent jamais documentées.** Les noms de colonnes sont des abréviations, en anglais, sans unités. Avant toute chose, il faut savoir ce qu'elles désignent :

Colonne	Signification	Unité
`mpg`	consommation	miles par gallon
`cyl`	nombre de cylindres	
`disp`	cylindrée	pouces cubes
`hp`	puissance	chevaux
`wt`	poids	milliers de livres
`qsec`	temps sur le quart de mile	secondes
`am`	boîte de vitesses	0 = automatique, 1 = manuelle

Nous n'utiliserons que `mpg`, `cyl` et `hp`.

3.2.2. 2. Regarder la forme du jeu de données

```
display(Markdown(f"{voitures.shape[0]} voitures, {voitures.shape[1]} caractéristiques"))
voitures[['mpg', 'cyl', 'hp']].describe().round(1)
```

32 voitures, 11 caractéristiques

	mpg	cyl	hp
count	32.0	32.0	32.0
mean	20.1	6.2	146.7
std	6.0	1.8	68.6
min	10.4	4.0	52.0
25%	15.4	4.0	96.5
50%	19.2	6.0	123.0
75%	22.8	8.0	180.0
max	33.9	8.0	335.0

3.2.3. 3. Créer une colonne exploitable

La consommation est donnée en *miles par gallon* : une unité où plus le nombre est grand, moins la voiture consomme. Convertissons-la en litres aux 100 km :

$$\text{L/100 km} = \frac{235{,}215}{\text{mpg}}$$

```
voitures['conso'] = 235.215 / voitures['mpg']

voitures[['cyl', 'hp', 'conso']].head()
```

	cyl	hp	conso
Mazda RX4	6	110	11.200714
Mazda RX4 Wag	6	110	11.200714
Datsun 710	4	93	10.316447
Hornet 4 Drive	6	110	10.991355
Hornet Sportabout	8	175	12.578342

```
display(Markdown(f"La plus sobre    : {voitures['conso'].idxmin():22} {voitures['conso'].min():.1f} L/100
↪  km"))
display(Markdown(f"La plus gourmande : {voitures['conso'].idxmax():22} {voitures['conso'].max():.1f}
↪  L/100 km"))
```

La plus sobre : Toyota Corolla 6.9 L/100 km

La plus gourmande : Cadillac Fleetwood 22.6 L/100 km

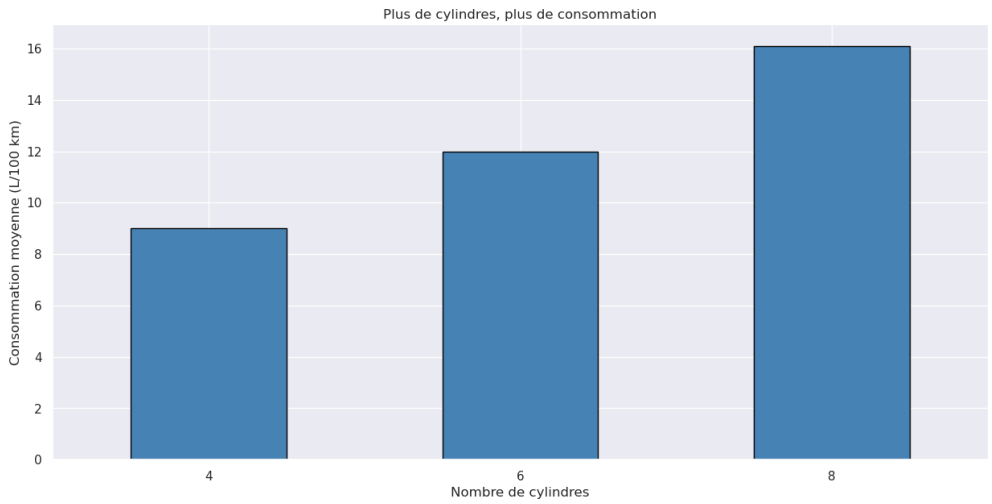
3.2.4. 4. Agréger : la consommation dépend-elle du nombre de cylindres ?

```
resume = voitures.groupby('cyl').agg(
    nombre            = ('conso', 'size'),
    conso_moyenne     = ('conso', 'mean'),
    puissance_moyenne = ('hp',    'mean')
).round(1)

resume
```

cyl	nombre	conso_moyenne	puissance_moyenne
4	11	9.0	82.6
6	7	12.0	122.3
8	14	16.1	209.2

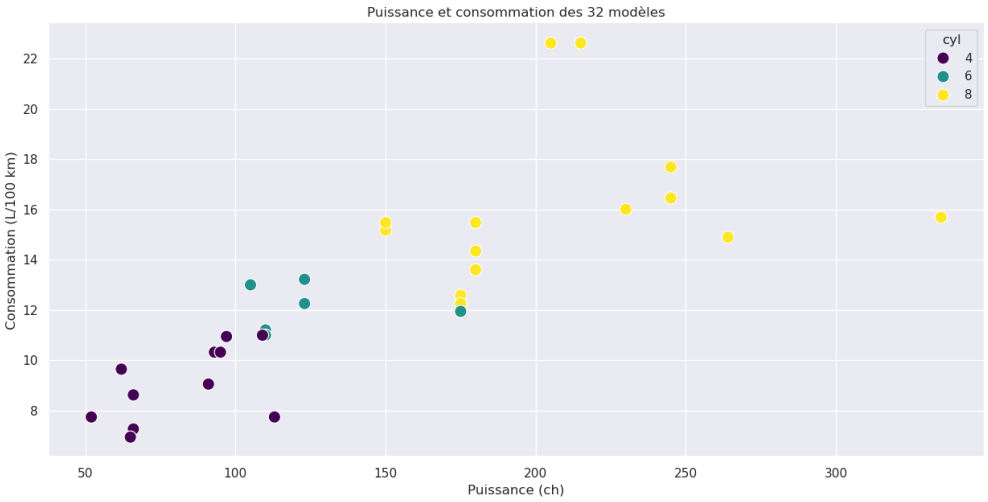
```
resume['conso_moyenne'].plot(kind='bar', color='steelblue', edgecolor='black', rot=0)
plt.xlabel("Nombre de cylindres")
plt.ylabel("Consommation moyenne (L/100 km)")
plt.title("Plus de cylindres, plus de consommation")
plt.show()
```



3.2.5. 5. Visualiser la relation complète

```
sns.scatterplot(data=voitures, x='hp', y='conso', hue='cyl', palette='viridis', s=120)
plt.xlabel("Puissance (ch)")
plt.ylabel("Consommation (L/100 km)")
plt.title("Puissance et consommation des 32 modèles")
plt.show()

display(Markdown(f"Corrélation entre puissance et consommation :
↪  {voitures['hp'].corr(voitures['conso']):.2f}"))
```



corrélation entre puissance et consommation : 0.76

3.2.6. 6. Conclure

Une 4 cylindres consomme en moyenne **9,0 L/100 km**, une 6 cylindres **12,0** et une 8 cylindres **16,1** : la consommation augmente de près de 80 % entre les deux extrêmes. Le nuage de points montre que la puissance explique largement ce phénomène : les deux grandeurs sont corrélées à **0,76** : et que le nombre de cylindres n'en est qu'un indicateur commode.

Rien de surprenant pour qui s'intéresse aux voitures. Ce qui doit retenir votre attention, c'est le **chemin parcouru** : d'un fichier texte de 32 lignes nous sommes arrivés à une affirmation chiffrée et à deux graphiques, en une quinzaine d'instructions.

3.2.7. Ce que nous venons de faire, et où vous l'apprendrez

Étape	Outil	Où
Lire un fichier de données	`read_csv`	chapitre *DataFrames*
Afficher, décrire, inspecter	`head`, `shape`, `describe`	chapitre *DataFrames*
Sélectionner des colonnes	`df[['a','b']]`	chapitres *Series* et *DataFrames*
Créer une colonne calculée	`df['new'] = ...`	chapitre *DataFrames*
Trouver un extremum	`idxmin`, `idxmax`	chapitre *DataFrames*
Regrouper et agréger	`groupby`, `agg`	chapitre *DataFrames*
Visualiser	`plot`, `seaborn`	tout au long du cours
Interpréter	votre tête	cours d'analyse de données
Prévoir	-	cours de modélisation

Chapitre 4.

Introduction aux structures de données pandas : les Series (tableaux unidimensionnels)

Nous allons découvrir deux structures de données de base de la bibliothèque Pandas : les `Series`, les `DataFrames` (éventuellement les `Index`).

Commençons par importer la bibliothèque Pandas (et la bibliotèque NumPy) avec les alias classiques `pd` et `np` :

```
import pandas as pd
import numpy as np
```

Un objet `Series` (une série) est un **tableau à une dimension** capable de stocker n'importe quel type de données (entiers, chaînes de caractères, flottants, objets Python, etc.).
Elle se distingue d'un ndarray NumPy par son indexation. Dans Numpy, l'indexation est implicite (pour accéder à une donnée on indique la position). Avec une Series, on peut bien sûr utiliser un indice de position mais on peut surtout faire appel à des indices plus explicites identifiés par des labels. Ces labels peuvent être des chaînes de caractères, des dates ou encore des nombres et peuvent être non ordonnés et non uniques.

```
# ?pd.Series
```

Chapitre 5.

Création d'une Series

On peut créée une série à partir

- d'une liste de valeurs ou d'un tableau NumPy (les labels sont alors générées automatiquement)
- deux listes : une liste (ou array NumPy) de valeurs et une liste de labels
- d'un dictionnaire

5.1. À partir d'une liste/array de valeurs

```
ma_series = pd.Series( data=[0.25, 0.5, 0.75, 1.0] )
display(ma_series)
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

On voit qu'une Series contient une séquence de valeurs et une séquence d'indices (ici générés automatiquement). Pour accéder à ces deux séquences, on peut utiliser les attributs `values` et `index` de la Series.

```
display(ma_series.values) # c'est un tableau numpy
display(ma_series.index) # c'est un objet de type Index, créé ici automatiquement
```

```
array([0.25, 0.5 , 0.75, 1.  ])
```

```
RangeIndex(start=0, stop=4, step=1)
```

On peut accéder à un élément de la série par son `index` (qui est un entier crée automatiquemnt ou passé explicitement) :

```
ma_series[1]
```

```
0.5
```

5.2. À partir de deux listes/array : une liste de valeurs et une liste d'index associés

```
ma_serie1 = pd.Series( data=[8, 70, 320, 1200],
                       index=["Suisse", "France", "USA", "Chine"])
ma_serie1
```

```
Suisse       8
France      70
USA        320
Chine     1200
dtype: int64
```

```
valeurs = ma_serie1.values   # c'est un tableau numpy
etiquettes = ma_serie1.index # c'est un objet de type Index
display(valeurs)
display(etiquettes)
```

```
array([   8,   70,  320, 1200])
```

```
Index(['Suisse', 'France', 'USA', 'Chine'], dtype='object')
```

On peut accéder à une valeur en utilisant son label, un peu comme dans un dictionnaire.

```
ma_serie1["France"]
```

```
70
```

Les objets `Index` (i.e. les labels) sont immutables et ne peuvent donc pas être modifiés par l'utilisateur. On peut accéder à une label selon sa position :

```
etiquettes[1]
```

```
'France'
```

Un `Index` **peut contenir des étiquettes dupliquées**. Les sélections avec des étiquettes en double renverront toutes les occurrences de cette étiquette.

```
ma_serie1 = pd.Series( data=[8, 70, 320, 1200 , 2000],
                       index=["Suisse", "France", "USA", "Chine", "France"])
display(ma_serie1)
display(ma_serie1["France"]) # renvoie les deux valeurs associées à l'étiquette "France"
```

```
Suisse       8
France      70
USA        320
Chine     1200
France    2000
dtype: int64
```

```
France      70
France    2000
dtype: int64
```

5.3. À partir d'un dictionnaire { label1: valeur1, label2: valeur2, ...}

```
ma_serie2 = pd.Series( { "France" : 70, "USA" : 320, "Suisse" : 8,"Chine" : 1200})
ma_serie2
```

```
France      70
USA        320
Suisse       8
Chine     1200
dtype: int64
```

```
ma_serie2["France"]
```

```
70
```

Attention, comme les éléments d'un dictionnaire ne sont pas ordonnés, dans l'objet Series les entrées seront ordonnées selon l'ordre d'ajout. On peut ensuite ordonner les éléments de la Series en utilisant la méthode `sort_index()`.

```
ma_serie2.sort_index()
```

```
Chine     1200
France      70
Suisse       8
USA        320
dtype: int64
```

Chapitre 6.

Slicing $\rightsquigarrow$ sélection avec `loc` et `iloc`

Pour l'indexation par étiquettes des DataFrame sur les lignes, les opérateurs d'indexation spéciaux `loc` et `iloc` permettent de sélectionner un sous-ensemble de lignes et de colonnes d'un DataFrame avec une notation de type NumPy en utilisant soit des étiquettes d'axes (`loc`), soit des entiers (`iloc`).

Une serie peut être vue comme un **dictionnaire ordonné** ou comme une **liste**, ainsi on peut accéder à un élément en utilisant son label ou son indice.

Considérons le cas suivant :

```
data = pd.Series(  data  = [0.25, 0.5, 0.75, 1.0],
                   index = ['a' , 'b', 'c' , 'd'])
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

```
# Si on voit la Series comme un dictionnaire :

display(data['b'])  # On peut accéder à un élément par son index
display('a' in data)  # On peut tester l'existence d'une clé
display(data.keys()) # On peut récupérer les index
display(list(data.items())) # On peut récupérer les couples (index, valeur)
data['e'] = 1.25 # On peut ajouter un élément
display(data)
```

```
0.5
```

```
True
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

```
# Si on voit la Series comme un tableau numpy/liste dont les indices peuvent être des chaînes :

display(data[1])  # On peut accéder à un élément par sa position
display(data[ ['a', 'e'] ]) # les éléments dont les indices sont 'a' et 'e'
display(data[(data > 0.3) & (data < 0.8)]) # masque : `&` pour le `and`, `|` pour le `or` et `~` pour le
 ↪  `not` ;  parenthèses obligatoires
```

```
/tmp/ipykernel_482666/2399729980.py:3: FutureWarning: Series.__getitem__ treating keys as positions i
  display(data[1])  # On peut accéder à un élément par sa position

0.5

a    0.25
e    1.25
dtype: float64

b    0.50
c    0.75
dtype: float64
```

Cette double vision des indices peut être source de confusion avec le slicing :

- si on utilise des indices automatiques, le slicing se fait comme dans Numpy (premier inclus, dernier exclu),
- si on utilise comme indices des labels, le dernier est aussi inclus.

```
display(data)
display(data[0:2]) # slicing explicite donc on prend les éléments de 0 inclus à 2 exclus
display(data['a':'c']) # slicing implicite donc on prend les éléments de 'a' à 'c' inclus
```

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64

a    0.25
b    0.50
dtype: float64

a    0.25
b    0.50
c    0.75
dtype: float64
```

Pire, si les indices sont des entiers, on ne sait pas si pandas interprètes ces nombre comme des indices ou des labels pour le slicing :

```
data2 = pd.Series( data=[0.25, 0.5, 0.75, 1.0],
                   index=[1000, 1100, 1200, 1300] )
data2
```

```
1000    0.25
1100    0.50
1200    0.75
1300    1.00
dtype: float64
```

```
display(data2[0:2])       # slicing explicite donc on prend les éléments de la poisition 0 incluse à la
 ↪  position 2 exclue
display(data2[1000:1200]) # slicing qui semble implicite, mais comme les indices sont des entiers, il
 ↪  agit comme un slicing explicite donc on prend les éléments de 1000 inclus à 1200 exclu
```

```
1000    0.25
1100    0.50
dtype: float64
```

```
Series([], dtype: float64)
```

La solution : `loc` et `iloc` :

- `loc` permet d'accéder aux éléments en utilisant les labels,
- `iloc` permet d'accéder aux éléments en utilisant les indices numériques.

```
data2.loc[1100] # la valeur de l'élément de label 1100
```

```
0.5
```

```
data2.iloc[1] # la valeur de l'élément d'indice 1
```

```
0.5
```

```
data2.iloc[:3] # les 3 premiers éléments, donc les éléments d'indice 0, 1 et 2
```

```
1000    0.25
1100    0.50
1200    0.75
dtype: float64
```

Chapitre 7.

Ajouter des éléments à une Series / Fusionner deux Series

Pour ajouter un élément à une Series, on peut utiliser la méthode `concat()` :

```
ma_serie1 = pd.Series( data=[8, 70, 320, 1200],
                       index=["Suisse", "France", "USA", "Chine"] )
ma_serie2 = pd.Series( data=[1,4,3,10],
                       index=["France", "USA", "Suisse", "Italie"] )
ma_serie3 = pd.concat([ma_serie1, ma_serie2])
ma_serie3
```

```
Suisse       8
France      70
USA        320
Chine     1200
France       1
USA          4
Suisse       3
Italie      10
dtype: int64
```

On note que certaines labels sont dupliquées, ce qui est autorisé dans une Series.

Chapitre 8.

Supprimer des éléments à une Series

Pour supprimer un élément d'une Series, on peut utiliser la méthode `drop()` et spécifier l'index de l'élément à supprimer.

```
ma_serie3.drop("France")
```

```
Suisse       8
USA        320
Chine     1200
USA          4
Suisse       3
Italie      10
dtype: int64
```

Par default la méthode drop ne modifie pas la Series, mais renvoie une nouvelle Series sans l'élément supprimé. Pour modifier la Series, on peut utiliser l'argument `inplace=True`.

```
ma_serie3
```

```
Suisse       8
France      70
USA        320
Chine     1200
France       1
USA          4
Suisse       3
Italie      10
dtype: int64
```

```
ma_serie3.drop("France", inplace=True)
ma_serie3
```

```
Suisse       8
USA        320
Chine     1200
USA          4
Suisse       3
Italie      10
dtype: int64
```

Chapitre 9.

Tableau comparatif entre `list`, `dict`, `numpy.ndarray` et `pandas.Series`

Aspect	`list`	`dict`	`np.ndarray`	`pd.Series`
Type d'accès	Par **indice entier**	Par **clé (=label)**	Par **indice entier**	Par **label** (`loc`) ou **position** (`iloc`)
Slicing	`lst[1:3]`	Impossible	`arr[1:3]`	`s.iloc[1:3]` (par positions) `s.loc["a":"c"]` (par labels)
Homogénéité	Hétérogène permis	Hétérogène permis	Homogène	Souvent homogène
Opérations vectorisées	Non	Non	Oui	Oui
Utilisation typique	Séquences	Mapping clé→valeur	Calcul scientifique	Données tabulaires 1D indexées

Note importante :

- `iloc[1:4]` exclut la fin → l'élément d'index 4 est non inclus.
- `loc["b":"d"]` inclut la fin → l'élément de label "d" est inclus.

```
# LISTE
# ====================================

lst = [10, 20, 30, 40, 50]

# Accès
display(lst[0])

# Slicing
display(lst[1:4])
```

```
10
```

```
[20, 30, 40]
```

```
# DICTIONNAIRE
# ====================================

d = {"a": 10, "b": 20, "c": 30}

# Accès
```

```
display(d["a"])
# d[0]        # Pas d'accès positionnel

# d["a":"c"]  # Pas de slicing
```

```
10
```

```
# TABLEAU NUMPY
# ====================================
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Accès
display(arr[0])

# Slicing
display(arr[1:4])
```

```
10
```

```
array([20, 30, 40])
```

```
# PANDAS SERIES
# ====================================
import pandas as pd

s = pd.Series([20, 40, 30, 50, 20],
              index=["a", "b", "c", "d", "e"])

# Accès
display(s.iloc[0]) # par position
display(s["a"])    # par label

# Slicing
display(s.iloc[1:4])    # par positions
display(s.loc["b":"d"]) # par labels (INCLUSIF !)
```

```
20
```

```
20
```

```
b    40
c    30
d    50
dtype: int64
```

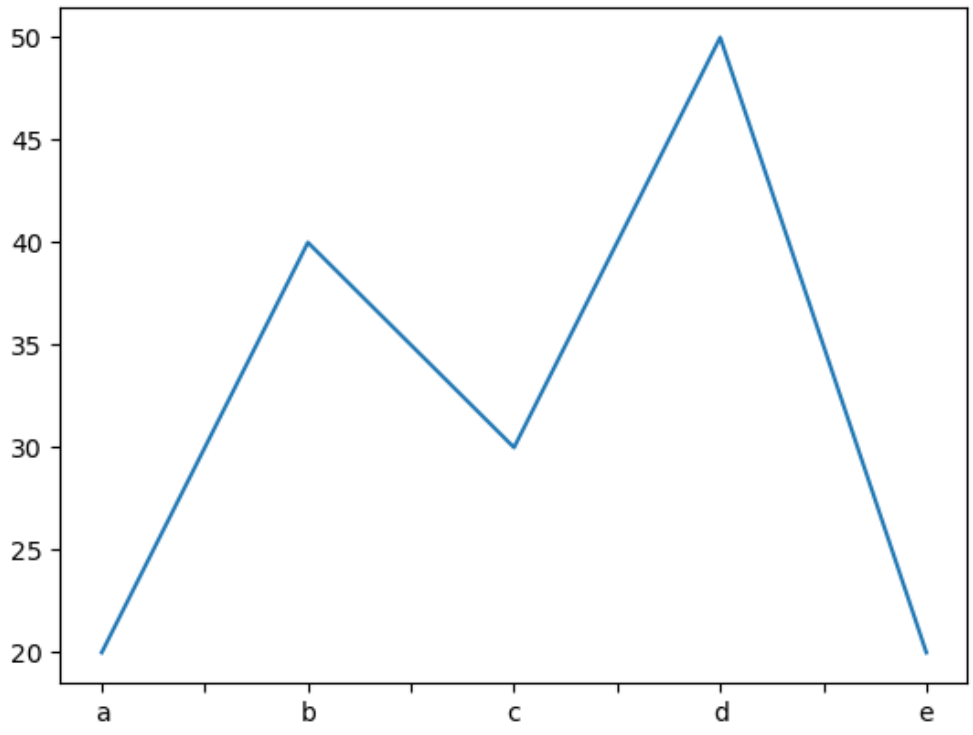
```
b    40
c    30
d    50
dtype: int64
```

Chapitre 10.

Représentation graphique d'une Series

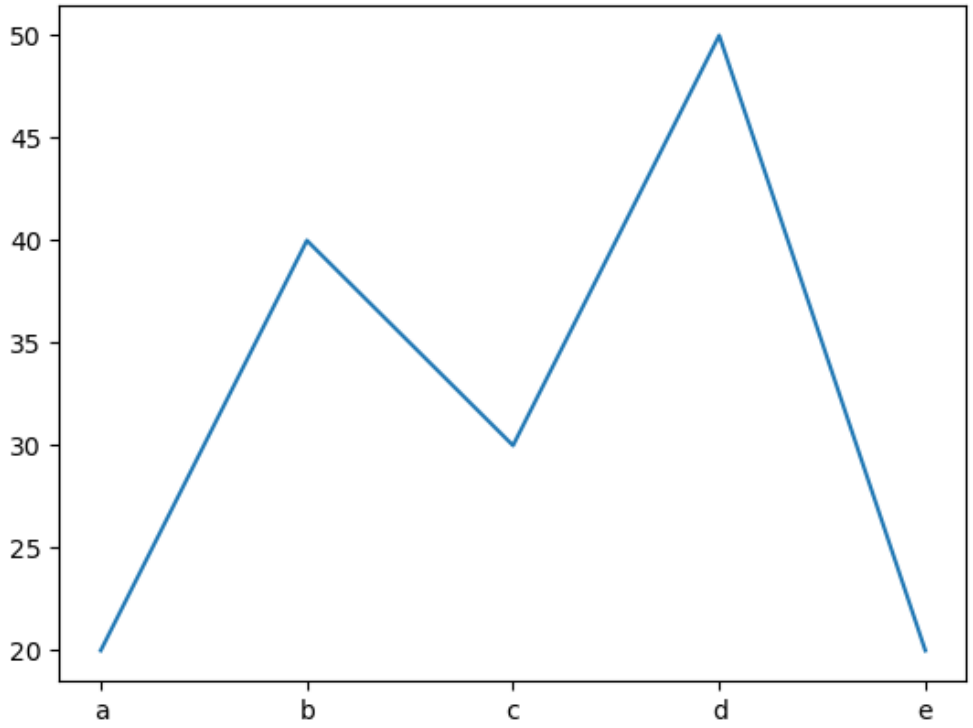
On peut appliquer la méthode `plot()` directement à une Serie :

```
s.plot();
```



Le code équivalent avec matplotlib serait :

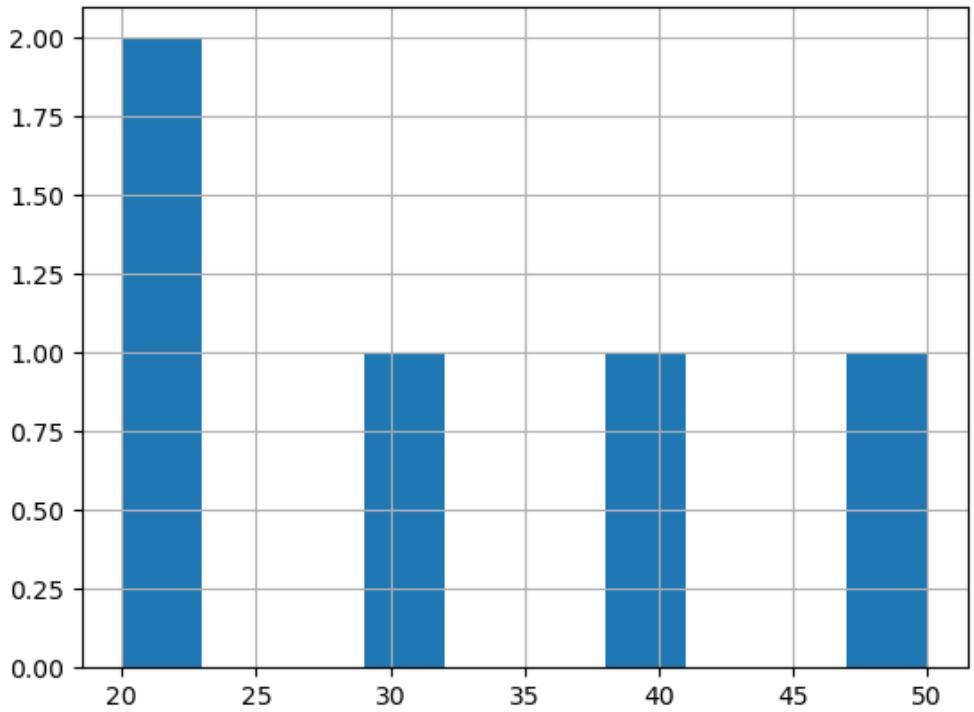
```
import matplotlib.pyplot as plt
plt.plot(s.index, s.values);
```



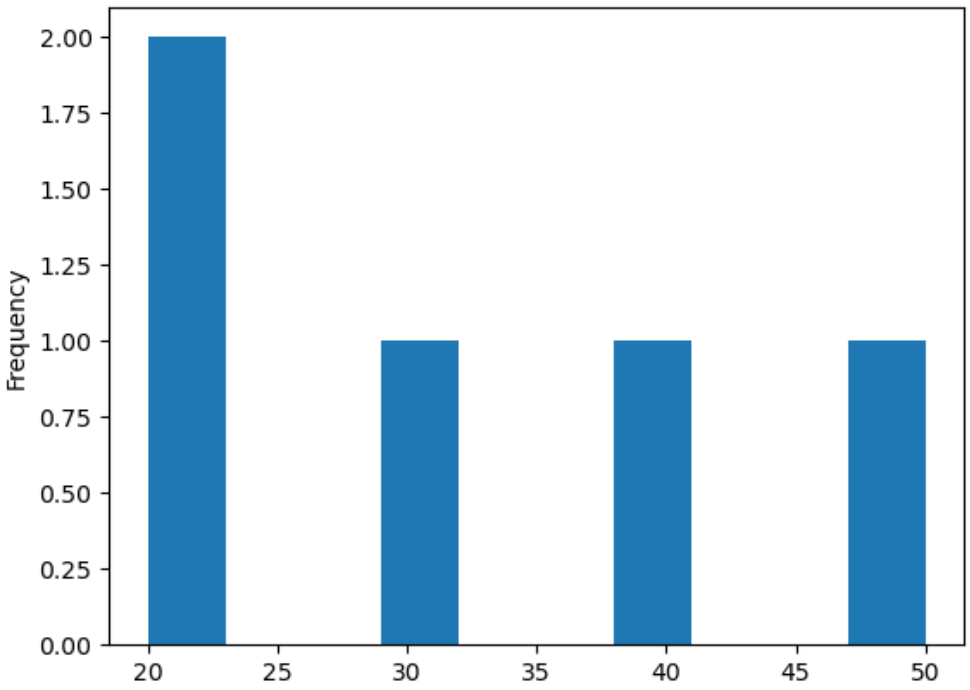
On peut personnaliser le graphique en utilisant les arguments de la méthode `plot()` de pandas ou en utilisant les fonctions de matplotlib après avoir récupéré les données de la Series.

On peut tracer différents types de graphiques en utilisant l'argument `kind` de la méthode `plot()` : `bar` pour un graphique en barres, `hist` pour un histogramme, `box` pour un diagramme en boîte, etc.

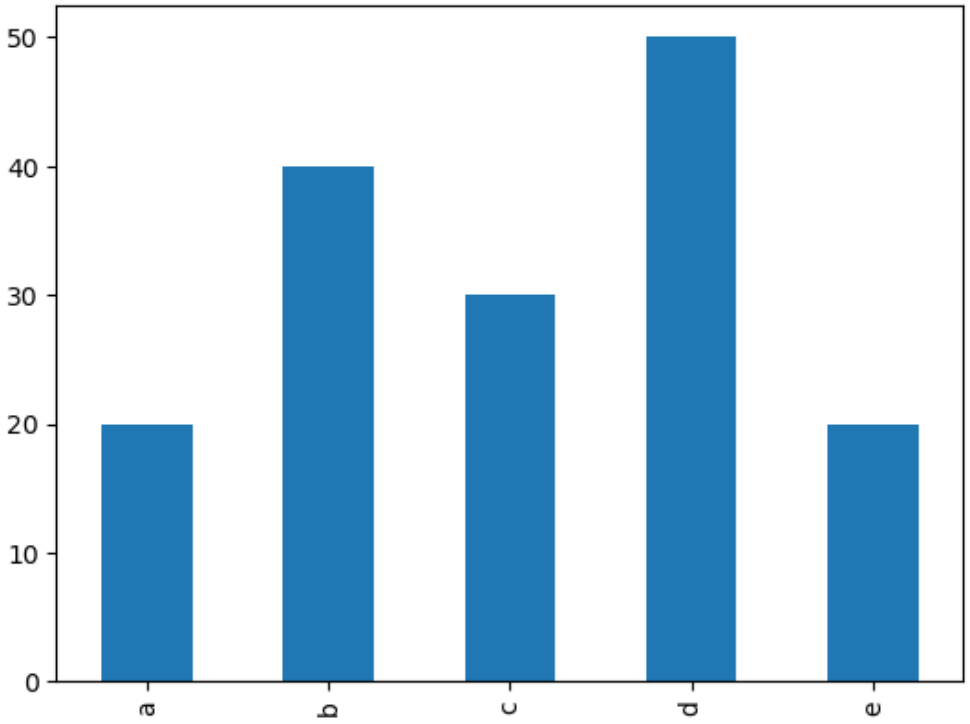
```
s.hist();
```




```
s.plot(kind='hist');
```




```
s.plot(kind='bar');
```



Ou encore avec plotly :

```
import plotly.express as px

# Pour la génération automatique de graphiques interactifs dans le polycopié
import plotly.io as pio
import os
if os.getenv("QUARTO_OUTPUT_FORMAT") is not None:
    # Export Quarto HTML
```

```
    pio.renderers.default = "iframe_connected"
else:
    # Notebook interactif
    pio.renderers.default = "notebook_connected"

px.bar(x=s.index, y=s.values)
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Chapitre 11.

Introduction aux structures de données pandas : les DataFrames (tableaux bidimensionnels)

Commençons par importer la bibliothèque Pandas (et la bibliotèque NumPy) avec les alias classiques `pd` et `np` :

```
import pandas as pd
import numpy as np

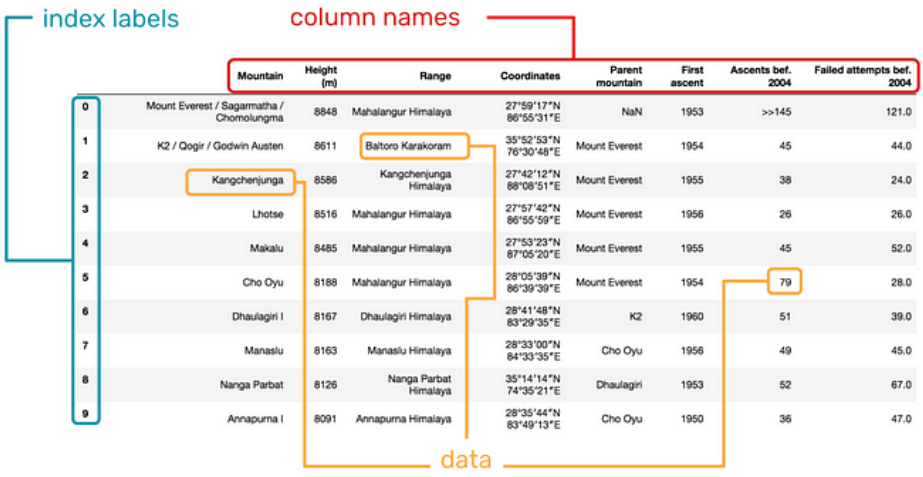
# Pour l'affichage dans les notebooks
from IPython.display import display, Markdown
```

La deuxième structure fondamentale de Pandas est le `DataFrame`. Elle peut être considérée comme une généralisation d'une matrice NumPy, où les lignes et les colonnes sont identifiées par des étiquettes, ou encore comme un dictionnaire de Series partageant le même index.

Contrairement à une matrice, les colonnes peuvent être de types différents.

Un DataFrame est composé des éléments suivants :

- l'indice de la ligne ;
- le nom de la colonne ;
- la valeur de la donnée ;



	Mountain	Height (m)	Range	Coordinates	Parent mountain	First ascent	Ascents bef. 2004	Failed attempts bef. 2004
0	Mount Everest / Sagarmatha / Chomolungma	8848	Mahalangur Himalaya	27°59'17"N 86°55'31"E	NaN	1953	>>145	121.0
1	K2 / Qogir / Godwin Austen	8611	Baltoro Karakoram	35°52'53"N 76°30'48"E	Mount Everest	1954	45	44.0
2	Kangchenjunga	8586	Kangchenjunga Himalaya	27°42'12"N 88°08'51"E	Mount Everest	1955	38	24.0
3	Lhotse	8516	Mahalangur Himalaya	27°57'42"N 86°55'59"E	Mount Everest	1956	26	26.0
4	Makalu	8485	Mahalangur Himalaya	27°53'23"N 87°05'20"E	Mount Everest	1955	45	52.0
5	Cho Oyu	8188	Mahalangur Himalaya	28°05'39"N 86°39'39"E	Mount Everest	1954	79	28.0
6	Dhaulagiri I	8167	Dhaulagiri Himalaya	28°41'48"N 83°29'35"E	K2	1960	51	39.0
7	Manaslu	8163	Manaslu Himalaya	28°33'00"N 84°33'35"E	Cho Oyu	1956	49	45.0
8	Nanga Parbat	8126	Nanga Parbat Himalaya	35°14'14"N 74°35'21"E	Dhaulagiri	1953	52	67.0
9	Annapurna I	8091	Annapurna Himalaya	28°35'44"N 83°49'13"E	Cho Oyu	1950	36	47.0

FIGURE 11.1. – DataFrame Pandas

```
# ?pd.DataFrame
```

Chapitre 12.

Création d'un DataFrame

On peut créer un DataFrame

- à partir d'une matrice NumPy (en précisant éventuellement les noms des colonnes et des lignes)
- à partir d'un fichier CSV, Excel, SQL, etc.
- colonne par colonne à partir
  - d'une seule Series
  - d'un dictionnaire de Series ou d'un dictionnaire de listes (chaque élément du dictionnaire est une colonne)
- ligne par ligne à partir
  - d'une liste de Series ou d'une liste de dictionnaires (chaque élément de la liste est une ligne)

12.1. À partir d'un tableau NumPy à deux dimensions

On peut aussi créer un DataFrame à partir d'un tableau NumPy à deux dimensions.

```
ARRAY = np.array([ [100,   50,  20, 100,  50],
                   [1000, 300, 400,  50, 200],
                   [10,    20,   5, 200,  25] ])

df3 = pd.DataFrame( data=ARRAY )
df3
```

	0	1	2	3	4
0	100	50	20	100	50
1	1000	300	400	50	200
2	10	20	5	200	25

On peut spécifier les noms des colonnes et des lignes :

```
ARRAY = np.array([ [100,   50,  20, 100,  50],
                   [1000, 300, 400,  50, 200],
                   [10,    20,   5, 200,  25] ])
columns = ["Facebook", "Twitter", "Instagram", "Linkedin", "Snapchat"]
indices = ["Budget", "Audience", "CPM"]

df3 = pd.DataFrame( data=ARRAY,
                    columns=columns,
                    index=indices
                    )
df3
```

	Facebook	Twitter	Instagram	Linkedin	Snapchat
Budget	100	50	20	100	50
Audience	1000	300	400	50	200
CPM	10	20	5	200	25

```
df4 = df3.T
df4
```

	Budget	Audience	CPM
Facebook	100	1000	10
Twitter	50	300	20
Instagram	20	400	5
Linkedin	100	50	200
Snapchat	50	200	25

12.2. Colonne par colonne à partir ...

12.2.1. ... d'une seule Series

```
population = pd.Series( data=[8.516, 67.06, 328.2, 1_386],
                        index=["Suisse", "France", "USA", "Chine"] # les index seront les labels des
 ↪ lignes
                        )

display( population )
# df0 = pd.DataFrame( data=population )
df0 = pd.DataFrame( data=population,
                    columns=["Population"]
                    )
display(df0)
```

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
dtype: float64
```

	Population
Suisse	8.516
France	67.060
USA	328.200
Chine	1386.000

```
display(df0.values) # un array 2D numpy
display(df0.index) # un objet de type Index
display(df0.columns) # un objet de type Index
display(df0["Population"]) # une Series
display(df0["Population"]["Suisse"]) # DataFrame["colonne"]["ligne"]
```

```
array([[   8.516],
       [  67.06 ],
       [ 328.2  ],
       [1386.   ]])
```

```
Index(['Suisse', 'France', 'USA', 'Chine'], dtype='object')
```

```
Index(['Population'], dtype='object')
```

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
Name: Population, dtype: float64
```

```
8.516
```

Si les éléments `index` et `columns` d'un DataFrame ont leurs attributs name définis, ceux-ci seront également affichés :

```
df0.index.name = "États"
df0
```

États	Population
Suisse	8.516
France	67.060
USA	328.200
Chine	1386.000

```
df0.columns.name = "Caractéristiques"
df0
```

Caractéristiques États	Population
Suisse	8.516
France	67.060
USA	328.200
Chine	1386.000

```
df0.T # transposée, comme en numpy
```

États Caractéristiques	Suisse	France	USA	Chine
Population	8.516	67.06	328.2	1386.0

12.2.2. … à partir d'un dictionnaire de Series `{label_1: serie_1, label_2: serie_2, etc}` ou de listes de valeurs `{label_1: [val_1, val_2, etc], label_2: [val_1, val_2, etc], etc}`

Chaque Series devient une **colonne** du DataFrame.

Les clés du dictionnaire sont les noms des colonnes et les valeurs associées sont les Series ou listes de valeurs pour chaque colonne.
Les index peuvent être passés ou générés automatiquement.
Depuis Python 3.7, les dictionnaires conservent l'ordre d'insertion des clés : les colonnes du DataFrame apparaissent donc dans l'ordre où elles ont été écrites dans le dictionnaire.

```
population = pd.Series([8.516, 67.06, 328.2, 1_386],            index=["Suisse", "France", "USA",
 ↪  "Chine"])
area       = pd.Series([41_285, 551_695, 3_796_742, 9_596_961], index=["Suisse", "France", "USA",
 ↪  "Chine"])

df1 = pd.DataFrame( { "Population" : population,
                      "Area" : area } )
df1
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

```
dico = { "RS"       : ["Facebook", "Twitter", "Instagram", "Linkedin", "Snapchat"],
         "Budget"   : [100, 50, 20, 100, 50],
         "Audience": [1000, 300, 400, 50, 200],
         "CPM"      : [10, 20, 5, 200, 25] }
df1bis = pd.DataFrame(dico)
df1bis
```

	RS	Budget	Audience	CPM
0	Facebook	100	1000	10
1	Twitter	50	300	20
2	Instagram	20	400	5
3	Linkedin	100	50	200
4	Snapchat	50	200	25

```
display(df1bis["RS"][2]) # dangereux : est-ce un label ou une position ?

# solution : utiliser .loc pour les labels ou .iloc pour les positions, on retrouve
# l'ordre habituel de numpy
# display(df1bis.loc[2, "RS"])
# display(df1bis.iloc[2, 0])
```

```
'Instagram'
```

12.3. Ligne par ligne à partir ...

12.3.1. ... d'une liste de Series `[serie_1, serie_2, etc]` ou d'une liste de dictionnaires `[dico_1, dico_2, etc]`

Chaque Series/Dictionnaire devient une **ligne** du DataFrame.

```
population = pd.Series(data=[8.516, 67.06, 328.2, 1_386],           index=["Suisse", "France", "USA",
 ↪  "Chine"])
area       = pd.Series(data=[41_285, 551_695, 3_796_742, 9_596_961], index=["Suisse", "France", "USA",
 ↪  "Chine"])
df1 = pd.DataFrame( data=[population, area],
                    index=["Population", "Area"])
df1
```

	Suisse	France	USA	Chine
Population	8.516	67.06	328.2	1386.0
Area	41285.000	551695.00	3796742.0	9596961.0

```
population = {"Suisse":8.516, "France":67.06, "USA":328.2, "Chine":1_386}
area       = {"Suisse":41_285, "France":551_695, "USA":3_796_742, "Chine":9_596_961}
df2 = pd.DataFrame( data=[population, area],
                    index=["Population", "Area"])
df2
```

	Suisse	France	USA	Chine
Population	8.516	67.06	328.2	1386
Area	41285.000	551695.00	3796742.0	9596961

Si les deux dictionnaires ne contiennent pas les mêmes clés, les valeurs manquantes sont remplacées par `NaN` :

```
population = {"Suisse":8.516, "France":67.06, "USA":328.2}
area       = {"Suisse":41_285, "France":551_695, "Chine":9_596_961}
df2bis = pd.DataFrame( data=[population, area],
                       index=["Population", "Area"])
df2bis
```

	Suisse	France	USA	Chine
Population	8.516	67.06	328.2	NaN
Area	41285.000	551695.00	NaN	9596961.0

```
display(df2bis.values) # tableau numpy à 2 dimensions
display(df2bis.index) # objet de type Index
display(df2bis.columns) # objet de type Index
display(df2bis["France"]) # DataFrame["colonne"] renvoie une Series !!!! NB
display(df2bis["France"]["Population"]) # DataFrame["colonne"]["ligne"] !!! NB l'ordre
```

```
array([[8.516000e+00, 6.706000e+01, 3.282000e+02,          nan],
       [4.128500e+04, 5.516950e+05,          nan, 9.596961e+06]])
```

```
Index(['Population', 'Area'], dtype='object')
```

```
Index(['Suisse', 'France', 'USA', 'Chine'], dtype='object')
```

```
Population        67.06
Area          551695.00
Name: France, dtype: float64
```

```
67.06
```

```
df2bis.T # transposée, comme en numpy
```

	Population	Area
Suisse	8.516	41285.0
France	67.060	551695.0
USA	328.200	NaN
Chine	NaN	9596961.0

12.4. À partir d'un fichier : `read_csv`

Dans la pratique, on construit rarement un DataFrame à la main : on le **lit depuis un fichier**. Pandas sait lire de très nombreux formats : `read_excel`, `read_json`, `read_sql`, `read_parquet`, `read_html`… : et chacun a sa fonction d'écriture correspondante (`to_excel`, `to_json`, etc.). Nous nous limiterons au format le plus répandu, le **CSV**.

Un fichier **CSV** (*Comma Separated Values*) est un simple fichier texte : une ligne par observation, des champs séparés par un caractère convenu. C'est à la fois son grand avantage : n'importe quel tableur, n'importe quel langage sait le lire : et sa grande faiblesse : **rien n'y est normalisé**, ni le séparateur, ni le marqueur décimal, ni l'encodage. Lire un CSV correctement, c'est donc d'abord regarder à quoi il ressemble.

Fabriquons-en un pour l'exemple :

```
contenu = """ville;population;superficie;recensement
Toulon;176 198;42,84;2021-01-01
Marseille;873 076;240,62;2021-01-01
Nice;342 669;71,92;2021-01-01
Hyères;57 635;132,38;
La Seyne-sur-Mer;65 691;22,02;2021-01-01
"""

with open("villes_demo.csv", "w", encoding="utf-8") as f:
    f.write(contenu)

display(contenu)
```

```
'ville;population;superficie;recensement\nToulon;176 198;42,84;2021-01-01\nMarseille;873 076;240,62;2
01-01\nNice;342 669;71,92;2021-01-01\nHyères;57 635;132,38;\nLa Seyne-sur-Mer;65 691;22,02;2021-
01-01\n'
```

Ce petit fichier cumule les habitudes françaises : séparateur `;`, **virgule** décimale, **espace** comme séparateur de milliers, et une date manquante. Voyons ce que donne une lecture naïve.

```
pd.read_csv("villes_demo.csv")
```

	ville ;population ;superficie ;recensement
Toulon ;176 198 ;42	84 ;2021-01-01

	ville ;population ;superficie ;recensement
Marseille ;873 076 ;240	62 ;2021-01-01
Nice ;342 669 ;71	92 ;2021-01-01
Hyères ;57 635 ;132	38 ;
La Seyne-sur-Mer ;65 691 ;22	02 ;2021-01-01

Une seule colonne ! C'est normal : `read_csv` attend par défaut la **virgule** comme séparateur. On le corrige avec `sep` :

```
villes_csv = pd.read_csv("villes_demo.csv", sep=";")

display(villes_csv)
display(villes_csv.dtypes)
```

	ville	population	superficie	recensement
0	Toulon	176 198	42,84	2021-01-01
1	Marseille	873 076	240,62	2021-01-01
2	Nice	342 669	71,92	2021-01-01
3	Hyères	57 635	132,38	NaN
4	La Seyne-sur-Mer	65 691	22,02	2021-01-01

```
ville          object
population     object
superficie     object
recensement    object
dtype: object
```

Le tableau a la bonne forme, mais regardez les types : `population` et `superficie` sont de type `object`, c'est-à-dire **du texte**. Impossible d'en calculer la moyenne. Deux raisons : l'espace des milliers et la virgule décimale, que pandas ne devine pas. On les déclare :

```
villes_csv = pd.read_csv("villes_demo.csv", sep=";", thousands=" ", decimal=",")

display(villes_csv)
display(villes_csv.dtypes)
```

	ville	population	superficie	recensement
0	Toulon	176198	42.84	2021-01-01
1	Marseille	873076	240.62	2021-01-01
2	Nice	342669	71.92	2021-01-01
3	Hyères	57635	132.38	NaN
4	La Seyne-sur-Mer	65691	22.02	2021-01-01

```
ville           object
population       int64
superficie     float64
recensement     object
dtype: object
```

Il reste la colonne `recensement`, toujours lue comme du texte. `parse_dates` la convertit en véritables dates : et la valeur absente devient `NaT` (*Not a Time*), l'équivalent de `NaN` pour les dates :

```
villes_csv = pd.read_csv("villes_demo.csv", sep=";", thousands=" ", decimal=",",
                         parse_dates=["recensement"])

display(villes_csv)
display(villes_csv.dtypes)
```

	ville	population	superficie	recensement
0	Toulon	176198	42.84	2021-01-01
1	Marseille	873076	240.62	2021-01-01
2	Nice	342669	71.92	2021-01-01
3	Hyères	57635	132.38	NaT
4	La Seyne-sur-Mer	65691	22.02	2021-01-01

```
ville                  object
population              int64
superficie            float64
recensement    datetime64[ns]
dtype: object
```

Enfin, `index_col` désigne directement la colonne qui servira d'index :

```
villes_csv = pd.read_csv("villes_demo.csv", sep=";", thousands=" ", decimal=",",
                         parse_dates=["recensement"], index_col="ville")
villes_csv
```

ville	population	superficie	recensement
Toulon	176198	42.84	2021-01-01
Marseille	873076	240.62	2021-01-01
Nice	342669	71.92	2021-01-01
Hyères	57635	132.38	NaT
La Seyne-sur-Mer	65691	22.02	2021-01-01

12.4.1. Les options à connaître

Option	Rôle
`sep`	caractère séparant les champs (`,` par défaut, souvent `;` en France)
`decimal`	marqueur décimal (`.` par défaut, `,` en France)
`thousands`	séparateur de milliers à supprimer
`encoding`	`"utf-8"` par défaut ; essayez `"latin-1"` sur un vieux fichier Windows
`index_col`	colonne à utiliser comme index
`usecols`	liste des colonnes à charger : précieux sur un gros fichier
`na_values`	valeurs à interpréter comme manquantes (`"N/A"`, `"-"`, `"inconnu"`…)
`parse_dates`	colonnes à convertir en dates
`dtype`	forcer le type d'une ou plusieurs colonnes
`nrows`	ne lire que les premières lignes, pour inspecter un gros fichier
`skiprows`	ignorer des lignes d'en-tête parasites
`header`	numéro de la ligne d'en-tête, ou `None` s'il n'y en a pas

Option	Rôle

Le réflexe à prendre. Ouvrez d'abord le fichier dans un éditeur de texte et regardez ses premières lignes ; écrivez l'appel à `read_csv` **ensuite**. Puis vérifiez systématiquement `.dtypes` : une colonne numérique lue comme `object` est de loin le symptôme le plus fréquent d'un mauvais paramétrage, et toutes vos statistiques en dépendent.

12.4.2. Un cas réel : l'espace insécable

Les exports de l'INSEE séparent les milliers par un **espace insécable**, un caractère invisible qui ressemble à une espace mais n'en est pas une (son code Unicode est 160). Vous le rencontrerez dans l'exercice sur la population des communes.

```
NBSP = chr(160)   # l'espace insécable

with open("insee_demo.csv", "w", encoding="utf-8") as f:
    f.write(f"commune;population\nToulon;176{NBSP}198\nNice;342{NBSP}669\n")

brut = pd.read_csv("insee_demo.csv", sep=";")

display(brut.dtypes)
display(Markdown("valeur brute : `" + repr(brut["population"][0]) + "`"))   # le \xa0 est bien là
```

```
commune       object
population    object
dtype: object
```

valeur brute : `'176\xa0198'`

```
# thousands ne connaît que l'espace ORDINAIRE : il ne règle pas le problème
display(pd.read_csv("insee_demo.csv", sep=";", thousands=" ").dtypes)

# il faut nettoyer la colonne à la main, puis convertir le type
propre = pd.read_csv("insee_demo.csv", sep=";")
propre["population"] = propre["population"].str.replace(chr(160), "", regex=False).astype(int)

display(propre.dtypes)
display(propre)
```

```
commune       object
population    object
dtype: object
```

```
commune       object
population     int64
dtype: object
```

	commune	population
0	Toulon	176198
1	Nice	342669

12.4.3. Exporter : `to_csv`

L'écriture suit la même logique, avec les mêmes options :

```
villes_csv.to_csv("villes_export.csv", sep=";", decimal=",", encoding="utf-8")

display(Markdown("```\n" + open("villes_export.csv", encoding="utf-8").read() + "\n```"))
```

```
ville;population;superficie;recensement
Toulon;176198;42,84;2021-01-01
Marseille;873076;240,62;2021-01-01
Nice;342669;71,92;2021-01-01
Hyères;57635;132,38;
La Seyne-sur-Mer;65691;22,02;2021-01-01
```

Par défaut l'index est écrit dans le fichier. Si c'est une simple numérotation sans signification, on l'omet avec `index=False` :

```
villes_csv.reset_index().to_csv("villes_export.csv", sep=";", decimal=",", index=False)

display(Markdown("```\n" + open("villes_export.csv", encoding="utf-8").read() + "\n```"))
```

```
ville;population;superficie;recensement
Toulon;176198;42,84;2021-01-01
Marseille;873076;240,62;2021-01-01
Nice;342669;71,92;2021-01-01
Hyères;57635;132,38;
La Seyne-sur-Mer;65691;22,02;2021-01-01
```

12.4.4. Bonus : manipuler des dates

Bonus : hors séance. Les dates sont omniprésentes dans les jeux de données ouverts ; cette section vous servira dès que vous en rencontrerez.

`parse_dates` convertit les dates **au moment de la lecture**. Quand la colonne a déjà été chargée comme du texte, `pd.to_datetime` fait la conversion après coup.

Une fois la colonne de type `datetime64`, l'accesseur `.dt` donne accès à l'année, au mois, au jour de la semaine : exactement comme `.str` donne accès aux méthodes de chaînes.

```
inscriptions = pd.Series(["2021-01-15", "2021-06-30", "2022-03-08"])
display(inscriptions.dtype)      # object : c'est du texte

inscriptions = pd.to_datetime(inscriptions)
display(inscriptions.dtype)      # datetime64[ns]
display(inscriptions)
```

```
dtype('O')
```

```
dtype('<M8[ns]')
```

```
0   2021-01-15
1   2021-06-30
2   2022-03-08
dtype: datetime64[ns]
```

```
display(inscriptions.dt.year)
display(inscriptions.dt.month)
display(inscriptions.dt.day_name())
```

```
0    2021
1    2021
2    2022
dtype: int32
```

```
0    1
1    6
2    3
dtype: int32
```

```
0       Friday
1    Wednesday
2      Tuesday
dtype: object
```

Si le format n'est pas reconnu automatiquement : typiquement le format français `jj/mm/aaaa` : on le précise avec `format` :

```
pd.to_datetime(pd.Series(["15/01/2021", "30/06/2021"]), format="%d/%m/%Y")
```

```
0   2021-01-15
1   2021-06-30
dtype: datetime64[ns]
```

On peut alors filtrer et regrouper sur le temps :

```
df_dates = pd.DataFrame({ "date"    : inscriptions,
                          "montant" : [10, 25, 40] })

display(df_dates[ df_dates["date"] >= "2021-06-01" ])
display(df_dates.groupby(df_dates["date"].dt.year)["montant"].sum())
```

	date	montant
1	2021-06-30	25
2	2022-03-08	40

```
date
2021    35
2022    40
Name: montant, dtype: int64
```

Chapitre 13.

Accés aux éléments

```
population = pd.Series([8.516, 67.06, 328.2, 1_386],           index=["Suisse", "France", "USA",
 ↪  "Chine"])
area       = pd.Series([41_285, 551_695, 3_796_742, 9_596_961], index=["Suisse", "France", "USA",
 ↪  "Chine"])

df1 = pd.DataFrame( { "Population" : population,
                      "Area" : area } )
```

```
display(df1)
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

```
display(df1.values)   # tableau numpy à 2 dimensions
display(df1.index)    # objet de type Index
display(df1.columns)  # objet de type Index
```

```
array([[8.516000e+00, 4.128500e+04],
       [6.706000e+01, 5.516950e+05],
       [3.282000e+02, 3.796742e+06],
       [1.386000e+03, 9.596961e+06]])

Index(['Suisse', 'France', 'USA', 'Chine'], dtype='object')

Index(['Population', 'Area'], dtype='object')
```

13.1. Selectionner une colonne : `df['nom_colonne']` ou `df.nom_colonne`

Pour séléctionner la colonne `col1` d'un DataFrame on peut utiliser `df.col1`, mais on préfèrera généralement `df["col1"]`. En effet, la première syntaxe ne fonctionne pas si le nom de la colonne contient des espaces ou des caractères spéciaux, ou si le nom de la colonne est identique à une méthode ou un attribut du DataFrame.
Le résultat est une Series.

```
# La colonne "Population" : c'est une Series
display(df1["Population"])

# Les colonnes "Population" et "Area" : c'est un DataFrame
display(df1[["Population", "Area"]])
```

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
Name: Population, dtype: float64
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

Attention :

- dans un tableau **numpy** à 2 dimensions, `A[0]` renvoi la première **ligne** du tableau `A`
- pour un objet **DataFrame**, l'écriture `A[0]` renvoi la première **colonne**.

C'est pourquoi il vaut mieux considérer un objet Dataframe comme un dictionnaire dont les clés sont les labels des colonnes plutôt que comme un tableau NumPy.

Une fois une colonne selectionnée (c'est une série), on peut selecionner un élément :

```
# La valeur de la cellule "Population" de la ligne "France"
col1 = "Population"
row1 = "France"
num_row1 = 1

display( df1[col1][row1] )
# display( df1[col1][num_row1] ) # deprecated
display( df1[col1].loc[row1] )
display( df1[col1].iloc[num_row1] )
```

```
67.06
```

```
67.06
```

```
67.06
```

Puisque les colonnes sont des Series, on peut utiliser les méthodes de Series sur les colonnes d'un DataFrame. En particulier, tout ce que l'on a dit sur le slicing des Series est valable pour les colonnes d'un DataFrame.

```
#  SLICING
# =========

# Slicing sur les lignes: les valeurs de la colonne "Population" des lignes de "France" à "USA" inclus
display(df1["Population"]["France":"USA"])

# La colonne "Population", toutes les lignes
display(df1["Population"][:]) # idem que df1["Population"]

# Les valeurs de la ligne France
# ERROR : on ne peut pas extraire une ligne avec [:] !!!
# display(df1[:]["France"])
```

```
France     67.06
USA       328.20
Name: Population, dtype: float64
```

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
Name: Population, dtype: float64
```

13.2. Selectionner une ligne : `loc` et `iloc`

Si on veut selectionner une ligne, on utilise

- soit la méthode `.loc[label]` qui utilise les labels et est la méthode recommandée ;
- soit la méthode `.iloc[indice]` qui utilise les indices de position (de 0 à N où N est égal à `df.shape[0]`) et qui est deconsillée car les indices peuvent changer si on modifie le DataFrame.

Dans les deux cas, on obtient une Series dont les index sont les noms des colonnes. Ensuite, on pourra utiliser les méthodes de Series pour accéder aux éléments de la ligne.

NB Avec `loc` ou `iloc`, on peut accéder à un élément d'un DataFrame avec la syntaxe `df.loc[label_ligne, nom_colonne]` ou `df.iloc[indice_ligne, indice_colonne]` et retrouver ainsi l'ordre d'indexation des lignes et des colonnes de numpy : `.iloc[i,j]` renvoi l'élément de la ligne `i` et de la colonne `j`.

```
df1
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

```
# Selectionner avec .loc (label de ligne)
# =======================================================

# Une ligne
display(df1.loc["France"]) # la ligne "France"

# Une cellule
display(df1.loc["France", "Population"]) # la valeur de la cellule "Population" de la ligne "France"

# Un slicing de lignes, une colonne
display(df1.loc["France":"USA", "Population"]) # les valeurs de la colonne "Population" des lignes de
 ↪  "France" à "USA" inclus

# Un slicing de lignes, plusieurs colonnes
display(df1.loc["France":"USA", ["Population", "Area"]]) # les valeurs des colonnes "Population" et
 ↪  "Area" des lignes de "France" à "USA" inclus

# Toutes les valeurs d'une colonne
display(df1.loc[:, "Population"]) # toutes les valeurs de la colonne "Population", idem que
 ↪  df1["Population"]
display(df1.iloc[:, 0]) # toutes les valeurs de la colonne "Population", idem que df1["Population"]
```

```
Population        67.06
Area          551695.00
Name: France, dtype: float64


67.06
```

```
France     67.06
USA       328.20
Name: Population, dtype: float64
```

	Population	Area
France	67.06	551695
USA	328.20	3796742

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
Name: Population, dtype: float64
```

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
Name: Population, dtype: float64
```

```
# Selectionner avec .iloc (indices de position)
# ===========================================================

# Une ligne
display(df1.iloc[1])   # la ligne d'indice 1, toutes les colonnes
display(df1.iloc[1,:]) # la ligne d'indice 1, toutes les colonnes

# Une cellule
display(df1.iloc[1, 0]) # la valeur de la cellule d'indice 1, 0

# Un slicing de lignes, une colonne
display(df1.iloc[1:3, 0]) # les valeurs de la colonne "Population" des lignes d'indice 1 à 3 exclus

# Un slicing de lignes, plusieurs colonnes
display(df1.iloc[1:3, [0, 1]]) # les valeurs des colonnes "Population" et "Area" des lignes d'indice 1 à
↪  3 exclus

# Toutes les valeurs d'une colonne
display(df1.iloc[:, 0]) # toutes les valeurs de la colonne "Population"  idem que df1["Population"]
```

```
Population        67.06
Area          551695.00
Name: France, dtype: float64
```

```
Population        67.06
Area          551695.00
Name: France, dtype: float64
```

```
67.06
```

```
France     67.06
USA       328.20
Name: Population, dtype: float64
```

	Population	Area
France	67.06	551695
USA	328.20	3796742

```
Suisse       8.516
France      67.060
USA        328.200
Chine     1386.000
Name: Population, dtype: float64
```

On peut alors utiliser les masques de numpy etc.

```
display(df1)

# les lignes dont la Population est supérieure à 100, les colonnes "Population" et "Area"
# df1.loc[ df1["Population"] > 100, ["Population", "Area"] ]
mask = df1["Population"] > 100
display(mask) # une Series de booléens
df1.loc[ mask, : ]
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

```
Suisse    False
France    False
USA        True
Chine      True
Name: Population, dtype: bool
```

	Population	Area
USA	328.2	3796742
Chine	1386.0	9596961

13.2.1. Tableau comparatif entre `numpy.ndarray` et `pandas.DataFrame`

Aspect	`numpy.ndarray`	`pandas.DataFrame`
Nature	Tableau N-D homogène	Tableau 2D étiqueté (lignes + colonnes)
Types des données	Un seul type par array	Un type **par colonne**
Ordonnancement	Ordonné	Ordonné (index + colonnes)
Labels	Aucun	Noms de colonnes + index
Accès positionnel	`arr[i, j]`	`df.iloc[i, j]`
Accès par label		`df.loc[row_label, col_label]` ou `df.at[row_label, col_label]`
Accès par colonne		`df["nom_colonne"]` (retourne une Series)
Accès par ligne		`df.iloc[i]` (par position), `df.loc[row_label]` (par label)

Aspect	`numpy.ndarray`	`pandas.DataFrame`
Slicing lignes	`arr[1:4]`	`df.iloc[1:4]` ou `df.loc["y":"z"]`
Slicing colonnes	`arr[:, 1:3]`	`df[["col1","col2"]]` ou `df.loc[:, "col1":"col3"]`
Vectorisation	Très rapide	Repose sur NumPy

13.2.2. Résumé des principales syntaxes d'accès aux éléments d'un DataFrame :

Syntaxe	Description
`df[eti_col]`	Sélectionne une colonne ou une séquence de colonnes. Peut aussi servir pour filtres booléens ou slices.
`df.loc[eti_ligne]`	Sélectionne une ou plusieurs lignes par leur **étiquette de ligne** (`eti_ligne`).
`df.loc[:, eti_col]`	Sélectionne une ou plusieurs colonnes par leurs **étiquettes de colonnes** (`eti_col`).
`df.loc[eti_ligne, eti_col]`	Sélectionne lignes (`eti_ligne`) et colonnes (`eti_col`) par leurs **étiquettes**.
`df.iloc[pos]`	Sélectionne une ou plusieurs lignes par leur **position entière** (`pos`).
`df.iloc[:, pos]`	Sélectionne une ou plusieurs colonnes par leur **position entière** (`pos`).
`df.iloc[pos_i, pos_j]`	Sélectionne lignes (`pos_i`) et colonnes (`pos_j`) par leurs **positions entières**.
`df.at[eti_ligne, eti_col]`	Accède à une **valeur scalaire unique** par étiquette de ligne (`eti_ligne`) et de colonne (`eti_col`).
`df.iat[i, j]`	Accède à une **valeur scalaire unique** par position entière de ligne (`i`) et de colonne (`j`).
Méthode `reindex`	Réorganise ou sélectionne lignes/colonnes par leurs **étiquettes** (`eti_ligne` et/ou `eti_col`).
Méthodes `get_value` / `set_value`	Accède ou modifie une valeur unique par étiquette (**obsolètes**, utilisez `at`/`iat`).

13.3. Ajout de lignes/colonnes

13.3.1. Ajout d'une colonne

On ajoute une colonne à un DataFrame en utilisant la notation `df['nom_colonne'] = serie`, eventuellement en utilisant une colonne existante pour calculer la nouvelle colonne.

```
df1 = pd.DataFrame( { "Population" : population,
                      "Area" : area } )
df1
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

```
df1["Density"] = df1["Population"] / df1["Area"]
df1
```

	Population	Area	Density
Suisse	8.516	41285	0.000206
France	67.060	551695	0.000122
USA	328.200	3796742	0.000086
Chine	1386.000	9596961	0.000144

Les colonnes sont toujours ajoutées à la fin du DataFrame. Si on veut spécifier une position (et décaler les autres colonnes) on utilise la méthode `.insert()` :

```
# ajout d'une colonne à une position donnée
df1.insert( 0, "DensityBIS", df1["Population"] / df1["Area"] )
df1
```

	DensityBIS	Population	Area	Density
Suisse	0.000206	8.516	41285	0.000206
France	0.000122	67.060	551695	0.000122
USA	0.000086	328.200	3796742	0.000086
Chine	0.000144	1386.000	9596961	0.000144

13.3.2. Ajout d'une ligne

```
df1 = pd.DataFrame( { "Population" : population,
                      "Area" : area } )
df1
```

	Population	Area
Suisse	8.516	41285
France	67.060	551695
USA	328.200	3796742
Chine	1386.000	9596961

Pour ajouter une ligne à un DataFrame, on utilise la méthode `.loc[label] = serie/liste/dict` :

```
df1.loc["Italie"] = [60.36, 301338]
df1
```

	Population	Area
Suisse	8.516	41285.0
France	67.060	551695.0
USA	328.200	3796742.0
Chine	1386.000	9596961.0
Italie	60.360	301338.0

13.4. Réordonner les observations

La méthode `sort_values` permet de réordonner les observations d'un DataFrame, en laissant l'ordre des colonnes identiques. On peut trier selon une ou plusieurs colonnes, dans l'ordre croissant ou décroissant.

```
df1_sorted_pop = df1.sort_values(by="Population")
df1_sorted_pop
```

	Population	Area
Suisse	8.516	41285.0
Italie	60.360	301338.0
France	67.060	551695.0
USA	328.200	3796742.0
Chine	1386.000	9596961.0

```
df1_sorted_area = df1.sort_values(by="Area", ascending=False)
df1_sorted_area
```

	Population	Area
Chine	1386.000	9596961.0
USA	328.200	3796742.0
France	67.060	551695.0
Italie	60.360	301338.0
Suisse	8.516	41285.0

13.5. Suppression de lignes/colonnes

Pour supprimer une ligne ou une colonne on utilise la méthode `.drop()` :

- `df.drop( nom_ligne, axis=0)` ou `df.drop(labels = nom_ligne)` pour supprimer une ligne
- `df.drop( nom_colonne, axis=1)` ou `df.drop( columns = nom_colonne)` pour supprimer une colonne

On ajoute l'option `inplace=True` pour modifier le DataFrame, `inplace=False` pour renvoyer un nouveau DataFrame sans modifier l'original.

Attention, si la la ligne ou la colonne n'existe pas, une erreur sera levée. Pour éviter cela, on peut utiliser l'option `errors='ignore'`.

```
# Création du DataFrame

df1 = pd.DataFrame( { "Population" : population,
                      "Area" : area } )
df1.loc["Italie"] = [ 60.36, 301338]

df1["Density"] = df1["Population"] / df1["Area"]
df1.insert( 0, "DensityBIS", df1["Population"] / df1["Area"] )

df1
```

	DensityBIS	Population	Area	Density
Suisse	0.000206	8.516	41285.0	0.000206
France	0.000122	67.060	551695.0	0.000122
USA	0.000086	328.200	3796742.0	0.000086
Chine	0.000144	1386.000	9596961.0	0.000144
Italie	0.000200	60.360	301338.0	0.000200

```
# Supprimoons une colonne

df1.drop( columns="DensityBIS", inplace=True ) # suppression d'une colonne
# del df1["Density"]  # autre méthode pour supprimer une colonne

df1
```

	Population	Area	Density
Suisse	8.516	41285.0	0.000206
France	67.060	551695.0	0.000122
USA	328.200	3796742.0	0.000086
Chine	1386.000	9596961.0	0.000144
Italie	60.360	301338.0	0.000200

```
# Supprimons une ligne

df1.drop(labels=["Chine"], inplace=True) # suppression d'une ligne
display(df1)
```

	Population	Area	Density
Suisse	8.516	41285.0	0.000206
France	67.060	551695.0	0.000122
USA	328.200	3796742.0	0.000086
Italie	60.360	301338.0	0.000200

Chapitre 14.

Aperçu des données

Créons d'abord un DataFrame un peu plus richement rempli (il pourrait tout aussi bien provenir d'un fichier CSV, comme vu plus haut) :

```
data = pd.DataFrame( {  "foo" : ["one", "one", "one", "two", "two", "two"],
                        "bar" : ["A", "A", "B", "A", "B", "B"],
                        "baz" : [1, 2, 3, 4, 5, 6],
                        "zoo" : ["x", "y", "z", "q", "w", 't'] } )

data
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

14.1. Afficher les premières/dernières lignes

Un bon réflexe à adopter après la création/l'importation d'un DataFrale, et après toute transformation importante, est de visualiser le jeu de données, ou du moins quelques lignes, afin de vérifier que tout s'est correctement déroulé. Pour cela, il existe deux méthodes principales :

- la méthode `.head()` permettant de sélectionner par défaut les 5 premières lignes du data frame. Il est possible de préciser entre parenthèses le nombre de lignes à afficher ;
- la méthode `.tail()` permettant de sélectionner par défaut les 5 dernières lignes du data frame. Il est également possible de préciser entre parenthèses le nombre de lignes à afficher.

Il n'est pas possible (par défaut) d'afficher plus de 60 lignes d'un data frame, afin de ne pas surcharger inutilement le notebook. De façon plus globale, chercher à visualiser l'ensemble d'un data frame n'est généralement pas une bonne pratique. Si cela est tout à fait envisageable avec quelques dizaines de lignes, ça devient vite impossible avec plusieurs millions de lignes ! Si vous cherchez à afficher plus de 60 lignes, vous aurez finalement comme résultat les 5 premières et dernières lignes du data frame.

```
# afficher les 5 premières lignes
display(data.head())
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q

	foo	bar	baz	zoo
4	two	B	5	w

```
# afficher les 2 dernières lignes
display(data.tail(2))
```

	foo	bar	baz	zoo
4	two	B	5	w
5	two	B	6	t

14.2. Attributs d'un DataFrame

Combien de lignes comporte le DataFrame ? Et combien de colonnes ? Tout comme avec les arrays NumPy, il est possible de répondre à ces questions via l'attribut `.shape`. Le résultat est un tuple : le premier élément correspond au nombre de lignes, et le second au nombre de colonnes. On peut naturellement stocker le résultat de cet attribut dans une variable pour réutiliser ces éléments ultérieurement.

```
display(Markdown("**`ndim` : nombre de dimensions (2 pour un DataFrame)**"))
display( data.ndim)

display(Markdown("**`shape` : (nombre de lignes, nombre de colonnes)**"))
display( data.shape )

display(Markdown("**`size` : nombre total d'éléments (lignes * colonnes)**"))
display( data.size )

display(Markdown("**`axes` : liste des axes (index des lignes, index des colonnes)**"))
display( data.axes)

display(Markdown("**`index`, `columns`, `values` : accès aux composants du DataFrame**"))
display( data.index )
display( data.columns )
display( data.values ) # tableau numpy 2D des valeurs
```

`ndim` : nombre de dimensions (2 pour un DataFrame)

```
2
```

`shape` : (nombre de lignes, nombre de colonnes)

```
(6, 4)
```

`size` : nombre total d'éléments (lignes * colonnes)

```
24
```

`axes` : liste des axes (index des lignes, index des colonnes)

```
[RangeIndex(start=0, stop=6, step=1),
 Index(['foo', 'bar', 'baz', 'zoo'], dtype='object')]
```

`index`, `columns`, `values` : accès aux composants du DataFrame

```
RangeIndex(start=0, stop=6, step=1)

Index(['foo', 'bar', 'baz', 'zoo'], dtype='object')

array([['one', 'A', 1, 'x'],
       ['one', 'A', 2, 'y'],
       ['one', 'B', 3, 'z'],
       ['two', 'A', 4, 'q'],
       ['two', 'B', 5, 'w'],
       ['two', 'B', 6, 't']], dtype=object)
```

On peut avoir envie de connaître les types de chacune de nos variables (un type par colonne). On peut accéder à cela très simplement à partir de l'attribut `.dtypes`. On obtient un objet Series :les index de cette série sont les étiquettes des colonnes, et les valeurs de cette série sont les types de données. Vous noterez que le type de `foo` est `objet`, alors que nous avons pourtant des chaînes de caractères. C'est une chose à connaître, mais le type objet de Pandas correspond en fait à une colonne de type chaîne de caractères.

```
data.dtypes
```

```
foo    object
bar    object
baz     int64
zoo    object
dtype: object
```

Les jeux de données réels sont rarement complets. Les valeurs manquantes sont affichées `NaN` et repérées par la méthode `.isnull()`, qui renvoie un tableau de booléens de même forme que le DataFrame.

Ici, le DataFrame `data` est complet : toutes les cases valent `False`. Nous verrons un peu plus bas, dans la section Les valeurs manquantes, comment les compter, les supprimer ou les remplacer.

```
data.isnull()
```

	foo	bar	baz	zoo
0	False	False	False	False
1	False	False	False	False
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False
5	False	False	False	False

Ces informations peuvent être combinées pour obtenir un résumé complet des données avec la méthode `.info()`.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   foo     6 non-null      object
 1   bar     6 non-null      object
 2   baz     6 non-null      int64
 3   zoo     6 non-null      object
dtypes: int64(1), object(3)
memory usage: 324.0+ bytes
```

```
data['baz'].mean()
```

```
3.5
```

14.3. Les valeurs manquantes

Les jeux de données réels sont rarement complets. Une valeur manquante peut recouvrir des réalités très différentes : information non collectée, question sans objet pour cet individu, erreur de saisie, perte lors d'un transfert. Pandas les représente toutes de la même façon, par `NaN` (*Not a Number*).

Le DataFrame `data` utilisé jusqu'ici n'en contient aucune. Construisons-en un qui en contient :

```
releve = pd.DataFrame({
    "station"     : ["Toulon", "Nice", "Marseille", "Ajaccio", "Bastia"],
    "temperature" : [18.2, np.nan, 21.0, 19.5, np.nan],
    "humidite"    : [65, 70, np.nan, 58, 62],
    "vent"        : [12, 8, 15, np.nan, 20] })

releve
```

	station	temperature	humidite	vent
0	Toulon	18.2	65.0	12.0
1	Nice	NaN	70.0	8.0
2	Marseille	21.0	NaN	15.0
3	Ajaccio	19.5	58.0	NaN
4	Bastia	NaN	62.0	20.0

14.3.1. Repérer les valeurs manquantes

`.isnull()` (ou son alias `.isna()`) renvoie un DataFrame de booléens de même forme. Combiné à `.sum()` il donne le **nombre** de valeurs manquantes par colonne ; combiné à `.mean()`, leur **proportion**.

```
display(releve.isnull())
display(releve.isnull().sum())
display((releve.isnull().mean() * 100).round(1))
```

	station	temperature	humidite	vent
0	False	False	False	False
1	False	True	False	False
2	False	False	True	False
3	False	False	False	True
4	False	True	False	False

```
station        0
temperature    2
humidite       1
vent           1
dtype: int64
```

```
station         0.0
temperature    40.0
humidite       20.0
vent           20.0
dtype: float64
```

`.info()` donne la même information sous une autre forme : le nombre de valeurs **non nulles** de chaque colonne.

```
releve.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   station      5 non-null      object
 1   temperature  3 non-null      float64
 2   humidite     4 non-null      float64
 3   vent         4 non-null      float64
dtypes: float64(3), object(1)
memory usage: 292.0+ bytes
```

Et pour examiner les lignes incomplètes :

```
masque = releve.isnull().any(axis=1)   # au moins une valeur manquante sur la ligne
releve[masque]
```

	station	temperature	humidite	vent
1	Nice	NaN	70.0	8.0
2	Marseille	21.0	NaN	15.0
3	Ajaccio	19.5	58.0	NaN
4	Bastia	NaN	62.0	20.0

`NaN` n'est égal à rien, pas même à lui-même. Un test `== np.nan` renvoie toujours `False`, et ne détecte donc jamais rien. Utilisez toujours `.isnull()` / `.isna()`.

```
display(Markdown(f"np.nan == np.nan : **{np.nan == np.nan}**"))

display(releve["temperature"] == np.nan)   # que des False : inutilisable
display(releve["temperature"].isnull())    # la bonne façon de faire
```

np.nan == np.nan : **False**

```
0    False
1    False
2    False
3    False
4    False
Name: temperature, dtype: bool
```

```
0    False
1     True
2    False
3    False
4     True
Name: temperature, dtype: bool
```

14.3.2. Comment les calculs les traitent

Les fonctions d'agrégation **ignorent** les valeurs manquantes. Ce n'est pas anodin : la moyenne est calculée sur les seules valeurs disponibles, et le dénominateur n'est donc pas celui que vous croyez.

```
display(releve["temperature"].mean())    # moyenne sur 3 valeurs, pas sur 5
display(releve["temperature"].sum())     # somme des 3 valeurs présentes
display(releve["temperature"].count())   # nombre de valeurs NON manquantes
```

```
19.566666666666666
```

```
58.7
```

```
3
```

En revanche, une opération arithmétique **propage** le `NaN` : dès qu'une des deux valeurs manque, le résultat manque.

```
releve["temperature"] + releve["vent"]
```

```
0    30.2
1     NaN
2    36.0
3     NaN
4     NaN
dtype: float64
```

14.3.3. Supprimer : `dropna`

Par défaut, `.dropna()` supprime toute **ligne** comportant au moins une valeur manquante. Les arguments utiles :

Argument	Effet
`axis=1`	supprimer les **colonnes** plutôt que les lignes
`how="all"`	ne supprimer que si **toutes** les valeurs manquent
`subset=[...]`	n'examiner que certaines colonnes
`thresh=n`	conserver les lignes ayant au moins `n` valeurs renseignées

```
display(releve.dropna())                          # ne reste que les lignes complètes
display(releve.dropna(subset=["temperature"]))    # on n'exige que la température
display(releve.dropna(axis=1))                    # on supprime les colonnes incomplètes
```

	station	temperature	humidite	vent
0	Toulon	18.2	65.0	12.0

	station	temperature	humidite	vent
0	Toulon	18.2	65.0	12.0
2	Marseille	21.0	NaN	15.0
3	Ajaccio	19.5	58.0	NaN

	station
0	Toulon
1	Nice
2	Marseille
3	Ajaccio
4	Bastia

14.3.4. Remplacer : `fillna`

```
display(releve.fillna(0))                                    # par une constante
display(releve.fillna(releve.mean(numeric_only=True)))       # par la moyenne de chaque colonne
```

	station	temperature	humidite	vent
0	Toulon	18.2	65.0	12.0
1	Nice	0.0	70.0	8.0
2	Marseille	21.0	0.0	15.0
3	Ajaccio	19.5	58.0	0.0
4	Bastia	0.0	62.0	20.0

	station	temperature	humidite	vent
0	Toulon	18.200000	65.00	12.00
1	Nice	19.566667	70.00	8.00
2	Marseille	21.000000	63.75	15.00
3	Ajaccio	19.500000	58.00	13.75
4	Bastia	19.566667	62.00	20.00

Pour des données ordonnées (une série temporelle, par exemple), on peut propager la dernière valeur connue avec `.ffill()`, ou la suivante avec `.bfill()` :

```
display(releve.ffill())   # forward fill  : on reprend la valeur du dessus
display(releve.bfill())   # backward fill : on reprend la valeur du dessous
```

	station	temperature	humidite	vent
0	Toulon	18.2	65.0	12.0
1	Nice	18.2	70.0	8.0
2	Marseille	21.0	70.0	15.0
3	Ajaccio	19.5	58.0	15.0
4	Bastia	19.5	62.0	20.0

	station	temperature	humidite	vent
0	Toulon	18.2	65.0	12.0
1	Nice	21.0	70.0	8.0
2	Marseille	21.0	58.0	15.0
3	Ajaccio	19.5	58.0	20.0
4	Bastia	NaN	62.0	20.0

Traiter les valeurs manquantes n'est pas un choix technique, c'est un choix méthodologique.

- **Supprimer** les lignes incomplètes biaise l'échantillon dès que les données ne manquent pas « au hasard » : si les personnes à hauts revenus refusent plus souvent de répondre, les supprimer fait baisser artificiellement le revenu moyen.
- **Remplacer par la moyenne** préserve cette moyenne, mais écrase la variance et invente des observations qui n'ont jamais été faites.

Il n'existe pas de bonne réponse générique. La seule règle vraiment solide : **regarder combien de lignes sont concernées avant de décider**, puis **dire explicitement ce que vous avez fait et pourquoi**.

14.4. Statistiques descriptives

On peut calculer la somme, la plus petite/grande valeur, la moyenne, la médiane, l'écart-type des élements de chaque colonne avec les méthodes `.sum()`, `.min()`, `.max()`, `.mean()`, `.median()` et `.std()`.
Rappel : la somme entre chaînes de caractères est une concaténation.

```
# Elles renvoient des Series dont les Index sont les colonnes du DataFrame

display(Markdown("**Opérations sur les DataFrames**"))

display(Markdown("**Somme**"))
display(data.sum()) # idem que data.sum(axis=0) ou data.sum(axis='rows')
display(Markdown("**Minimum**"))
display(data.min())
display(Markdown("**Maximum**"))
display(data.max())
display(Markdown("**Moyenne**"))
# display(data.mean())
display(data.mean(numeric_only=True))
display(Markdown("**Médiane**"))
# display(data.median())
display(data.median(numeric_only=True))
display(Markdown("**Écart-type**"))
# display(data.std())
display(data.std(numeric_only=True))
display(Markdown("**Comptage : nombre de valeurs non manquants par colonne**"))
display(data.count())
```

Opérations sur les DataFrames

Somme

```
foo    oneoneonetwotwotwo
bar                AABABB
baz                    21
zoo                xyzqwt
dtype: object
```

Minimum

```
foo    one
bar      A
baz      1
zoo      q
dtype: object
```

Maximum

```
foo    two
bar      B
baz      6
zoo      z
dtype: object
```

Moyenne

```
baz    3.5
dtype: float64
```

Médiane

```
baz    3.5
dtype: float64
```

Écart-type

```
baz    1.870829
dtype: float64
```

Comptage : nombre de valeurs non manquants par colonne

```
foo    6
bar    6
baz    6
zoo    6
dtype: int64
```

Pour obtenir un résumé statistique des variables numériques, on peut utiliser la méthode `.describe()` :

```
data.describe()
```

	baz
count	6.000000
mean	3.500000
std	1.870829
min	1.000000
25%	2.250000
50%	3.500000
75%	4.750000
max	6.000000

Cette méthode renvoie un objet de type DataFrame, où les statistiques descriptives sont calculées pour chaque colonne numérique. Pour les variables catégorielles, on peut obtenir le nombre de valeurs uniques, la valeur la plus fréquente et sa fréquence :

```
data[['foo', 'bar', 'zoo']].describe()
```

	foo	bar	zoo

	foo	bar	zoo
count	6	6	6
unique	2	2	6
top	one	A	x
freq	3	3	1

```
data.describe(include="all")
```

	foo	bar	baz	zoo
count	6	6	6.000000	6
unique	2	2	NaN	6
top	one	A	NaN	x
freq	3	3	NaN	1
mean	NaN	NaN	3.500000	NaN
std	NaN	NaN	1.870829	NaN
min	NaN	NaN	1.000000	NaN
25%	NaN	NaN	2.250000	NaN
50%	NaN	NaN	3.500000	NaN
75%	NaN	NaN	4.750000	NaN
max	NaN	NaN	6.000000	NaN

On peut modifier le type de données d'une colonne avec la méthode `.astype()` et demander à Pandas d'y associer des codes numériques. Si on ne veut pas ecraser le DataFrame initial, on peut utiliser la méthode `.copy()`.

```
data_bis = data.copy()
for col in data_bis.select_dtypes(include='object').columns:
    data_bis[col] = data_bis[col].astype('category').cat.codes

display(data)
display(data_bis)
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

	foo	bar	baz	zoo
0	0	0	1	3
1	0	0	2	4
2	0	1	3	5
3	1	0	4	0
4	1	1	5	2
5	1	1	6	1

```
data_bis.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   foo     6 non-null      int8
 1   bar     6 non-null      int8
 2   baz     6 non-null      int64
 3   zoo     6 non-null      int8
dtypes: int64(1), int8(3)
memory usage: 198.0 bytes
```

```
data_bis.describe()
```

	foo	bar	baz	zoo
count	6.000000	6.000000	6.000000	6.000000
mean	0.500000	0.500000	3.500000	2.500000
std	0.547723	0.547723	1.870829	1.870829
min	0.000000	0.000000	1.000000	0.000000
25%	0.000000	0.000000	2.250000	1.250000
50%	0.500000	0.500000	3.500000	2.500000
75%	1.000000	1.000000	4.750000	3.750000
max	1.000000	1.000000	6.000000	5.000000

14.5. Comptage : valeurs uniques, dénombrement de valeurs et appartenance

Une autre catégorie de méthodes connexes permet d'extraire des informations sur les valeurs contenues dans une série unidimensionnelle. Il s'agit des méthodes `.count()`, `.unique()`, `.nunique()` et `.value_counts()`.

```
data
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

```
# Combien de valuers : ici , on peut également utiliser .size(), puisqu'il n'y a pas de valeurs
↪ manquantes
data['foo'].count()
```

```
6
```

Pour déterminer le nombre de valeurs uniques d'une variable, plutôt que chercher à écrire soi-même une fonction, on utilise la méthode `nunique`.

```
# On affiche combien de valeurs uniques par colonne :
data.nunique()
```

```
foo    2
bar    2
baz    6
zoo    6
dtype: int64
```

```
# Pour chaque colonne selectionnée, on affiche les valeurs uniques :
for col in data.columns:
    print((f"\nValeurs uniques dans la colonne {col} :"))
    print(data[col].unique())
```

```

Valeurs uniques dans la colonne foo :
['one' 'two']

Valeurs uniques dans la colonne bar :
['A' 'B']

Valeurs uniques dans la colonne baz :
[1 2 3 4 5 6]

Valeurs uniques dans la colonne zoo :
['x' 'y' 'z' 'q' 'w' 't']
```

```
# On peut obtenir ces deux informations en une seule instruction :
for col in data.columns:
    print((f"\nValeurs et occurrences dans la colonne {col} :"))
    print(data[col].value_counts())
```

```

Valeurs et occurrences dans la colonne foo :
foo
one    3
two    3
Name: count, dtype: int64

Valeurs et occurrences dans la colonne bar :
bar
A    3
B    3
Name: count, dtype: int64

Valeurs et occurrences dans la colonne baz :
baz
1    1
2    1
3    1
4    1
5    1
6    1
Name: count, dtype: int64

Valeurs et occurrences dans la colonne zoo :
```

```
zoo
x    1
y    1
z    1
q    1
w    1
t    1
Name: count, dtype: int64
```

Chapitre 15.

Filtres et opérations

L'opération de sélection de lignes s'appelle Filtrer. Elle s'utilise en fonction d'une condition logique, on sélectionne les données sur une condition logique.

Il existe plusieurs méthodes en Pandas. La plus simple est d'utiliser les boolean mask, déjà vus en numpy.

La plupart de ce qu'on a pu voir avec les arrays de NumPy est applicable aux objets Series et DataFrames de Pandas. On peut par exemple effectuer des opérations arithmétiques sur les objets, ou encore appliquer des fonctions mathématiques.

15.1. Opérations sur les DataFrames

On peut appliquer des opérations arithmétiques sur les DataFrames, de la même manière que pour les Series. Les opérations sont effectuées élément par élément, en alignant les lignes et les colonnes par leurs étiquettes respectives.

```
df1['NewCol1'] = df1['Population'] * 2
df1['NewCol2'] = df1['Population']/df1['Area']
df1
```

	Population	Area	Density	NewCol1	NewCol2
Suisse	8.516	41285.0	0.000206	17.032	0.000206
France	67.060	551695.0	0.000122	134.120	0.000122
USA	328.200	3796742.0	0.000086	656.400	0.000086
Italie	60.360	301338.0	0.000200	120.720	0.000200

15.2. Filtrer un DataFrame : masque booléen *vs* `query()`

Pandas propose deux façons principales de filtrer un DataFrame : les masques booléens "à la NumPy", très flexibles, et la méthode `query()`, plus lisible et proche du SQL.

Voici un exemple de DataFrame que nous allons utiliser pour illustrer ces deux approches :

```
data
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

15.2.1. Filtre “à la NumPy” (masques booléens)

Avec cette méthode, on crée un masque booléen en appliquant une condition logique sur une ou plusieurs colonnes du DataFrame. On utilise ensuite ce masque pour sélectionner les lignes correspondantes.

- Avantages
  - Très flexible (méthodes pandas/NumPy, fonctions Python).
  - Compatible avec tous les noms de colonnes.
- Inconvénients
  - Syntaxe moins intuitive car `& | ~` au lieu de `and or not` et parenthèses obligatoires.

```
mask = data["bar"]=="A"
mask
```

```
0     True
1     True
2    False
3     True
4    False
5    False
Name: bar, dtype: bool
```

```
data[mask]
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
3	two	A	4	q

```
# `&` pour le `and`
# `|` pour le `or`
# `~` pour le `not`
# parenthèses obligatoires
mask = (data["bar"]=="A") & (data["baz"]>=2)
data[mask]
```

	foo	bar	baz	zoo
1	one	A	2	y
3	two	A	4	q

Remarque : dans l’exemple ci-dessous, on utilise la méthode `str.startswith('t')` pour vérifier si les chaînes de caractères de la colonne ‘foo’ commencent par la lettre ‘t’. `.str.` est une méthode particulière qui permet de traiter chaque valeur d’une colonne comme un string natif en Python sur lequel appliquer une méthode ultérieure (en l’occurrence `startswith`).

```
mask = data['foo'].str.startswith('t')
data[mask]
```

	foo	bar	baz	zoo
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

	foo	bar	baz	zoo

15.2.2. Filtre avec `query()`

- Avantages
  - Syntaxe type SQL : plus lisible.
  - Pas de parenthèses autour des conditions.
- Inconvénients
  - Limitations si noms de colonnes avec espaces ou fonctions Python complexes.
  - Expression passée en chaîne de caractères.

```
data.query("bar=='A'")
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
3	two	A	4	q

```
data.query("bar=='A' & baz>=2")
# idem que
# data[(data["bar"]=="A") & (data["baz"]>=2)]
```

	foo	bar	baz	zoo
1	one	A	2	y
3	two	A	4	q

Chapitre 16.

Aggregations

16.1. `groupby` : analyse par groupes

La méthode `groupby` permet de grouper les données suivant certains critères.

Commençons par un exemple simple :

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])

display(Markdown("**DataFrame initial**"))
display(df)
display(Markdown("**GroupBy sum sur la colonne 'key'**"))
display(df.groupby('key').sum())
```

DataFrame initial

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

GroupBy sum sur la colonne 'key'

	data
key	
A	3
B	5
C	7

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6),
                   'toto': ['a','a','b','b','a','b']},
                   columns=['key', 'data', 'toto'])

display(Markdown("**DataFrame initial**"))
display(df)
display(Markdown("**GroupBy sum sur la colonne 'key'**"))
display(df.groupby('key').sum())
```

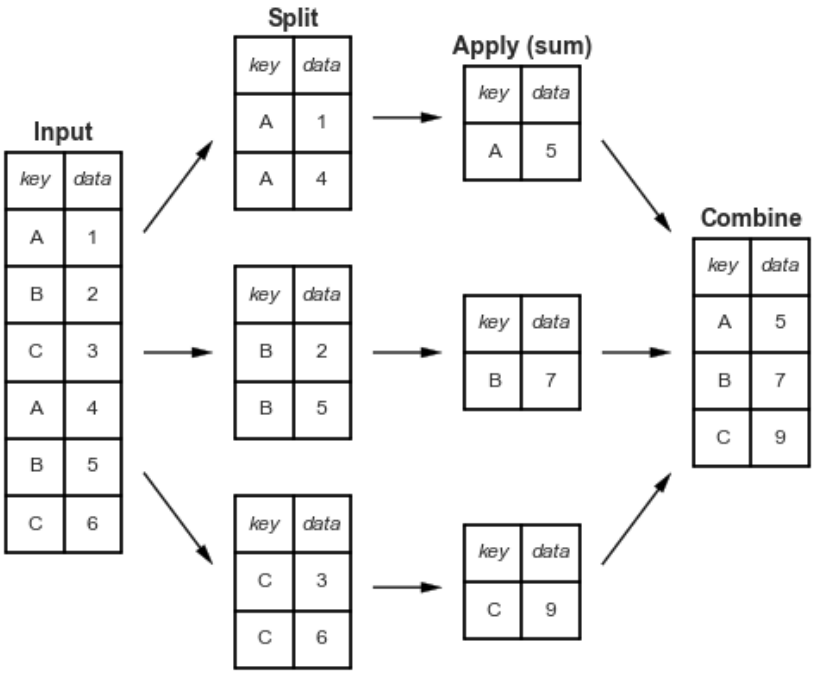
DataFrame initial

	key	data	toto
0	A	0	a
1	B	1	a
2	C	2	b
3	A	3	b
4	B	4	a
5	C	5	b

GroupBy sum sur la colonne 'key'

key	data	toto
A	3	ab
B	5	aa
C	7	bb

C'est ce que l'on appelle le paradigme « split-apply-combine » (diviser-appliquer-combiner) : d'abord on génère un groupe de DataFrame en fonction de la valeur de la clé spécifiée. À chaque groupe, on applique ensuite une fonction d'agrégation (ici la somme) pour obtenir une seule ligne. Enfin on fusionne les lignes obtenues dans un nouveau DataFrame.



Passons à un exemple plus riche.

```
data = pd.DataFrame( {  "foo" : ["one", "one", "one", "two", "two", "two"],
                        "bar" : ["A", "A", "B", "A", "B", "B"],
                        "baz" : [1, 2, 3, 4, 5, 6],
                        "zoo" : ["x", "y", "z", "q", "w", 't'] } )

data
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

On va regrouper les données en fonction de la colonne `foo`.

```
data_gbfoo = data.groupby("foo")
data_gbfoo
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f29dba4faa0>
```

On obtient un objet `DataFrameGroupBy`, c'est-à-dire **une sorte de dictionnaire de DatFrame dont les clés sont les valeurs uniques de la colonne `foo` et les valeurs sont les DataFrame correspondant à chaque groupe** :

```
for name, group in data_gbfoo:
    display(Markdown(f"**Nom du groupe : {name}**"))
    display(group)
```

Nom du groupe : one

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z

Nom du groupe : two

	foo	bar	baz	zoo
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

On peut regrouper les données en fonction de plusieurs colonnes. Par exemple, on peut regrouper les données en fonction des colonnes `foo` et `bar` :

```
data_gbfoobar =  data.groupby(['foo', 'bar'])
for name, group in data_gbfoobar:
    display(Markdown(f"**Nom du groupe : {name}**"))
    display(group)
```

Nom du groupe : ('one', 'A')

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y

Nom du groupe : ('one', 'B')

	foo	bar	baz	zoo
2	one	B	3	z

Nom du groupe : ('two', 'A')

	foo	bar	baz	zoo
3	two	A	4	q

Nom du groupe : ('two', 'B')

	foo	bar	baz	zoo
4	two	B	5	w
5	two	B	6	t

On peut appliquer une même opération sur chaque groupe en même temps. Par exemple, on peut extraire une seule colonne de cet objet, et on obtient un objet `SeriesGroupBy` :

```
data_gbfoo_baz = data_gbfoo["baz"]
data_gbfoo_baz
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7f29dba4ee10>
```

On obtient ainsi un objet `SeriesGroupBy`, qui est une sorte de dictionnaire de Series. Chaque Series correspond à un groupe.

```
for name, group in data_gbfoo_baz:
    display(Markdown(f"**Nom du groupe : {name}**"))
    display(group)
```

Nom du groupe : one

```
0    1
1    2
2    3
Name: baz, dtype: int64
```

Nom du groupe : two

```
3    4
4    5
5    6
Name: baz, dtype: int64
```

Pour ne pas itérer sur les groupes mais juste connaître des informations sur les groupes, on peut utiliser :

- `ngroups` permet de connaître le nombre de groupes obtenus (ici 2 car on a deux valeurs uniques dans la colonne `foo`)
- `size` combien d'éléments sont dans chaque groupe
- `groups` les indices des lignes de chaque groupe avec la méthode.

```
# Combien de groupes ?
nb_groupe = data_gbfoo.ngroups
display(nb_groupe)

# Combien d'éléments par groupe ?
nb_element_par_groupe = data_gbfoo.size()
display(nb_element_par_groupe)

# Quelles sont les indices des lignes de chaque groupe ?
group_indices = data_gbfoo.groups
display(group_indices)
```

```
2
```

```
foo
one    3
two    3
dtype: int64
```

```
{'one': [0, 1, 2], 'two': [3, 4, 5]}
```

L'object `DataFrameGroupBy` peut être immaginé comme un dictionnaire de DataFrames où chaque clé est un nom de groupe et chaque valeur est le DataFrame correspondant à ce groupe. Cependant, ce c'est pas un vrai dictionnaire, en particulier on ne peut pas accéder aux groupes avec la syntaxe des dictionnaires `data_gbfoo['one']`.

Pour récuperer les dataframes on doit utiliser la méthode `get_group()` :

```
data_one = data_gbfoo.get_group("one") # les lignes du groupe "one" constitue un dataframe
data_two = data_gbfoo.get_group("two") # les lignes du groupe "two" constitue un dataframe
display(data_one, data_two)
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z

	foo	bar	baz	zoo
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

L'utilisation principale de `groupby` nest pas la simple création de sous-DataFrames. Généralement **on applique des fonctions d'agrégation** sur les groupes obtenus. **On obtient un unique DataFrame** avec une ligne par groupe.

Par exemple, on peut calculer la somme des éléments de chaque groupe (on se rappelle que l'operation "somme" entre chaînes de caractères correspond à la concaténation de ces chaînes) :

```
data_gbfoo.sum()
```

foo	bar	baz	zoo
one	AAB	6	xyz
two	ABB	15	qwt

```
data_gbfoo['baz'].sum()
```

```
foo
one     6
two    15
Name: baz, dtype: int64
```

On peut aussi **regrouper par plusieurs colonnes**, puis appliquer une fonction d'agrégation.

Exemple : regroupons les données de `data` en utilisant les colonnes `foo` et `bar`, puis calculons la somme de la colonne `baz` pour chaque groupe unique de `foo` et `bar`.

Voici les étapes décomposées :

1. **`data_grfoobar = data.groupby(['foo', 'bar'])`** : regroupe le DataFrame par les colonnes `foo` et `bar`, en obtenant autant de groupes que de couples différents ;
2. **`data_grfoobar['baz'].sum()`** : calcule la somme de `baz` pour chaque groupe.
3. Facultatif : **`.unstack()`** qui transforme le résultat en une table pivotée, en déplaçant les valeurs uniques de `bar` en colonnes. Ainsi, pour chaque valeur de `foo`, on obtient les sommes de `baz` réparties par les différentes valeurs de `bar`.

Le résultat est un DataFrame avec les valeurs de `foo` en index et celles de `bar` en colonnes, où chaque cellule contient la somme correspondante de `baz`.

```
# on regroupe les couples (foo, bar)
data_grfoobar = data.groupby(['foo', 'bar'])

display(data_grfoobar.get_group(('one', 'A'))) # df pour qui foo == 'one' et bar == 'A'
display(data_grfoobar.get_group(('one', 'B'))) # df pour qui foo == 'one' et bar == 'B'
display(data_grfoobar.get_group(('two', 'A'))) # df pour qui foo == 'two' et bar == 'A'
display(data_grfoobar.get_group(('two', 'B'))) # df pour qui foo == 'two' et bar == 'B'
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y

	foo	bar	baz	zoo
2	one	B	3	z

	foo	bar	baz	zoo
3	two	A	4	q

	foo	bar	baz	zoo
4	two	B	5	w
5	two	B	6	t

```
# Pour chaque groupe, on somme les valeurs de la colonne "baz"
data_gbfoobarbaz = data_grfoobar['baz'].sum()
data_gbfoobarbaz # c'est une Series avec un MultiIndex
```

```
foo  bar
one  A       3
     B       3
two  A       4
     B      11
Name: baz, dtype: int64
```

```
data_gbfoobarbaz.index
```

```
MultiIndex([('one', 'A'),
            ('one', 'B'),
            ('two', 'A'),
            ('two', 'B')],
           names=['foo', 'bar'])
```

Avec `.unstack()`, on peut transformer la Series MultiIndex en DataFrame pour une meilleure lisibilité :

```
data_gbfoobarbaz.unstack()
```

bar foo	A	B
one	3	3
two	4	11

16.2. `pivot_table` : tableaux croisés

Le résultat obtenu avec l'instruction :

`data.groupby(['col1', 'col2'])['col3'].sum().unstack()`

est un **tableau croisé dynamique** : à chaque couple `(col1, col2)`, on associe la **somme des valeurs de `col3`**.

Si le DataFrame `data` contient les colonnes `col1`, `col2` et `col3`, on peut imaginer que chaque ligne du DataFrame correspond à un triplet `(x_i, y_i, z_i)`. On peut alors representer cela comme un point en 3D :

- `col1` est la coordonnée `x_i`
- `col2` est la coordonnée `y_i`
- `col3` est la coordonnée `z_i`

On peut imaginer que l'on cherche à représenter une fonction de deux variables $z = f(x, y)$, où x et y sont les coordonnées dans le plan horizontal, et z est la hauteur associée à chaque point (x, y). **Problème** : si le couple `(x_i, y_i)` est unique dans le DataFrame, on a un seul point pour cette position. Mais si plusieurs lignes j ont le même couple `(x_i, y_i)` mais des valeurs différentes `(z_i)_j`, cela correspond à plusieurs point empilés verticalement, ce qui ne représente plus une fonction de $\mathbb{R}^2$ vers $\mathbb{R}$. Dans ce cas, pour obtenir une valeur unique $\tilde{z}_i$, il faut **agréger** ces différentes valeurs. C'est là qu'intervient l'agrégation (ici la somme) :

$$(x_i, y_i) \mapsto \tilde{z}_i = \sum_j (z_i)_j$$

On pourrait également utiliser d'autres fonctions d'agrégation comme `mean`, `max`, `min`, etc.

C'est une opération très courante en analyse de données. Pandas propose donc une méthode dédiée : `pivot_table`.

```
pivot = data.pivot_table(
    index='col1',
    columns='col2',
    values='col3',
    aggfunc='sum'
)
```

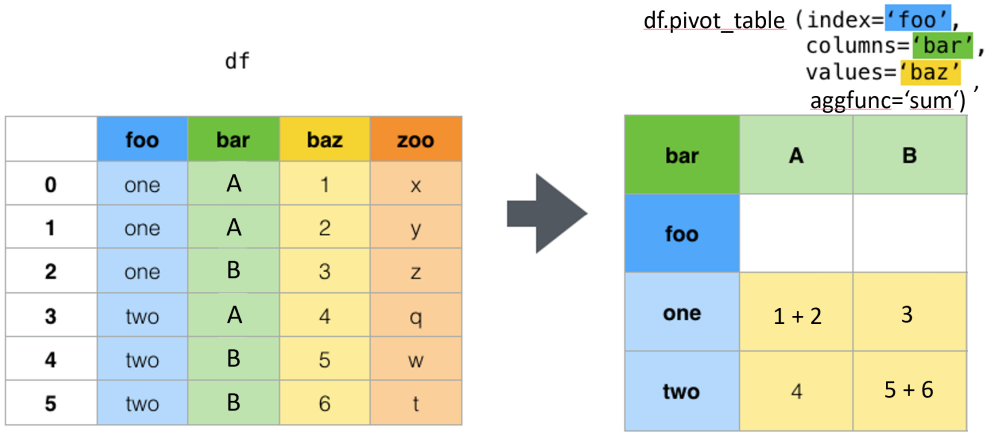
On applique la méthode `pivot_table` sur le DataFrame `data` en précisant :

- `index` : le nom de la colonne dans de DataFrame qui sera la ligne du nouveau tableau
- `columns` : le nom de la colonne dans le DataFrame qui sera la colonne du nouveau tableau
- `values` : le nom de la colonne sur laquelle appliquer l'agrégation
- `aggfunc` : la fonction d'agrégation (`sum`, `mean`, `count`, `min`, `max`, etc.). Si ce n'est pas spécifié, par défaut on calcule la moyenne (mean) des valeurs.

Avec `pivot_table`, on obtient un résultat équivalent à

`groupby(...).agg(...).unstack()`

mais plus lisible et directement structuré sous forme de tableau croisé.




```
data
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

```
data.pivot_table(index='foo', columns='bar', values='baz', aggfunc='sum')
```

bar	A	B
foo		
one	3	3
two	4	11

Lorsqu'on a besoin de calculer des sous-totaux, on peut utiliser l'option `margins=True` :

```
data.pivot_table(index='foo',
                 columns='bar',
                 values='baz',
                 aggfunc='sum',
                 margins=True)
```

bar foo	A	B	All
one	3	3	6
two	4	11	15
All	7	14	21

16.3. `crosstab` : tableaux croisés

`crosstab` est une fonction principalement utilisée pour **compter les occurrences ou les fréquences** des combinaisons de valeurs de deux colonnes.

```
pd.crosstab(data['col1'], data['col2'])
```

Cette fonction crée un tableau croisé indipendamment d'un DataFrame, en utilisant deux séries ou colonnes comme entrées :

- `index` : les données stockées dans une séries ou une colonne d'un DataFrame qui sera la ligne du nouveau tableau
- `columns` : les données stockées dans une séries ou une colonne d'un DataFrame qui sera la colonne du nouveau tableau

Pour chaque combinaison unique de valeurs dans `col1` et `col2`, `crosstab` compte le nombre d'occurrences et remplit le tableau avec ces comptes.

`crosstab` peut également agréger les valeurs d'une colonne supplémentaire, mais la syntaxe est différente de `pivot_table` :

```
pd.crosstab(
    index=data['col1'],
    columns=data['col2'],
    values=data['col3'],
    aggfunc=sum
)
```

- `index` et `columns` : comme précédemment
- `values` : les données stockées dans une séries ou une colonne d'un DataFrame qui sera la colonne à agréger
- `aggfunc` : fonction d'agrégation (par défaut `count` pour compter les occurrences, mais on peut utiliser `sum`, `mean`, etc.)

Enfin, `crosstab` propose un argument très pratique, `normalize`, qui permet d'afficher les résultats en pourcentages (par rapport à la somme totale, aux lignes ou aux colonnes) en une seule instruction, sans calcul supplémentaire.

```
data
```

	foo	bar	baz	zoo
0	one	A	1	x
1	one	A	2	y
2	one	B	3	z

	foo	bar	baz	zoo
3	two	A	4	q
4	two	B	5	w
5	two	B	6	t

La commande `pd.crosstab(data['foo'], data['bar'])` crée un tableau croisé (ou "contingence") en comptant les occurrences de chaque combinaison unique de valeurs dans les colonnes `foo` et `bar` du DataFrame `data`.

1. **`pd.crosstab(data['foo'], data['bar'])`** : compte le nombre de fois que chaque paire de valeurs (`foo`,`bar`) apparaît dans le DataFrame.
2. Le résultat est un nouveau DataFrame avec :
   - Les valeurs uniques de `foo` comme index (lignes).
   - Les valeurs uniques de `bar` comme colonnes.
   - Les cellules contiennent le nombre d'occurrences de chaque combinaison `(foo, bar)`.

```
pd.crosstab(data['foo'], data['bar'])
```

bar foo	A	B
one	2	1
two	1	2

```
# idem que
data.pivot_table(values='baz', index='foo', columns='bar', aggfunc='count')
```

bar foo	A	B
one	2	1
two	1	2

C'est utile pour obtenir une vue d'ensemble des fréquences d'association entre les deux variables. Pour obtenir les fréquences au lieu des comptes bruts dans un tableau croisé de pandas, on ajoute l'argument `normalize=True`, chaque valeur est alors divisée par le total de toutes les combinaisons. Cela affichera chaque valeur comme une proportion du total, ce qui correspond aux fréquences relatives.

```
pd.crosstab(data['foo'], data['bar'], normalize=True)
```

bar foo	A	B
one	0.333333	0.166667
two	0.166667	0.333333

On peut aussi normaliser par ligne ou par colonne :

- Par ligne (chaque ligne totalisera 1) : `normalize='index'`
- Par colonne (chaque colonne totalisera 1) : `normalize='columns'`

```
pd.crosstab(data['foo'], data['bar'], normalize='index')
```

bar foo	A	B
one	0.666667	0.333333
two	0.333333	0.666667

16.4. Bonus : `melt`, du format large au format long

Bonus : hors séance. Cette section n'est pas nécessaire pour la suite du cours, mais elle vous servira dès que vous voudrez tracer des données groupées avec Seaborn ou Plotly.

`pivot_table` transforme un tableau **long** (une observation par ligne) en tableau **large** (une modalité par colonne). `melt` réalise l'opération **inverse**.

Pourquoi en aurait-on besoin ? Parce que Seaborn et Plotly attendent presque toujours le format **long** : une colonne qui dit *de quelle variable il s'agit*, une colonne qui dit *quelle en est la valeur*.

```
notes_large = pd.DataFrame({ "etudiant" : ["Alice", "Bob", "Chloé"],
                             "algebre"  : [12, 15, 9],
                             "analyse"  : [14, 8, 16] })
notes_large
```

	etudiant	algebre	analyse
0	Alice	12	14
1	Bob	15	8
2	Chloé	9	16

```
notes_long = notes_large.melt( id_vars    = "etudiant",
                               var_name   = "matiere",
                               value_name = "note" )
notes_long
```

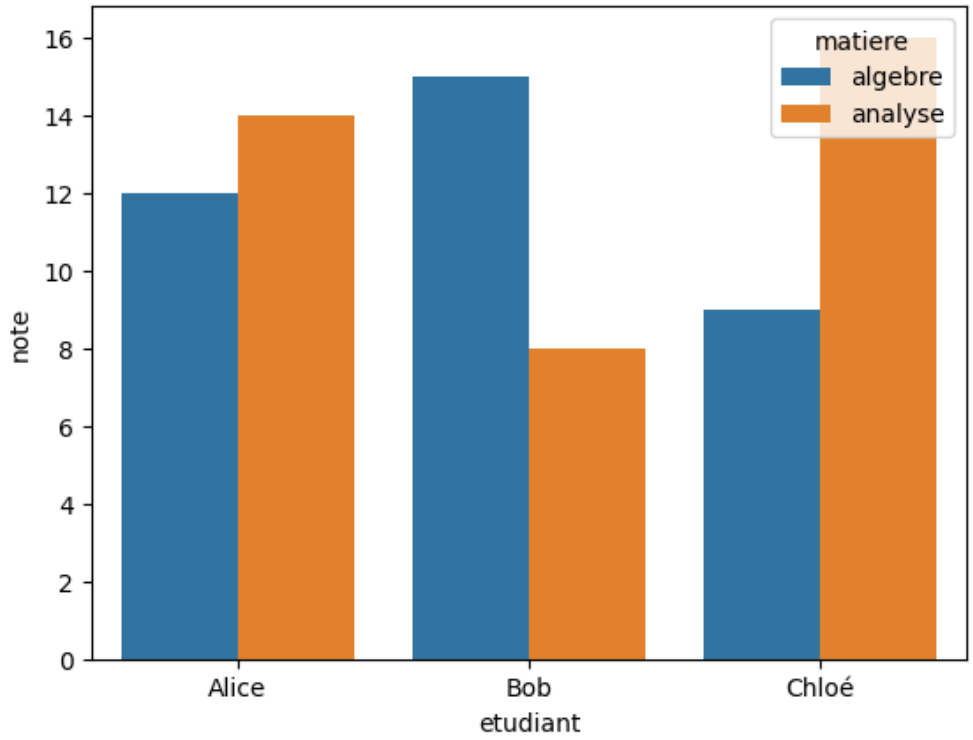
	etudiant	matiere	note
0	Alice	algebre	12
1	Bob	algebre	15
2	Chloé	algebre	9
3	Alice	analyse	14
4	Bob	analyse	8
5	Chloé	analyse	16

- `id_vars` : les colonnes à **conserver telles quelles** (ici l'identifiant de l'étudiant) ;
- `var_name` : le nom de la nouvelle colonne qui recevra les **anciens noms de colonnes** ;
- `value_name` : le nom de la nouvelle colonne qui recevra les **valeurs**.

Le format long se prête directement à une visualisation groupée :

```
import seaborn as sns

sns.barplot(data=notes_long, x="etudiant", y="note", hue="matiere");
```



Et `pivot_table` permet de revenir au format large :

```
notes_long.pivot_table(index="etudiant", columns="matiere", values="note")
```

matiere etudiant	algebre	analyse
Alice	12.0	14.0
Bob	15.0	8.0
Chloé	9.0	16.0

16.5. Arithmétique et alignement des données : différences avec `Numpy`

Il existe une différence très importante : lorsque vous faites une opération entre deux objets Series, **l'opération se fait terme à terme, mais les termes sont identifiés par leur index.** Si les index ne correspondent pas, le résultat sera un objet Series avec un index qui est l'union des deux index initiaux. Les valeurs correspondant à des index qui n'étaient pas présents dans les deux objets initiaux seront des valeurs manquantes, notées `NaN` (pour Not a Number).

```
ma_serie4 = pd.Series( np.random.randn(5), index=["A","B","C","D","E"] )
display(ma_serie4)
ma_serie5 = pd.Series( np.random.randn(4), index=["A","B","C","F"] )
display(ma_serie5)
ma_serie_somme = ma_serie4 + ma_serie5
display(ma_serie_somme)
```

```
A   -0.405428
B   -0.741871
C   -0.080453
D    0.626063
E   -1.865330
dtype: float64
```

```
A    0.286204
B    0.812590
C    0.220832
F   -0.737247
dtype: float64
```

```
A   -0.119224
B    0.070718
C    0.140379
D         NaN
E         NaN
F         NaN
dtype: float64
```

On voit ici que les deux Series ont en commun A, B et C. La somme donne bien une valeur qui est la somme des deux objets Series. Pour D, E et F, on obtient une valeur manquante car l'un des deux objets Series ne comprend pas d'index D, E ou F. On a donc la somme d'une valeur manquante et d'une valeur présente qui logiquement donne une valeur manquante. Si vous voulez faire une somme en supposant que les données manquantes sont équivalentes à 0, il faut utiliser : `s1.add(s2, fill_value=0)` :

```
ma_serie_somme_BIS = ma_serie4.add(ma_serie5, fill_value=0)
display(ma_serie_somme_BIS)
```

```
A   -0.119224
B    0.070718
C    0.140379
D    0.626063
E   -1.865330
F   -0.737247
dtype: float64
```

16.6. L'indexation des DataFrames

Vous pouvez avoir besoin de modifier les index de vos données. Pour cela, différentes options s'offrent à vous :

- `.reindex()` permet de sélectionner des colonnes et de réordonner les colonnes et les lignes. Si une colonne ou une ligne n'est pas présente dans le DataFrame initial, elle sera ajoutée avec des valeurs manquantes.

```
df5 = pd.DataFrame( np.random.randn(5,2), index=["a","b","c","d","e"], columns=["Col 1","Col 2"])
df5
```

	Col 1	Col 2
a	1.108168	0.632504
b	1.556843	1.267146
c	1.032881	-1.404079
d	-0.025248	0.105628
e	-1.924212	1.302465

```
# on ne garde que les lignes "c","d","e" mais on les réordonne
# on garde toutes les colonnes mais on les réordonne aussi
df6 = df5.reindex(index=["e","c","d"], columns=["Col 2","Col 1"])
df6
```

	Col 2	Col 1
e	1.302465	-1.924212
c	-1.404079	1.032881
d	0.105628	-0.025248

- `.rename()` permet de renommer les colonnes ou les lignes.

```
df7 = df6.rename(index={"e":"E", "c":"C"}, columns={"Col 1":"Premiere", "Col 2":"Seconde"})
df7
```

	Seconde	Premiere
E	1.302465	-1.924212
C	-1.404079	1.032881
d	0.105628	-0.025248

```
df8 = df7.rename(index=str.upper, columns=str.title)
df8
```

	Seconde	Premiere
E	1.302465	-1.924212
C	-1.404079	1.032881
D	0.105628	-0.025248

```
df9 = df8.rename(mapper=lambda x: x.upper(), axis=1)
df9
```

	SECONDE	PREMIERE
E	1.302465	-1.924212
C	-1.404079	1.032881
D	0.105628	-0.025248

16.6.1. Bonus : `set_index` et `reset_index`

Bonus : hors séance, mais très utile en pratique : `groupby` produit constamment des index qu'il faut ensuite remettre en colonnes.

- `.set_index("col")` promeut une **colonne** au rang d'index ;
- `.reset_index()` fait l'inverse : il remet l'index **dans une colonne** et renumérote les lignes.

```
villes = pd.DataFrame({ "ville"      : ["Toulon", "Nice", "Marseille"],
                        "dep"        : ["83", "06", "13"],
                        "population" : [176198, 342669, 873076] })

display(villes)
display(villes.set_index("ville"))
```

	ville	dep	population
0	Toulon	83	176198
1	Nice	06	342669

	ville	dep	population
2	Marseille	13	873076

ville	dep	population
Toulon	83	176198
Nice	06	342669
Marseille	13	873076

Une fois la colonne devenue index, elle est utilisable avec `loc` :

```
villes_idx = villes.set_index("ville")
villes_idx.loc["Nice"]
```

```
dep               06
population    342669
Name: Nice, dtype: object
```

`reset_index()` est l'opération réciproque. C'est le réflexe à avoir après un `groupby`, dont le résultat a pour index la clé de regroupement :

```
resume = villes.groupby("dep")["population"].sum()

display(resume)                # une Series indexée par le département
display(resume.reset_index())  # un DataFrame à deux colonnes, plus commode à réutiliser
```

```
dep
06    342669
13    873076
83    176198
Name: population, dtype: int64
```

	dep	population
0	06	342669
1	13	873076
2	83	176198

Enfin, `reset_index(drop=True)` **jette** l'ancien index au lieu d'en faire une colonne : pratique après un filtrage qui a laissé des trous dans la numérotation.

```
filtre = villes[ villes["population"] > 200_000 ]

display(filtre)                         # les index 1 et 2 d'origine ont été conservés
display(filtre.reset_index(drop=True))  # renumérotation de 0 à n-1
```

	ville	dep	population
1	Nice	06	342669
2	Marseille	13	873076

	ville	dep	population
0	Nice	06	342669
1	Marseille	13	873076

16.7. Copie *vs* vue

Lorsqu'on crée un objet à partir d'un autre : `df2 = df1`, `df2 = df1[["col"]]`, ou un DataFrame construit sur un tableau NumPy : pandas ne duplique pas systématiquement les données. Tantôt vous obtenez une **copie** indépendante, tantôt une **vue** qui partage la même zone mémoire que l'original : modifier l'une modifie alors l'autre, parfois à votre insu.

La règle à retenir. Si vous voulez un objet réellement indépendant de l'original, dites-le explicitement :

```
df2 = df1.copy()
```

```
df_origine = pd.DataFrame({ "a" : [1, 2, 3],
                            "b" : [4, 5, 6] })

df_copie = df_origine.copy()
df_copie.loc[0, "a"] = 999

display(df_origine)   # intact
display(df_copie)     # seul le second a changé
```

	a	b
0	1	4
1	2	5
2	3	6

	a	b
0	999	4
1	2	5
2	3	6

16.7.1. Pourquoi ? Ce qui se passe réellement

Un DataFrame construit à partir d'un tableau NumPy **partage la mémoire** de ce tableau. Modifier le DataFrame modifie donc aussi le tableau d'origine :

```
arr1 = np.arange(6).reshape(3, 2)
df1  = pd.DataFrame(arr1)

print("Avant modification")
display(arr1)
display(df1)

df1.iloc[1, 1] = 22

print("Après modification d'une valeur du DataFrame")
display(df1)
display(arr1)   # ici, l'array a été modifié lui aussi
```

```
Avant modification
```

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

	0	1
0	0	1
1	2	3
2	4	5

```
Après modification d'une valeur du DataFrame
```

	0	1
0	0	1
1	2	22
2	4	5

```
array([[ 0,  1],
       [ 2, 22],
       [ 4,  5]])
```

Le même phénomène explique l'avertissement `SettingWithCopyWarning`, que vous rencontrerez tôt ou tard : pandas vous prévient qu'il ne sait pas si l'objet que vous modifiez est une vue ou une copie, et donc si votre modification atteindra l'original ou se perdra.

16.7.2. Ce qui change avec le *Copy-on-Write*

Depuis pandas 2.0, un mode appelé **Copy-on-Write** (CoW) est disponible ; il devient le comportement **par défaut à partir de pandas 3.0**. Sa règle est simple et uniforme : *toute opération se comporte comme si elle renvoyait une copie*. Le partage de mémoire continue d'exister en interne, pour la performance, mais il n'est plus jamais observable : et l'avertissement `SettingWithCopyWarning` disparaît.

Activons-le le temps d'une démonstration, avec `pd.option_context` qui restaure automatiquement le réglage à la sortie du bloc :

```
with pd.option_context("mode.copy_on_write", True):

    arr2 = np.arange(6).reshape(3, 2)
    df2  = pd.DataFrame(arr2)
    df2.iloc[1, 1] = 22

    display(df2)    # le DataFrame est bien modifié
    display(arr2)   # ici, en revanche, l'array n'est PLUS modifié

display(Markdown(f"Copy-on-Write actif en dehors du bloc ? **{pd.options.mode.copy_on_write}**"))
```

	0	1
0	0	1
1	2	22
2	4	5

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

Copy-on-Write actif en dehors du bloc ? **False**

	pandas 2.x (comportement actuel)	Copy-on-Write (défaut en pandas 3.0)
Modification « à distance »	possible, et silencieuse	impossible
`SettingWithCopyWarning`	fréquent	n'existe plus
Rôle de `.copy()`	**indispensable** pour s'isoler	explicite votre intention

Dans les deux cas, la règle du début de section reste la bonne : **si vous voulez une copie, écrivez `.copy()`.**

Chapitre 17.

Combiner des tables

Jusqu'ici nous avons travaillé sur un seul DataFrame. En pratique, l'information est presque toujours répartie dans **plusieurs tables** qu'il faut rapprocher. Deux situations, très différentes, se présentent.

- **Empiler** des tables qui décrivent *les mêmes variables* pour *des individus différents* : les ventes de janvier et celles de février, les étudiants du groupe A et ceux du groupe B. C'est le rôle de `pd.concat`.
- **Joindre** des tables qui décrivent *des variables différentes* pour *les mêmes individus*, en s'appuyant sur une colonne commune appelée **clé** : une table des communes et une table des départements, reliées par le code du département. C'est le rôle de `pd.merge`.

Retenez la distinction : **empiler ajoute des lignes, joindre ajoute des colonnes.**

17.1. Empiler des tables : `pd.concat`

`pd.concat` prend une **liste** de DataFrame et les empile. Par défaut l'empilement est vertical (`axis=0`) : les lignes du second viennent sous celles du premier, et les colonnes sont appariées **par leur nom**.

```
ventes_janvier = pd.DataFrame({ "produit"  : ["stylo", "cahier", "gomme"],
                                "quantite" : [120, 80, 45] })

ventes_fevrier = pd.DataFrame({ "produit"  : ["stylo", "cahier", "règle"],
                                "quantite" : [95, 110, 30] })

display(ventes_janvier, ventes_fevrier)
```

	produit	quantite
0	stylo	120
1	cahier	80
2	gomme	45

	produit	quantite
0	stylo	95
1	cahier	110
2	règle	30

```
ventes = pd.concat([ventes_janvier, ventes_fevrier])
ventes
```

	produit	quantite
0	stylo	120
1	cahier	80
2	gomme	45

	produit	quantite
0	stylo	95
1	cahier	110
2	règle	30

Remarquez les **index dupliqués** : chaque DataFrame a conservé le sien (0, 1, 2 puis 0, 1, 2). C'est rarement souhaitable. Deux solutions :

- `ignore_index=True` renumérote les lignes de 0 à $n-1$;
- `keys=[...]` conserve la **provenance** de chaque ligne, dans un index à deux niveaux.

```
pd.concat([ventes_janvier, ventes_fevrier], ignore_index=True)
```

	produit	quantite
0	stylo	120
1	cahier	80
2	gomme	45
3	stylo	95
4	cahier	110
5	règle	30

```
ventes_par_mois = pd.concat([ventes_janvier, ventes_fevrier], keys=["Janvier", "Février"])
ventes_par_mois
```

		p
Janvier	0	s
	1	c
	2	g
Février	0	s
	1	c
	2	r

Si les deux tables n'ont pas exactement les mêmes colonnes, `concat` fait l'**union** des colonnes et comble les trous par des `NaN` :

```
ventes_mars = pd.DataFrame({ "produit"  : ["stylo", "trousse"],
                             "quantite" : [60, 12],
                             "remise"   : [0.1, 0.2] })   # colonne supplémentaire

pd.concat([ventes_janvier, ventes_mars], ignore_index=True)
```

	produit	quantite	remise
0	stylo	120	NaN
1	cahier	80	NaN
2	gomme	45	NaN
3	stylo	60	0.1
4	trousse	12	0.2

Avec `axis=1`, l'empilement devient **horizontal** : les colonnes sont juxtaposées et les lignes appariées **par leur index**, exactement selon la règle d'alignement vue plus haut.

```
prix  = pd.DataFrame({ "prix"  : [1.2, 2.5, 0.6] }, index=["stylo", "cahier", "gomme"])
stock = pd.DataFrame({ "stock" : [300, 150, 80]  }, index=["stylo", "cahier", "gomme"])

pd.concat([prix, stock], axis=1)
```

	prix	stock
stylo	1.2	300
cahier	2.5	150
gomme	0.6	80

Attention : `pd.concat(..., axis=1)` apparie les lignes **par leur index**, et non par leur ordre d'apparition. Si les index ne coïncident pas, vous obtiendrez des `NaN` sans le moindre message d'erreur.

```
stock_desordre = pd.DataFrame({ "stock" : [80, 300, 150] }, index=["gomme", "stylo", "cahier"])

# L'ordre des lignes diffère, mais l'appariement se fait sur l'index : le résultat est correct
display(pd.concat([prix, stock_desordre], axis=1))

stock_partiel = pd.DataFrame({ "stock" : [300, 150] }, index=["stylo", "règle"])

# Ici "cahier" et "gomme" n'ont pas de stock, "règle" n'a pas de prix : des NaN apparaissent
display(pd.concat([prix, stock_partiel], axis=1))
```

	prix	stock
stylo	1.2	300
cahier	2.5	150
gomme	0.6	80

	prix	stock
stylo	1.2	300.0
cahier	2.5	NaN
gomme	0.6	NaN
règle	NaN	150.0

17.2. Joindre deux tables : `pd.merge`

Changeons de situation. Nous disposons maintenant de deux tables qui décrivent **des choses différentes** :

- une table des **communes**, avec leur population et le code de leur département ;
- une table des **départements**, qui donne le nom du département et sa région.

La colonne `dep` est présente dans les deux : c'est la **clé** de la jointure.

```
communes = pd.DataFrame({
    "commune"    : ["Toulon", "Hyères", "Nice", "Marseille", "Lyon"],
    "dep"        : ["83", "83", "06", "13", "69"],
    "population" : [176198, 57635, 342669, 873076, 522250] })

departements = pd.DataFrame({
    "dep"             : ["06", "13", "69", "83"],
    "nom_departement" : ["Alpes-Maritimes", "Bouches-du-Rhône", "Rhône", "Var"],
    "region"          : ["PACA", "PACA", "Auvergne-Rhône-Alpes", "PACA"] })

display(communes, departements)
```

	commune	dep	population
0	Toulon	83	176198
1	Hyères	83	57635
2	Nice	06	342669
3	Marseille	13	873076
4	Lyon	69	522250

	dep	nom_departement	region
0	06	Alpes-Maritimes	PACA
1	13	Bouches-du-Rhône	PACA
2	69	Rhône	Auvergne-Rhône-Alpes
3	83	Var	PACA

```
pd.merge(communes, departements, on="dep")
```

	commune	dep	population	nom_departement	region
0	Toulon	83	176198	Var	PACA
1	Hyères	83	57635	Var	PACA
2	Nice	06	342669	Alpes-Maritimes	PACA
3	Marseille	13	873076	Bouches-du-Rhône	PACA
4	Lyon	69	522250	Rhône	Auvergne-Rhône-Alpes

Que s'est-il passé ? Pour **chaque ligne** de la table de gauche, pandas a cherché dans la table de droite les lignes ayant la **même valeur de clé**, puis a recollé les colonnes. La table des départements comptait 4 lignes, celle des communes 5 ; le résultat en compte 5, car plusieurs communes partagent le même département.

On peut écrire indifféremment `pd.merge(communes, departements, on="dep")` ou `communes.merge(departements, on="dep")`.

17.2.1. Les quatre types de jointure : l'argument `how`

Que faire des lignes qui **n'ont aucune correspondance** dans l'autre table ? C'est tout l'objet de l'argument `how` :

`how`	Lignes conservées	Équivalent SQL
`"inner"` *(défaut)*	seulement les clés présentes **dans les deux** tables	`INNER JOIN`
`"left"`	**toutes** celles de la table de gauche	`LEFT JOIN`
`"right"`	**toutes** celles de la table de droite	`RIGHT JOIN`
`"outer"`	**toutes**, des deux côtés	`FULL OUTER JOIN`

Les colonnes sans correspondance sont remplies par des `NaN`.

```
# On ajoute une commune dont le département est absent de la table des départements (Ajaccio, 2A),
# et un département dans lequel aucune commune n'est listée (Paris, 75).

communes2 = pd.concat(
    [communes, pd.DataFrame({"commune":["Ajaccio"], "dep":["2A"], "population":[71361]})],
    ignore_index=True)

departements2 = pd.concat(
```

```
    [departements, pd.DataFrame({"dep":["75"], "nom_departement":["Paris"],
↪   "region":["Île-de-France"]})],
    ignore_index=True)

display(communes2, departements2)
```

	commune	dep	population
0	Toulon	83	176198
1	Hyères	83	57635
2	Nice	06	342669
3	Marseille	13	873076
4	Lyon	69	522250
5	Ajaccio	2A	71361

	dep	nom_departement	region
0	06	Alpes-Maritimes	PACA
1	13	Bouches-du-Rhône	PACA
2	69	Rhône	Auvergne-Rhône-Alpes
3	83	Var	PACA
4	75	Paris	Île-de-France

```
for methode in ["inner", "left", "right", "outer"]:
    display(Markdown(f"**`how = \"{methode}\"`**"))
    display(pd.merge(communes2, departements2, on="dep", how=methode))
```

how = "inner"

	commune	dep	population	nom_departement	region
0	Toulon	83	176198	Var	PACA
1	Hyères	83	57635	Var	PACA
2	Nice	06	342669	Alpes-Maritimes	PACA
3	Marseille	13	873076	Bouches-du-Rhône	PACA
4	Lyon	69	522250	Rhône	Auvergne-Rhône-Alpes

how = "left"

	commune	dep	population	nom_departement	region
0	Toulon	83	176198	Var	PACA
1	Hyères	83	57635	Var	PACA
2	Nice	06	342669	Alpes-Maritimes	PACA
3	Marseille	13	873076	Bouches-du-Rhône	PACA
4	Lyon	69	522250	Rhône	Auvergne-Rhône-Alpes
5	Ajaccio	2A	71361	NaN	NaN

how = "right"

	commune	dep	population	nom_departement	region
0	Nice	06	342669.0	Alpes-Maritimes	PACA
1	Marseille	13	873076.0	Bouches-du-Rhône	PACA
2	Lyon	69	522250.0	Rhône	Auvergne-Rhône-Alpes

	commune	dep	population	nom_departement	region
3	Toulon	83	176198.0	Var	PACA
4	Hyères	83	57635.0	Var	PACA
5	NaN	75	NaN	Paris	Île-de-France

`how = "outer"`

	commune	dep	population	nom_departement	region
0	Toulon	83	176198.0	Var	PACA
1	Hyères	83	57635.0	Var	PACA
2	Nice	06	342669.0	Alpes-Maritimes	PACA
3	Marseille	13	873076.0	Bouches-du-Rhône	PACA
4	Lyon	69	522250.0	Rhône	Auvergne-Rhône-Alpes
5	Ajaccio	2A	71361.0	NaN	NaN
6	NaN	75	NaN	Paris	Île-de-France

Comparez attentivement ces quatre tableaux :

- `inner` **supprime silencieusement** Ajaccio (département inconnu) *et* Paris (aucune commune) ;
- `left` conserve Ajaccio, avec `NaN` pour le nom du département et la région ;
- `right` conserve Paris, avec `NaN` pour la commune et la population ;
- `outer` conserve les deux.

Le choix n'est pas anodin : `inner`, la valeur **par défaut**, est précisément celle qui perd des données sans prévenir.

17.2.2. Quand les colonnes clés n'ont pas le même nom

Les deux tables ne nomment pas toujours la clé de la même façon. On précise alors `left_on` et `right_on` :

```
departements_bis = departements.rename(columns={"dep": "code_departement"})

pd.merge(communes, departements_bis,
         left_on="dep", right_on="code_departement")
```

	commune	dep	population	code_departement	nom_departement	region
0	Toulon	83	176198	83	Var	PACA
1	Hyères	83	57635	83	Var	PACA
2	Nice	06	342669	06	Alpes-Maritimes	PACA
3	Marseille	13	873076	13	Bouches-du-Rhône	PACA
4	Lyon	69	522250	69	Rhône	Auvergne-Rhône-Alpes

La clé apparaît alors en double (`dep` et `code_departement`) ; on supprime la colonne redondante :

```
pd.merge(communes, departements_bis,
         left_on="dep", right_on="code_departement").drop(columns="code_departement")
```

	commune	dep	population	nom_departement	region
0	Toulon	83	176198	Var	PACA
1	Hyères	83	57635	Var	PACA
2	Nice	06	342669	Alpes-Maritimes	PACA
3	Marseille	13	873076	Bouches-du-Rhône	PACA
4	Lyon	69	522250	Rhône	Auvergne-Rhône-Alpes

	commune	dep	population	nom_departement	region

17.2.3. Vérifier une jointure

Une jointure **réussit toujours** : y compris quand elle ne fait pas ce que vous croyez. Deux réflexes :

1. comparer le **nombre de lignes** avant et après ;
2. utiliser `indicator=True`, qui ajoute une colonne `_merge` indiquant l'origine de chaque ligne.

```
resultat = pd.merge(communes2, departements2, on="dep", how="outer", indicator=True)

display(resultat)
display(resultat["_merge"].value_counts())
```

	commune	dep	population	nom_departement	region	_merge
0	Toulon	83	176198.0	Var	PACA	both
1	Hyères	83	57635.0	Var	PACA	both
2	Nice	06	342669.0	Alpes-Maritimes	PACA	both
3	Marseille	13	873076.0	Bouches-du-Rhône	PACA	both
4	Lyon	69	522250.0	Rhône	Auvergne-Rhône-Alpes	both
5	Ajaccio	2A	71361.0	NaN	NaN	left_only
6	NaN	75	NaN	Paris	Île-de-France	right_only

```
_merge
both          5
left_only     1
right_only    1
Name: count, dtype: int64
```

Les deux pièges classiques de `merge`

1. **Des lignes disparaissent.** Avec `how="inner"` (le défaut), toute clé absente d'un côté fait disparaître la ligne, sans message. Vérifiez systématiquement le nombre de lignes avant et après.
2. **Des lignes se dupliquent.** Si la clé n'est **pas unique** dans la table de droite, chaque ligne de gauche est répétée autant de fois qu'il y a de correspondances : et tous les totaux calculés ensuite sont faux.

```
# La clé "83" apparaît deux fois à droite : la table des départements est mal construite
departements_doublon = pd.concat(
    [departements, pd.DataFrame({"dep":["83"], "nom_departement":["Var (doublon)"], "region":["PACA"]})],
    ignore_index=True)

jointure = pd.merge(communes, departements_doublon, on="dep")

display(Markdown(f"communes          : {len(communes)} lignes"))
display(Markdown(f"après la jointure : **{len(jointure)}** lignes, "
                 "car Toulon et Hyères ont été dupliquées"))
display(jointure)
```

communes : 5 lignes

après la jointure : **7** lignes, car Toulon et Hyères ont été dupliquées

	commune	dep	population	nom_departement	region
0	Toulon	83	176198	Var	PACA

	commune	dep	population	nom_departement	region
1	Toulon	83	176198	Var (doublon)	PACA
2	Hyères	83	57635	Var	PACA
3	Hyères	83	57635	Var (doublon)	PACA
4	Nice	06	342669	Alpes-Maritimes	PACA
5	Marseille	13	873076	Bouches-du-Rhône	PACA
6	Lyon	69	522250	Rhône	Auvergne-Rhône-Alpes

L'argument `validate` permet de **déclarer la relation attendue**. Pandas lève alors une erreur si l'hypothèse est fausse, au lieu de produire silencieusement un résultat faux :

```
# On déclare : plusieurs communes pour un même département, donc "many_to_one"
try:
    pd.merge(communes, departements_doublon, on="dep", validate="many_to_one")
except Exception as e:
    display(Markdown(f"{type(e).__name__} : {e}"))
```

MergeError : Merge keys are not unique in right dataset ; not a many-to-one merge

17.2.4. `join` : la variante qui travaille sur l'index

La méthode `.join()` fait la même chose que `merge`, mais en utilisant par défaut l'**index** des deux tables plutôt qu'une colonne. C'est commode lorsque la clé est déjà l'index :

```
communes_idx     = communes.set_index("dep")
departements_idx = departements.set_index("dep")

communes_idx.join(departements_idx)
```

dep	commune	population	nom_departement	region
06	Nice	342669	Alpes-Maritimes	PACA
13	Marseille	873076	Bouches-du-Rhône	PACA
69	Lyon	522250	Rhône	Auvergne-Rhône-Alpes
83	Toulon	176198	Var	PACA
83	Hyères	57635	Var	PACA

17.2.5. En résumé

Objectif	Outil
Ajouter des **lignes** (mêmes colonnes)	`pd.concat([df1, df2])`
Ajouter des **colonnes**, appariées par l'index	`pd.concat([df1, df2], axis=1)`
Ajouter des **colonnes**, appariées par une clé commune	`pd.merge(df1, df2, on="cle", how=...)`
Ajouter des **colonnes**, appariées par l'index	`df1.join(df2)`

Et le réflexe à garder : **après chaque jointure, vérifiez le nombre de lignes.**

Chapitre 18.

Analyse statistique et visualisation : le jeu de données du Titanic

Nous allons effectuer une analyse exploratoire approfondie (EDA) de l'ensemble de données relatives au naufrage du Titanic. Notre objectif est d'étudier les données en détail et de voir comment même des observations simples peuvent révéler des informations précieuses.

Le naufrage du Titanic est l'un des plus célèbres de l'histoire. Le **15 avril 1912**, lors de son voyage inaugural, le RMS Titanic, un paquebot transatlantique britannique, a sombré après être entré en collision avec un iceberg, **tuant 1502 passagers et membres d'équipage sur 2224**. (Selon Wikipedia il y avait 1316 passagers et 889 membres d'équipage, ce qui donne 2205 personnes) L'une des raisons pour lesquelles le naufrage a causé tant de morts est qu'il n'y avait pas assez de canots de sauvetage pour les passagers et l'équipage. On a toujours entendu dire que certains groupes de personnes étaient plus susceptibles de survivre que d'autres, comme les femmes, les enfants et les classes supérieures. Nous allons analyser les données pour vérifier ces affirmations.

Dans cet exemple, nous allons récupérer un jeu de données (liste de personnes, caractéristiques, survivants ou non…) et analyser quelles catégories de personnes ont survécu. Nous utiliserons les bibliothèques **pandas**, **Matplotlib**, **Seaborn** et **Plotly** pour explorer et visualiser ces données.

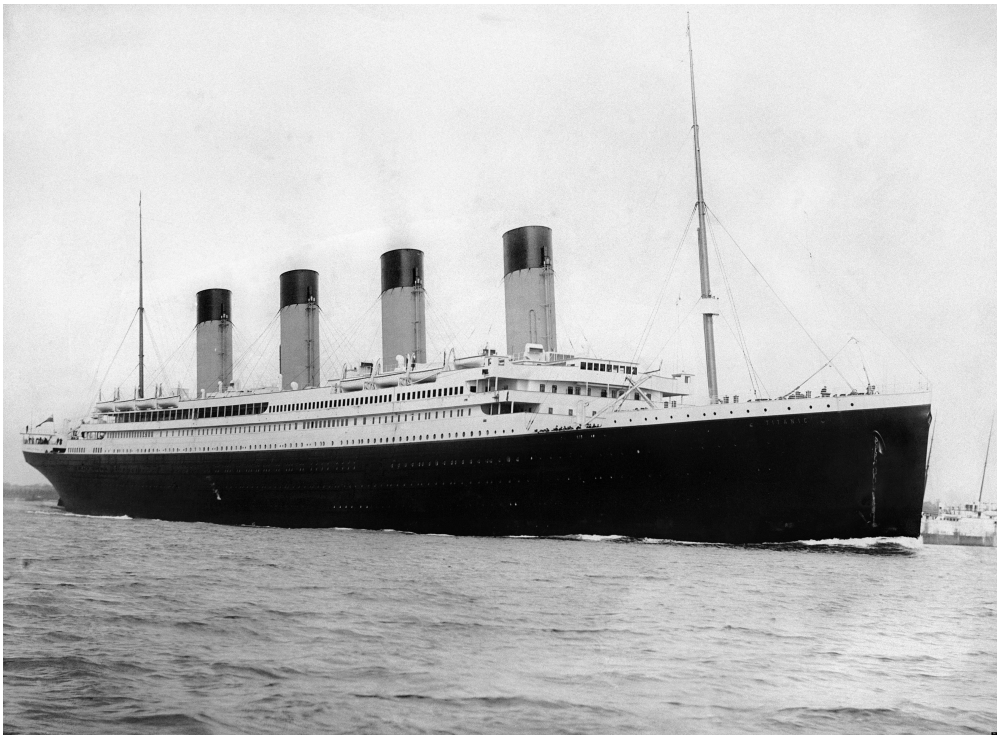


FIGURE 18.1. – Le Titanic à Southampton le 10 avril 1912

18.1. Importation des données

```
# Import data analysis libraries
# ==============================
import numpy as np
import pandas as pd
pd.set_option("display.max_columns", None)


# Import visualization libraries and set favourite style
# ======================================================
import matplotlib.pyplot as plt
plt.rcParams['font.size'] = 12
plt.rcParams['figure.figsize'] = (10, 6) # set default size of plots
# plt.style.use('ggplot') # set ggplot style

import seaborn as sns
sns.set_theme(style = "ticks", # "ticks", "whitegrid", "darkgrid", "white", "dark"
              palette = "pastel", # None, Set2, husl, deep, muted, bright, pastel, dark, colorblind
              rc = {"axes.spines.right": False, "axes.spines.top": False})


import plotly.express as px
from plotly.subplots import make_subplots

# Import Notebook display library
# ===============================
from IPython.display import display, Markdown


# Pour la génération automatique de graphiques interactifs dans le polycopié
# ==========================
import plotly.io as pio
import os
if os.getenv("QUARTO_OUTPUT_FORMAT") is not None:
    # Export Quarto HTML
    pio.renderers.default = "iframe_connected"
else:
    # Notebook interactif
    pio.renderers.default = "notebook_connected"
```

L'une des forces de pandas est l'importation et l'exportation des données. Ce package possède un ensemble de fonctions très large pour charger des données en mémoire et les exporter dans divers formats.

Pandas dédie un sous-répertoire entier du package à l'importation et à l'exportation vers des formats de données exploitables avec d'autres outils. On peut citer les formats csv, txt, Excel®, SAS®, SQL, HDF5 entre autres. Suivant le format, les outils seront différents, mais les principes restent les mêmes. Ainsi, nous allons considérer uniquement le format `csv`.

Un fichier **CSV** (.csv) est un fichier de données tabulaires. Le sigle CSV signifie *Comma Separated Values* qui se traduit par « valeurs séparées par des virgules ». L'avantage de ce type de fichier est qu'il s'agit d'un fichier texte qui ne conserve que les données du tableau (pas de mise en page) et peut être lu par n'importe quel tableur. Les données d'une même ligne sont souvent séparées par des points-virgules ou des virgules.

Pour importer les données stockées dans un fichier csv dont les éléments sont séparés par des virgules, on utilisera :

```
df = pd.read_csv('nom_du_fichier.csv', sep=',')
```

Dans l'exemple suivant nous allons importer et afficher un fichier csv contenant des données sur le Titanic.

```
df = pd.read_csv('titanic.csv')
```

18.2. Étude globale de l'ensemble de données

Dans cette partie nous allons utiliser les méthodes suivantes :

```
.shape
.info()
.head() .tail()
.columns
.isnull() .dropna() .fillna()
.unique() .nunique() .value_counts()
.describe() .min(), .max(), .mean(), .median(), .std()
```

La première étape de tout projet d'analyse de données consiste à examiner les données. Nous devons voir combien d'observations (lignes) et combien d'entités (colonnes) sont contenues, ce que signifient ces colonnes, et ainsi de suite. Cela nous aidera à nous familiariser avec l'ensemble de données, et pourrait même nous aider à évaluer quelles informations sont importantes et lesquelles ne le sont pas.

```
df.shape # tailles du dataset : 1309 lignes et 7 colonnes
# df.columns
```

```
(1309, 7)
```

Un moyen rapide de vérifier le contenu consiste à appeler les 5 premières lignes à l'aide de la méthode `.head()` sans spécifier l'argument entre parenthèses. Si nous voulons vérifier les 20 premières lignes, nous mettons 20 comme argument.

```
# df
df.head(20)
# df.tail(10)
```

	classe	survivant	nom	sexe	age	prix	port_depart
0	1	1	Allen, Miss. Elisabeth Walton	F	29.0000	211.3375	S
1	1	1	Allison, Master. Hudson Trevor	H	0.9167	151.5500	S
2	1	0	Allison, Miss. Helen Loraine	F	2.0000	151.5500	S
3	1	0	Allison, Mr. Hudson Joshua Creighton	H	30.0000	151.5500	S
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	F	25.0000	151.5500	S
5	1	1	Anderson, Mr. Harry	H	48.0000	26.5500	S
6	1	1	Andrews, Miss. Kornelia Theodosia	F	63.0000	77.9583	S
7	1	0	Andrews, Mr. Thomas Jr	H	39.0000	0.0000	S
8	1	1	Appleton, Mrs. Edward Dale (Charlotte Lamson)	F	53.0000	51.4792	S
9	1	0	Artagaveytia, Mr. Ramon	H	71.0000	49.5042	C
10	1	0	Astor, Col. John Jacob	H	47.0000	227.5250	C
11	1	1	Astor, Mrs. John Jacob (Madeleine Talmadge Force)	F	18.0000	227.5250	C
12	1	1	Aubart, Mme. Leontine Pauline	F	24.0000	69.3000	C
13	1	1	Barber, Miss. Ellen "Nellie"	F	26.0000	78.8500	S
14	1	1	Barkworth, Mr. Algernon Henry Wilson	H	80.0000	30.0000	S
15	1	0	Baumann, Mr. John D	H	NaN	25.9250	S
16	1	0	Baxter, Mr. Quigg Edmond	H	24.0000	247.5208	C
17	1	1	Baxter, Mrs. James (Helene DeLaudeniere Chaput)	F	50.0000	247.5208	C
18	1	1	Bazzani, Miss. Albina	F	32.0000	76.2917	C
19	1	0	Beattie, Mr. Thomson	H	36.0000	75.2417	C

On compte 7 caractéristiques (colonnes) décrivant chaque personne à bord du Titanic. Notre caractéristique cible (également appelée variable dépendante) est la colonne `survivant`, qui vaut 1 si la personne a survécu et 0 sinon. Dans ce cas, il est facile de déduire que cette colonne ne contient que des valeurs numériques 0 et 1. Cependant, pour d'autres caractéristiques (ou variables indépendantes), il peut ne pas être possible de déduire au premier coup d'œil le type de données contenues.

Afin de générer rapidement un tableau des types de données contenus dans chaque colonne, nous utilisons la méthode `.info()`.

```
# Info about data frame dimensions, column types, and file size

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   classe       1309 non-null   int64
 1   survivant    1309 non-null   int64
 2   nom          1309 non-null   object
 3   sexe         1309 non-null   object
 4   age          1046 non-null   float64
 5   prix         1308 non-null   float64
 6   port_depart  1307 non-null   object
dtypes: float64(2), int64(2), object(3)
memory usage: 71.7+ KB
```

18.2.1. Données manquantes ?

Combien de valeurs manquantes dans chaque colonne ?

```
df.isnull().sum().sort_values(ascending=False).to_frame('Données manquantes')
```

	Données manquantes
age	263
port_depart	2
prix	1
classe	0
survivant	0
nom	0
sexe	0

Il est utile d'utiliser la méthode `.isnull()` combinée avec `.sum()` ou `.mean()` pour connaître l'ampleur des valeurs manquantes dans un jeu de données.

```
(df.isnull().mean().sort_values(ascending = False) *
 ↪  100).round(2).sort_values(ascending=False).to_frame('% données manquantes')
```

	% données manquantes
age	20.09
port_depart	0.15
prix	0.08

	% données manquantes
classe	0.00
survivant	0.00
nom	0.00
sexe	0.00

Nous pouvons voir que la colonne `port_depart` contient 2 valeurs manquantes (indiquées par `NaN`). Qui sont les passagers pour lesquels on n'a pas ces informations ?

```
# mask = df['age'].isnull()
# df.loc[mask] # il y a 263 valeurs manquantes pour l'age, car 1309-1046=263

mask = df['port_depart'].isnull()
df.loc[mask]
```

	classe	survivant	nom	sexe	age	prix	port_depart
168	1	1	Icard, Miss. Amelie	F	38.0	80.0	NaN
284	1	1	Stone, Mrs. George Nelson (Martha Evelyn)	F	62.0	80.0	NaN

De même, nous pouvons voir que dans la colonne `age` il manque 263 données (nous n'allons pas les afficher). Peut-être que ces données n'ont pas été collectées ou ont été perdues. Que faire ? Nous pourrions soit supprimer ces lignes, soit remplacer les valeurs manquantes par la moyenne de l'âge des passagers par exemple. Les choix relatifs au traitement des valeurs manquantes ne sont pas des choix méthodologiques neutres. Pandas donne les outils techniques pour faire ceci mais la question de la légitimité de ces choix et de leur pertinence est propre à chaque jeu de données.

Pour le moment nous n'allons pas modifier les données.

```
# Option 1 : éliminer les lignes avec des valeurs manquantes
# df = df.dropna(axis=0)
# df.shape  # (1046, 7)

# Option 2 : remplacer par la moyenne (Attention : on modifie la réalité)
# df.fillna(df['age'].mean(), inplace=True)
```

18.2.2. Bilan sur l'ensemble de données

Conclusion : nous savons maintenant que ce jeu de données comporte 1309 lignes et 7 colonnes. 4 colonnes sont de type numérique (flottants et entiers), 3 colonnes sont non numériques :

- `classe` : valeurs possibles 1, 2 ou 3
- `survivant` : 0 (décédé), 1 (a survécu)
- `nom` : nom, prénom et titre
- `sexe` : H (homme), F (femme)
- `age` : en années
- `prix` : prix du ticket
- `port_depart` : trois ports d'embarquement, C = Cherbourg (France), S = Southampton (Angleterre), Q = Queenstown (Irlande)

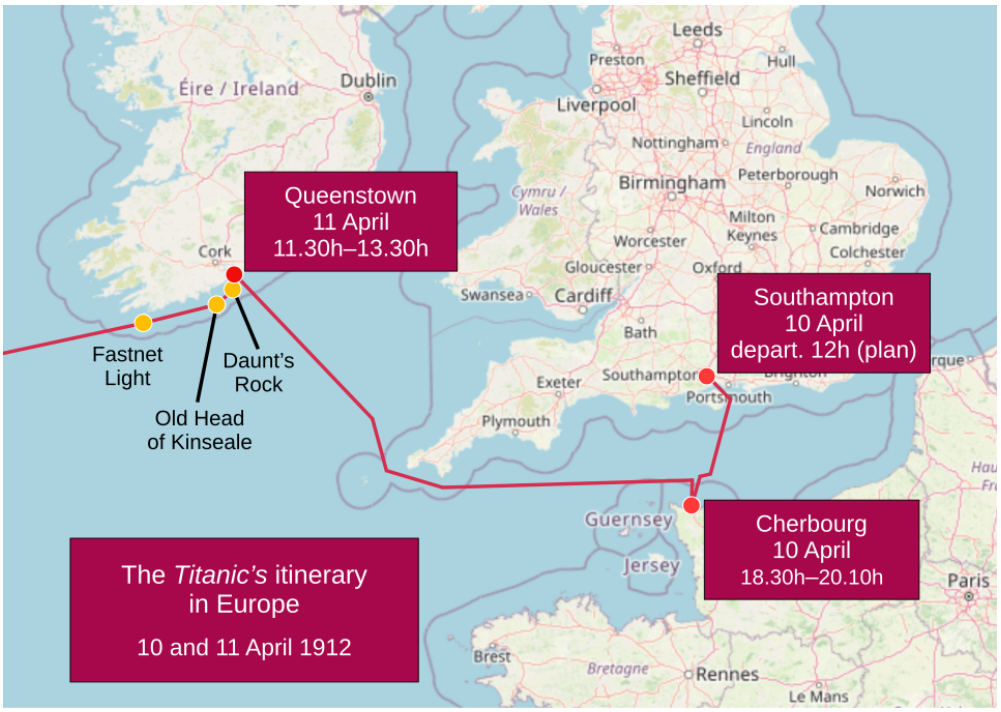


FIGURE 18.2. – Itinéraire en Europe

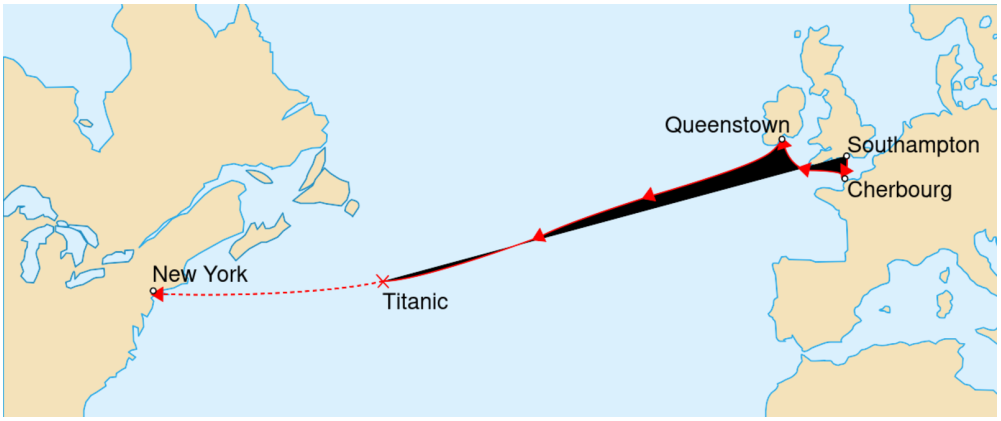


FIGURE 18.3. – Itinéraire prévu

18.2.3. Statistiques descriptives

Pour les colonnes numériques telles que `age`, `prix`, etc., nous aimerions connaître leurs valeurs moyennes, maximales et minimales pour voir si les données sont raisonnablement distribuées ou s'il y a des anomalies. Pour afficher un tableau de statistiques pour chaque colonne numérique, nous utilisons la méthode `.describe()`.

```
# df.describe(include='all')
display(df.describe().T)
display(df.describe(include='object').T)
# df.describe(include="all")
```

	count	mean	std	min	25%	50%	75%	max
classe	1309.0	2.294882	0.837836	1.0000	2.0000	3.0000	3.000	3.0000
survivant	1309.0	0.381971	0.486055	0.0000	0.0000	0.0000	1.000	1.0000

	count	mean	std	min	25%	50%	75%	max
age	1046.0	29.881135	14.413500	0.1667	21.0000	28.0000	39.000	80.0000
prix	1308.0	33.295479	51.758668	0.0000	7.8958	14.4542	31.275	512.3292

	count	unique	top	freq
nom	1309	1307	Connolly, Miss. Kate	2
sexe	1309	2	H	843
port_depart	1307	3	S	914

On note que la colonne `prix` a une valeur minimale de 0, ce qui semble étrange. Cela pourrait signifier que certaines personnes ont voyagé gratuitement.

18.3. Analyse univariée (chaque colonne individuellement)

Commençons par l'analyse univariée, qui consiste à explorer une seule variable à la fois. Cela nous aide à comprendre la distribution, les tendances et les caractéristiques sous-jacentes de cette variable. En traçant et en analysant les courbes d'une caractéristique, nous pouvons obtenir des informations précieuses sur les données avant de passer à des relations plus complexes.

Dans cette partie nous allons utiliser les méthodes suivantes :

```
.drop() .replace()
.value_counts()
.agg()
.cut() .map()
.plot(kind='bar') .plot(kind='hist')
sns.countplot() sns.catplot() sns.histplot()
px.histogram() px.pie()
```

Tout d'abord nous allons examiner la colonne `nom` pour retrouver des informations intéressantes sur les passagers (en vérifiant les informations indiquées sur le site https://titanicfacts.net/titanic-passengers/). Ensuite, nous analyserons les autres colonnes une par une. Nous nous intéresserons particulièrement aux colonnes `classe`, `sexe`, `survivant`, `age` et verrons ensuite comment ces caractéristiques influencent les chances de survie des passagers.

18.3.1. Colonne `nom`

18.3.1.1. Plus jeune et plus vieux passagers

Une autre valeur qui attire l'attention est l'âge minimum, soit 0,1667 ans, avec un éventuel bébé de 9 semaines à bord. De plus amples informations sur https://titanicfacts.net/titanic-passengers/ indiquent qu'il s'agit de Elizabeth Gladys "Milvina" Dean, une fille âgée de 2 mois et 13 jours au moment du naufrage (qui est décédée en 2009 à l'âge de 97 ans). Vérifions si cette information est correcte :

```
# mask_idx = df['age'].idxmin()  # index de la personne la plus jeune
# df.iloc[mask_idx]
```

```
mask = df['age'] == df['age'].min()
df.loc[mask]
```

	classe	survivant	nom	sexe	age	prix	port_depart
763	3	1	Dean, Miss. Elizabeth Gladys ”Millvina”	F	0.1667	20.575	S

Et le plus jeune passager garçon ? Toujours selon le site précédent, il s'agit de Master Gilbert Sigvard Emanuel Danbom, âgé de 4 mois et 29 jours, qui n'a malheureusement pas survécu. Vérifions si cette information est correcte :

```
# Filtrer les individus dont sex == "H"
filtered_df = df.loc[df['sexe'] == "H"]
# Trouver l'âge minimum dans cette sous-table
min_age = filtered_df['age'].min()
# Sélectionner les lignes correspondant à cet âge minimum
youngest = filtered_df.loc[filtered_df['age'] == min_age]
# Afficher les résultats
youngest
```

	classe	survivant	nom	sexe	age	prix	port_depart
747	3	0	Danbom, Master. Gilbert Sigvard Emanuel	H	0.3333	14.4	S

Et le passager le plus âgé ?

```
df.loc[df['age']==df['age'].max()]
```

	classe	survivant	nom	sexe	age	prix	port_depart
14	1	1	Barkworth, Mr. Algernon Henry Wilson	H	80.0	30.0	S

Selon le site précédent, il s'agit de Mr Johan Svensson, âgé de 74 ans et 10 mois, qui n'a pas survécu. Cependant, notre jeu de données ne contient pas cette information :

```
df[df['nom'].str.contains('Svensson', case=False, na=False)]
```

	classe	survivant	nom	sexe	age	prix	port_depart
979	3	0	Lundahl, Mr. Johan Svensson	H	51.0	7.0542	S
1235	3	0	Svensson, Mr. Johan	H	74.0	7.7750	S
1236	3	1	Svensson, Mr. Johan Cervin	H	14.0	9.2250	S
1237	3	0	Svensson, Mr. Olof	H	24.0	7.7958	S

Et la femme la plus âgée ?

```
# Filtrer les individus dont sex == "F"
filtered_df = df.loc[df['sexe'] == "F"]
# Trouver l'âge maximum dans cette sous-table
max_age = filtered_df['age'].max()
# Sélectionner les lignes correspondant à cet âge maximum
oldest = filtered_df.loc[filtered_df['age'] == max_age]
# Afficher les résultats
oldest
```

	classe	survivant	nom	sexe	age	prix	port_depart
61	1	1	Cavendish, Mrs. Tyrell William (Julia Florence...	F	76.0	78.85	S

Selon le site il s'agit de Mrs Mary Eliza Compton, âgée de 64 ans et 8 mois, qui a survécu. À nouveau, notre jeu de données ne contient pas cette information :

```
df[df['nom'].str.contains('Compton', case=False, na=False)]
```

	classe	survivant	nom	sexe	age	prix	port_depart
76	1	1	Compton, Miss. Sara Rebecca	F	39.0	83.1583	C
77	1	0	Compton, Mr. Alexander Taylor Jr	H	37.0	83.1583	C
78	1	1	Compton, Mrs. Alexander Taylor (Mary Eliza Ing...	F	64.0	83.1583	C

18.3.1.2. Plus jeune et plus vieux passagers à avoir survécu

```
# parmi les survivants, qui est le plus jeune et qui est le plus âgé ?
survivants = df[df['survivant'] == 1]
# plus jeune
min_age_survivant = survivants['age'].min()
plus_jeune_survivant = survivants[survivants['age'] == min_age_survivant]
display(Markdown("**Plus jeune survivant :**"))
display(plus_jeune_survivant)
# plus âgé
max_age_survivant = survivants['age'].max()
plus_vieux_survivant = survivants[survivants['age'] == max_age_survivant]
display(Markdown("**Plus vieux survivant :**"))
display(plus_vieux_survivant)
```

Plus jeune survivant :

	classe	survivant	nom	sexe	age	prix	port_depart
763	3	1	Dean, Miss. Elizabeth Gladys ”Millvina”	F	0.1667	20.575	S

Plus vieux survivant :

	classe	survivant	nom	sexe	age	prix	port_depart
14	1	1	Barkworth, Mr. Algernon Henry Wilson	H	80.0	30.0	S

18.3.1.3. Plus jeune et plus vieux passagers à être décédé

```
# parmi les decede, qui est le plus jeune et qui est le plus âgé ?
decede = df[df['survivant'] == 0]
# plus jeune
min_age_decede = decede['age'].min()
plus_jeune_decede = decede[decede['age'] == min_age_decede]
display(Markdown("**Plus jeune décédé :**"))
display(plus_jeune_decede)
# plus âgé
max_age_decede = decede['age'].max()
plus_vieux_decede = decede[decede['age'] == max_age_decede]
display(Markdown("**Plus vieux décédé :**"))
display(plus_vieux_decede)
```

Plus jeune décédé :

	classe	survivant	nom	sexe	age	prix	port_depart
747	3	0	Danbom, Master. Gilbert Sigvard Emanuel	H	0.3333	14.4	S

Plus vieux décédé :

	classe	survivant	nom	sexe	age	prix	port_depart
1235	3	0	Svensson, Mr. Johan	H	74.0	7.775	S

18.3.1.4. Le capitaine du Titanic ?

Si on cherche dans notre jeu de données le titre capitaine, on trouve le passager suivant :

```
df[df['nom'].str.contains('Capt', case=False, na=False)]
```

	classe	survivant	nom	sexe	age	prix	port_depart
81	1	0	Crosby, Capt. Edward Gifford	H	70.0	71.0	S

Ce n'est pas le capitaine du Titanic qui était Edward John Smith (1850-1912) mais il n'apparaît pas dans notre jeu de données.

```
df[df['nom'].str.contains('Smith', case=True, na=False)]
```

	classe	survivant	nom	sexe	age	prix	port_depart
267	1	0	Smith, Mr. James Clinch	H	56.0	30.6958	C
268	1	0	Smith, Mr. Lucien Philip	H	24.0	60.0000	S
269	1	0	Smith, Mr. Richard William	H	NaN	26.0000	S
270	1	1	Smith, Mrs. Lucien Philip (Mary Eloise Hughes)	F	18.0	60.0000	S
564	2	1	Smith, Miss. Marion Elsie	F	40.0	13.0000	S
1215	3	0	Smith, Mr. Thomas	H	NaN	7.7500	Q

18.3.1.5. Curiosités

Sur Wikipedia, à la page https://fr.m.wikipedia.org/wiki/Violet_Constance_Jessop il est indiqué que **Violet Constance Jessop** (2 octobre 1887 - 5 mai 1971), une hôtesse et infirmière britannique de la White Star Line, a survécu aux trois naufrages des navires de la classe Olympic. En effet, elle sert comme hôtesse à bord de l'*Olympic* quand il heurte le croiseur HMS Hawke le 20 septembre 1911. Elle se trouve également à bord du *Titanic* lorsqu'il fait naufrage le 15 avril 1912, et sert en tant qu'infirmière volontaire à bord du *Britannic* lorsqu'il fait naufrage en mer Égée le 21 novembre 1916.

Vérifions si elle apparaît dans notre jeu de données :

```
df[df['nom'].str.contains('Violet', case=False, na=False)]
```

	classe	survivant	nom	sexe	age	prix	port_depart
501	2	1	Mellinger, Miss. Madeleine Violet	F	13.0	19.5	S

La seule Violet dans notre jeu de données est une passagère de 13 ans, qui a embarqué à Southampton en 2ème classe. Elle a survécu au naufrage du Titanic mais n'est probablement pas la même personne que Violet Jessop.

Et **Molly (Margaret Tobin) Brown** (1867-1932) ? Elle est connue pour être l'une des rescapées du naufrage du Titanic. Sauvée à bord du canot n° 6, dans lequel elle déplore le comportement du quartier-maître Robert Hichens, elle participe ensuite à la création du Comité des Survivants. Un siècle après le naufrage, Margaret Brown est presque exclusivement surnommée l'« Insubmersible Molly Brown », bien qu'elle n'ait jamais été appelée ainsi de son vivant. Il s'agit d'une invention du cinéma hollywoodien, qui s'empare de son histoire pour en faire un mythe, parfois très éloigné de la vérité, notamment dans la comédie musicale La Reine du Colorado (1964) et le drame Titanic (1997).

```
df[df['nom'].str.contains('Tobin', case=False, na=False)]
```

	classe	survivant	nom	sexe	age	prix	port_depart
41	1	1	Brown, Mrs. James Joseph (Margaret Tobin)	F	44.0	27.7208	C
1249	3	0	Tobin, Mr. Roger	H	NaN	7.7500	Q

18.3.1.6. Doublons ?

```
df['nom'].nunique(), df['nom'].count()  # 1307 noms différents pour 1309 passagers
```

```
(1307, 1309)
```

On constate que la colonne `nom` contient 1307 valeurs uniques pour 1309 lignes, ce qui indique la présence de doublons. Affichons les lignes concernées pour examiner ces cas particuliers :

```
# keep=False signifie que chaque valeur de la colonne qui apparaît plus d'une fois sera marquée comme
↪ True
# Les valeurs uniques (apparaissant une seule fois) seront marquées comme False
mask = df['nom'].duplicated(keep=False)
df[mask]
```

	classe	survivant	nom	sexe	age	prix	port_depart
725	3	1	Connolly, Miss. Kate	F	22.0	7.7500	Q
726	3	0	Connolly, Miss. Kate	F	30.0	7.6292	Q
924	3	0	Kelly, Mr. James	H	34.5	7.8292	Q
925	3	0	Kelly, Mr. James	H	44.0	8.0500	S

18.3.1.7. Suppression de la colonne `nom`

Dans notre jeu de données, la colonne `nom` ne semble plus être pertinente pour l'analys qui va suivre. Nous pouvons la supprimer en utilisant la méthode `.drop()`. En effet, certaines colonnes peuvent être négligées si elles ne sont pas pertinentes pour l'analyse et inutile de les garder dans le DataFrame.

```
# errors="ignore" pour éviter une erreur si la colonne n'existe pas
df.drop('nom', axis='columns', inplace=True, errors="ignore")
```

18.3.2. Préambule : valeurs uniques dans chaque colonne

Pour chaque colonne, nous voulons connaître **combien de valeurs différentes** elle contient, **lesquelles** sont ces valeurs et **combien de fois chacune apparaît**.

Pour le nombre de valeurs différentes par colonne, nous avons :

```
df.nunique()
```

```
classe           3
survivant        2
sexe             2
age             98
prix           281
port_depart      3
dtype: int64
```

On note que la colonne `classe` ne contient que 3 valeurs différentes, la colonne `survivant` ne contient que 2 valeurs différentes, la colonne `sexe` ne contient que 2 valeurs différentes, la colonne `port_depart` ne contient que 3 valeurs différentes. Pour ces colonnes ayant moins de 5 valeurs différentes, nous pouvons lister quelles sont ces valeurs différentes en utilisant la méthode `.unique()` :

```
filtre = [df[col].nunique() < 5 for col in df.columns]

for col in df[df.columns[filtre]]:
    print(col, df[col].unique())
```

```
classe [1 2 3]
survivant [1 0]
sexe ['F' 'H']
port_depart ['S' 'C' nan 'Q']
```

Pour une colonne donnée, nous pouvons lister non seulement les valeurs différentes en utilisant la méthode `.unique()`, mais aussi compter le nombre d'occurrences de chaque valeurs différentes en utilisant la méthode `.value_counts()`.

Pour analyser plus en profondeur une colonne catégorielle, il est utile d'afficher à la fois : - Le **nombre absolu** d'occurrences de chaque catégorie - Le **pourcentage** que représente chaque catégorie par rapport au total - Le traitement des **valeurs manquantes** (si présentes)

Comme ces informations sont intéressantes à afficher pour toute colonne catégorielle ou numérique discrète, nous allons créer une fonction réutilisable nommée `afficher_compte_et_pourcentage()` qui : 1. Calcule le nombre d'occurrences de chaque valeur unique 2. Calcule le pourcentage correspondant 3. Gère l'affichage des valeurs manquantes de manière explicite 4. Retourne un DataFrame synthétique facile à lire

Cette fonction sera ensuite appelée pour différentes colonnes en changeant simplement le nom de la colonne passée en paramètre.

```
def afficher_compte_et_pourcentage(df, colonne):
    compte = df[colonne].value_counts(dropna=False)
    pourcentage = df[colonne].value_counts(normalize=True, dropna=False) * 100
    resultat = pd.DataFrame({'Nb occurrences': compte, 'Pourcentage (%)': round(pourcentage, 2)})
    # Remplacer 'nan' dans l'index par 'Valeur manquante'
    resultat.index = resultat.index.map(lambda x: 'Valeur manquante' if pd.isna(x) else str(x))
    return resultat
```

18.3.3. Colonne `classe`

Combien de passagers étaient dans chaque classe ?

Dans la colonne `classe` seulement trois catégories sont possibles : 1, 2 ou 3. La méthode `.value_counts()` permet de compter le nombre d'occurrences pour chaque catégorie unique dans une colonne.

```
# Résumé statistique de la colonne 'classe'
# ================================================================

colonne = df['classe']

display(Markdown("**Résumé statistique de la colonne 'classe'**"))
display(colonne.describe().to_frame())

display(Markdown("**Nombre de valeurs uniques**"))
display(colonne.nunique())

display(Markdown("**Valeurs uniques**"))
display(colonne.unique())

display(Markdown("**Combien d'occurrences de chaque valeur unique**"))
display(colonne.value_counts().to_frame('Occurrences'))

display(Markdown("**Proportions des valeurs uniques**"))
display(colonne.value_counts(normalize=True).to_frame('Proportions'))  # proportions
```

Résumé statistique de la colonne 'classe'

	classe
count	1309.000000
mean	2.294882
std	0.837836
min	1.000000
25%	2.000000
50%	3.000000
75%	3.000000
max	3.000000

Nombre de valeurs uniques

```
3
```

Valeurs uniques

```
array([1, 2, 3])
```

Combien d'occurrences de chaque valeur unique

classe	Occurrences
3	709
1	323
2	277

Proportions des valeurs uniques

classe	Proportions
3	0.541635
1	0.246753
2	0.211612

Selon le site Titanic Facts il est indiqué qu'il y avait 2222 personnes à bord du Titanic (passagers et équipage) dont 1327 de passagers effectivement à bord et répartis comme suit : 324 en première classe, 284 en deuxième classe et 709 en troisième classe. Il y avait 49% de places passagers inutilisées. Nos données semblent cohérentes avec ces informations.

```
afficher_compte_et_pourcentage(df, 'classe')
```

classe	Nb occurrences	Pourcentage (%)
3	709	54.16
1	323	24.68
2	277	21.16

Ce tableau montre que la majorité des passagers (plus précisément 54 %) étaient de 3e classe. Fait intéressant, il y avait plus de passagers en première classe (323 passagers) qu'en seconde classe (277 passagers), même si la différence n'est pas très grande.

Pour avoir une idée visuelle du nombre de passagers par classe, on peut afficher ces informations

- soit avec la fonction `.plot()` de pandas en spécifiant le type de graphique `bar` ou `pie`,
- soit avec la fonction `countplot()` de Seaborn

- soit avec les fonctions `histogram()` ou `pie()` de Plotly.

```
# Combien de fois chaque valeur unique apparaît dans la colonne ?
# ================================================================

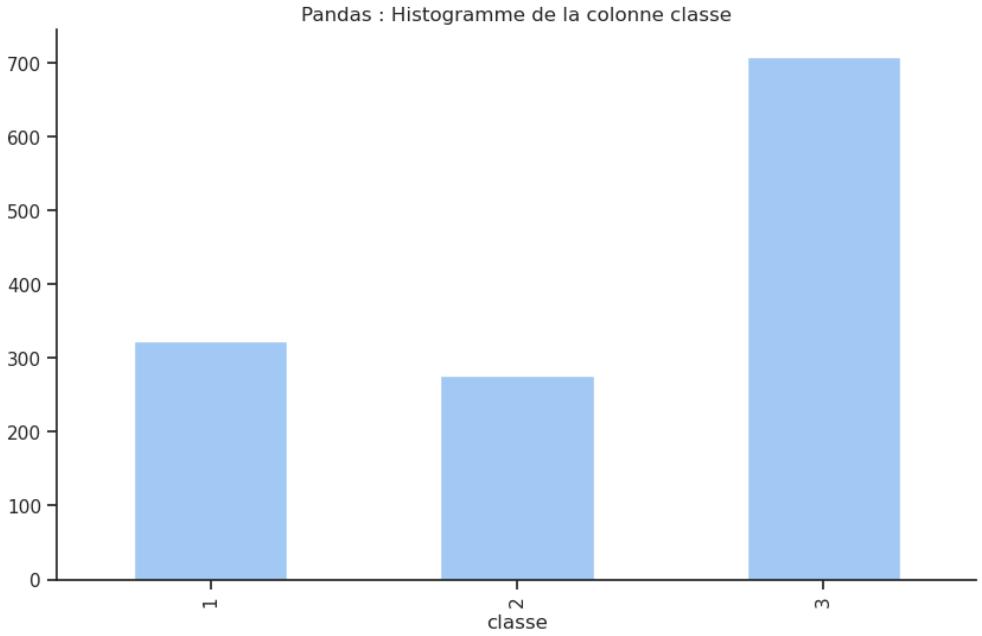
colonne = 'classe'

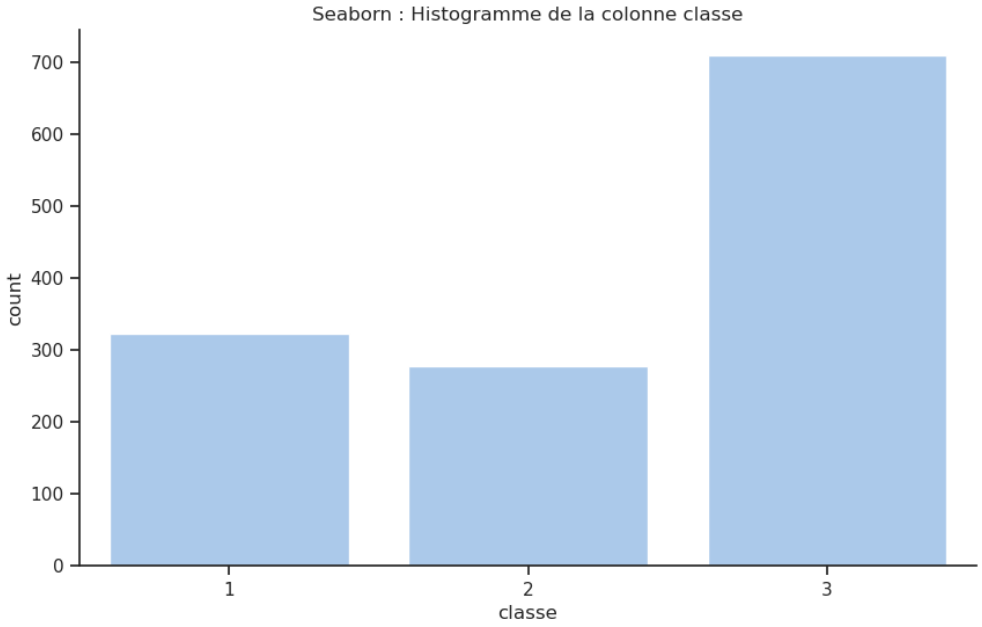
# Avec Pandas
# -----------------------------

# Avec la méthode value_counts() de la série, on peut obtenir le compte de chaque valeur unique dans la
↪  colonne.
df[colonne].value_counts().sort_index().plot(kind='bar');  # histogramme
plt.title(f'Pandas : Histogramme de la colonne {colonne}')
plt.show()


# Avec seaborn
# -----------------------------
ax = sns.countplot(data=df, x=colonne); # histogramme
plt.title(f'Seaborn : Histogramme de la colonne {colonne}')
plt.show()
# Ajouter les annotations directement sur les barres
# for p in ax.patches:  # Les rectangles/barres sont dans ax.patches
#     ax.annotate(f'{int(p.get_height())}',  # Le texte est la hauteur de la barre
#                 (p.get_x() + p.get_width() / 2., p.get_height()),  # Position : centre de la barre
#                 ha='center', va='bottom', fontsize=10, color='black')  # Alignement et style
```

```
# Avec plotly.express
# ------------------------------
px.histogram(df, x=colonne, text_auto=True, title=f'Plotly : Histogramme de la colonne
 ↪  {colonne}').update_layout(bargap=0.2).show()
```






```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Et en pourcentage ?

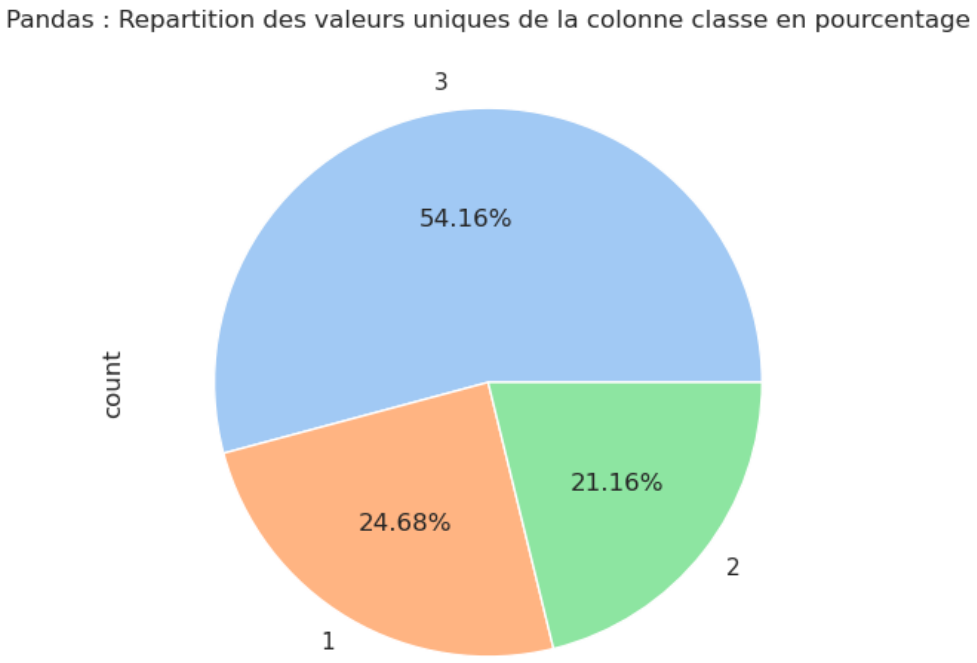
```
# Repartition des valeurs uniques en pourcentage
# ================================================================

colonne = 'classe'
```

```
# Avec Pandas et dans un camembert
# ------------------------------
df[colonne].value_counts().plot(kind='pie',
                                autopct='%1.2f%%',
                                #explode=[0.1]*len(df[colonne].unique())
                                )
plt.title(f'Pandas : Repartition des valeurs uniques de la colonne {colonne} en pourcentage')
plt.show();


# Avec seaborn
# ------------------------------
ax = sns.countplot(data=df, x=colonne, stat='percent'); # histogramme en pourcentage
plt.title(f'Seaborn : Repartition des valeurs uniques de la colonne {colonne} en pourcentage')
plt.show()
ax = sns.countplot(data=df, x=colonne, stat='proportion'); # histogramme en ptorportion
plt.title(f'Seaborn : Repartition des valeurs uniques de la colonne {colonne} en proportion')
plt.show()
# Note: Seaborn n'a pas de fonction native pour les diagrammes circulaires (pie charts)
# Il est recommandé d'utiliser matplotlib ou plotly pour ce type de visualisation


# Avec plotly.express
# ------------------------------
px.histogram(df, x='classe', histnorm='percent',title=f'Plotly : Repartition des valeurs uniques de la
 ↪  colonne {colonne} en pourcentage').update_traces(texttemplate='%{y:.2f} %',
 ↪  textposition='outside').update_layout(bargap=0.2).show()
#
px.pie(df, names=colonne,title=f'Plotly : Repartition des valeurs uniques de la colonne {colonne} en
 ↪  pourcentage').show()
```








```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

18.3.4. Colonne `sexe`

Si nécessaire, la colonne `sexe` peut être au préalable transformée en une colonne numérique en utilisant la méthode `.replace()` ou encore en utilisant la méthode `.map()` ou `.astype().cat.codes`.

```
# Catégorisation de la variable 'sexe' en 'sexe_code'
# ===================================================
# df['sexe_code'] = df['sexe'].astype('category').cat.codes
# df['sexe_code'] = df['sexe'].map({'H':0, 'F':1})
# df['sexe_code'] = df['sexe'].replace({'H':0, 'F':1})
```

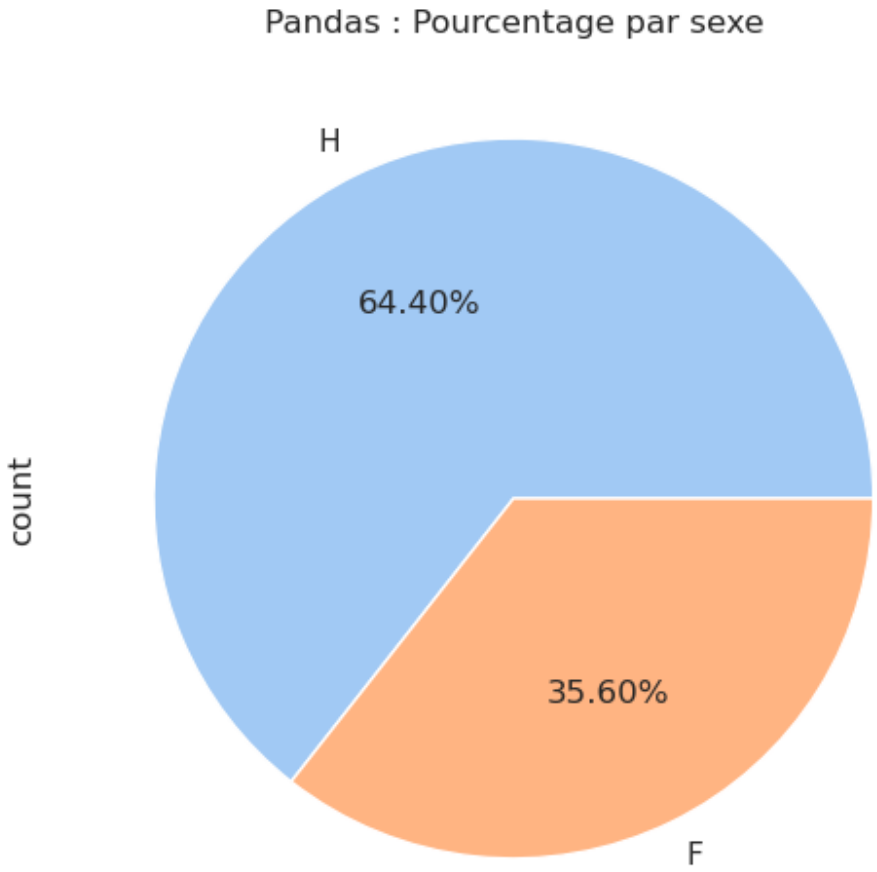
```
# Nombre d'occurrences et pourcentage de chaque catégorie
afficher_compte_et_pourcentage(df, 'sexe')
```

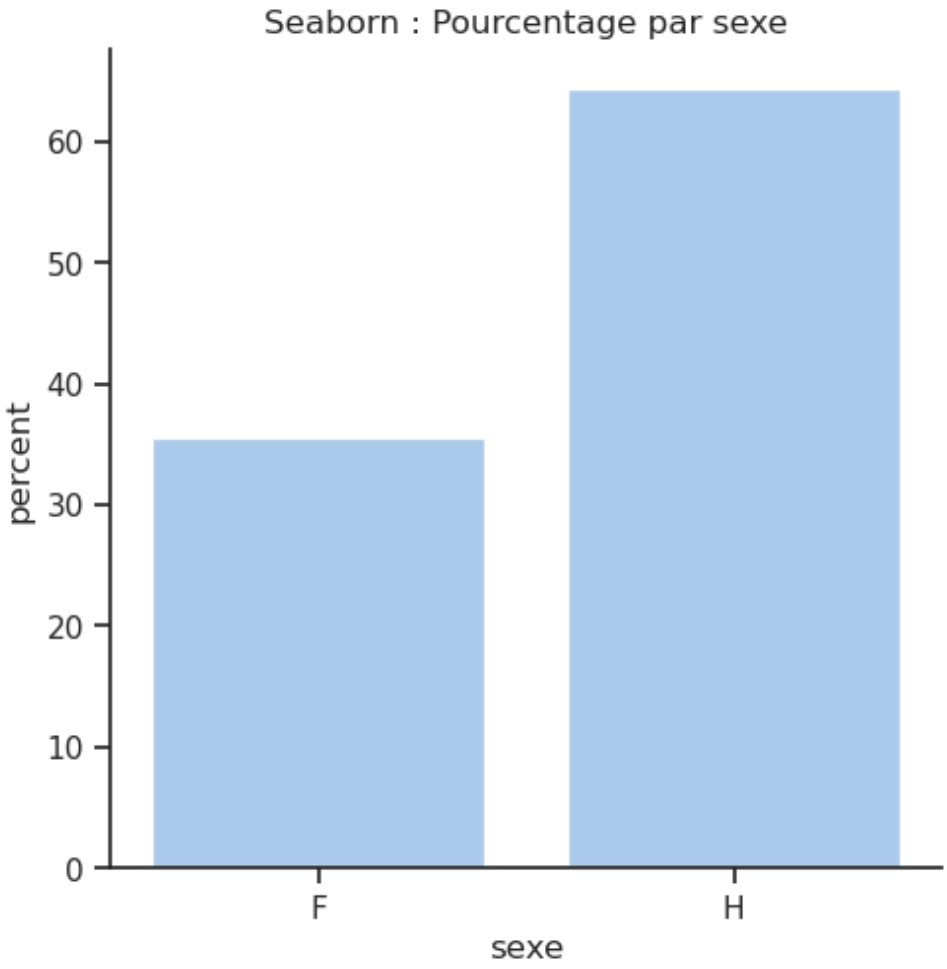
sexe	Nb occurrences	Pourcentage (%)
H	843	64.4
F	466	35.6

```
# Avec Pandas et dans un camembert
# -------------------------------
df['sexe'].value_counts().plot(kind='pie', autopct='%1.2f%%')
plt.title('Pandas : Pourcentage par sexe')
plt.show()

# Avec Seaborn
# -------------------------------
# 'catplot()': Figure-level interface for drawing categorical plots onto a FacetGrid.
# sns.catplot(x='sexe', data=df, kind='count');
sns.catplot(x='sexe', data=df, kind='count', stat='percent')
plt.title('Seaborn : Pourcentage par sexe')
plt.show()

# Avec plotly.express
# -------------------------------
# px.pie(df, names='sexe').show()
# px.histogram(df, x='sexe', text_auto=True).show()
px.pie(df,
       names='sexe',
       color='sexe',
       title='Plotly : Pourcentage par sexe',
         ).update_traces( texttemplate="%{label} : %{value} (%{percent})" ,
                          textposition="outside",    # texte à l'extérieur du secteur
                          pull=[0.05]*len(df['sexe'].unique())# léger décalage pour chaque secteur
                        ).show()
```


Pandas : Pourcentage par sexe
H
64.40%
count
35.60%
F




```
Unable to display output for mime type(s): text/html
```

18.3.5. Colonne `survivant`

Selon le site Titanic Facts, 706 personnes ont survécu, dont 492 passagers et 214 membres d'équipage. Vérifions si nos données confirment cette information :

```
# Compte et le pourcentage de chaque catégorie

afficher_compte_et_pourcentage(df, 'survivant')
```

survivant	Nb occurrences	Pourcentage (%)
0	809	61.8
1	500	38.2

Seulement environ 38 % des personnes ont survécu. Existe-t-il un moyen de savoir quels types de personnes ont eu le plus de chances de survivre ? Y a-t-il des caractéristiques particulières partagées par les survivants ? Pour répondre à ces questions, nous examinerons plus tard les autres colonnes et leurs relations avec la colonne survivant.

```
# Barres : matplotlib ou seaborn

# df['survivant'].value_counts().plot(kind='bar');
# sns.countplot(data=df, x='survivant');

# df['survivant'].value_counts(normalize=True).plot(kind='bar');
# sns.countplot(data=df, x='survivant', stat='percent');
# sns.countplot(data=df, x='survivant', stat='proportion');
```

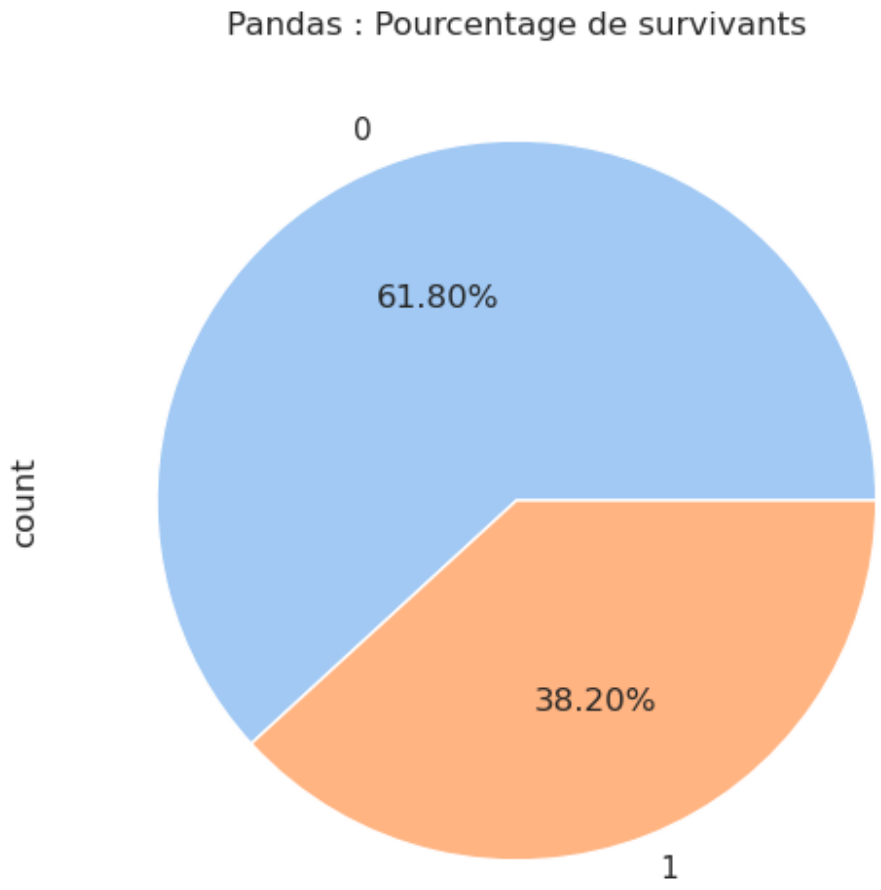
```
# Pourcentage dans un camembert
# ================================================================

# Avec Pandas
# ------------------------------
df['survivant'].value_counts().plot(kind='pie', autopct='%1.2f%%')
plt.title('Pandas : Pourcentage de survivants')
plt.show()


# Avec plotly.express
# ------------------------------

# px.pie(df, names='survivant')

# Mieux : chaque secteur affiche directement le nom (1 ou 0), le nombre d'occurrences et le pourcentage.
px.pie(df,
       names='survivant',
       color='survivant',
       color_discrete_sequence=px.colors.qualitative.Pastel,
       title='Plotly : Pourcentage de survivants'
       ).update_traces( texttemplate="%{label} : %{value} (%{percent})" ,
                        textposition="outside",    # texte à l'extérieur du secteur
                        pull=[0.05]*len(df['survivant'].unique())# léger décalage pour chaque secteur)
                        ).show()
```




```
Unable to display output for mime type(s): text/html
```

18.3.6. Colonne age

Analysons d'abord les données numériques pour la colonne de l'âge. On peut choisir de n'afficher que certaines informations (par exemple, la moyenne et l'écart-type) en utilisant la méthode `.agg()` :

```
# Statistiques descriptives pour une variable numérique
# ============================================================

# Méthode 1 : la méthode describe()
# ---------------------------------
# df[['age']].describe()

# Méthode 2 : en choisissant les statistiques souhaitées
# ------------------------------------------------------
# df[['age']].agg(['count','mean','median','std','min','max','nunique'])

# Méthode 3 : encore mieux, en définissant une fonction pour le nombre de valeurs manquantes et
 ↪  l'utilisant dans agg()
#
 ↪  ------------------------------------------------------------------------------------------------------------------------

missing_percent = lambda x: x.isna().mean() * 100
missing_percent.__name__ = 'missing_percent'   # pour un joli nom dans le résultat

def missing(x):
```

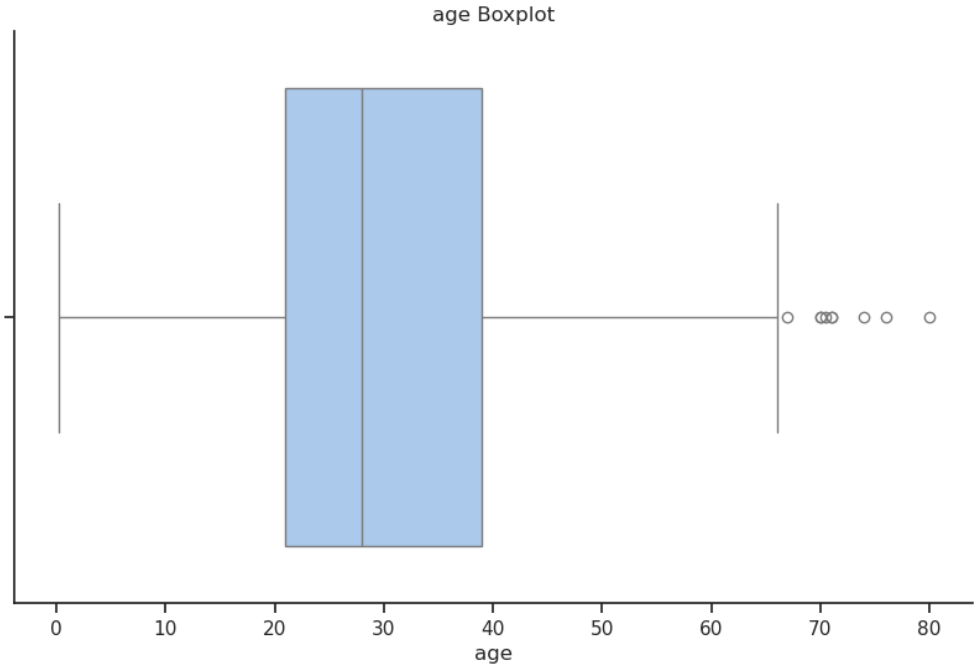
```
    return x.isna().sum()

df[['age']].agg({
    'age': ['count', missing, missing_percent, 'mean', 'median', 'std', 'min', 'max', 'nunique']
}).rename(index={'missing': 'missing_count'}).T
```

	count	missing_count	missing_percent	mean	median	std	min	max	nunique
age	1046.0	263.0	20.091673	29.881135	28.0	14.4135	0.1667	80.0	98.0

Comme on peut le voir, l'âge moyen des passagers est d'environ 29,9 ans, avec un écart-type de 14,5 ans. Cependant, il manque 263 valeurs dans cette colonne, ce qui représente environ 20 % des données. Nous pourrions soit supprimer ces lignes, soit remplacer les valeurs manquantes par la moyenne de l'âge des passagers par exemple ou simplement ne pas les prendre en compte dans notre analyse. Pour le moment nous n'allons pas modifier les données.

```
# Boxplot pour la variable 'age' :
# quartiles, médiane, valeurs extrêmes et outliers
# ================================================================
sns.boxplot(x=df["age"])
plt.title("age Boxplot")
plt.show()
```



On peut afficher l'histogramme de la distribution de l'âge des passagers en utilisant le type de graphique `hist` de la fonction `.plot()` ou utiliser la fonction `histplot()` de Seaborn ou `histogram()` de Plotly Express. Dans les trois cas on peut décider de regrouper les âges par intervalles de 10 ans en utilisant l'argument `bins`.

On peut aussi ajouter une courbe de densité (KDE = Kernel Density Estimation) : c'est une courbe lissée qui représente la distribution d'un ensemble de données. Elle sert à visualiser la forme générale de la distribution, sans dépendre des bins de l'histogramme.

```
# Pandas
# ------------------------------
# Rque : Pandas n'a pas de kde=True dans plot(kind='hist'), il a une fonction KDE séparée.
# Utiliser density=True pour que l'histogramme et le KDE soient sur la même échelle
# Dans ce cas en ordonnée on a la densité de probabilité (somme des barres = 1) et non plus le nombre
↪  d'occurrences
```

```
df['age'].plot(kind='hist',  bins=range(0, 90), edgecolor='white', title="Histogramme avec plot,
 ↪  bins=range(0, 90)") #, density=True)
# df['age'].plot(kind='kde')
plt.show()

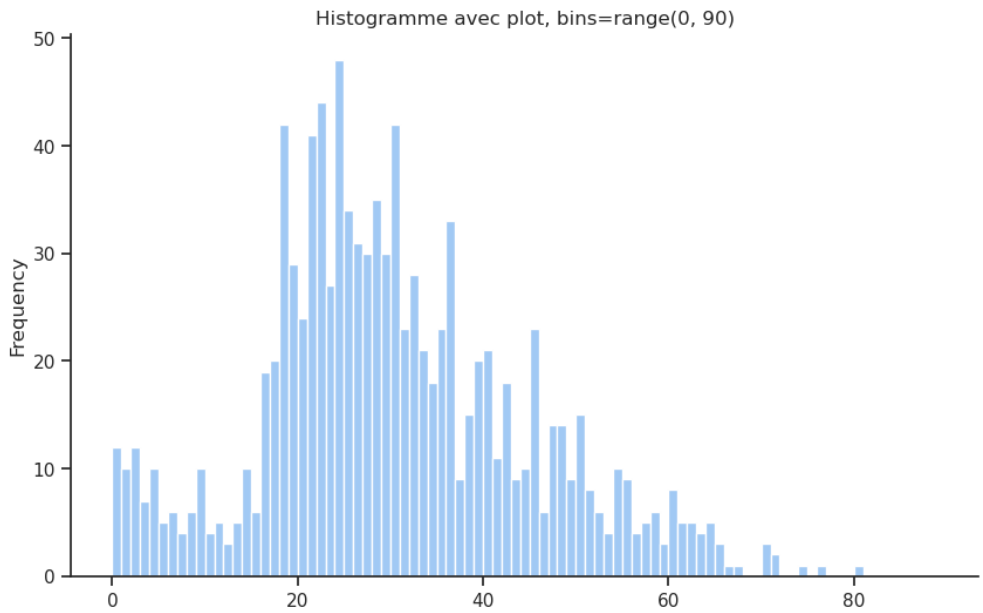
df['age'].plot(kind='hist', bins=range(0, 90, 10), edgecolor='white', title="Histogramme avec plot,
 ↪  bins=range(0, 90, 10)") #, density=True)
# df['age'].plot(kind='kde')
plt.show()

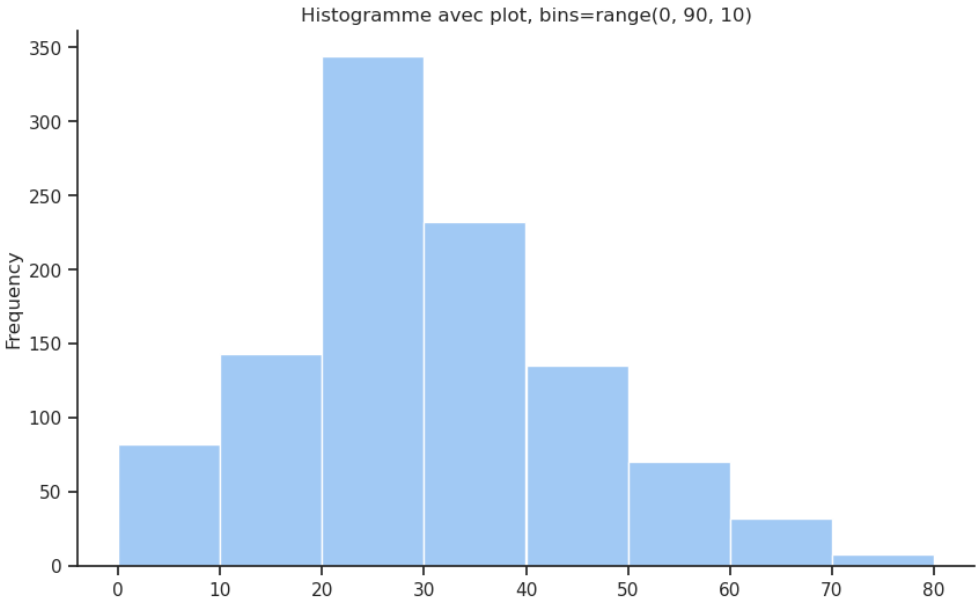

# Seaborn
# ------------------------------
# Rque : seaborn a kde=True dans histplot() et le rescale automatiquement
sns.histplot(data=df, x='age',  bins=range(0, 90), kde=True);
plt.title("Histogramme avec histplot de Seaborn, bins=range(0, 90)");
plt.show()

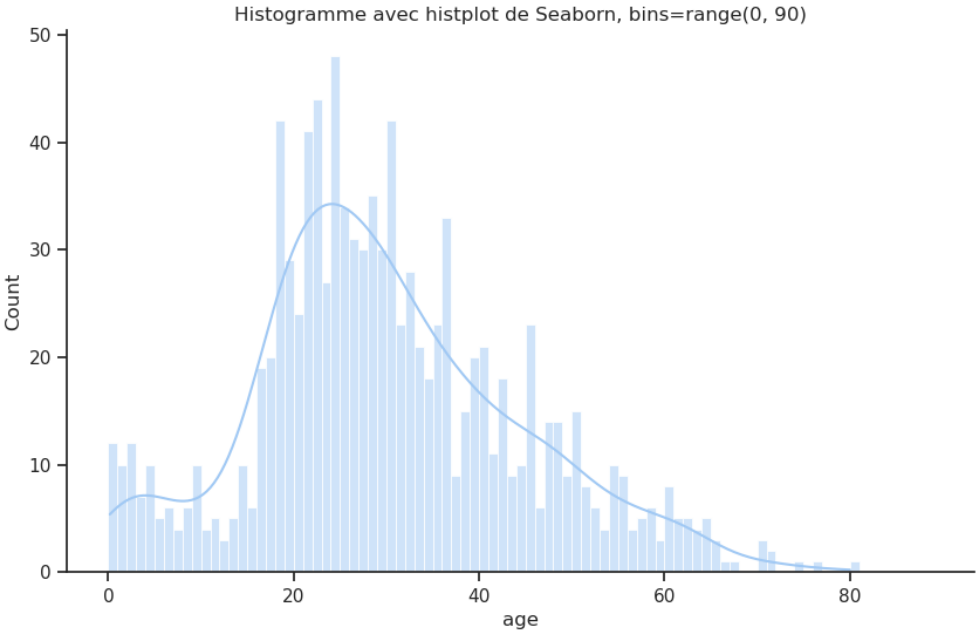
sns.histplot(df['age'], bins=range(0, 90, 10), edgecolor='white', kde=True);
plt.title("Histogramme avec histplot de Seaborn, bins=range(0, 90, 10)")
plt.show()

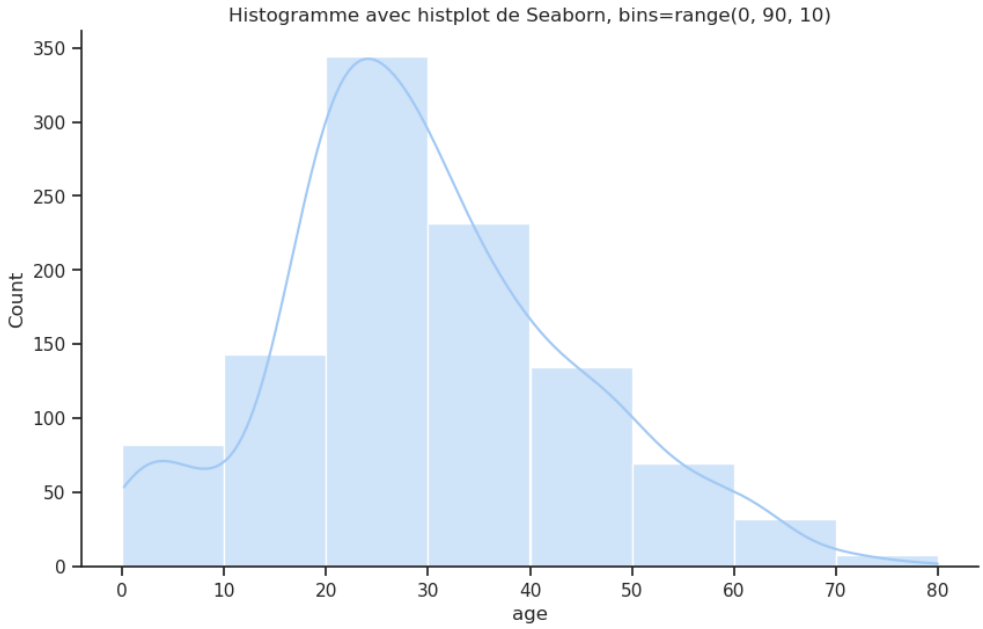

# Avec plotly.express
# ------------------------------
px.histogram(df, x='age', nbins=90, title='Histogramme des âges').show()

# NB Avec plotly.express, la gestion des bins est différente de matplotlib ou seaborn :
# il faut préciser xbins pour définir les bornes des classes.
px.histogram(
    df,
    x='age',
    title="Histogramme Plotly : bins de 10 ans"
).update_traces( xbins=dict( start=0, end=90, size=10 ) ).update_layout(bargap=0.2).show()
```




Histogramme avec plot, bins=range(0, 90, 10)
Frequency
0
50
100
150
200
250
300
350
0
10
20
30
40
50
60
70
80


Histogramme avec histplot de Seaborn, bins=range(0, 90)
Count
0
10
20
30
40
50
0
20
40
60
80
age




```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

La répartition par âge est globalement en forme de cloche, avec un pic autour de la tranche d'âge 20-30 ans. Il est intéressant de noter qu'il y a un pic local vers la gauche dans la tranche d'âge 0-5 ans, ce qui indique la présence de nombreux nourrissons et enfants en bas âge à bord. D'un point de vue historique, cela est logique, car à cette époque, les voyages concernaient souvent de jeunes adultes qui commençaient une nouvelle vie, parfois accompagnés d'enfants. Pour l'analyse de survie, cela est important, car les enfants étaient souvent prioritaires lors des opérations de sauvetage.

18.3.7. Nouvelle Colonne `age_group`

On peut regrouper les données en découpant un ensemble de valeurs selon des intervalles, comme on l'a vu sur les graphiques précédents. Cela semble pertinent pour l'âge. On va donc ajouter une colonne `age_group` à notre DataFrame qui contiendra l'âge regroupé par intervalles de 10 ans. Pour cela,

1. on peut utiliser la méthode `pd.cut()` en spécifiant les valeurs à découper et les étiquettes des intervalles,
2. on peut passer à `pd.map()` une fonction que nous aurons définie pour regrouper les âges par intervalles de 10 ans.

```
# Première méthode
# ------------------------------

df['age_group'] = pd.cut(df['age'], bins=range(0, 90, 10), labels=[f'{i}-{i+10}' for i in range(0, 80,
 ↪  10)])
df['age_group'] = df['age_group'].astype(str).replace('nan', 'Unknown')

# Deuxième méthode
# ------------------------------
# df['age_group'] = df['age'].map(lambda x: f'{int(x/10)*10}-{int(x/10)*10+10}' if not np.isnan(x) else
 ↪  'Unknown')

display(df)
```

	classe	survivant	sexe	age	prix	port_depart	age_group
0	1	1	F	29.0000	211.3375	S	20-30
1	1	1	H	0.9167	151.5500	S	0-10
2	1	0	F	2.0000	151.5500	S	0-10
3	1	0	H	30.0000	151.5500	S	20-30
4	1	0	F	25.0000	151.5500	S	20-30
...	...	...	...	...	...	...	...
1304	3	0	F	14.5000	14.4542	C	10-20
1305	3	0	F	NaN	14.4542	C	Unknown
1306	3	0	H	26.5000	7.2250	C	20-30
1307	3	0	H	27.0000	7.2250	C	20-30
1308	3	0	H	29.0000	7.8750	S	20-30

```
# Nb d'occurrences et pourcentage de chaque catégorie (age_group)

afficher_compte_et_pourcentage(df, 'age_group')
```

age_group	Nb occurrences	Pourcentage (%)
20-30	361	27.58
Unknown	263	20.09
30-40	210	16.04
10-20	162	12.38
40-50	132	10.08
0-10	86	6.57
50-60	62	4.74
60-70	27	2.06
70-80	6	0.46

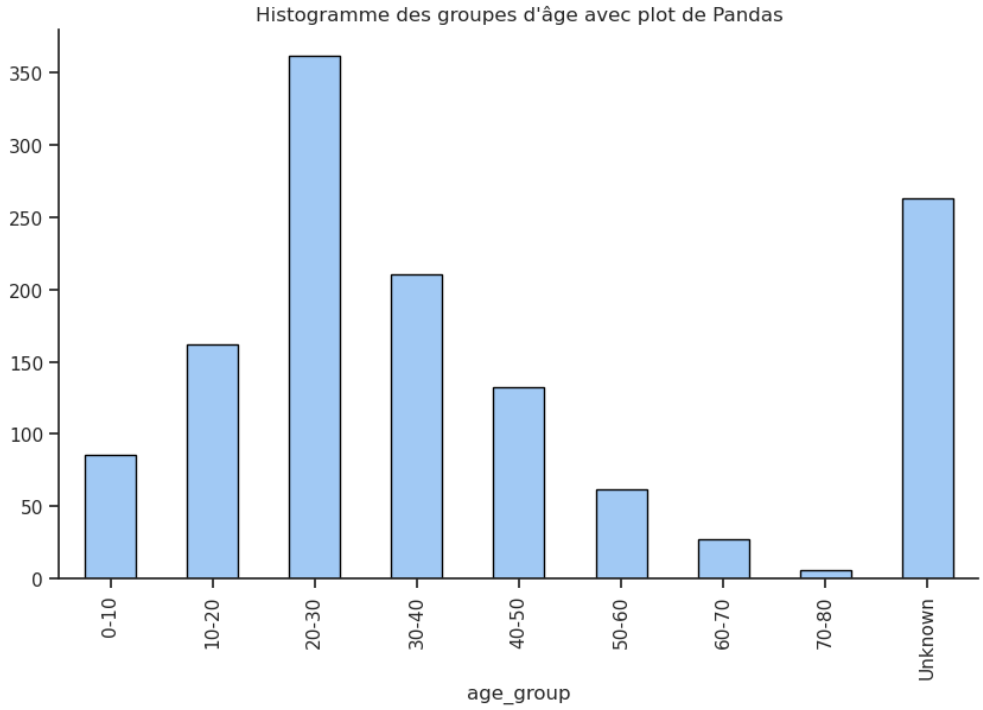
Histogramme des âges :

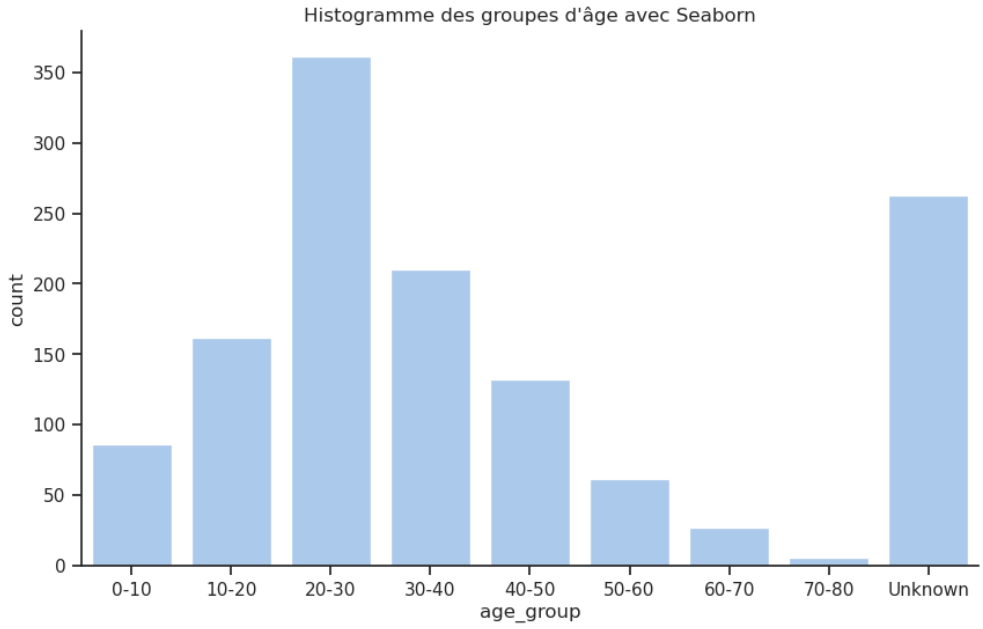
```
# Méthode 1 : Utiliser pandas pour créer un graphique en barres trié
# ---------------------------------------------------------------------

# df['age_group'].plot(kind='bar', edgecolor='black'); # ne fonctionne pas car les valeurs sont
 ↪  catégorielles
# df['age_group'].value_counts().plot(kind='bar', edgecolor='black'); # ok mais pas trié par ordre
 ↪  croissant
df['age_group'].value_counts().sort_index().plot(kind='bar', edgecolor='black')
plt.title("Histogramme des groupes d'âge avec plot de Pandas")
plt.show()

# Méthode 2 : Utiliser seaborn pour créer un countplot trié
# ---------------------------------------------------------------------
sns.countplot(data=df, x='age_group', order=sorted(df['age_group'].unique()));
plt.title("Histogramme des groupes d'âge avec Seaborn")
plt.show()

# Méthode 3 : Utiliser plotly.express
# ---------------------------------------------------------------------
px.histogram(df,
             x='age_group',
             title="Histogramme des groupes d'âge avec Plotly",
             category_orders={'age_group': sorted(df['age_group'].unique())}, text_auto=True).show()
```






```
Unable to display output for mime type(s): text/html
```

Pourcentage d'occurrences dans chaque groupe d'âge :

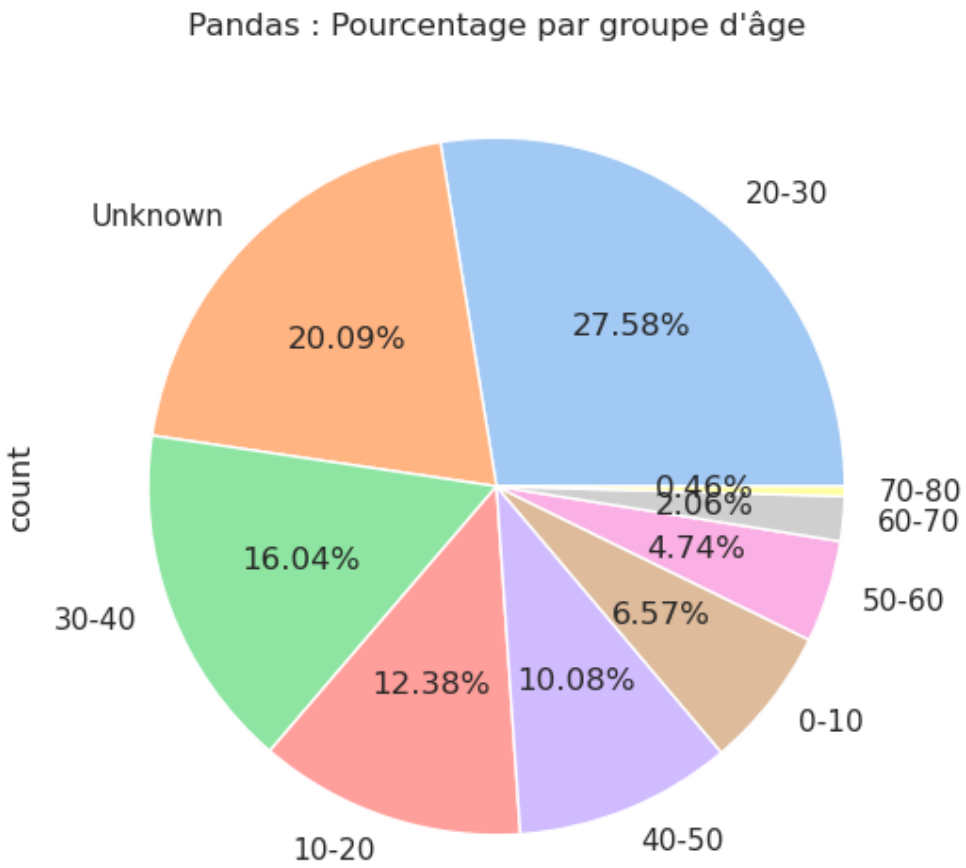
```
# Avec Pandas et dans un camembert
# --------------------------------
df['age_group'].value_counts().plot(kind='pie',
                                    autopct='%1.2f%%',
                                    #explode=[0.1]*len(df['age_group'].unique())
                                    )
plt.title("Pandas : Pourcentage par groupe d'âge")
plt.show()

# Avec plotly.express
```

```
# ----------------------------

# px.pie(df, names='age_group')

# Mieux
px.pie(df,
       names='age_group',
       color='age_group',
       title="Plotly : Pourcentage par groupe d'âge",
       color_discrete_sequence=px.colors.qualitative.Pastel
       ).update_traces( texttemplate="%{label} : %{value} (%{percent})" ,
                        textposition="outside",    # texte à l'extérieur du secteur
                        pull=[0.05]*len(df['age_group'].unique()))  # léger décalage pour chaque secteur)
```




```
Unable to display output for mime type(s): text/html
```

18.4. Liens entre deux ou plusieurs colonnes

Jusqu'à présent, nous n'avons exploré que les statistiques descriptives, mais les véritables informations apparaissent lorsque nous relions ces caractéristiques à la variable cible : `survivant`.
Les passagers plus jeunes ont-ils un taux de survie plus élevé ? Les passagers payant plus cher, souvent ceux de première classe, avaient-ils un meilleur accès aux canots de sauvetage ? Ce sont les questions que nous allons aborder ensuite en superposant les résultats de survie à nos distributions. Intuitivement, on pourrait s'attendre à ce que les enfants et les passagers de classe supérieure bénéficient d'un avantage en termes de survie.

Dans cette partie nous allons utiliser les méthodes suivantes :

```
.crosstab()
.plot(kind='bar') .plot(kind='hist') .value_counts().plot(kind='pie')
sns.countplot() sns.histplot() sns.catplot()
px.histogram() px.pie()
```

18.4.1. Analyse bivariée, vue d'ensemble

Pour une vue d'ensemble, nous pouvons utiliser la fonction `pairplot()` de Seaborn : elle crée une matrice de graphiques où chaque variable numérique du dataset est comparée à toutes les autres. Chaque ligne et colonne correspond à une variable. Les graphes diagonaux montrent généralement la distribution de chaque variable (histogramme ou KDE). Les graphes hors diagonale montrent la relation entre chaque paire de variables (scatter plot par défaut).

Avec ce pairplot, on peut visualiser d'un coup d'œil les patterns les plus importants comme la survie selon le sexe et la classe ou encore l'âge des survivants comparé à celui des non-survivants.

```
df['sexe_num'] = df['sexe'].astype('category').cat.codes
# df['port_depart_num'] = df['port_depart'].astype('category').cat.codes
```

```
# Sélection des variables numériques et catégorielles codées
# ============================================================
pairplot_vars = ['age', 'prix', 'classe', 'sexe_num']

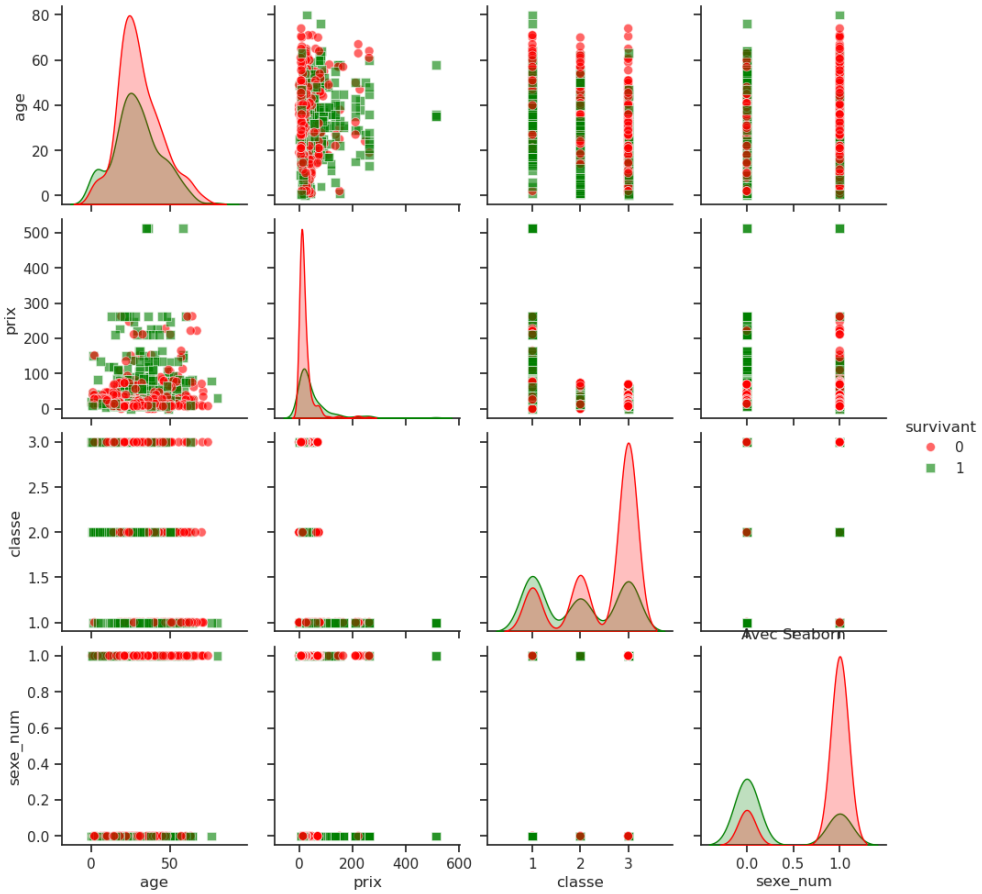


# Avec Seaborn
# ------------------------------

g = sns.pairplot(
    data=df[pairplot_vars + ['survivant']],  # inclure la variable cible
    hue='survivant',                         # couleur selon survie
    palette={0: 'red', 1: 'green'},          # couleurs personnalisées
    diag_kind='kde',                         # KDE pour la diagonale
    markers=['o', 's'],                      # forme différente selon la classe
    plot_kws={'alpha':0.6, 's':50}           # transparence et taille des points
)
# Ajuster les légendes
# for t, l in zip(g._legend.texts, ["Décédé", "Survécu"]):
#     t.set_text(l)
plt.title('Avec Seaborn')
plt.show()


# Avec plotly.express
# ------------------------------
# fig = px.scatter_matrix(
#     df,
#     dimensions=pairplot_vars,  # variables à afficher
#     color=df['survivant'].map({0: 'Décédé', 1: 'Survécu'}),
#     color_discrete_map={'Décédé': 'red', 'Survécu': 'green'},
#     symbol='survivant',  # forme différente selon la classe
#     symbol_map={0: 'circle', 1: 'square'},
#     opacity=0.6,  # transparence
#     height=800,   # hauteur de la figure
#     title='Avec Plotly'
# )
# fig.update_traces(
#     marker=dict(size=5),       # taille des points
#     diagonal_visible=False     # masquer la diagonale (pas de KDE natif))
# )
```

```
# fig.update_layout(width=1000, height=1000)
# fig.show()
```



Caveat

Le paradoxe de Simpson : la difficulté de comparer des pourcentages

L'étude des relations entre 3 ou plus caractéristiques (par exemple **sexe**, **classe** et **survie**), peut être délicat. En effet, il existe un phénomène statistique contre-intuitif appelé **paradoxe de Simpson**.
Ce paradoxe se produit lorsqu'une tendance observée dans **chaque sous-groupe** disparaît ou **s'inverse** lorsque l'on regroupe les données.
Ainsi, les pourcentages globaux peuvent raconter une histoire différente (et parfois totalement fausse) de celle révélée par les sous-groupes.

Exemple illustratif : l'inversion des pourcentages

Considérons deux médicaments (A et B) testés sur des hommes et des femmes. Les tableaux suivants indiquent le nombre de personnes traitées et le nombre de guérisons.

Médicament A

Médicament B

Genre

Total

Guéris

Total

Guéris

Hommes

5

1

10

3

Femmes

9

4

2

1

Taux de guérison

Groupe	Médicament A	Médicament B
Hommes	$1/5 = 20\%$	$3/10 = 30\%$
Femmes	$4/9 = 44\%$	$1/2 = 50\%$
Total	$5/14 = 36\%$	$4/12 = 33\%$

Conclusion :

- chez les *hommes*, le médicament B guérit davantage que A ;
- chez les *femmes*, le médicament B guérit encore davantage que A ;

- mais **une fois les deux groupes combinés**, le médicament A apparaît meilleur que B.

Le classement s'inverse au niveau global : c'est le **paradoxe de Simpson**. Le renversement provient de la **répartition très inégale des tailles de groupes**. Mathématiquement, le paradoxe vient du fait que les inégalités

$$\frac{a}{b} < \frac{A}{B} \quad \text{et} \quad \frac{c}{d} < \frac{C}{D}$$

n'impliquent pas que

$$\frac{a+c}{b+d} < \frac{A+C}{B+D}.$$

Dans la suite, il faudra faire très attention à comparer uniquement les pourcentages globaux de survivants par catégorie d'une autre variable si les distributions des sous-groupes sont très différentes.

18.4.2. Classe / Survie

Observons la colonne `classe`.
Il y a trois catégories dans cette colonne : les passagers de 1re classe, de 2e classe et de 3e classe représentés respectivement par les chiffres 1, 2 et 3.
Est-ce que les passagers de 1re classe ont eu la priorité pour monter à bord du navire et peut-être la priorité pour être sauvés dans les canots de sauvetage ? Autrement-dit, est-ce que les passagers de 1re classe ont un taux de survie plus élevé par rapport à ceux de 2e ou 3e classe ? Vérifions si c'est le cas.

La méthode `.crosstab()` permet de créer un **tableau de contingence** pour voir combien de personnes de chaque classe ont survécu ou non.

```
pd.crosstab(df['classe'], df['survivant'], margins=True, margins_name='Total')
```

survivant classe	0	1	Total
1	123	200	323
2	158	119	277
3	528	181	709
Total	809	500	1309

Même résultat avec `.pivot_table()` : la fonction `size` compte le nombre d'occurrences dans chaque groupe et `fill_value=0` remplace les NaN par 0 (utile si certaines combinaisons de valeurs n'existent pas). La fonction `pivot_table()` ne propose pas de manière directe pour ajouter des marges comme `crosstab()`, il faut donc additionner les marges séparément.

```
pt = df.pivot_table(index='classe', columns='survivant', aggfunc='size', fill_value=0)
pt['Total'] = pt.sum(axis=1)  # Total par ligne
pt.loc['Total'] = pt.sum(axis=0)  # Total par colonne
pt
```

survivant classe	0	1	Total
1	123	200	323
2	158	119	277
3	528	181	709
Total	809	500	1309

On peut également afficher le **pourcentage de survie pour chaque classe**, mais il faut choisir entre 3 options :

- le pourcentage de survivants dans chaque classe

- la répartition de chaque classe parmi les survivants/non-survivants
- le pourcentage de couple (classe, survie) parmi le total

```
# Nomenclature :
# -------------------------
# tc = tableau croisé
# cl_sur = classe survivant
# norm = normalizé

# Pourcentage par ligne
display(Markdown("**Pourcentage par ligne : combien de survivants parmi chaque classe**"))
tc_cl_sur_norm_lignes= pd.crosstab(df['classe'], df['survivant'], margins=True, margins_name='Total',
 ↪  normalize='index')
display(tc_cl_sur_norm_lignes)

# Pourcentage par colonne
display(Markdown("**Pourcentage par colonne : proportion de chaque classe parmi les survivants et les non
 ↪  survivants**"))
tc_cl_sur_norm_colonnes = pd.crosstab(df['classe'], df['survivant'], margins=True, margins_name='Total',
 ↪  normalize='columns')
display(tc_cl_sur_norm_colonnes)

# Pourcentage total
display(Markdown("**Pourcentage total : proportion de chaque combinaison classe-survivant par rapport à
 ↪  l'ensemble des passagers**"))
tc_cl_sur_norm_all = pd.crosstab(df['classe'], df['survivant'], margins=True, margins_name='Total',
 ↪  normalize='all')
display(tc_cl_sur_norm_all)
```

Pourcentage par ligne : combien de survivants parmi chaque classe

survivant classe	0	1
1	0.380805	0.619195
2	0.570397	0.429603
3	0.744711	0.255289
Total	0.618029	0.381971

Pourcentage par colonne : proportion de chaque classe parmi les survivants et les non survivants

survivant classe	0	1	Total
1	0.152040	0.400	0.246753
2	0.195303	0.238	0.211612
3	0.652658	0.362	0.541635

Pourcentage total : proportion de chaque combinaison classe-survivant par rapport à l'ensemble des passagers

survivant classe	0	1	Total
1	0.093965	0.152788	0.246753
2	0.120703	0.090909	0.211612
3	0.403361	0.138273	0.541635
Total	0.618029	0.381971	1.000000

Le tableau "par ligne" indique que les passagers de 1re classe avaient un taux de survie d'environ 62 %, tandis que ceux de 2e classe avaient un taux de survie d'environ 43 %, et ceux de 3e classe seulement environ 26 %. Cela confirme l'idée que les passagers de première classe avaient un avantage significatif en termes de survie par rapport aux autres classes.

Le tableau "par colonne" montre que parmi les survivants, une proportion plus élevée venait de la 1re classe (40 %), suivi de la 3e classe (36 %) et enfin de la 2e classe (24 %). La difficulté avec ce tableau est qu'il est influencé par la répartition initiale des passagers dans chaque classe (cf. paradoxe de Simpson).

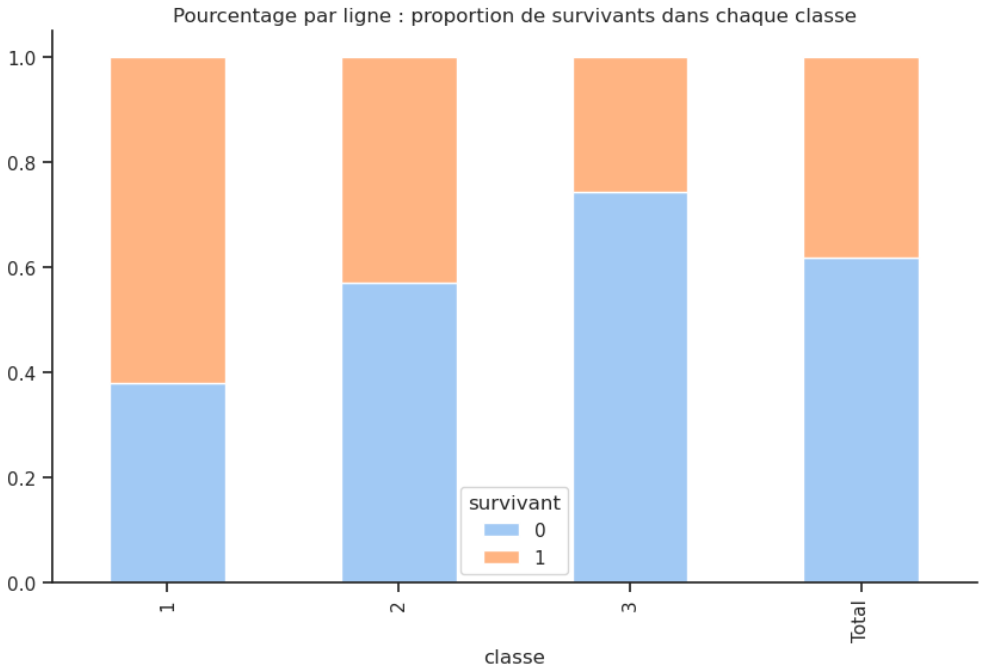
Le tableau "total" montre la répartition globale des passagers par classe, indépendamment de leur survie. Cela donne une idée aussi de la composition initiale des passagers à bord du Titanic.

On peut ensuite afficher ces informations sous forme de graphique :

```
# Avec Pandas et dans un graphique en barres empilées
# ---------------------------------------------------------------
tc_cl_sur_norm_lignes.plot(kind='bar', stacked=True, title='Pourcentage par ligne : proportion de
 ↪  survivants dans chaque classe');

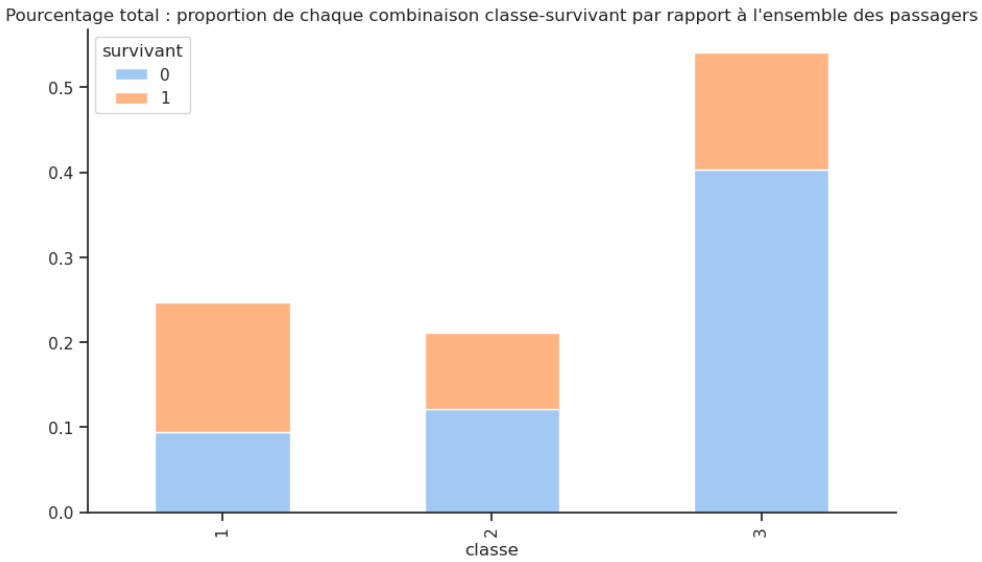
# NB Normalisation par ligne : chaque barre a la même hauteur (1 ou 100%)
# et les segments représentent la proportion de survivants dans chaque classe.

# On ne peut pas faire cela avec plotly.express directement.
```




```
# Pourcentage total sans les marges
tc_cl_sur_norm_all = pd.crosstab(df['classe'], df['survivant'], normalize='all')
display(tc_cl_sur_norm_all)
tc_cl_sur_norm_all.plot(kind='bar', stacked=True, title='Pourcentage total : proportion de chaque
↪  combinaison classe-survivant par rapport à l\'ensemble des passagers');
```

survivant classe	0	1
1	0.093965	0.152788
2	0.120703	0.090909
3	0.403361	0.138273



Quel graphe est le plus pertinent pour comparer les survivants et les non-survivants en fonction de la classe ?

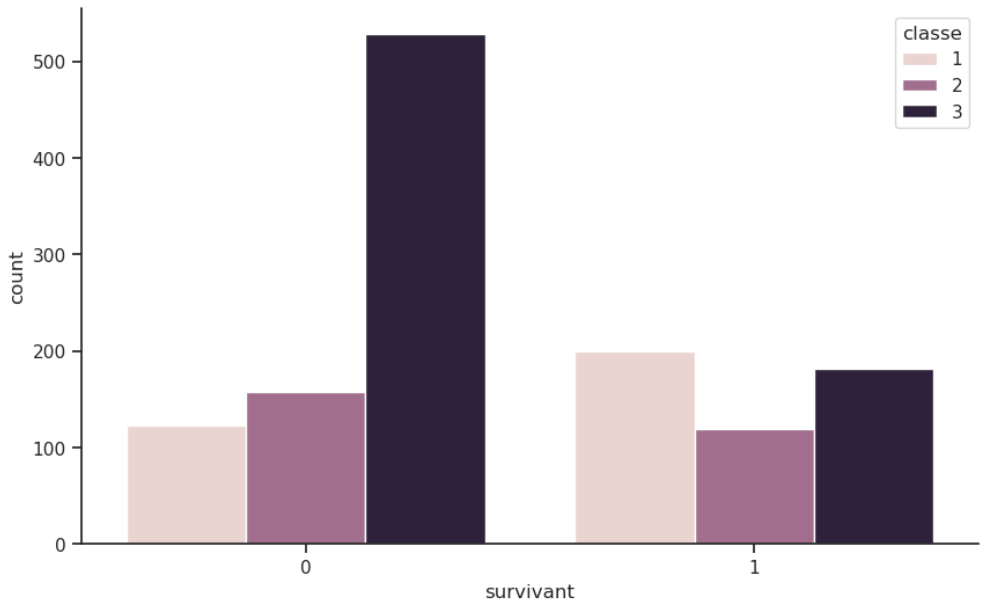
```
# Quel graphe est le plus pertinent pour comparer les survivants et les non-survivants en fonction de la
 ↪ classe ?
#
 ↪ ============================================================================================================

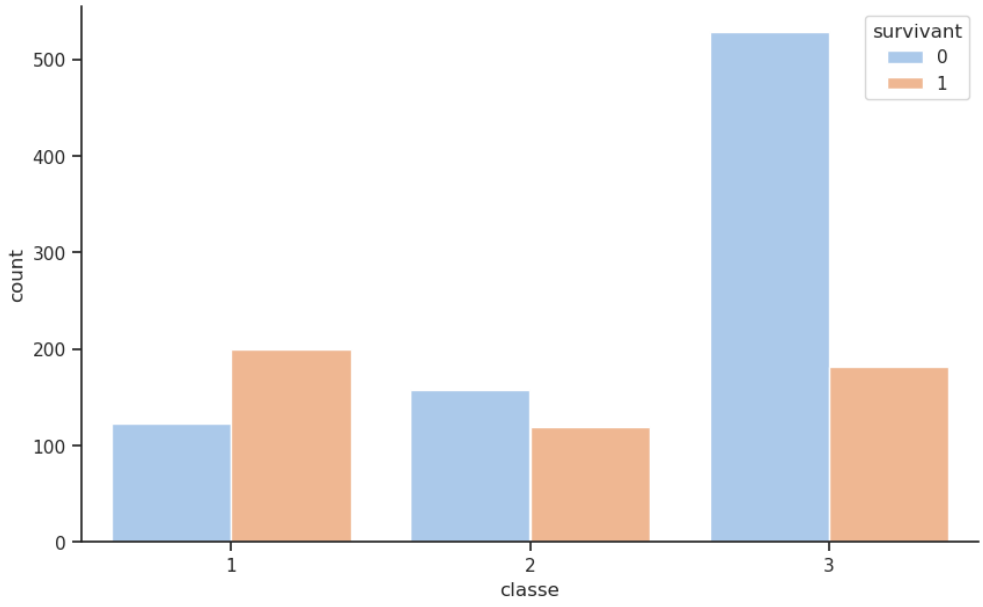
# sns.countplot(data=df, x='survivant')

# plt.figure();
# Chaque statut (survivant / décédé) est divisé en trois barres (classe 1, 2, 3)
sns.countplot(data=df, x='survivant', hue='classe');

plt.figure();
# Chaque classe est divisée en deux barres (survivant et non-survivant)
sns.countplot(data=df, x='classe', hue='survivant');

# plt.figure();
# sns.histplot(data=df, x='classe', hue='survivant', multiple='stack'); # idem empilées
```





```
px.histogram(df, x='survivant', color='classe', barmode='group', text_auto=True).show()

px.histogram(df, x='classe', color='survivant', barmode='group', text_auto=True).show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

```
# Taille de la figure et disposition
rows, cols = 2, 2
fig, axx = plt.subplots(rows, cols, figsize=(14, 10))
axx = axx.flatten()  # Pour accéder facilement aux axes

# Proportion par classe
sns.countplot(data=df, x='classe', stat='proportion', ax=axx[0])
axx[0].grid()

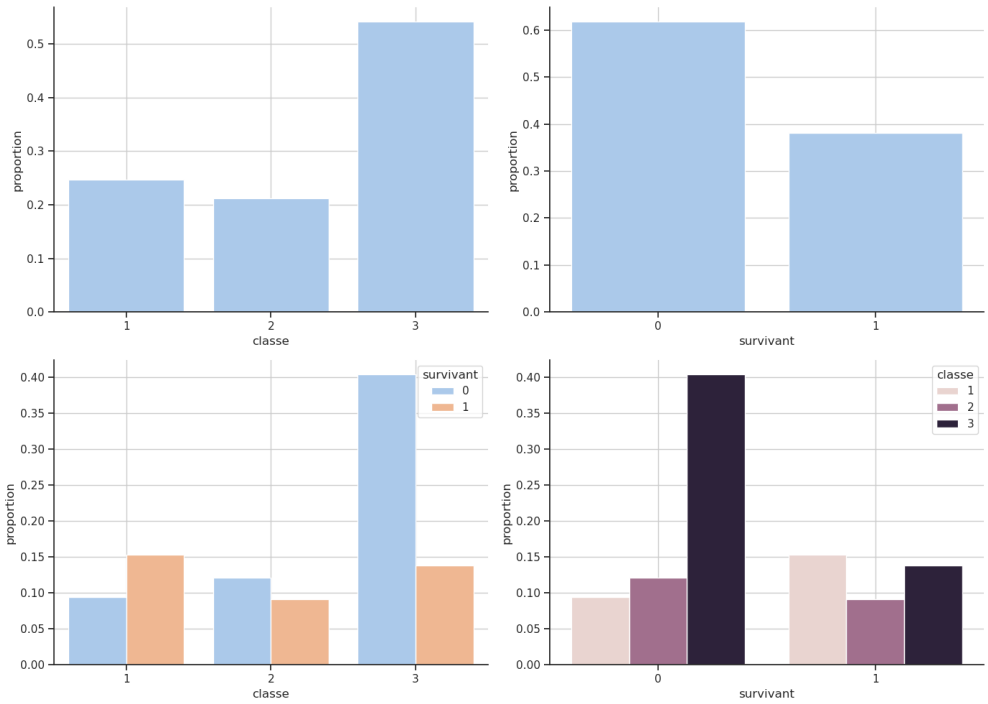
# Proportion par survivant
sns.countplot(data=df, x='survivant', stat='proportion', ax=axx[1])
axx[1].grid()

# Classe par survivant
sns.countplot(data=df, x='classe', hue='survivant', stat='proportion', ax=axx[2])
# sns.histplot(data=df, x='classe', hue='survivant', stat='proportion', multiple='stack', ax=axx[2])
axx[2].grid()

# Survivant par classe
sns.countplot(data=df, x='survivant', hue='classe', stat='proportion', ax=axx[3])
# sns.histplot(data=df, x='survivant', hue='classe', stat='proportion', multiple='stack', ax=axx[3])
axx[3].grid()



# Ajuster l'espacement
plt.tight_layout()
plt.show()
```



```
# import plotly.graph_objects as go

# fig = make_subplots(
#     rows=2, cols=2,
#     subplot_titles=(
#         "Proportion par classe",
#         "Proportion par survivant",
#         "Classe par survivant",
#         "Survivant par classe",
#     )
# )

# # Proportion par classe
# tc_cl = df['classe'].value_counts().sort_index()
# tc_cl_norm = tc_cl / tc_cl.sum()
# fig.add_trace(go.Bar(
#     x=tc_cl.index,
#     y=tc_cl_norm.values,
#     text=[f"{label}: {val} ({pct:.2%})" for label, val, pct in zip(tc_cl.index, tc_cl.values,
 ↪  tc_cl_norm.values)],
#     textposition="inside",
#     showlegend=False
# ), row=1, col=1)

# # Proportion par survivant
# tc_surv = df['survivant'].value_counts().sort_index()
# tc_surv_norm = tc_surv / tc_surv.sum()
# fig.add_trace(go.Bar(
#     x=tc_surv.index,
#     y=tc_surv_norm.values,
#     text=[f"{label}: {val} ({pct:.2%})" for label, val, pct in zip(tc_surv.index, tc_surv.values,
 ↪  tc_surv_norm.values)],
#     textposition="inside",
#     showlegend=False
# ), row=1, col=2)

# # Classe par survivant (empilé)
# tc_cl_surv = pd.crosstab(df['classe'], df['survivant'])
# tc_cl_surv_norm = tc_cl_surv.div(tc_cl_surv.sum(axis=1), axis=0)
# for col in tc_cl_surv.columns:
#     fig.add_trace(go.Bar(
#         x=tc_cl_surv.index,
#         y=tc_cl_surv_norm[col],
#         text=[f"{col}: {val} ({pct:.2%})" for val, pct in zip(tc_cl_surv[col], tc_cl_surv_norm[col])],
#         textposition="inside",
#         showlegend=False
#     ), row=2, col=1)

# # Survivant par classe (empilé)
# tc_surv_cl = pd.crosstab(df['survivant'], df['classe'])
# tc_surv_cl_norm = tc_surv_cl.div(tc_surv_cl.sum(axis=1), axis=0)
# for col in tc_surv_cl.columns:
#     fig.add_trace(go.Bar(
#         x=tc_surv_cl.index,
#         y=tc_surv_cl_norm[col],
#         text=[f"{col}: {val} ({pct:.2%})" for val, pct in zip(tc_surv_cl[col], tc_surv_cl_norm[col])],
#         textposition="inside",
#         showlegend=False
#     ), row=2, col=2)

# fig.update_layout(
#     height=800,
#     width=1000,
```

```
#     barmode='stack'
# )

# fig.show()
```

Nous allons généraliser cette approche en créant une fonction qui prend en argument une colonne et affiche le taux de survie pour chaque catégorie unique dans cette colonne, ainsi que les deux tableaux de contingence.

```
def plot_proportions(df, col1, col2):
    plt.figure(figsize=(14, 10))

    # Pie chart col1
    plt.subplot(2, 2, 1)
    df[col1].value_counts(normalize=True).plot(
        kind='pie', autopct='%1.2f%%', startangle=90, legend=False
    )
    plt.title(f"Répartition de '{col1}'")

    # Pie chart col2
    plt.subplot(2, 2, 2)
    df[col2].value_counts(normalize=True).plot(
        kind='pie', autopct='%1.2f%%', startangle=90, legend=False
    )
    plt.title(f"Répartition de '{col2}'")

    # Bar plot col2 par col1 (proportion)
    plt.subplot(2, 2, 3)
    sns.countplot(data=df, x=col2, hue=col1, stat='proportion')
    plt.title(f"'{col2}' par '{col1}'")

    # Bar plot col1 par col2 (proportion)
    plt.subplot(2, 2, 4)
    sns.countplot(data=df, x=col1, hue=col2, stat='proportion')
    plt.title(f"'{col1}' par '{col2}'")

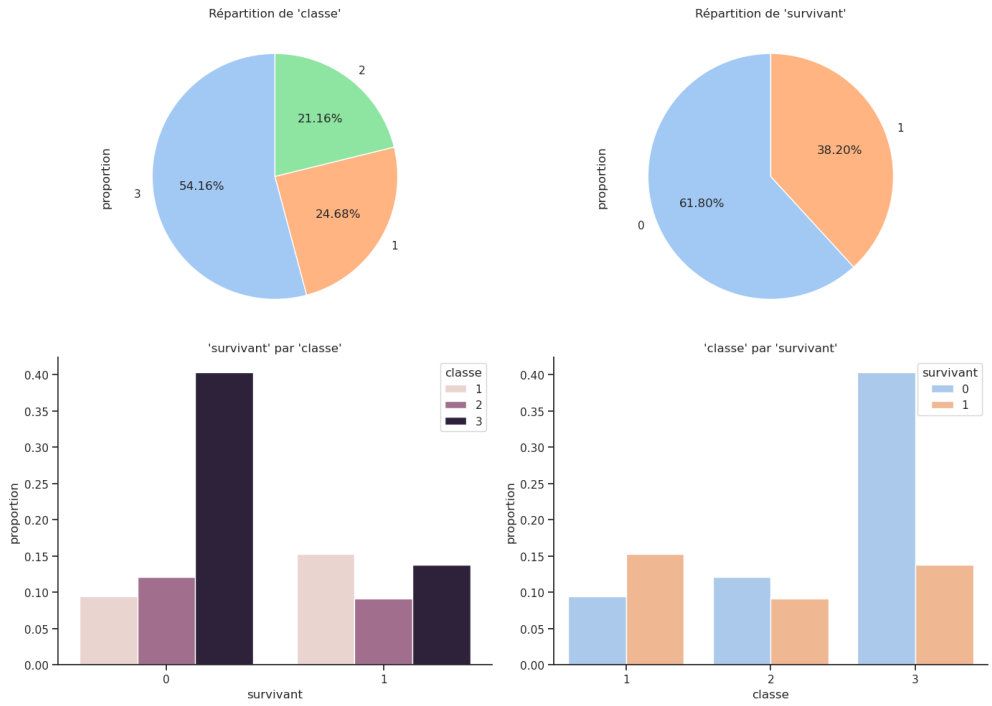
    plt.tight_layout()
    plt.show()

    # Crosstabs
    tc = pd.crosstab(df[col1], df[col2], margins=True, margins_name='Total') # pourcentage total
    tc_norm = pd.crosstab(df[col1], df[col2], margins=True, margins_name='Total', normalize='index') #
↪   pourcentage par ligne
    # tc_norm_all = pd.crosstab(df[col1], df[col2], margins=True, normalize='all') # pourcentage total

    display(Markdown("Crosstab absolu :"))
    display(tc)
    display(Markdown("Crosstab normalisé par ligne :"))
    display(tc_norm)

    # Bar plot empilé du crosstab normalisé
    tc_norm.plot(kind='bar', stacked=True)
    plt.ylabel("Proportion par ligne")
    plt.title(f"'{col1}' par '{col2}' (proportion par ligne)")
    plt.show()


# TEST
plot_proportions(df, 'classe', 'survivant')
```

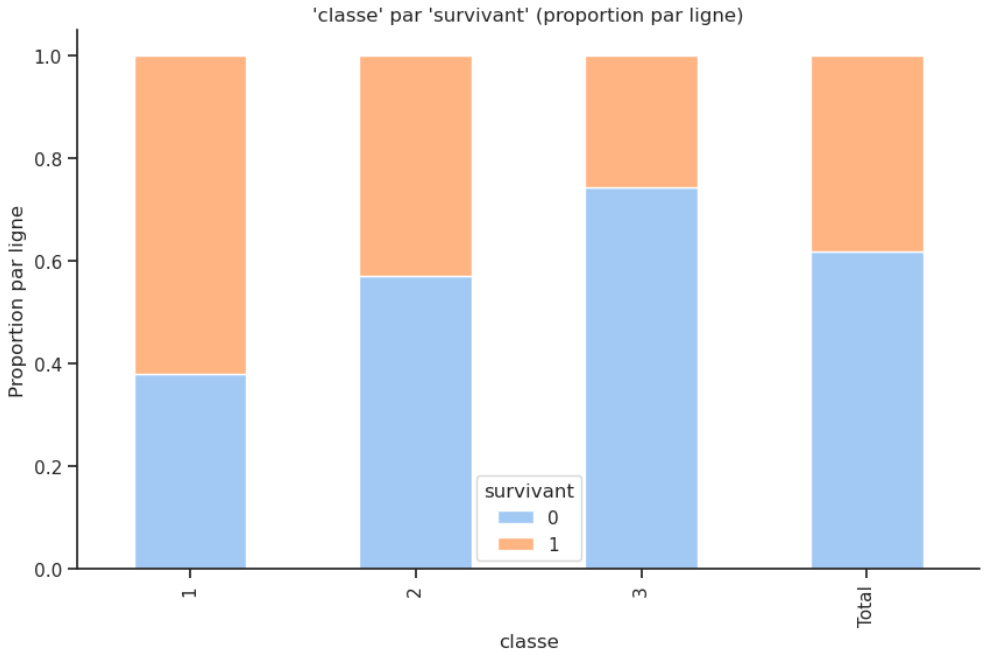


Crosstab absolu :

survivant classe	0	1	Total
1	123	200	323
2	158	119	277
3	528	181	709
Total	809	500	1309

Crosstab normalisé par ligne :

survivant classe	0	1
1	0.380805	0.619195
2	0.570397	0.429603
3	0.744711	0.255289
Total	0.618029	0.381971



18.4.3. Sexe / Survie

Nous menons une enquête similaire pour la colonne **sexe**. L'article de Wikipédia sur le RMS Titanic indiquait que « le protocole "les femmes et les enfants d'abord" était généralement suivi lors du chargement des canots de sauvetage ». Nous pouvions donc nous attendre à un taux de survie plus élevé pour les femmes et les enfants. Vérifions si cela est vrai.

Taux de survie par sexe.

```
# Avec groupby
# ------------------------------------------------
display( df.groupby('sexe')['survivant'].mean() )  # proportion de survivants par sexe

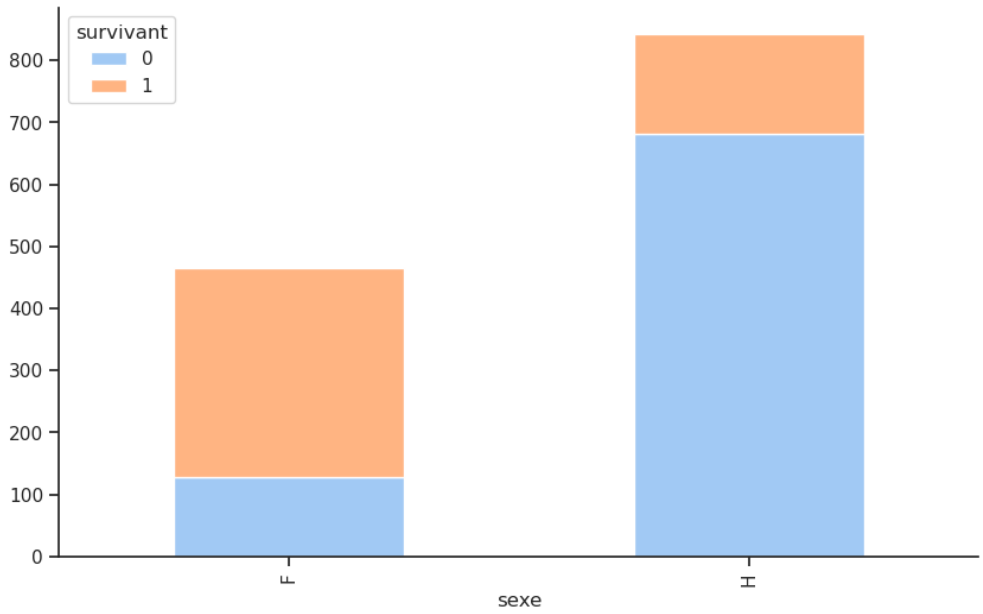
# Avec crosstab
# ------------------------------------------------
sex_survivant = pd.crosstab(df['sexe'],
            df['survivant'],
            # normalize='index'
            # normalize='columns'
            # normalize='all'
            )
display(sex_survivant)

# Affichage avec Pandas
# ------------------------------------------------
sex_survivant.plot(kind='bar', stacked=True)
plt.show()

# Avec plotly.express
# ------------------------------------------------
px.bar(sex_survivant, barmode='stack', text_auto=True).show()
```

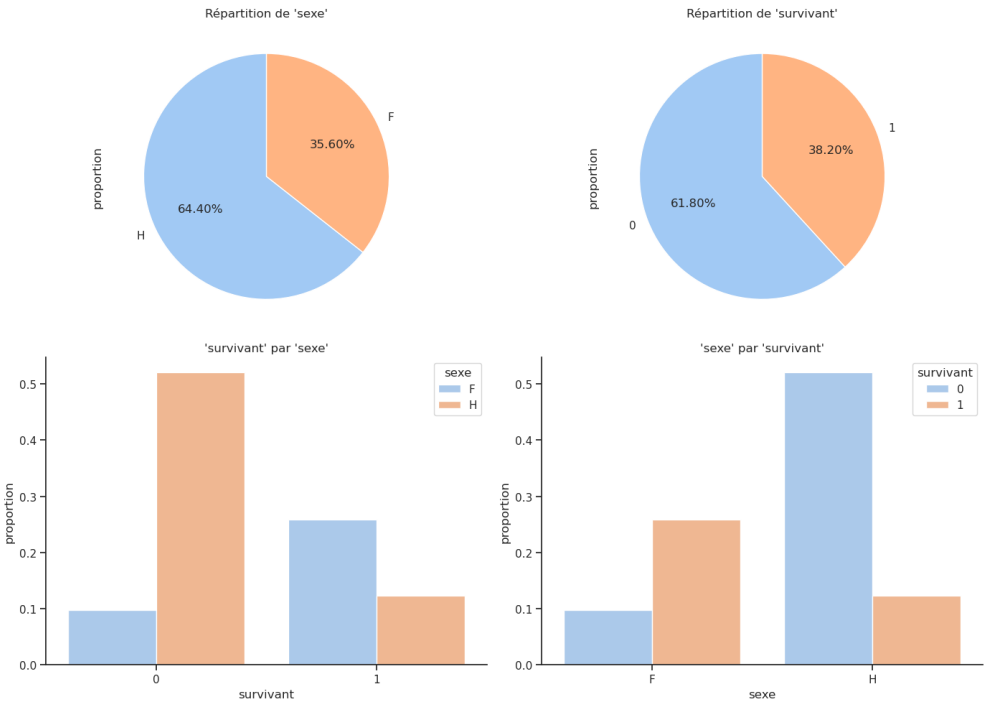
```
sexe
F    0.727468
H    0.190985
Name: survivant, dtype: float64
```

survivant sexe	0	1
F	127	339
H	682	161




```
Unable to display output for mime type(s): text/html
```

```
plot_proportions(df, 'sexe', 'survivant')
```

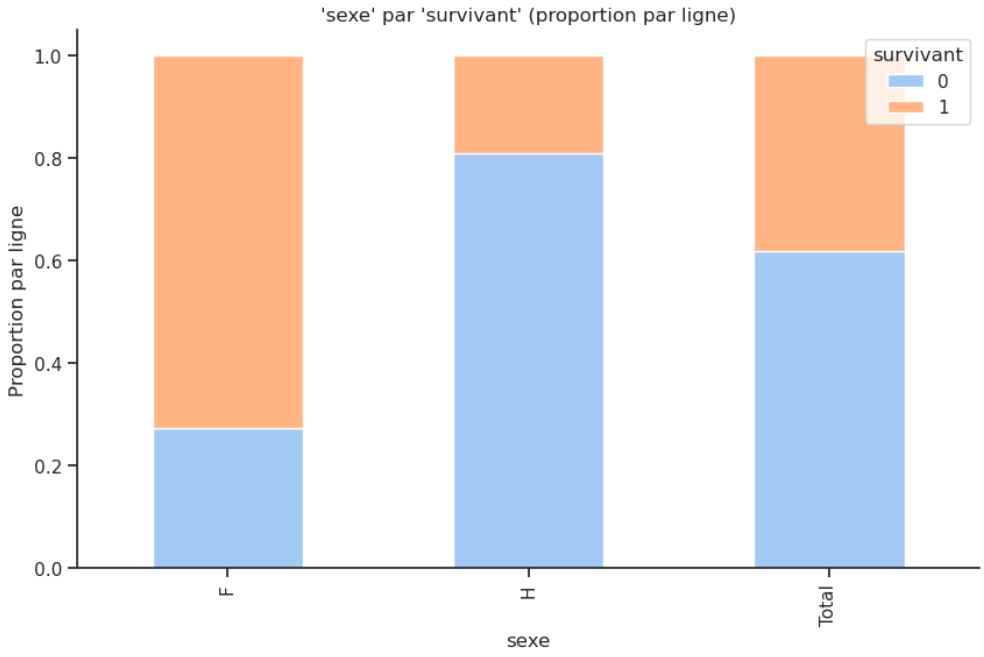


Crosstab absolu :

survivant sexe	0	1	Total
F	127	339	466
H	682	161	843
Total	809	500	1309

Crosstab normalisé par ligne :

survivant sexe	0	1
F	0.272532	0.727468
H	0.809015	0.190985
Total	0.618029	0.381971



Parmi les femmes, la majorité a survécu, avec un taux de survie de 75 %, contre un taux de survie de 20 % pour les hommes.

Notons que le résultat obtenu avec `crosstab()` peut s'obtenir avec des masques.

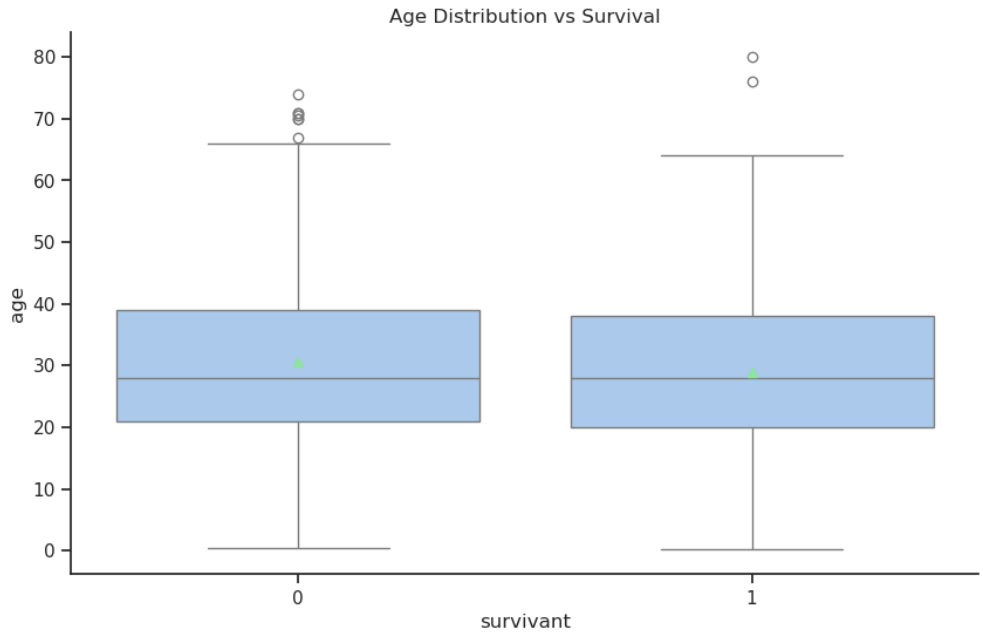
Par exemple, parmi les survivants, combien étaient des hommes et combien étaient des femmes ? Le tableau précédent indique que sur les 500 survivants, 339 étaient des femmes et 161 des hommes. Voici comment faire avec une masque :

```
mask = (df['survivant'] == 1)
df[mask]['sexe'].value_counts()
# df[mask]['sexe'].value_counts().plot(kind='bar') ;
```

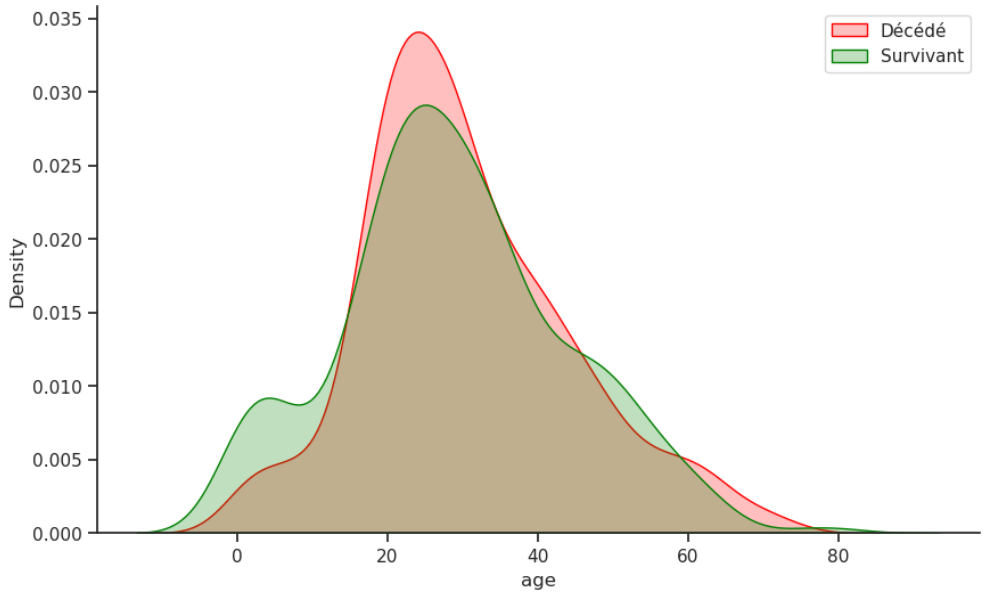
```
sexe
F    339
H    161
Name: count, dtype: int64
```

18.4.4. Age / Survie

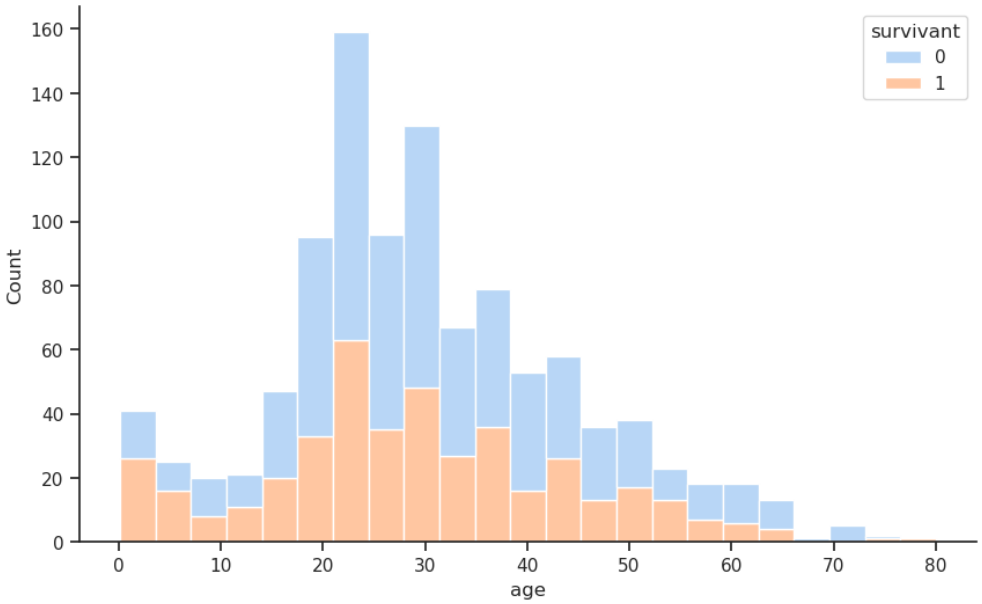
```
sns.boxplot(data=df, x="survivant", y="age", showmeans=True)
plt.title("Age Distribution vs Survival")
plt.show()
```




```
sns.kdeplot(df[df['survivant']==0]['age'].dropna(), fill=True, alpha=0.25, label='Décédé', color='red')
sns.kdeplot(df[df['survivant']==1]['age'].dropna(), fill=True, alpha=0.25, label='Survivant',
 ↪  color='green')
plt.legend();
```

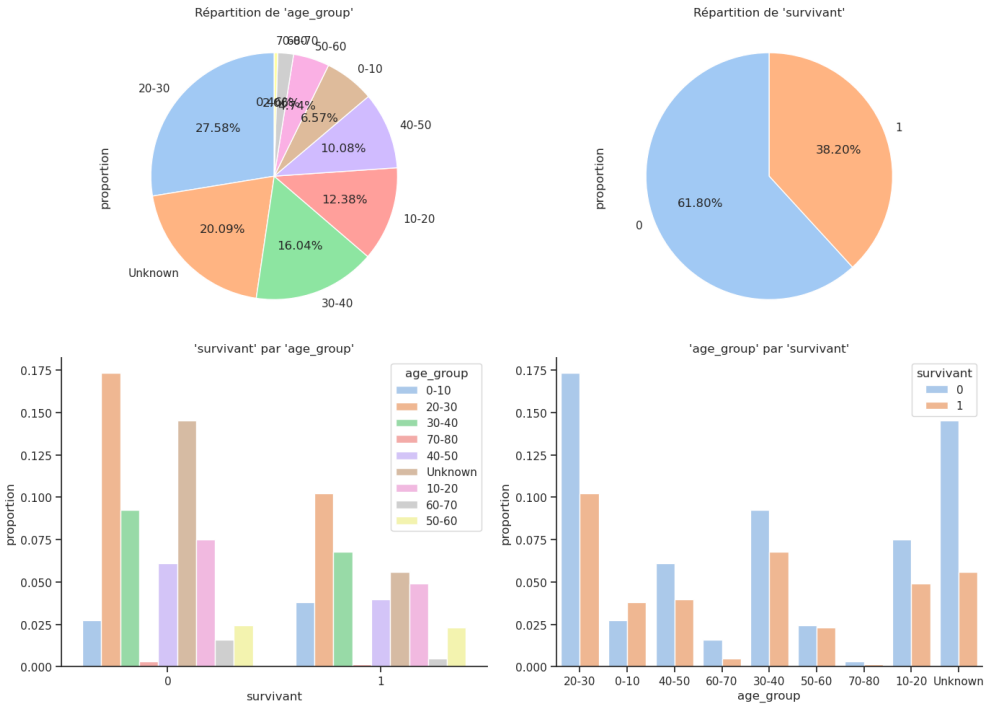



```
sns.histplot(data=df, x="age", hue="survivant", multiple="stack");
```



18.4.5. Age_group / Survie

```
plot_proportions(df, 'age_group', 'survivant')
```



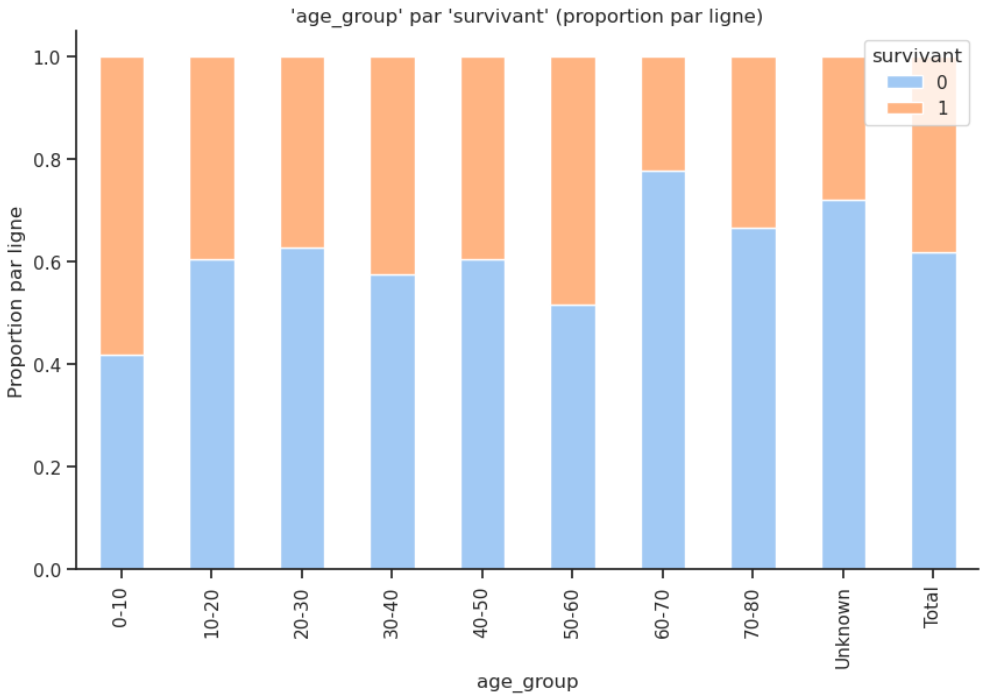
Crosstab absolu :

survivant age_group	0	1	Total
0-10	36	50	86
10-20	98	64	162
20-30	227	134	361
30-40	121	89	210

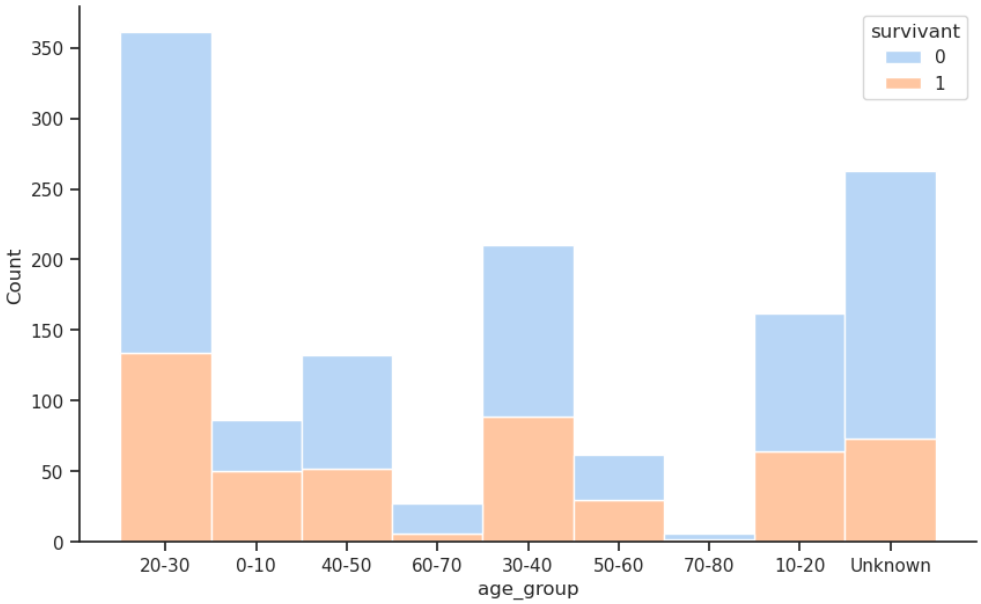
survivant age_group	0	1	Total
40-50	80	52	132
50-60	32	30	62
60-70	21	6	27
70-80	4	2	6
Unknown	190	73	263
Total	809	500	1309

Crosstab normalisé par ligne :

survivant age_group	0	1
0-10	0.418605	0.581395
10-20	0.604938	0.395062
20-30	0.628809	0.371191
30-40	0.576190	0.423810
40-50	0.606061	0.393939
50-60	0.516129	0.483871
60-70	0.777778	0.222222
70-80	0.666667	0.333333
Unknown	0.722433	0.277567
Total	0.618029	0.381971




```
sns.histplot(data=df, x="age_group", hue="survivant", multiple="stack");
```



Taux de survie par âge.

```
# observed=False pour inclure les catégories sans valeurs
df.groupby('age_group', observed=False)['survivant'].mean()
# df.groupby('age_group', observed=False)['survivant'].mean().plot(kind='bar', edgecolor='black');
```

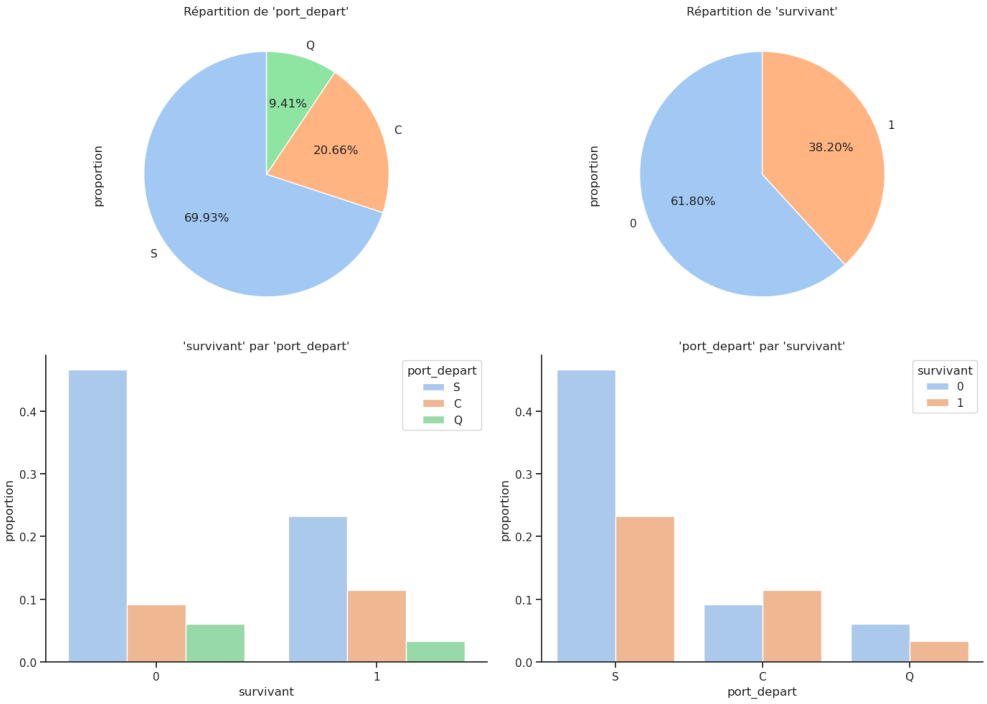
```
age_group
0-10        0.581395
10-20       0.395062
20-30       0.371191
30-40       0.423810
40-50       0.393939
50-60       0.483871
60-70       0.222222
70-80       0.333333
Unknown     0.277567
Name: survivant, dtype: float64
```

Plus de la moitié des moins de 10 ans ont survécu.

18.4.6. Port de départ / Survie

La dernière caractéristique catégorique que nous pouvons examiner est le port d'embarquement. Les passagers ont embarqué depuis trois ports différents nommés Cherbourg, Queenstown et Southampton, abrégés respectivement par les lettres C, Q et S. Nous avons déjà vu que le sexe et la classe influencent le taux de survie des passagers. Le port d'embarquement influence-t-il la survie ?

```
plot_proportions(df, 'port_depart', 'survivant')
```

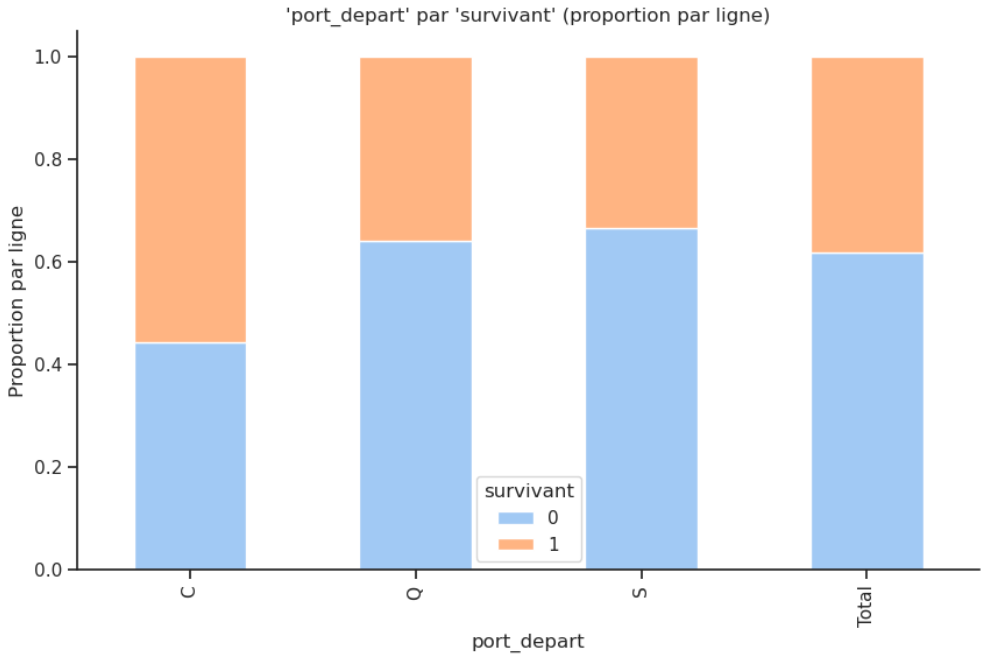


Crosstab absolu :

survivant port_depart	0	1	Total
C	120	150	270
Q	79	44	123
S	610	304	914
Total	809	498	1307

Crosstab normalisé par ligne :

survivant port_depart	0	1
C	0.444444	0.555556
Q	0.642276	0.357724
S	0.667396	0.332604
Total	0.618975	0.381025



D'après ces graphiques, environ 70 % des passagers ont embarqué depuis S(outhampton), 20 % depuis C(herbourg) et 10 % depuis Q(ueenstown). Cependant, le taux de survie le plus élevé, soit 55 %, est obtenu pour les passagers ayant embarqué à Cherbourg, puis environ 35 % à Queenstown et 30 % à Southampton.

18.4.7. Bonus : nouvelle colonne femmes / hommes / enfants

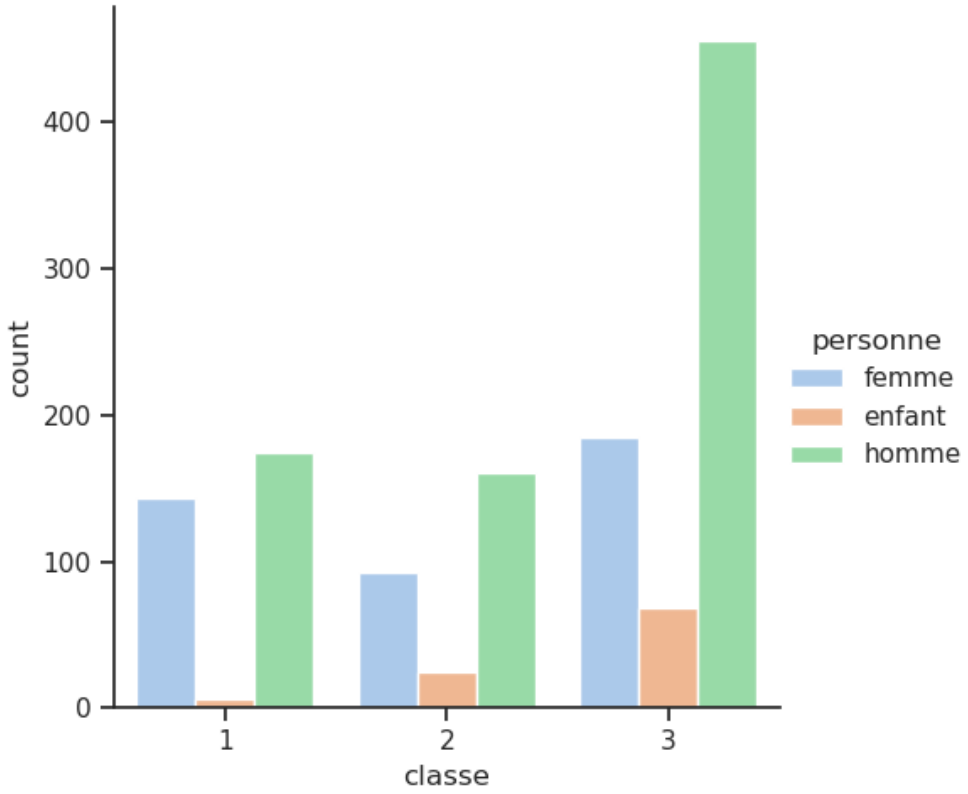
Il pourrait être utile de connaître la répartition entre les hommes, les femmes et les enfants. Pour cela, nous allons créer une nouvelle colonne `personne` qui prendra les valeurs `homme`, `femme` ou `enfant` en fonction de l'âge. Nous allons ensuite afficher le taux de survie pour chaque catégorie de personnes.

Cependant, la priorité pour l'évacuation selon le fameux principe « les femmes et les enfants d'abord » ne définissait pas exactement un âge officiel pour les enfants. Les archives et témoignages historiques montrent que les enfants étaient généralement considérés comme ceux de moins de 14 ans. Certains témoignages mentionnent que des adolescents (12-15 ans) étaient parfois traités comme des adultes selon le contexte et la cabine. Les décisions étaient aussi un peu arbitraires : la distinction reposait souvent sur l'apparence et la taille plutôt que sur un âge strict. Donc, pour les analyses de survie ou des modèles statistiques sur le Titanic, beaucoup de chercheurs prennent <15 ans comme enfant pour appliquer ce critère.

```
df['personne'] = df['sexe']
df['personne'] = df['personne'].replace({'H':'homme', 'F':'femme'})
df.loc[df['age']<14, 'personne'] = 'enfant'

sns.catplot(x='classe', data=df, hue='personne', kind='count');

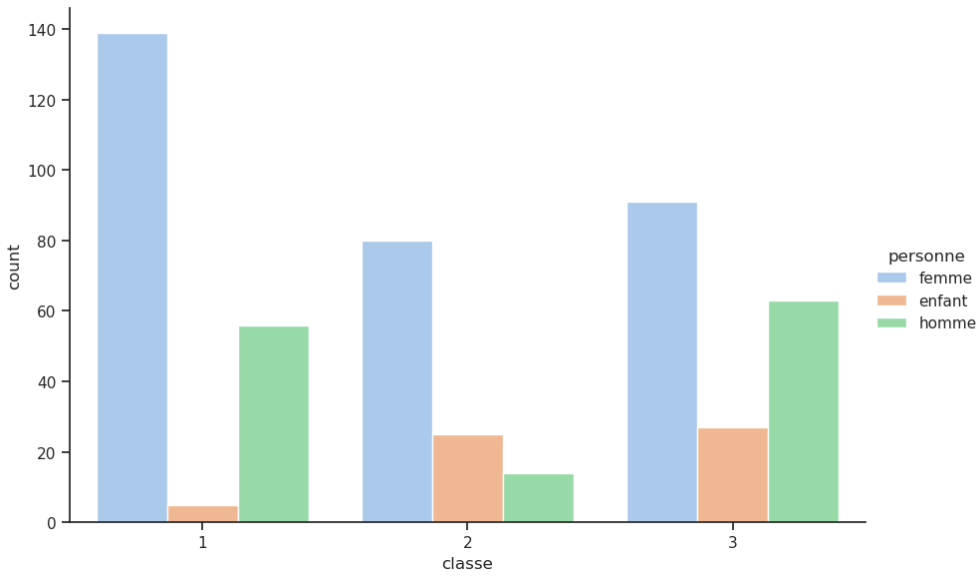
# sns.catplot(x='classe', y='survivant', data=df, hue='personne', kind='point')
```




```
px.histogram(df, x='classe', color='personne', text_auto=True).update_layout(bargap=0.2).show()
```

```
Unable to display output for mime type(s): text/html
```

```
# Graphique pour le nombre absolu de survivants
sns.catplot(
    x='classe',
    hue='personne',
    data=df[df['survivant'] == 1],
    kind='count',
    height=6,
    aspect=1.5
);
```



```
df_survivants = df[df['survivant'] == 1]
px.histogram(df_survivants, x='classe', color='personne',
 ↪  text_auto=True).update_layout(bargap=0.2).show()
```

```
Unable to display output for mime type(s): text/html
```

18.4.8. Classe / Âge / Survie

Quelle est la distribution de l'âge des passagers en fonction de la classe ?

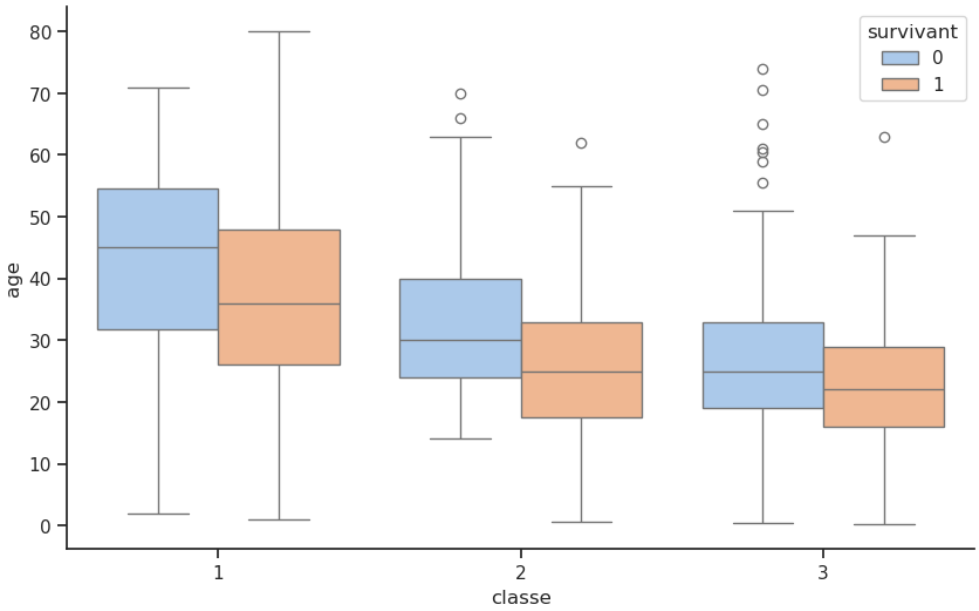
```
# df.groupby('classe')['age'].mean()
df.pivot_table(values='age', index='classe', aggfunc='mean')
```

classe	age
1	39.159918
2	29.506705
3	24.816367

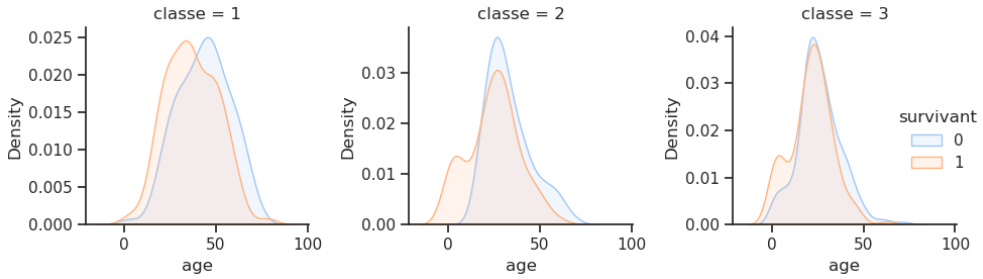
```
# plot_proportions(df, 'age_group', 'classe')
```

Nous souhaitons analyser comment une variable continue (comme l'âge) varie au sein de groupes catégoriels (comme la classe), tout en distinguant les sous-groupes à l'intérieur de chaque catégorie à l'aide d'une variable de différenciation (ici, le statut de survivant). Ce type de graphique permet d'avoir une vue claire des distributions, de voir si certaines catégories ont des valeurs extrêmes ou des tendances particulières, et de faire des comparaisons entre des groupes sous différents critères.

```
sns.boxplot(data=df, x='classe', y='age', hue='survivant');
```

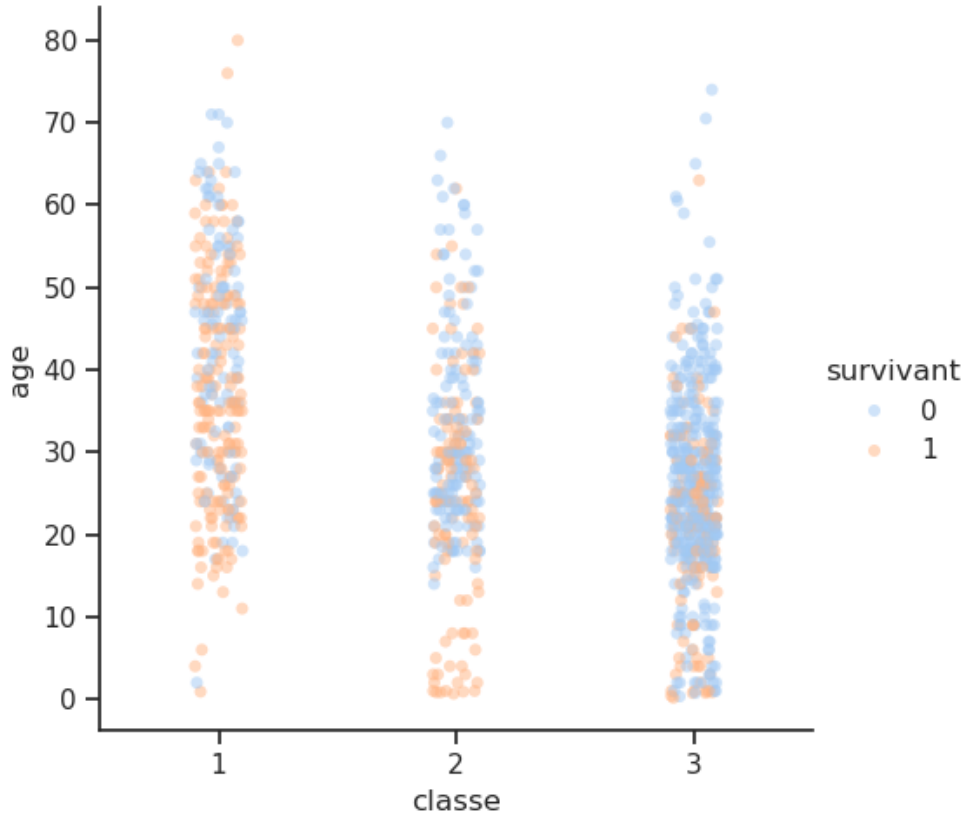


```
sns.FacetGrid(df,
              col='classe',
              hue='survivant',
              sharex=True,
              sharey=False,
              ).map(sns.kdeplot, 'age', fill=True, alpha=0.15).add_legend();
plt.tight_layout()
```



Avantage de la première classe à tous les âges : pour les passagers de première classe, la courbe de densité de survie (orange) reste systématiquement supérieure à la courbe des non-survivants (bleue) dans presque toutes les tranches d'âge. Cela indique que le statut de première classe conférait un fort avantage en termes de survie, quel que soit l'âge.
Dans les classes moyenne et inférieure, la courbe de survie atteint un pic marqué chez les plus jeunes (0-10 ans), confirmant que la politique « les femmes et les enfants d'abord » offrait un avantage en termes de survie aux enfants, même dans les classes inférieures.
Les graphiques mettent en évidence comment la combinaison de l'âge et de la classe sociale a déterminé les chances de survie : un jeune enfant, quelle que soit sa classe sociale, avait plus de chances de survivre qu'un jeune adulte de la même classe sociale. Un jeune adulte de troisième classe avait beaucoup moins de chances de survivre qu'un jeune adulte de première classe, comme le montre clairement le pic bleu proéminent pour Pclass=3 chez les 20-30 ans.

```
# cat pour categorical, comme la variable 'classe'
sns.catplot(data=df, x='classe', y='age', hue='survivant', alpha=0.5);
```

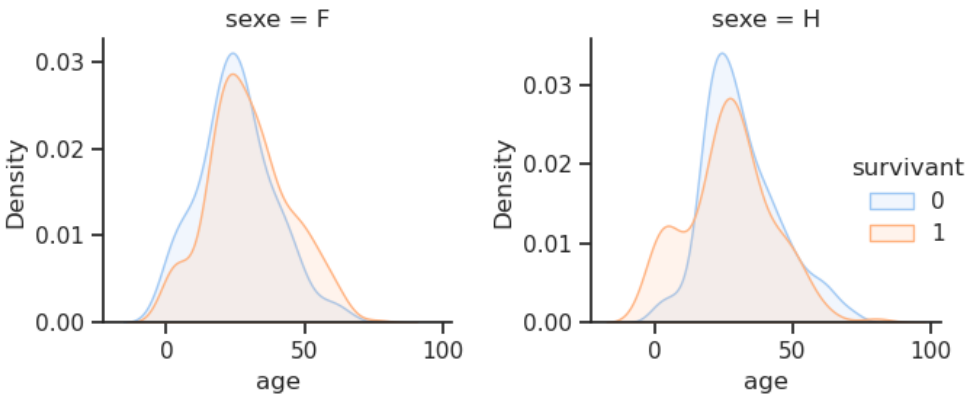



```
px.strip(df, x='classe', y='age', color='survivant')
```

```
Unable to display output for mime type(s): text/html
```

18.4.9. Sexe / Âge / Survie

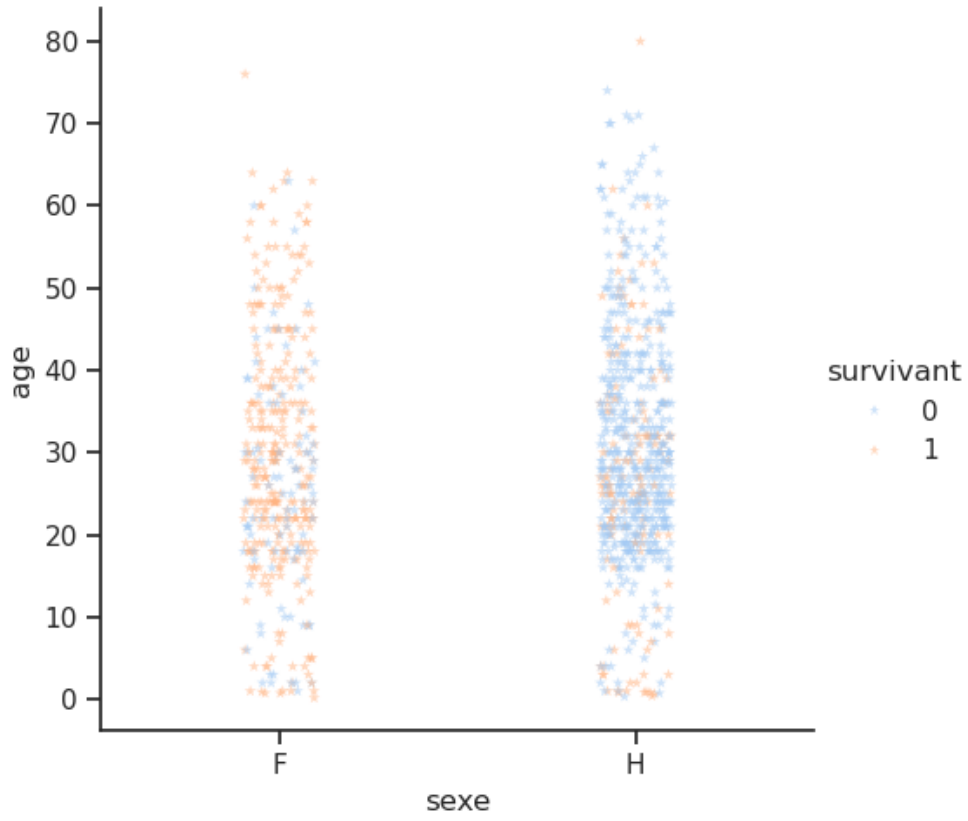
```
sns.FacetGrid(df,
              col='sexe',
              hue='survivant',
              sharex=True,
              sharey=False,
              ).map(sns.kdeplot, 'age', fill=True, alpha=0.15).add_legend();
plt.tight_layout()
```



Le graphique de survie des passagers masculins révèle un désavantage prononcé. La courbe des non-survivants (en bleu) reste systématiquement plus élevée que celle des survivants (en orange) pour presque tous les âges, du jeune âge adulte à la vieillesse. La seule exception est un petit pic dans la courbe de survie autour de 0 à 5 ans, qui représente les jeunes garçons ayant bénéficié de la politique « les femmes et les enfants d'abord ». Le pic le plus marqué de la courbe des hommes non survivants se situe au début de la vingtaine, ce qui souligne que les jeunes hommes constituaient le groupe le plus vulnérable. Ils étaient les derniers à avoir accès aux canots de sauvetage et devaient faire face à des difficultés physiques importantes pour s'échapper des cabines situées sur les ponts inférieurs.

En revanche, le graphique de survie des passagères montre que la courbe des survivantes (en orange) est supérieure à celle des non-survivantes (en bleu) pour la majorité des tranches d'âge. Les chances de survie des femmes étant nettement supérieures à celles des hommes.

```
# cat pour categorical, comme la variable 'sexe'
sns.catplot(data=df, x='sexe', y='age', hue='survivant', marker="*", alpha=0.5);
```

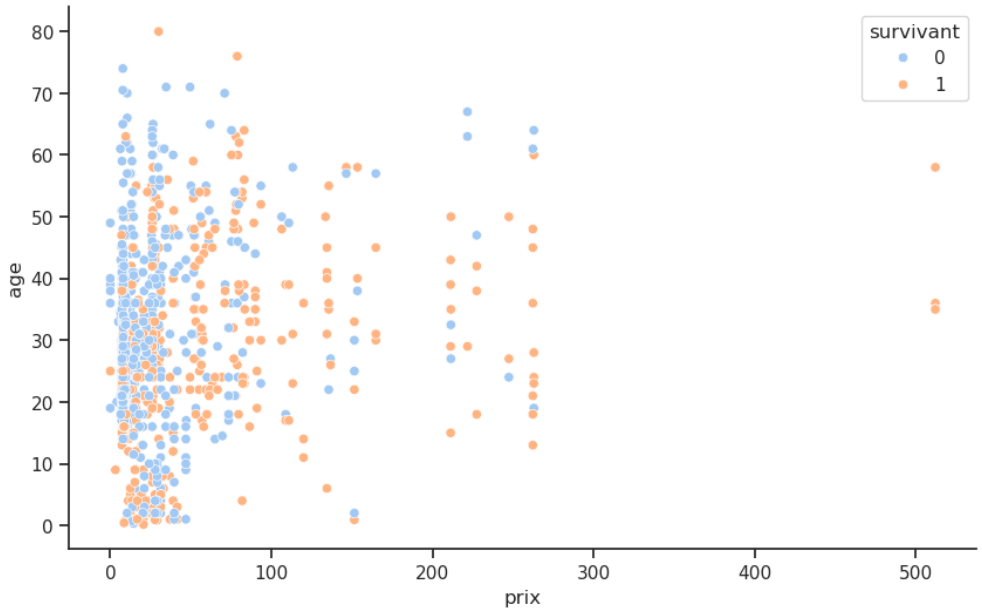



```
px.strip(df, x='sexe', y='age', color='survivant')
```

```
Unable to display output for mime type(s): text/html
```

18.4.10. Prix / Âge / Survie

```
sns.scatterplot(data=df, x='prix', y='age', hue='survivant');
```

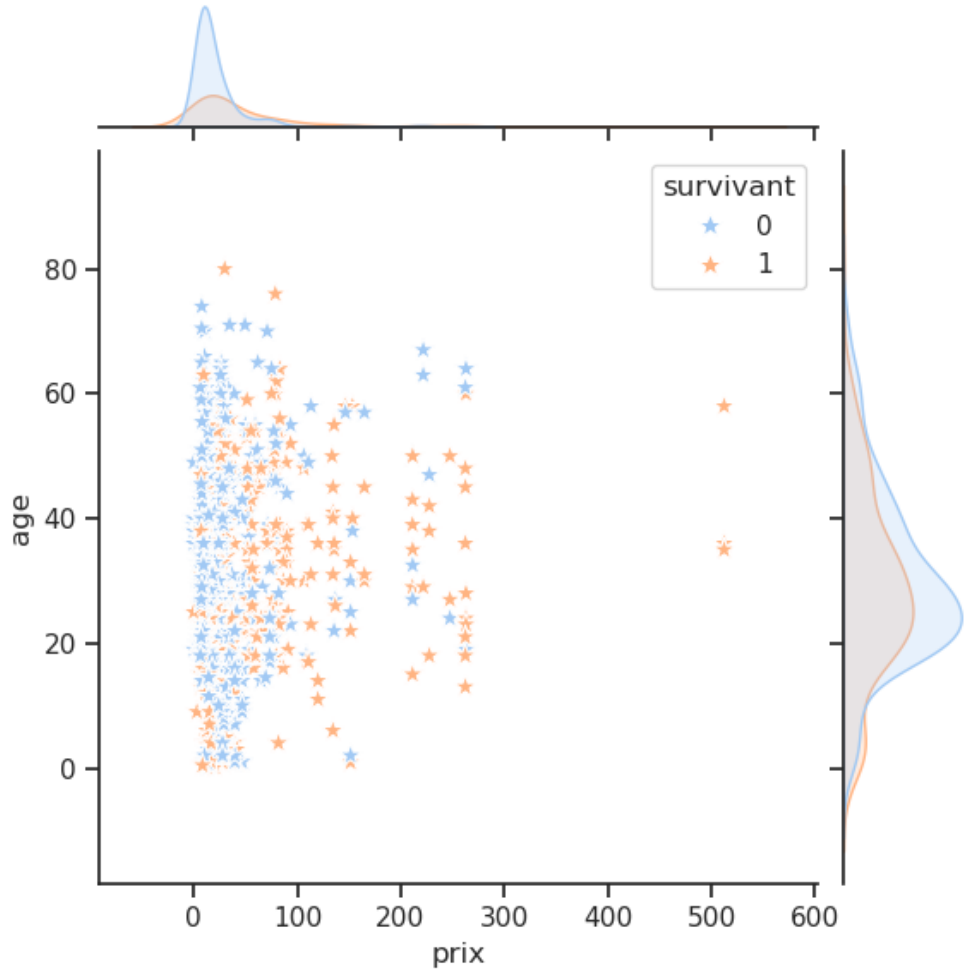



```
px.scatter(df, x='prix', y='age', color='survivant', opacity=0.5)
```

```
Unable to display output for mime type(s): text/html
```

La fonction `jointplot` ajoute des informations sur leurs distributions respectives sur les bords du graphique (grâce à des histogrammes ou des KDEs).

```
# joint
sns.jointplot(data=df, x='prix', y='age', hue='survivant', marker="*", s=100);
```




```
px.scatter(df, x='prix', y='age', color='survivant', marginal_x='histogram', marginal_y='histogram')
```

```
Unable to display output for mime type(s): text/html
```

18.4.11. Classe / Sexe / Survie

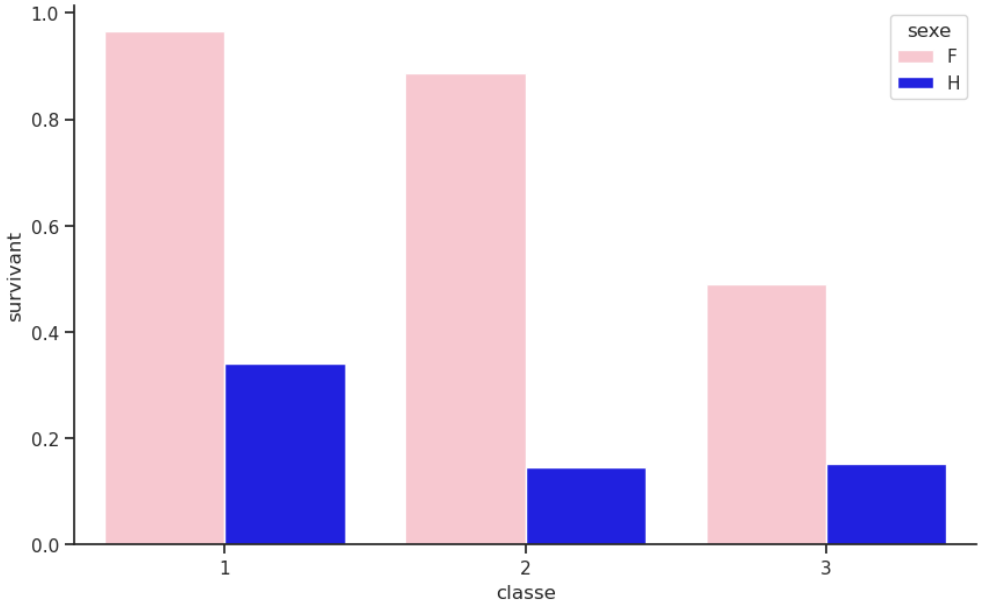
Regroupons les données d'abord par sexe et classe, puis sélectionnons le taux de survie et enfin calculons le taux de survie moyen pour chaque groupe.

```
df.groupby(['sexe','classe'])['survivant'].agg('mean').unstack()
```

classe sexe	1	2	3
F	0.965278	0.886792	0.490741
H	0.340782	0.146199	0.152130

On note que la classe sociale a un impact sur la survie. Les femmes de première classe ont survécu à 96 %, contre 50 % pour celle de troisième classe. En ce qui concerne les hommes, 36 % des hommes de première classe ont survécu, contre 13 % pour ceux de troisième classe.

```
sns.barplot(data=df, x='classe', y='survivant', hue='sexe',
            errorbar=None, palette={'F': 'pink', 'H': 'blue'});
```



Pour obtenir les mêmes résultats que précédemment, on peut utiliser la méthode `pivot_table()` qui permet de créer un tableau croisé dynamique. Voici un petit dessin pour comprendre :

Combien de survivants par classe et sexe ?

```
df.pivot_table(index='sexe', columns='classe', values='survivant', aggfunc='sum')
```

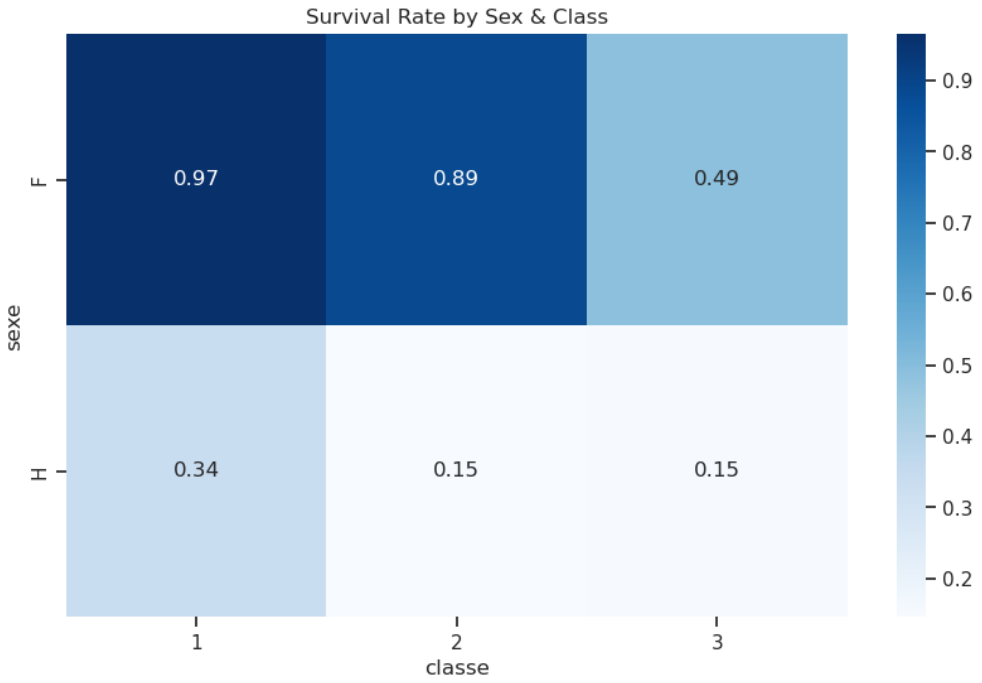
classe sexe	1	2	3
F	139	94	106
H	61	25	75

```
df.pivot_table(index='sexe', columns='classe', values='survivant', aggfunc='mean')
```

classe sexe	1	2	3
F	0.965278	0.886792	0.490741
H	0.340782	0.146199	0.152130

La même information peut être affichée sous forme de mappe de chaleur (heatmap) :

```
pivot = df.pivot_table(values="survivant", index="sexe", columns="classe", aggfunc="mean")
sns.heatmap(pivot, annot=True, cmap="Blues", fmt=".2f")
plt.title("Survival Rate by Sex & Class")
plt.show()
```



On peut spécifier plusieurs niveaux d'index pour obtenir des résultats plus détaillés, ici on a ajouté le groupe d'âge et, pour chaque groupe, le sexe.

```
df.pivot_table(index=['age_group','sexe'], columns='classe', values='survivant', aggfunc='sum',
↪  observed=False)
```

age_group	classe sexe
0-10	F
	H
10-20	F
	H
20-30	F
	H
30-40	F
	H
40-50	F
	H
50-60	F
	H
60-70	F
	H
70-80	F
	H
Unknown	F
	H

La même technique peut s'appliquer pour les colonnes :

```
df.pivot_table(index=['age_group','sexe'], columns=['classe','port_depart'], values='survivant',
↪  aggfunc='sum', observed=False)
```

age_group	classe port_depart sexe
0-10	F
	H
10-20	F
	H
20-30	F
	H
30-40	F
	H
40-50	F
	H
50-60	F
	H
60-70	F
	H
70-80	F
	H
Unknown	F
	H

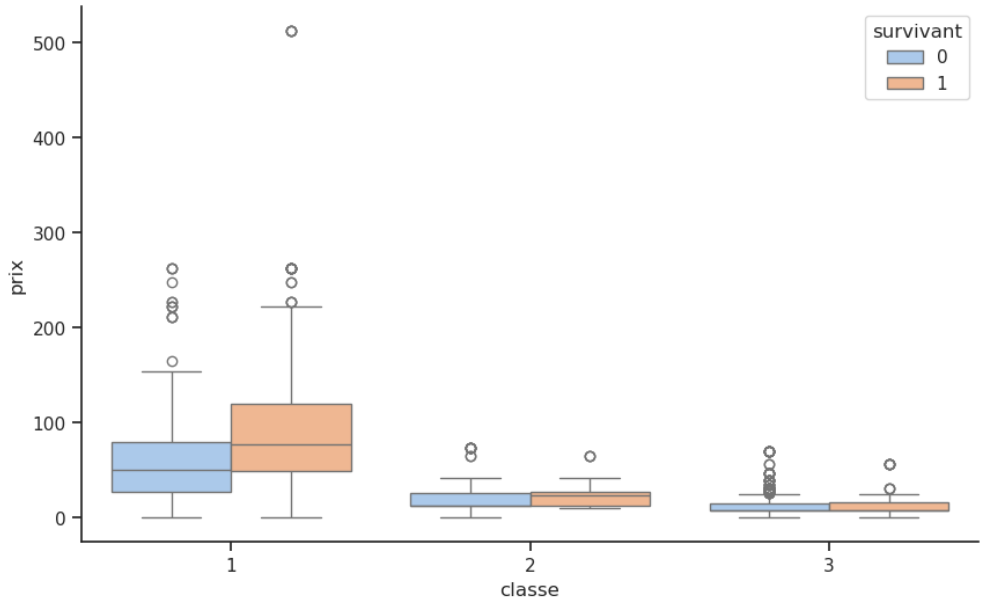
Pourcentage de survivants par sexe et age :

```
df.pivot_table(index='sexe', columns='age_group', values='survivant', aggfunc='sum', observed=False)
# Comme la colonne 'survivant' vaut 0 si décédé et 1 si survivant, la somme donne le nombre de survivants
```

age_group sexe	0-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	Unknown
F	25	50	91	61	36	23	5	1	47
H	25	14	43	28	16	7	1	1	26

18.4.12. Classe / Prix / Survie

```
sns.boxplot(data=df, x='classe', y='prix', hue='survivant');
```



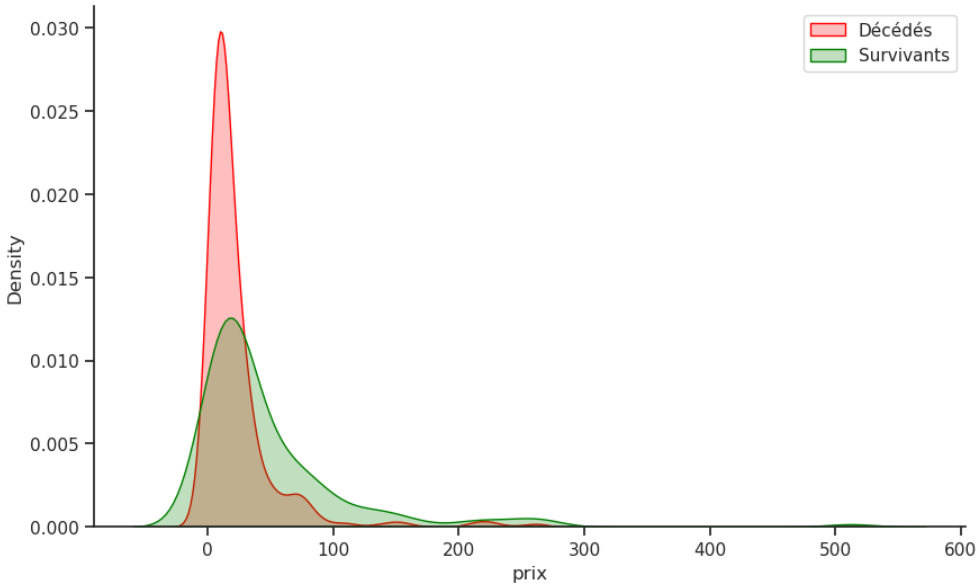
Le diagramme en boîte « Répartition des tarifs par classe et survie » offre une perspective plus détaillée sur la manière dont la richesse, telle que reflétée par le prix du billet, a influencé les chances de survie.

Première classe (classe = 1) : parmi les passagers de première classe, les survivants (boîte orange) avaient un tarif médian nettement plus élevé que les non-survivants (boîte bleue). Cela suggère que même au sein de ce groupe privilégié, les passagers ayant payé plus cher ont bénéficié de meilleures chances de survie, probablement grâce à des cabines situées plus près des ponts supérieurs et des canots de sauvetage. Les valeurs extrêmes à l'extrémité supérieure du graphique des tarifs des survivants indiquent que les passagers les plus riches, qui ont payé des tarifs exceptionnellement élevés, ont presque tous survécu.

Deuxième classe (classe = 2) : la différence entre les tarifs médians des survivants et des non-survivants est moins frappante, mais toujours présente. Les survivants ont tendance à avoir des tarifs légèrement plus élevés, ce qui suggère un léger avantage lié au fait de payer plus cher, bien que cet effet soit moins marqué qu'en première classe.

Troisième classe (classe = 3) : pour les passagers de troisième classe, la répartition des tarifs est uniformément basse, avec presque aucune différence entre les survivants et les non-survivants. Dans ce groupe, le prix du billet a eu un impact minime sur la survie. Les obstacles physiques importants que représentaient les ponts inférieurs et l'accès limité aux canots de sauvetage ont éclipsé les différences économiques mineures.

```
sns.kdeplot(df[df['survivant']==0]['prix'], fill=True, alpha=0.25, label='Décédés', color='red')
sns.kdeplot(df[df['survivant']==1]['prix'], fill=True, alpha=0.25, label='Survivants', color='green')
plt.legend();
```



18.4.13. Prix / Âge / Sexe / Survie

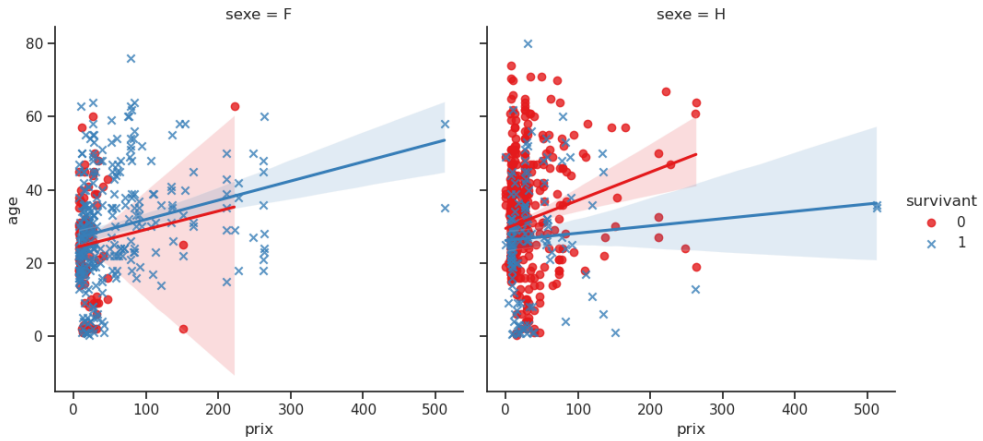
Nous souhaitons explorer la relation entre le prix payé et l'âge, tout en tenant compte du statut de survie (`survivant`) et du sexe.

La commande `lmplot()` est utilisée pour afficher une **régression linéaire** entre deux variables continues tout en permettant de visualiser la relation à l'aide de couleurs et de facettes.

- Le graphique affiche une régression linéaire entre les deux variables continues `prix` et `age`. Cela signifie qu'une droite (ou un autre modèle de régression si spécifié) est tracée pour chaque groupe de la variable `survivant`. La ligne de régression montre la tendance générale de la relation entre prix et âge. Par exemple, on observe que les personnes plus âgées ont tendance à payer plus cher.
- En fonction de la variable `hue='survivant'`, chaque groupe (survivant vs non-survivant) aura une couleur différente. Cela permet de comparer les relations entre prix et âge pour chaque groupe de manière visuellement distincte.
- Grâce à l'argument `col='sexe'`, on a des graphiques séparés pour chaque niveau de la variable sexe. Cela permet de visualiser comment la relation entre prix et âge varie entre les hommes et les femmes, par exemple.
- Le paramètre `markers=['o', 'x']` permet d'utiliser différents types de marqueurs pour les points représentant les survivants et les non-survivants, facilitant ainsi la distinction visuelle entre les deux groupes.

Ce graphique est pertinent si on pense que la relation entre deux variables continues est linéaire (ou presque linéaire) et souhaitons l'examiner graphiquement tout en incluant des groupes distincts (par exemple, selon `survivant`).

```
# On ajoute une quatrième dimension : le sexe
sns.lmplot(data=df, x='prix', y='age', hue='survivant', col='sexe', markers=['o', 'x'], palette='Set1');
```



Un graphique 3D avec la couleur comme 4e dimension est également possible :

```
# df['survivant'] = df['survivant'].astype('category')

df['survivant_code'] = df['survivant'].map({0: 'Non', 1: 'Oui'})

px.scatter_3d(
    df,
    x='prix',
    y='age',
    z='sexe',
    # color='survivant',
    color='survivant_code',
    # Si la colonne est numérique, on peut la colorer mais il faut color_continuous_scale
    # color_continuous_scale='sunset',
    # On l'a transformée en catégorielle pour utiliser color_discrete_sequence
    color_discrete_sequence=['green', 'red'],  # il faut que 'survivant' soit une variable catégorielle !
    # labels={'prix': 'Prix', 'age': 'Âge', 'sexe': 'Sexe', 'survivant_code': 'Survivant'},
    # title='Classe vs Age vs Sexe selon le statut de survie',
)
```

```
Unable to display output for mime type(s): text/html
```

18.5. Miscellanea

18.5.1. `groupby()`

Cout moyen (en livres d'époque) des billets par classe.

Selon (Wikipedia)[https://fr.wikipedia.org/wiki/Passagers_du_Titanic#Le_naufrage_et_son_bilan], le prix moyen des billets était de 7 livres de l'époque pour la troisième classe. Notre jeu de données indique un prix moyen de 13,30 livres pour la troisième classe, ce qui semble un peu élevé. Cependant, il est possible que notre jeu de données ne contienne pas tous les passagers de troisième classe.

```
df.groupby('classe')['prix'].mean().to_frame()
```

classe	prix
1	87.508992
2	21.179196
3	13.302889

18.5.2. Filtres et `value_counts`

Parmi les enfants, les filles ont-elles été plus sauvées que les garçons ?

```
mask_mineur_homme = (df['age'] < 10) & (df['sexe']=="H")
df[mask_mineur_homme]['survivant'].value_counts()
```

```
survivant
1    25
0    18
Name: count, dtype: int64
```

```
mask_mineur_femme = (df['age'] < 10) & (df['sexe']=="F")
df[mask_mineur_femme]['survivant'].value_counts()
```

```
survivant
1    25
0    14
Name: count, dtype: int64
```

18.5.3. Corrélation entre les données : `corr`

La corrélation est le degré auquel deux ou plusieurs attributs ou mesures sur le même groupe d'éléments ont tendance à varier ensemble.

Les corrélations sont parmi les éléments les plus courants et les plus utiles de l'analyse des données. Qu'est-ce qui bouge avec quoi ? Quelles variables sont « dépendantes » et lesquelles sont « indépendantes » ? Quelles sont donc les questions pour lesquelles nous pourrions vouloir trouver des corrélations ?

Quelles sont les corrélations que nous pourrions rechercher ?

- Existe-t-il une corrélation entre l'âge et le prix du billet ?
- Existe-t-il une corrélation entre la classe et la survie ? Les riches ont-ils survécu davantage que les travailleurs ?
- Y a-t-il une corrélation entre l'âge et la survie ?
- Les femmes et les enfants ont-ils vraiment le droit de passer en premier ?

Nous connaissons tous le vieil adage selon lequel, lorsqu'un navire coule, ce sont « les femmes et les enfants d'abord » qui montent à bord des canots de sauvetage. Ce vieil adage s'est-il vérifié sur le Titanic ? Les femmes et les enfants ont-ils été plus nombreux à survivre que les hommes ? Le sexe, l'âge ou la classe sociale des passagers du Titanic ont-ils joué un rôle déterminant dans leur survie ?

Si les femmes et les enfants ont survécu plus que les hommes, il devrait y avoir une corrélation positive entre la survie et le sexe et la survie et l'âge.

La méthode `.corr()` de pandas permet de trouver la corrélation entre deux caractéristiques quelconques d'un jeu de données.

```
df['age'].corr(df['survivant'])
```

```
-0.055512520192146246
```

Une valeur +1 signifie qu'il existe une corrélation positive parfaite entre deux caractéristiques, c'est-à-dire que lorsqu'une caractéristique augmente, l'autre augmente exactement dans les mêmes proportions.

Zéro signifie qu'il n'y a pas de corrélation entre deux caractéristiques. Elles se déplacent complètement au hasard l'une par rapport à l'autre.

Une valeur -1 signifie qu'il existe une corrélation parfaite, négative ou inverse entre deux caractéristiques. Lorsqu'une caractéristique augmente, l'autre diminue et vice versa.

D'après la corrélation entre le sexe et la survie (53 %) et l'âge et la survie (-0,05 %), nous pouvons affirmer certaines choses :

- le fait qu'un passager soit un homme était fortement corrélé négativement avec sa survie à bord du Titanic.
- le fait qu'un passager soit plus âgé est très faiblement corrélé négativement avec sa survie.

Mais « l'âge » n'est pas « les enfants », n'est-ce pas ? Comment savoir si le fait d'avoir moins de 10 ans augmentait les chances de survie ? Quelle était la probabilité qu'une personne de moins de 10 ans survive par rapport à une personne de plus de 10 ans ?

```
masque = df['age'] < 10
df[masque]['age'].corr(df['survivant'])
```

```
-0.14006219496281994
```

On ne peut pas calculer la corrélation avec le sexe car c'est une variable catégorielle. On peut utiliser la méthode `groupby()` pour regrouper les données par sexe et calculer la moyenne de survie pour chaque groupe. Ou sinon ajouter une colonne où les valeurs catégorielles sont encodées par des valeurs numériques, par exemple 0 pour les hommes et 1 pour les femmes.

```
# Catégorisation de la variable 'sexe' en 'sexe_code'
df['sexe_code'] = df['sexe'].astype('category').cat.codes
# df['sexe_code'] = df['sexe'].map({'H': 0, 'F': 1})
# df['sexe_code'] = df['sexe'].replace({'H': 0, 'F': 1})
df
```

	classe	survivant	sexe	age	prix	port_depart	age_group	sexe_num	personne	survivant_c
0	1	1	F	29.0000	211.3375	S	20-30	0	femme	Oui
1	1	1	H	0.9167	151.5500	S	0-10	1	enfant	Oui
2	1	0	F	2.0000	151.5500	S	0-10	0	enfant	Non
3	1	0	H	30.0000	151.5500	S	20-30	1	homme	Non
4	1	0	F	25.0000	151.5500	S	20-30	0	femme	Non
...	...	...	...	...	...	...	...	...	...	...
1304	3	0	F	14.5000	14.4542	C	10-20	0	femme	Non
1305	3	0	F	NaN	14.4542	C	Unknown	0	femme	Non
1306	3	0	H	26.5000	7.2250	C	20-30	1	homme	Non
1307	3	0	H	27.0000	7.2250	C	20-30	1	homme	Non
1308	3	0	H	29.0000	7.8750	S	20-30	1	homme	Non

Pour l'instant, examinons la matrice de corrélation.

```
corr_matrix = df[['survivant', 'sexe_code', 'age', 'classe', 'prix']].corr()

# Ordonner les colonnes et les lignes selon la corrélation avec la variable 'survivant'
sorted_columns = corr_matrix['survivant'].abs().sort_values(ascending=False).index
sorted_corr_matrix = corr_matrix.loc[sorted_columns, sorted_columns]

corr_matrix['survivant']
```

```
survivant    1.000000
sexe_code   -0.528693
age         -0.055513
classe      -0.312469
prix         0.244265
Name: survivant, dtype: float64
```

```
# Avec Seaborn
# ---------------------------
# sns.heatmap(sorted_corr_matrix, linewidths=0.5, annot=True, cbar=True, cmap="coolwarm")
# plt.show()

#Avec plotly.express
# ---------------------------
px.imshow(sorted_corr_matrix,
          text_auto='.2f',
          color_continuous_scale='RdBu_r',
          aspect='auto').show()
```

```
Unable to display output for mime type(s): text/html
```

18.5.3.1. Attention : toutes les variables codées ne se valent pas

Nous venons de remplacer une variable catégorielle par des codes numériques pour la faire entrer dans une matrice de corrélation. C'est une manipulation **très courante** : et qui n'est légitime que dans certains cas. Il faut distinguer trois situations.

Type de variable	Exemple ici	Corrélation de Pearson ?
Numérique	`age`, `prix`	oui, c'est son usage normal
Binaire (2 modalités)	`sexe` → 0 / 1	oui : c'est la *corrélation point-bisériale*. Seul le **signe** dépend de la convention de codage
Ordinale (modalités ordonnées)	`classe` → 1, 2, 3	acceptable, car l'ordre a un sens : mais on ne mesure que la part **monotone** de la relation
Nominale (3 modalités non ordonnées)	`port_depart` → C, Q, S	**non**, le résultat n'a aucun sens

Pourquoi c'est faux pour une variable nominale. Coder C, Q, S par 0, 1, 2 revient à affirmer que Queenstown est « entre » Cherbourg et Southampton, et que l'écart C→Q est le même que l'écart Q→S. Ces affirmations n'ont aucun sens : l'ordre choisi est purement arbitraire, hérité de l'alphabet.

La conséquence est facile à vérifier : **changez l'ordre du codage, et le coefficient change.**

```
# Le même port de départ, codé de deux façons également arbitraires
codage_1 = df['port_depart'].map({'C': 0, 'Q': 1, 'S': 2})   # ordre alphabétique
codage_2 = df['port_depart'].map({'S': 0, 'C': 1, 'Q': 2})   # un autre ordre, tout aussi valable

display(Markdown(f"codage C,Q,S = 0,1,2  ->  corrélation avec la survie =
 ↪  {codage_1.corr(df['survivant']): .4f}"))
display(Markdown(f"codage S,C,Q = 0,1,2  ->  corrélation avec la survie =
 ↪  {codage_2.corr(df['survivant']): .4f}"))
```

codage C,Q,S = 0,1,2 -> corrélation avec la survie = -0.1768

codage S,C,Q = 0,1,2 -> corrélation avec la survie = 0.0998

Mêmes données, même question : et non seulement la valeur change, mais **le signe s'inverse**. Le coefficient ne mesure donc rien du port d'embarquement : il mesure l'ordre dans lequel nous avons écrit les lettres.

Notez que `.astype('category').cat.codes`, bien pratique, applique exactement le codage alphabétique : il produit silencieusement le premier des deux résultats.

La bonne façon de faire avec une variable nominale est celle que nous avons déjà employée dans la section *Port de départ / Survie* : comparer le **taux de survie modalité par modalité**, avec `groupby` ou un tableau croisé.

```
df.groupby('port_depart')['survivant'].agg(
    taux_de_survie = 'mean',
    effectif       = 'count'
).round(3)
```

port_depart	taux_de_survie	effectif
C	0.556	270
Q	0.358	123
S	0.333	914

On lit directement le résultat, sans aucune convention arbitraire : **55,6 %** de survie pour Cherbourg, **35,8 %** pour Queenstown, **33,3 %** pour Southampton. L'information est là, et elle est interprétable.

À retenir pour vos propres analyses : une matrice de corrélation ne doit contenir que des variables **numériques, binaires ou ordinales**. Pour les autres, un tableau croisé des taux dit bien davantage : et dit vrai.

```
# df['personne_code'] = df['personne'].astype('category').cat.codes
# corr_matrix = df[['survivant','personne_code','age','classe']].corr()
# sorted_columns = corr_matrix['survivant'].abs().sort_values(ascending=False).index
# sorted_corr_matrix = corr_matrix.loc[sorted_columns, sorted_columns]
# plt.figure(figsize=(10, 10))
# sns.heatmap(sorted_corr_matrix, linewidths=0.5, annot=True, cbar=True, cmap="coolwarm");
```

```
# dftemp = df[['survivant','sexe_code','age','classe']]
# corr_matrix = dftemp.corr()['survivant'].sort_values(ascending=False)

# # Sélection des colonnes ordonnées par corrélation avec 'survivant' (à l'exclusion de 'survivant'
 ↪  elle-même)
# sorted_columns = corr_matrix.index.tolist()[1:]

# # Création du heatmap avec Seaborn
# sns.heatmap(dftemp[sorted_columns].corr(), annot=True, linewidths=0.5, cmap="coolwarm", cbar=True);
```

18.6. Autres ressources

Voici trois excellentes vidéos (en français) de la chaîne YouTube **Machine Learnia** qui expliquent comment faire une analyse statistique du jeu de données du Titanic :

1. Vidéo 1 : Analyse exploratoire de données
2. Vidéo 2 : Data visualization (à partir de 19:50)
3. Vidéo 3 : Feature engineering (à partir de 6:35)

18.7. Pour aller plus loin : analyse exploratoire → prédiction

Hors séance : pour votre curiosité. Ce qui suit dépasse le cadre de ce cours : il s'agit d'une ouverture vers l'apprentissage automatique, présentée pour montrer à quoi sert le travail de préparation des données que nous venons de faire.

Rien de cette section n'est exigible, et l'examen ne porte que sur Pandas. Le code utilise la bibliothèque `scikit-learn`, qui n'est pas au programme.

Dans cette exploraiton du dataset Titanic avec Pandas, nous avons vu comment filtrer et sélectionner des données, faire des statistiques descriptives, croiser et agréger des variables, visualiser les résultats avec différents types de graphiques. Tout cela nous permet de **comprendre les facteurs qui influencent la survie** des passagers. Mais ces outils restent des analyses descriptives : ils montrent des relations, mais ne permettent pas de prédire directement le destin d'un passager inconnu.

Une fois que l'on a identifié les facteurs importants, on peut se poser la question : **"Peut-on prédire si un passager va survivre ou non à partir de ses caractéristiques (âge, sexe, classe, tarif, etc.) ?"**

La prédiction cherche à **estimer une valeur future ou inconnue** sur la base de données existantes. Les étapes générales pour prédire sont les suivantes :

1. **Préparer et explorer les données**.
2. **Choisir un modèle** : régression logistique, arbres de décision, réseaux de neurones, etc.
3. Séparer les données en **train set** (pour entraîner le modèle) et **test set** (pour évaluer la performance)
4. **Entraîner le modèle** sur les données existantes (*train set*)
5. **Évaluer le modèle** sur des données jamais vues (*test set*) et évaluer la performance avec des métriques adaptées (accuracy, precision, recall)

Pandas est utilisé pour la **préparation des données** avant de les passer à un modèle de machine learning.

Dans le cas du Titanic :

- Variable cible : `survivant` (0 = mort, 1 = survécu)
- Features (variables explicatives) : `classe`, `sexe`, `age`, `port_depart`, etc.

C'est un **problème de classification** : prédire une catégorie (`0` ou `1`) à partir des autres caractéristiques.

Observations après l'analyse exploratoire :

- les femmes ont survécu plus souvent que les hommes
- les passagers de première classe ont survécu plus souvent que ceux de troisième classe
- les enfants ont un taux de survie légèrement supérieur à celui des adultes

Un modèle de prédiction pourrait apprendre ces patterns et estimer la probabilité de survie d'un passager inconnu.

Conclusion
L'analyse exploratoire avec Pandas est la première étape pour comprendre les données et identifier les facteurs influençant la survie des passagers. Pour aller plus loin, on peut passer à la prédiction : estimer si un passager inconnu survivra ou non à partir de ses caractéristiques.

Cette étape implique le ***machine learning***, où l'on choisit un modèle (régression logistique, arbres de décision, réseau de neurones…), on entraîne ce modèle sur un train set, puis on évalue sa performance sur un test set jamais vu.

Les modèles de machine learning permettent non seulement de quantifier l'impact de chaque variable, mais aussi de capturer des relations complexes entre elles. Les réseaux de neurones (*deep learning*) sont particulièrement utiles lorsque les patterns sont non linéaires ou que les datasets sont volumineux.

Ainsi, le workflow complet va de l'observation des données à la prise de décision prédictive, illustrant comment l'intelligence artificielle appliquée aux données réelles peut aider à estimer des probabilités et guider des actions.

Voici plusieurs modèles de prédiction avec la bibliothèque Scikit-Learn pour les entraîner sur le dataset Titanic :

```
from sklearn.model_selection import train_test_split

from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import SGDClassifier
# from sklearn.cluster import KMeans
# from sklearn.decomposition import PCA

from sklearn.metrics import accuracy_score, confusion_matrix, precision_score, recall_score, f1_score
from sklearn.preprocessing import StandardScaler

# ------------------------------------------------------------
# 1. PRÉPARATION DES DONNÉES (COMMUNE AUX MODÈLES)
# ------------------------------------------------------------

data = df.copy()

# Remplacer les valeurs manquantes
data['age'] = data['age'].fillna(data['age'].median())

# Encoder les variables catégorielles
data['sexe_num'] = data['sexe'].astype('category').cat.codes

# Features et cible
X = data[['classe', 'sexe_num', 'age']]
y = data['survivant']

# Train/Test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42,
    stratify=y # pour conserver la même proportion de survivants dans train et test
)
display(Markdown(f"Dimension de X_train : {X_train.shape}, Dimension de X_test : {X_test.shape}"))

# Normalisation (utile pour MLP, k-NN, SVC)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)



# ------------------------------------------------------------
# 2. LISTE DES MODÈLES
# ------------------------------------------------------------

models = [
    ("Baseline (Dummy)", DummyClassifier(strategy='most_frequent', random_state=42)),
    ("Régression logistique", LogisticRegression(max_iter=1000)),
    ("Réseau de neurones (MLP)", MLPClassifier(hidden_layer_sizes=(32,16),
                                               activation='logistic', # 'relu', 'tanh'
                                               solver='adam',
                                               learning_rate='adaptive',
                                               learning_rate_init=0.01,
                                               early_stopping=True,
                                               n_iter_no_change=20,
                                               max_iter=500,
                                               random_state=42)),
```

```
    ("k-Nearest Neighbors", KNeighborsClassifier(n_neighbors=5)),
    ("Random Forest", RandomForestClassifier(criterion='gini', n_estimators=450, random_state=1,
 ↪  max_depth=20, min_samples_leaf=3, min_samples_split=2) ),
    ("Decision Tree", DecisionTreeClassifier(random_state=42)),
    ("Stochastic Gradient Descent", SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)),
    ("Gradient Boosting", GradientBoostingClassifier(n_estimators=100, random_state=42)),
    ("Support Vector Classifier", SVC(kernel='rbf', probability=True, random_state=42)),
    ("Naive Bayes (Gaussian)", GaussianNB())
]

# ---------------------------------------------------------
# 3. ENTRAÎNEMENT ET STOCKAGE DES RÉSULTATS
# ---------------------------------------------------------

results = []

for name, model in models:
    if name in ["Régression logistique", "Réseau de neurones (MLP)", "k-Nearest Neighbors", "Support
     ↪  Vector Classifier"]:
        model.fit(X_train_scaled, y_train)
        y_pred = model.predict(X_test_scaled)
    else:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, zero_division=0)
    rec = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)

    # Extraction des valeurs pour classification binaire
    if cm.shape == (2,2):
        TN, FP, FN, TP = cm.ravel()
    else:
        TN = FP = FN = TP = None

    results.append({
        "Modèle": name,
        "Accuracy": acc,
        "Précision": prec,
        "Rappel": rec,
        "F1-Score": f1,
        "Vrai 0 / Prédit 0": TN,
        "Vrai 1 / Prédit 1": TP,
        "Vrai 0 / Prédit 1": FP,
        "Vrai 1 / Prédit 0": FN
    })

final_df = pd.DataFrame(results)
final_df.sort_values(by='Accuracy', ascending=False)
```

Dimension de X_train : (1047, 3), Dimension de X_test : (262, 3)

	Modèle	Accuracy	Précision	Rappel	F1-Score	Vrai 0 / Prédit 0	Vrai 1 / Prédit 1	Vrai 0 / F
8	Support Vector Classifier	0.835878	0.880000	0.66	0.754286	153	66	9
7	Gradient Boosting	0.828244	0.848101	0.67	0.748603	150	67	12
4	Random Forest	0.820611	0.762376	0.77	0.766169	138	77	24
1	Régression logistique	0.816794	0.776596	0.73	0.752577	141	73	21
5	Decision Tree	0.801527	0.750000	0.72	0.734694	138	72	24
9	Naive Bayes (Gaussian)	0.793893	0.739583	0.71	0.724490	137	71	25

	Modèle	Accuracy	Précision	Rappel	F1-Score	Vrai 0 / Prédit 0	Vrai 1 / Prédit 1	Vr
3	k-Nearest Neighbors	0.774809	0.688073	0.75	0.717703	128	75	34
2	Réseau de neurones (MLP)	0.732824	0.894737	0.34	0.492754	158	34	4
6	Stochastic Gradient Descent	0.706107	0.810811	0.30	0.437956	155	30	7
0	Baseline (Dummy)	0.618321	0.000000	0.00	0.000000	162	0	0

18.7.1. Mesures de fiabilité des prédictions

Pour évaluer la pertinence du modèle de machine learning appliqué au Titanic, nous utilisons les **métriques dérivées de la matrice de confusion**.

La matrice de confusion est un tableau qui permet de visualiser les performances d'un algorithme de classification. Elle compare les prédictions du modèle avec les résultats réels :

Prévu non-survécu

Prévu survécu

Réel non-survécu

Vrais négatifs

Faux positifs

Réel survécu

Faux négatifs

Vrais positifs

Par exemple, dans le cas du modèle SVC appliqué au dataset Titanic, la matrice de confusion est la suivante :

- **Vrais négatifs (VN)** : le modèle a correctement prédit que 153 passagers n'ont pas survécu.
- **Faux positifs (FP)** : le modèle a prédit que 9 passagers survivraient alors qu'ils sont morts.
- **Faux négatifs (FN)** : le modèle a prédit que 34 passagers ne survivraient pas alors qu'ils ont survécu.
- **Vrais positifs (VP)** : le modèle a correctement prédit que 66 passagers ont survécu.

Métriques de performance

1. **Accuracy** (précision globale)
   Proportion de bonnes prédictions parmi l'ensemble des prédictions :
$$\frac{VN + VP}{VN + VP + FP + FN}$$
2. **Précision (precision)**
   Mesure la proportion de prédictions positives correctes. Elle pénalise les **faux positifs**.
$$\text{precision} = \frac{VP}{VP + FP}$$
3. **Rappel (recall ou sensibilité)**
   Mesure la capacité du modèle à identifier correctement les survivants (classe positive).
$$\text{recall} = \frac{VP}{VP + FN}$$
4. **F1-score**
   Moyenne harmonique entre la précision et le rappel.
$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

5. **Spécificité**
   Proportion de vrais négatifs correctement identifiés.

$$\text{spécificité} = \frac{VN}{VN + FP}$$

Interprétation des résultats

Accuracy : parmi les modèles testés, le SVC obtient la meilleure performance avec une précision globale de **83,6%**. Cela signifie que **83,6% des prédictions du modèle sont correctes**.
Cependant, cette mesure ne distingue pas les erreurs concernant les **survivants** et les **non-survivants**.
La matrice de confusion montre que le modèle dentifie correctement une majorité de non-survivants, **sous-prédit la survie**, avec **34 faux négatifs**, ce qui signifie qu'un nombre significatif de survivants réels sont classés comme non-survivants.
Globalement, le modèle est **performant**, mais la sous-détection des survivants peut être problématique selon les objectifs de l'étude.

Troisième partie

Exercices

Chapitre 19.

Premières manipulations avec Pandas

19.1. Exercice : premières manipulations de Series

- Construire la Series suivante :

Series	
index	value
0	12
1	-4
2	7
3	9

- Modifier la valeur de `value` à 10 pour l'élément en position 3.
- Afficher les éléments avec `value` supérieur à 5.
- Modifier la valeur de `value` à 0 pour les éléments avec un `value` inférieur à 5.
- Ajouter un élément avec un `value` de 3.5 à la Serie (cet élément doit avoir un label différent).
- Modifier les indices de la Serie pour qu'ils soient des lettres de l'alphabet.

```
import pandas as pd

# help(pd.Series)
```

```
from IPython.display import display, Markdown
```

```
# creation et affichage de la serie
# s = pd.Series([12, -4, 7, 9])
s = pd.Series(data=[12, -4, 7, 9])
s
```

```
0    12
1    -4
2     7
3     9
dtype: int64
```

```
# affichage des valeurs de la serie
s.values
```

```
array([12, -4,  7,  9])
```

```
# modification de la valeur de l'element d'indice 3
# s[3] = 10      # ambiguïté possible entre position et étiquette
# s.iloc[3] = 10 # modification par position
s.loc[3] = 10    # modification par étiquette
s
```

```
0    12
1    -4
2     7
3    10
dtype: int64
```

```
# filtre sur les valeurs de la serie
mask = s > 5 # création du masque
display(mask) # affichage du masque = c'est une serie de booléens
display(s[mask]) # affichage des valeurs filtrées
```

```
0     True
1    False
2     True
3     True
dtype: bool
```

```
0    12
2     7
3    10
dtype: int64
```

```
# filtre sur les valeurs de la serie et modification des valeurs
s[ s<5 ] = 0
s
```

```
0    12
1     0
2     7
3    10
dtype: int64
```

```
# Ajout d'une valeur à la serie
#
# VISION NUMPY
# Concatenantion de deux series dont une contenant une seule valeur

# # L = pd.Series([3.5])
# # # s = pd.concat( [s,L] ) # PB !!!! deux fois le même index
# # s = pd.concat( [s,L] , ignore_index=True)
# # display(s)

# # Autre idée :
# L = pd.Series(data=[3.5], index=[len(s)])
# s = pd.concat( [s,L] )
# display(s)



# VISION DICTIONNAIRE
s[len(s)] = 100
display(s)
```

```
0     12
1      0
2      7
3     10
4    100
dtype: int64
```

```
# changement des indices
s.index = ['a', 'b', 'c', 'd', 'e']
s
```

```
a     12
b      0
c      7
d     10
e    100
dtype: int64
```

19.2. Exercice : premières manipulations de DataFrame

- Construire le DataFrame suivant :

	color	object	price
0	red	pen	1.2
1	green	pencil	0.6
2	blue	marker	1.0
3	yellow	eraser	0.7
4	black	ruler	1.5

- Afficher les lignes correspondant aux objets dont le prix est supérieur ou égale à 1.
- Modifier le prix de l'objet "eraser" pour le passer à 0.2.

La construction du DataFrame peut se faire en deux étapes. D'abord, on crée un dictionnaire avec les données (colonne par colonne), puis on crée le DataFrame à partir de ce dictionnaire.

```
import pandas as pd

# un dictionaire de listes, chaque liste representant une colonne
data = {'color' :  ['red', 'green', 'blue', 'yellow', 'black'],
        'object' : ['pen', 'pencil', 'marker', 'eraser', 'ruler'],
        'price' :  [1.2, 0.6, 1.0, 0.7, 1.5]}

df = pd.DataFrame(data)
df
```

	color	object	price
0	red	pen	1.2
1	green	pencil	0.6
2	blue	marker	1.0
3	yellow	eraser	0.7
4	black	ruler	1.5

On va séléctionner les lignes du DataFrame en utilisant un masque :

```
mask = (df['price'] > 1.0)
df[ mask ] # application du masque à toutes les colonnes
```

	color	object	price
0	red	pen	1.2
4	black	ruler	1.5

```
# pour comprendre
mask
# Dans la sortie on voit l'indice de la ligne et un booléen qui indique si la condition est vraie ou
 ↪  fausse pour cette ligne
```

```
0     True
1    False
2    False
3    False
4     True
Name: price, dtype: bool
```

On va modifier la valeur d'une cellule : avec `.loc` on peut accéder à une cellule en utilisant les indices de la ligne et de la colonne. Pour trouver l'indice de la ligne correspondant à l'objet "eraser", on peut utiliser un masque.

```
label_de_la_ligne = (df['object']=='eraser')
label_de_la_colonne = 'price'
df.loc[ label_de_la_ligne, label_de_la_colonne ] = 0.2
df
```

	color	object	price
0	red	pen	1.2
1	green	pencil	0.6
2	blue	marker	1.0
3	yellow	eraser	0.2
4	black	ruler	1.5

Pour comprendre :

```
df['object']=='eraser'
```

```
0    False
1    False
2    False
3     True
4    False
Name: object, dtype: bool
```

```
df.loc[ df['object']=='eraser', : ]
```

	color	object	price
3	yellow	eraser	0.2

19.3. Exercice : dict de dict en DataFrame

Transformer en dataframe le dictionnaire de dictionnaires suivant :

```
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio'  : {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Une autre forme courante de données est un dictionnaire qui contient plusieurs autres dictionnaires. Si le dictionnaire priciapl est passé au DataFrame, pandas interprétera les clés du dictionnaire extérieur comme étant les colonnes, et les clés intérieures comme étant les indices des lignes :

```
import pandas as pd

pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio'  : {2000: 1.5, 2001: 1.7, 2002: 3.6}}

pd.DataFrame(data=pop)
```

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

19.4. Exercice : régression linéaire

Source : problème 32 de *"Python & Pandas et les 36 problèmes de Data Science"*, Frédéric Bro et Chantal Remy

Dans *The Song of Insects*, George W. Pierce a mesuré la fréquence des stridulations : le son produit par le frottement des ailes : de plusieurs insectes, ainsi que la température à laquelle ces stridulations ont été enregistrées. L'idée est ancienne et connue des naturalistes : un insecte stridulerait d'autant plus vite qu'il fait chaud, au point de pouvoir servir de thermomètre.

Le fichier `grillons.csv` contient **15 relevés** pour le **grillon rayé** (*Nemobius fasciatus*) : le nombre de stridulations par seconde et la température en degrés Fahrenheit.

1. Charger le fichier et afficher les données.
2. Établir par **régression linéaire** la relation entre la fréquence des stridulations et la température. Autrement dit, en notant x la fréquence et y la température, on cherche a et b tels que $y \approx ax + b$. La méthode des moindres carrés consiste à minimiser la somme des carrés des écarts entre valeurs observées et valeurs prédites :

$$(a, b) \qquad \text{tels que} \qquad \sum_{i=1}^{n} (y_i - (ax_i + b))^2 \quad \text{soit minimale.}$$

3. Quelle température prédit-on lorsque le grillon stridule 19 fois par seconde ?
4. Vérifier que la droite de régression passe par le point moyen du nuage.
5. Calculer le coefficient de corrélation linéaire r et conclure sur la qualité de l'ajustement.

Bonus. Le fichier `cigales.csv` contient **19 relevés** analogues pour la **cigale** *Tibicen linnei*. Refaire la même analyse et comparer les deux espèces : la cigale est-elle un aussi bon thermomètre que le grillon ?

Les coefficients de la droite s'obtiennent avec la fonction `polyfit` de NumPy, qui prend les abscisses, les ordonnées et le degré du polynôme à ajuster : ici 1. Elle renvoie les coefficients par puissances décroissantes :

```
import numpy as np
a, b = np.polyfit(x_points, y_points, 1)
```

19.5. Le grillon rayé

Le fichier est au format français : séparateur `;` et virgule décimale. On vérifie les types après lecture, comme toujours.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

grillons = pd.read_csv('grillons.csv', sep=';', decimal=',')

display(grillons)
display(grillons.dtypes)
```

	stridulations_par_seconde	temperature_F
0	20.0	88.6
1	16.0	71.6
2	19.8	93.3
3	18.4	84.3
4	17.1	80.6
5	15.5	75.2
6	14.7	69.7
7	17.1	82.0
8	15.4	69.4
9	16.2	83.3
10	15.0	79.6
11	17.2	82.6
12	16.0	80.6
13	17.0	83.5
14	14.4	76.3

```
stridulations_par_seconde    float64
temperature_F                float64
dtype: object
```

```
x_points = grillons['stridulations_par_seconde']
y_points = grillons['temperature_F']

a, b = np.polyfit(x_points, y_points, 1)
f = lambda x : a*x + b

display(Markdown(f"droite de régression : y = {a:.3f} x + {b:.2f}"))
```

droite de régression : y = 3.291 x + 25.23

Prédiction pour un grillon qui stridule 19 fois par seconde :

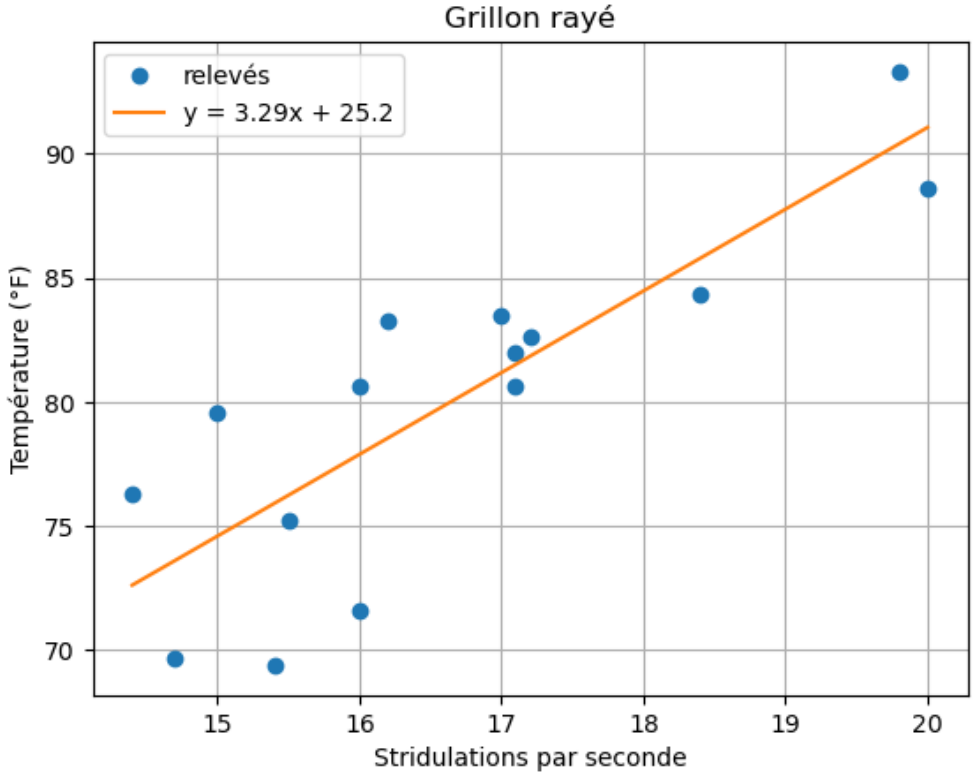
```
f(19)   # identique à np.polyval([a, b], 19)
```

```
87.76310079558164
```

```
plt.plot(x_points, y_points, 'o', label='relevés')

xx = np.linspace(x_points.min(), x_points.max(), 101)
plt.plot(xx, f(xx), label=f'y = {a:.2f}x + {b:.1f}')

plt.xlabel('Stridulations par seconde')
plt.ylabel('Température (°F)')
plt.title('Grillon rayé')
plt.legend(); plt.grid(); plt.show()
```



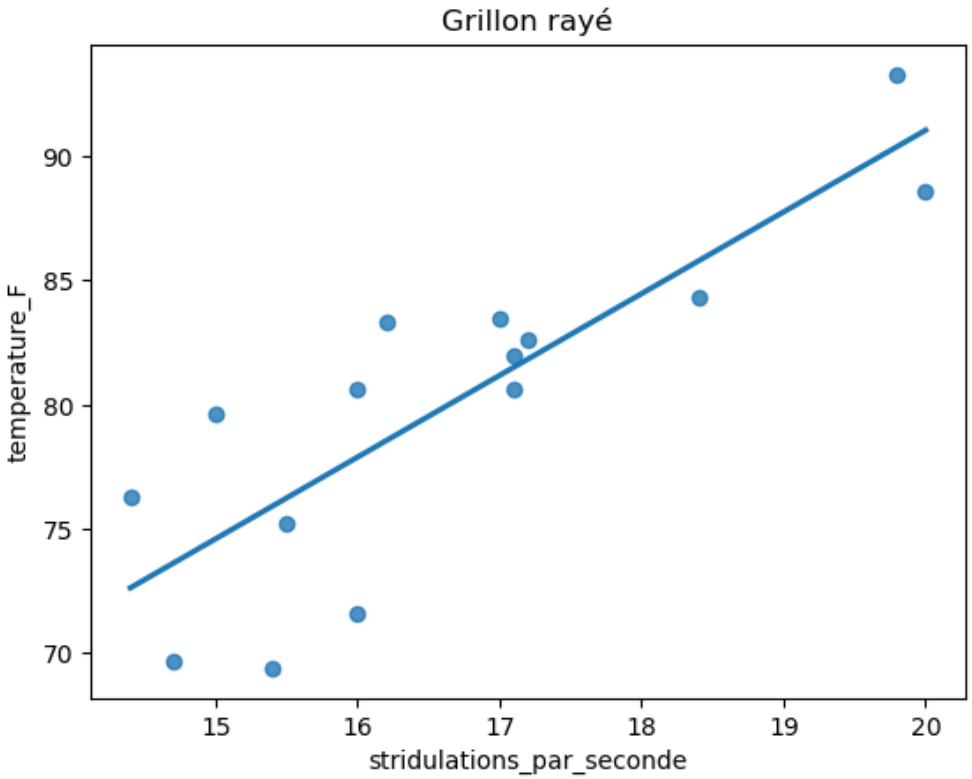
Seaborn sait calculer et tracer la droite directement, avec `regplot` :

```
import seaborn as sns
sns.regplot(x='...', y='...', data=df)
```

Pour consulter la documentation d'une fonction, on utilise le point d'interrogation : `?sns.regplot`.

```
import seaborn as sns

sns.regplot(x='stridulations_par_seconde', y='temperature_F', data=grillons, ci=None)
plt.title('Grillon rayé'); plt.show()
```



Rappel : la droite de régression passe par le point moyen du nuage. Vérifions-le : l'écart doit être nul, à l'erreur d'arrondi près.

```
x_moy = x_points.mean()
y_moy = y_points.mean()

y_moy - f(x_moy)
```

```
-7.105427357601002e-14
```

Rappel : la qualité de l'ajustement se mesure par le coefficient de corrélation linéaire r, compris entre -1 et 1. Plus $|r|$ est proche de 1, plus les points sont alignés.

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

```
r = ( ((x_points - x_moy) * (y_points - y_moy)).sum()
      / np.sqrt(((x_points - x_moy)**2).sum())
      / np.sqrt(((y_points - y_moy)**2).sum()) )

display(Markdown(f"r  = {r:.3f}"))
display(Markdown(f"r² = {r**2:.3f}"))

# pandas sait le faire directement :
# x_points.corr(y_points)
```

r = 0.835

r^2 = 0.697

$r \approx 0{,}84$: la relation est forte et positive. Environ **70 %** de la variation de température est expliquée par la fréquence des stridulations. Le grillon est effectivement un thermomètre acceptable.

19.6. Bonus : et la cigale ?

```
cigales = pd.read_csv('cigales.csv', sep=';', decimal=',')

xc = cigales['stridulations_par_minute']
yc = cigales['temperature_F']

ac, bc = np.polyfit(xc, yc, 1)
rc = xc.corr(yc)

display(Markdown(f"droite de régression : y = {ac:.3f} x + {bc:.2f}"))
display(Markdown(f"r  = {rc:.3f}"))
display(Markdown(f"r² = {rc**2:.3f}"))
```

droite de régression : y = -0.154 x + 89.96

r = -0.207

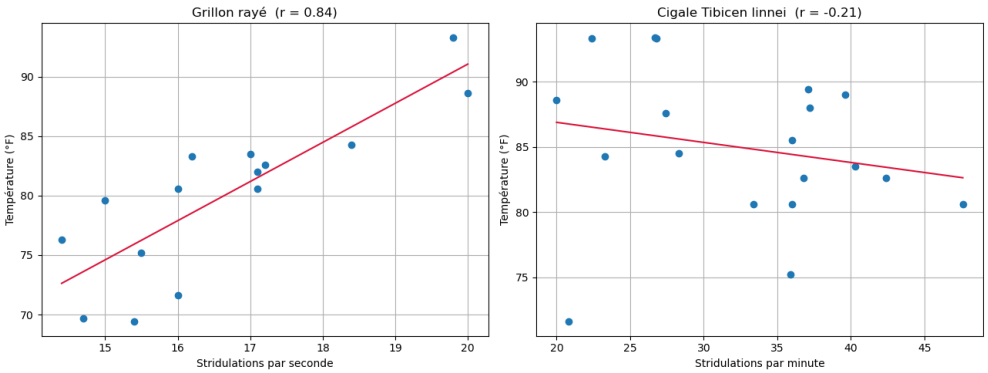
r^2 = 0.043

```
fig, (g1, g2) = plt.subplots(1, 2, figsize=(13, 5))

g1.plot(x_points, y_points, 'o')
g1.plot(xx, f(xx), color='crimson')
g1.set(title=f"Grillon rayé  (r = {r:.2f})",
       xlabel='Stridulations par seconde', ylabel='Température (°F)')
g1.grid()

xxc = np.linspace(xc.min(), xc.max(), 101)
g2.plot(xc, yc, 'o')
g2.plot(xxc, ac*xxc + bc, color='crimson')
g2.set(title=f"Cigale Tibicen linnei  (r = {rc:.2f})",
       xlabel='Stridulations par minute', ylabel='Température (°F)')
g2.grid()

plt.tight_layout(); plt.show()
```



La droite existe toujours ; la relation, non.

`polyfit` a renvoyé une droite dans les deux cas, sans le moindre avertissement. Mais les deux situations n'ont rien à voir :

	r	r^2	pente
Grillon rayé	0,84	0,70	+3,29
Cigale *Tibicen linnei*	−0,21	0,04	−0,15

Pour la cigale, $r^2 \approx 0{,}04$: la fréquence des stridulations n'explique que **4 %** de la variation de température. Pire, la pente est **négative** : le modèle prétend qu'il fait d'autant plus froid que la cigale stridule vite, ce qui n'a aucun sens biologique. Le nuage de droite est d'ailleurs sans structure visible.

La leçon vaut pour tout votre travail d'analyse : **on peut toujours ajuster une droite sur un nuage de points.** Regarder le nuage et le coefficient r avant de conclure n'est pas une formalité, c'est ce qui distingue une analyse d'un calcul.

Chapitre 20.

Notes et moyennes pondérées en L3 Mathématiques

20.1. Exercice

On s'intéresse aux résultats d'un étudiant (ou de deux étudiants) en L3 Mathématiques, pour les deux semestres de l'année universitaire.

On vous fournit les ECUE, leurs coefficients, et les notes (aléatoires) de chaque étudiant.

20.1.1. Semestre 1

- Créér une **Series** contenant les coefficients des ECUE. Les indices doivent être les noms des ECUE.
- Crééer une **Series** contenant les notes des ECUE (entiers aléatoires entre 0 et 20), avec les mêmes indices que pour les coefficients.
- Construire un **DataFrame** à partir de ces deux Series.

Exemple attendu :

	Coefficients	Étudiant 1
M51 Intégration	8	11
M52 Topologie	8	15
M53 Calcul différentiel	8	16
M54 Algèbre 5	4	7
M55 DS&SC 5	2	11
RDOC	0	3
PPE	0	20

Bonus : ajouter un deuxième étudiant.

20.1.2. Semestre 2

Faire la même chose pour les ECUE du semestre 2.

Exemple attendu :

	Coefficients	Étudiant 1
M61 EDO	5	6
M62 Analyse Complexe	9	12
M63 Modélisation	4	9
M64 DS&SC 6	4	6
UE65	6	6
UE66	2	18

Bonus : ajouter un deuxième étudiant.

20.1.3. Calculs à effectuer à partir des deux DataFrames

- Calculer la **moyenne pondérée** du semestre 1 :

$$\bar{x} = \frac{\sum_i \text{note}_i \times \text{coeff}_i}{\sum_i \text{coeff}_i}$$

- Calculer la **moyenne pondérée** du semestre 2
- Fusionner les deux DataFrames (par concaténation) puis calculer la moyenne pondérée globale. Exemple attendu :

	Coefficients	Étudiant 1
M51 Intégration	8	11
M52 Topologie	8	15
M53 Calcul différentiel	8	16
M54 Algèbre 5	4	7
M55 DS&SC 5	2	11
RDOC	0	3
PPE	0	20
M61 EDO	5	6
M62 Analyse Complexe	9	12
M63 Modélisation	4	9
M64 DS&SC 6	4	6
UE65	6	6
UE66	2	18

- Pour chaque étudiant, déterminer l'ECUE où il a eu sa pire note et l'ECUE où il a eu sa meilleure note.

```
import pandas as pd
import numpy as np

# Graine fixée : les notes sont tirées au hasard, mais on veut que le
# notebook donne le même résultat à chaque exécution.
np.random.seed(0)

from IPython.display import display, Markdown
```

```
# --- Semestre 1 : définition des ECUE, coefficients, et notes des étudiants ---

# Liste des ECUE du semestre 1
ecue_names_semester1 = [
    "M51 Intégration",
    "M52 Topologie",
    "M53 Calcul différentiel",
    "M54 Algèbre 5",
    "M55 DS&SC 5",
    "RDOC",
    "PPE"
]

nb_ecue_semester1 = len(ecue_names_semester1)

# Coefficients des ECUE (indexés par les noms des ECUE)
```

```
coefficients_semester1 = pd.Series(
    [8, 8, 8, 4, 2, 0, 0],
    index=ecue_names_semester1
)

# Notes aléatoires pour deux étudiants
etu_1_semester1 = pd.Series(
    np.random.randint(0, 21, size=nb_ecue_semester1),
    index=ecue_names_semester1
)

etu_2_semester1 = pd.Series(
    np.random.randint(0, 21, size=nb_ecue_semester1),
    index=ecue_names_semester1
)

# Construction du DataFrame
df_semester1 = pd.DataFrame({
    "Coefficients": coefficients_semester1,
    "Étudiant 1": etu_1_semester1,
    "Étudiant 2": etu_2_semester1
})

display(df_semester1)
```

	Coefficients	Étudiant 1	Étudiant 2
M51 Intégration	8	12	19
M52 Topologie	8	15	18
M53 Calcul différentiel	8	0	4
M54 Algèbre 5	4	3	6
M55 DS&SC 5	2	3	12
RDOC	0	7	1
PPE	0	9	6

```
# --- Semestre 2 : définition des ECUE, coefficients, et notes des étudiants ---

# Liste des ECUE du semestre 2
ecue_names_semester2 = [
    "M61 EDO",
    "M62 Analyse Complexe",
    "M63 Modélisation",
    "M64 DS&SC 6",
    "UE65",
    "UE66"
]

nb_ecue_semester2 = len(ecue_names_semester2)

# Coefficients des ECUE (indexés par les noms des ECUE)
coefficients_semester2 = pd.Series(
    [5, 9, 4, 4, 6, 2],
    index=ecue_names_semester2
)

# Notes aléatoires pour deux étudiants
etu_1_semester2 = pd.Series(
    np.random.randint(0, 21, size=nb_ecue_semester2),
    index=ecue_names_semester2
)
```

```
etu_2_semester2 = pd.Series(
    np.random.randint(0, 21, size=nb_ecue_semester2),
    index=ecue_names_semester2
)

# Construction du DataFrame
df_semester2 = pd.DataFrame({
    "Coefficients": coefficients_semester2,
    "Étudiant 1": etu_1_semester2,
    "Étudiant 2": etu_2_semester2
})

display(df_semester2)
```

	Coefficients	Étudiant 1	Étudiant 2
M61 EDO	5	7	9
M62 Analyse Complexe	9	14	20
M63 Modélisation	4	17	19
M64 DS&SC 6	4	5	16
UE65	6	13	19
UE66	2	8	5

```
# Calcul de la moyenne pondérée pour un étudiant donné au semestre 1

# df_semester1["Étudiant 1"].mean()
# Non pertinent : cette moyenne n'est pas pondérée,
# donc elle donne le même poids à un ECUE coefficient 8 qu'à un ECUE coefficient 0.
```

```
# Fonction qui pour un DataFrame donné et un étudiant donné
# calcule la moyenne pondérée

def weighted_average(df, student_name):
    notes = df[student_name]
    coeffs = df["Coefficients"]

    # --- Somme pondérée ---
    return (notes * coeffs).sum() / coeffs.sum()

    # --- Alternatives  ---
    # return notes.dot(coeffs) / coeffs.sum()
    # return np.average(notes, weights=coeffs)
```

```
# Moyenne pondérée pour chaque étudiant au semestre 1
print("Semestre 1")
for etu in ["Étudiant 1", "Étudiant 2"]:
    print(f"{etu} : {weighted_average(df_semester1, etu):.2f}")
```

```
Semestre 1
Étudiant 1 : 7.80
Étudiant 2 : 12.53
```

```
# Moyenne pondérée pour chaque étudiant au semestre 2
print("Semestre 2")
for etu in ["Étudiant 1", "Étudiant 2"]:
    print(f"{etu} : {weighted_average(df_semester2, etu):.2f}")
```

```
Semestre 2
Étudiant 1 : 11.43
Étudiant 2 : 16.30
```

Notes et coefficients pour l'année entière :

```
# ========================================
# Fusion des semestres pour l'année entière
# ========================================

# Option 1 : concaténation simple (les lignes sont juste empilées)
df_year = pd.concat([df_semester1, df_semester2])
display(df_year)

# Option 2 : concaténation avec MultiIndex pour identifier le semestre
# df_year = pd.concat([df_semester1, df_semester2],
#                     keys=["Semestre 1", "Semestre 2"], axis=0)
# display(df_year)

# Avantages :
# - Option 1 : plus simple pour calculer directement les moyennes pondérées
# - Option 2 : permet de conserver l'information du semestre dans l'index
```

	Coefficients	Étudiant 1	Étudiant 2
M51 Intégration	8	12	19
M52 Topologie	8	15	18
M53 Calcul différentiel	8	0	4
M54 Algèbre 5	4	3	6
M55 DS&SC 5	2	3	12
RDOC	0	7	1
PPE	0	9	6
M61 EDO	5	7	9
M62 Analyse Complexe	9	14	20
M63 Modélisation	4	17	19
M64 DS&SC 6	4	5	16
UE65	6	13	19
UE66	2	8	5

```
# Calculs de la pire et meilleure note pour chaque étudiant au semestre 1

for etu in ["Étudiant 1", "Étudiant 2"]:
    min_idx = df_semester1[etu].idxmin()
    max_idx = df_semester1[etu].idxmax()
    print(f"{etu} :")
    print(f"  Pire note au semestre 1 : {df_semester1.loc[min_idx, etu]} à {min_idx}")
    print(f"  Meilleure note au semestre 1 : {df_semester1.loc[max_idx, etu]} à {max_idx}")
    print()
```

```
Étudiant 1 :
  Pire note au semestre 1 : 0 à M53 Calcul différentiel
  Meilleure note au semestre 1 : 15 à M52 Topologie

Étudiant 2 :
  Pire note au semestre 1 : 1 à RDOC
  Meilleure note au semestre 1 : 19 à M51 Intégration
```

```
# ==============================
# DataFrame récapitulatif annuel
# ==============================

students = ["Étudiant 1", "Étudiant 2"]

df_summary = pd.DataFrame({
    etu: {
        "Pire note": df_year[etu].min(),
        "ECUE pire note": df_year[etu].idxmin(),
        "Meilleure note": df_year[etu].max(),
        "ECUE meilleure note": df_year[etu].idxmax(),
        "Moyenne annuelle pondérée": weighted_average(df_year, etu),
        "Moyenne semestre 1": weighted_average(df_semester1, etu),
        "Moyenne semestre 2": weighted_average(df_semester2, etu)
    }
    for etu in students
})

display(df_summary)
```

	Étudiant 1	Étudiant 2
Pire note	0	1
ECUE pire note	M53 Calcul différentiel	RDOC
Meilleure note	17	20
ECUE meilleure note	M63 Modélisation	M62 Analyse Complexe
Moyenne annuelle pondérée	9.616667	14.416667
Moyenne semestre 1	7.8	12.533333
Moyenne semestre 2	11.433333	16.3

Chapitre 21.

Ventes et pluie : agréger, c'est choisir

Les chapitres précédents ont présenté la **mécanique** de l'agrégation : `groupby`, `pivot_table`, `crosstab`. Cet exercice porte sur autre chose : **la lecture du résultat**.

Vous allez constater, sur deux jeux de données différents, qu'une même question posée à des données identiques reçoit des réponses **opposées** selon la grandeur que l'on choisit d'agréger. Agréger n'est pas une opération neutre : c'est déjà une décision d'analyse.

21.1. Exercice

21.1.1. Partie A : Ventes de deux magasins

Le fichier `ventes.csv` donne, pour deux magasins et quatre trimestres, les quantités vendues de trois produits et leur prix unitaire.

1. Charger le fichier et **vérifier le type de chaque colonne** avant toute chose.
2. Ajouter une colonne `montant`, égale à la quantité multipliée par le prix unitaire.
3. Quel magasin réalise le plus gros **chiffre d'affaires** ?
4. Croiser magasin et produit : quel produit porte le chiffre d'affaires de chaque magasin ?
5. Tracer l'évolution du chiffre d'affaires par trimestre pour chaque magasin.
6. **Quel magasin vend le plus d'articles ?** Comparer avec la réponse de la question 3 et commenter.

21.1.2. Partie B : Pluviométrie de trois villes

Le fichier `pluviometrie.csv` donne, pour Brest, Paris et Nice, la hauteur de pluie et le nombre de jours de pluie de chaque mois.

1. Charger le fichier.
2. Classer les trois villes par **cumul annuel de pluie**.
3. Classer les trois villes par **nombre de jours de pluie**.
4. Comparer les deux classements. Que peut-on en conclure ?
5. Calculer la hauteur moyenne de pluie **par jour de pluie** dans chaque ville.
6. Construire un tableau croisé station $\times$ mois. Attention à l'ordre des mois.
7. Pour chaque ville, déterminer le mois le plus arrosé.

Sur les données. Les ventes sont fictives. Pour la pluviométrie, les **totaux annuels** sont proches des normales publiées par Météo-France (Brest 1200 mm, Nice 800 mm, Paris 640 mm), mais la **répartition mensuelle a été reconstruite** pour les besoins de l'exercice : ne citez pas ces chiffres comme des relevés officiels.

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

sns.set_theme(context="notebook", style="whitegrid", palette="pastel")
```

```
plt.rcParams['figure.figsize'] = (12, 5)

from IPython.display import display, Markdown
```

Chapitre 22.

Partie A : Ventes de deux magasins

22.1. 1. Charger le fichier et vérifier les types

Premier réflexe, avant même de lire les données : regarder à quoi ressemble le fichier. Ici le séparateur est `;` et le marqueur décimal est la virgule.

```
display(Markdown("```\n" + open('ventes.csv', encoding='utf-8').read()[:220] + "\n```"))
```

```
magasin;trimestre;produit;quantite;prix_unitaire
Nord;T1;stylo;400;1,50
Nord;T1;cahier;120;3,20
Nord;T1;classeur;30;8,90
Nord;T2;stylo;420;1,50
Nord;T2;cahier;130;3,20
Nord;T2;classeur;25;8,90
Nord;T3;stylo;380;1,50
Nord
```

```
ventes = pd.read_csv('ventes.csv', sep=';', decimal=',')

display(ventes.head())
display(ventes.dtypes)
```

	magasin	trimestre	produit	quantite	prix_unitaire
0	Nord	T1	stylo	400	1.5
1	Nord	T1	cahier	120	3.2
2	Nord	T1	classeur	30	8.9
3	Nord	T2	stylo	420	1.5
4	Nord	T2	cahier	130	3.2

```
magasin           object
trimestre         object
produit           object
quantite           int64
prix_unitaire    float64
dtype: object
```

`prix_unitaire` est bien en `float64`. Si nous avions oublié `decimal=','`, la colonne aurait été lue comme du texte et toute la suite aurait été fausse : c'est la vérification à ne jamais sauter.

22.2. 2. Ajouter la colonne `montant`

```
ventes['montant'] = ventes['quantite'] * ventes['prix_unitaire']
ventes.head()
```

	magasin	trimestre	produit	quantite	prix_unitaire	montant
0	Nord	T1	stylo	400	1.5	600.0
1	Nord	T1	cahier	120	3.2	384.0
2	Nord	T1	classeur	30	8.9	267.0
3	Nord	T2	stylo	420	1.5	630.0
4	Nord	T2	cahier	130	3.2	416.0

22.3. 3. Quel magasin réalise le plus gros chiffre d'affaires ?

```
ca = ventes.groupby('magasin')['montant'].sum()
display(ca)
display(Markdown(f"Le magasin {ca.idxmax()} réalise le plus gros chiffre d'affaires : {ca.max():.2f} €"))
```

```
magasin
Nord    5264.0
Sud     8920.5
Name: montant, dtype: float64
```

Le magasin Sud réalise le plus gros chiffre d'affaires : 8920.50 €

22.4. 4. Quel produit porte chaque magasin ?

```
tableau = ventes.pivot_table(index='magasin', columns='produit',
                             values='montant', aggfunc='sum', margins=True)
tableau.round(2)
```

produit magasin	cahier	classeur	stylo	All
Nord	1632.0	1157.0	2475.0	5264.0
Sud	1360.0	6630.5	930.0	8920.5
All	2992.0	7787.5	3405.0	14184.5

Plus lisible en pourcentage de chaque ligne :

```
part = ventes.pivot_table(index='magasin', columns='produit', values='montant', aggfunc='sum')
(part.div(part.sum(axis=1), axis=0) * 100).round(1)
```

produit magasin	cahier	classeur	stylo
Nord	31.0	22.0	47.0
Sud	15.2	74.3	10.4

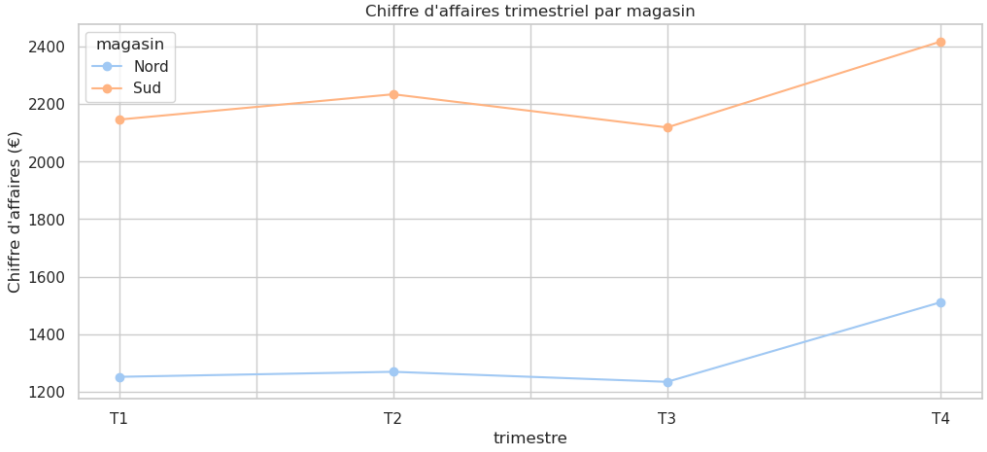
La lecture est immédiate : le Sud réalise l'essentiel de son chiffre d'affaires sur les **classeurs**, le Nord est beaucoup plus équilibré, avec une part importante de **stylos**.

22.5. 5. Évolution trimestrielle

```
evolution = ventes.pivot_table(index='trimestre', columns='magasin',
                               values='montant', aggfunc='sum')
display(evolution.round(2))

evolution.plot(marker='o')
plt.ylabel("Chiffre d'affaires (€)")
plt.title("Chiffre d'affaires trimestriel par magasin")
plt.show()
```

magasin trimestre	Nord	Sud
T1	1251.0	2147.0
T2	1268.5	2235.0
T3	1233.5	2119.5
T4	1511.0	2419.0



22.6. 6. Et en nombre d'articles ?

Jusqu'ici nous avons agrégé les **euros**. Agrégeons maintenant les **articles** :

```
bilan = ventes.groupby('magasin').agg(
    articles          = ('quantite', 'sum'),
    chiffre_affaires  = ('montant',  'sum')
)
bilan['panier_moyen'] = (bilan['chiffre_affaires'] / bilan['articles']).round(2)
bilan
```

magasin	articles	chiffre_affaires	panier_moyen
Nord	2290	5264.0	2.30
Sud	1790	8920.5	4.98

```
display(Markdown(f"Plus d'articles vendus     : {bilan['articles'].idxmax()}"))
display(Markdown(f"Plus gros chiffre d'affaires : {bilan['chiffre_affaires'].idxmax()}"))
```

Plus d'articles vendus : Nord

Plus gros chiffre d'affaires : Sud

Les deux classements s'opposent. Le magasin **Nord** vend 2 290 articles contre 1 790 pour le Sud, mais le Sud réalise un chiffre d'affaires de 8 920,50 € contre 5 264 € pour le Nord.

Il n'y a aucune contradiction : le Nord écoule des stylos à 1,50 €, le Sud des classeurs à 8,90 €. La dernière colonne le résume : le prix moyen d'un article vendu est près de deux fois plus élevé au Sud.

La question « quel est le meilleur magasin ? » n'a donc **pas de réponse** tant qu'on n'a pas dit ce qu'on mesure. C'est à l'analyste de le préciser, pas à `groupby`.

Chapitre 23.

Partie B : Pluviométrie de trois villes

23.1. 1. Charger le fichier

```
pluie = pd.read_csv('pluviometrie.csv', sep=';')

display(pluie.head())
display(pluie.shape)
```

	station	mois	pluie_mm	jours_pluie
0	Brest	janvier	150	17
1	Brest	février	110	14
2	Brest	mars	100	14
3	Brest	avril	85	13
4	Brest	mai	75	12

```
(36, 4)
```

23.2. 2. et 3. Deux classements

```
bilan_villes = pluie.groupby('station').agg(
    cumul_mm = ('pluie_mm',    'sum'),
    jours    = ('jours_pluie', 'sum')
)
bilan_villes
```

station	cumul_mm	jours
Brest	1225	162
Nice	797	63
Paris	655	111

```
display(Markdown("**Classement par hauteur de pluie :** "
                 + " > ".join(bilan_villes['cumul_mm'].sort_values(ascending=False).index)))
display(Markdown("**Classement par jours de pluie :** "
                 + " > ".join(bilan_villes['jours'].sort_values(ascending=False).index)))
```

Classement par hauteur de pluie : Brest > Nice > Paris

Classement par jours de pluie : Brest > Paris > Nice

23.3. 4. Comparaison des deux classements

Paris et Nice échangent leur place.

- En **hauteur de pluie**, il tombe nettement plus d'eau à Nice qu'à Paris : environ 800 mm contre 655 mm.
- En **nombre de jours**, c'est l'inverse : il pleut à Paris environ 111 jours par an, contre 63 à Nice.

« Il pleut plus à Paris qu'à Nice » est donc **vrai et faux à la fois**, selon la grandeur choisie. Les deux affirmations décrivent la même réalité : à Nice il pleut rarement, mais fort ; à Paris il pleut souvent, mais peu.

Brest, elle, est première dans les deux classements : il n'y a pas toujours d'ambiguïté, mais il faut toujours vérifier.

23.4. 5. L'intensité des pluies

Le rapport des deux grandeurs donne directement la hauteur moyenne tombée un jour de pluie :

```
bilan_villes['mm_par_jour_de_pluie'] = (bilan_villes['cumul_mm'] / bilan_villes['jours']).round(1)
bilan_villes.sort_values('mm_par_jour_de_pluie', ascending=False)
```

station	cumul_mm	jours	mm_par_jour_de_pluie
Nice	797	63	12.7
Brest	1225	162	7.6
Paris	655	111	5.9

Une pluie niçoise apporte en moyenne **plus du double** d'une pluie parisienne. C'est cette troisième grandeur, construite à partir des deux premières, qui réconcilie les classements et explique le phénomène.

23.5. 6. Tableau croisé station × mois

Première tentative, et premier piège :

```
pluie.pivot_table(index='station', columns='mois', values='pluie_mm')
```

station	août	avril	décembre	février	janvier	juillet	juin	mai	mars	novembre	octobre	septembre
Brest	55.0	85.0	165.0	110.0	150.0	50.0	55.0	75.0	100.0	155.0	140.0	85.0
Nice	20.0	70.0	95.0	45.0	70.0	12.0	35.0	45.0	40.0	140.0	150.0	75.0
Paris	60.0	52.0	58.0	45.0	50.0	60.0	55.0	65.0	48.0	52.0	62.0	48.0

Les mois sont rangés par ordre **alphabétique** : août, avril, décembre… Pandas ne peut pas deviner qu'il s'agit d'un calendrier. Il faut le lui dire, en déclarant la colonne comme une variable **catégorielle ordonnée** :

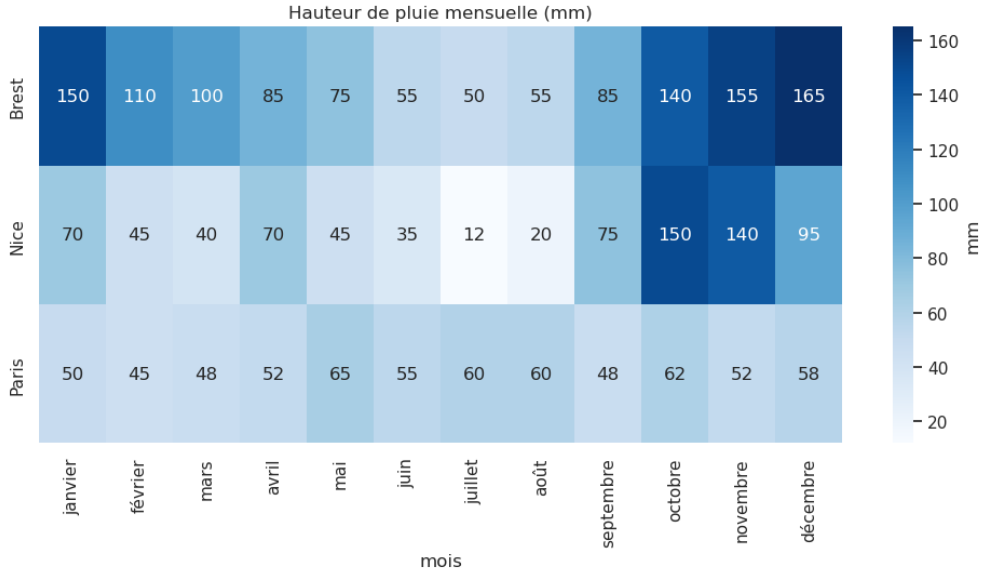
```
calendrier = ["janvier","février","mars","avril","mai","juin",
              "juillet","août","septembre","octobre","novembre","décembre"]

pluie['mois'] = pd.Categorical(pluie['mois'], categories=calendrier, ordered=True)

croise = pluie.pivot_table(index='station', columns='mois', values='pluie_mm', observed=True)
croise
```

mois station	janvier	février	mars	avril	mai	juin	juillet	août	septembre	octobre	novembre	décembre
Brest	150.0	110.0	100.0	85.0	75.0	55.0	50.0	55.0	85.0	140.0	155.0	165.0
Nice	70.0	45.0	40.0	70.0	45.0	35.0	12.0	20.0	75.0	150.0	140.0	95.0
Paris	50.0	45.0	48.0	52.0	65.0	55.0	60.0	60.0	48.0	62.0	52.0	58.0

```
sns.heatmap(croise, annot=True, fmt='.0f', cmap='Blues', cbar_kws={'label': 'mm'})
plt.title("Hauteur de pluie mensuelle (mm)")
plt.ylabel("")
plt.show()
```



La carte de chaleur rend les trois régimes immédiatement visibles : Brest arrosée toute l'année avec un creux estival, Paris remarquablement régulière, et Nice avec un été quasi sec suivi d'un automne très violent.

23.6. 7. Le mois le plus arrosé de chaque ville

```
croise.idxmax(axis=1).to_frame('mois le plus arrosé')
```

station	mois le plus arrosé
Brest	décembre
Nice	octobre
Paris	mai

Chapitre 24.

Conclusion

Ce que cet exercice voulait montrer

Dans les deux cas, le code était trivial : un `groupby` et une somme. C'est l'**interprétation** qui demandait du travail.

- Partie A : « quel est le meilleur magasin ? » dépend de ce qu'on somme, les euros ou les articles.
- Partie B : « où pleut-il le plus ? » dépend de ce qu'on somme, les millimètres ou les jours.

Retenez le réflexe : avant d'agréger, demandez-vous **quelle question vous posez réellement**, et après avoir agrégé, demandez-vous si **une autre grandeur donnerait une autre réponse**. C'est précisément ce qu'on attend de vous à l'examen : non pas produire un tableau, mais dire ce qu'il signifie.

Chapitre 25.

Population des communes de France

Le fichier `population.csv` contient les séries historiques de population (1876 à 2021) par Commune de France hors Mayotte. Il a été mis en ligne en décembre 2023 (source : Insee, recensements de la population).

Les colonnes correspondent à :

- le code géographique,
- la région,
- le département,
- le libellé géographique
- la population de 1876 à 2021.

Remarque : les treize régions métropolitaines (dont la Corse) et les cinq régions ultramarines ont les codes INSE suivants :

- 84 Auvergne-Rhône-Alpes
- 27 Bourgogne-Franche-Comté
- 53 Bretagne
- 24 Centre-Val de Loire
- 94 Corse
- 44 Grand Est
- 32 Hauts-de-France
- 11 Île-de-France
- 28 Normandie
- 75 Nouvelle-Aquitaine
- 76 Occitanie
- 52 Pays de la Loire
- 93 Provence-Alpes-Côte d'Azur
- 01 Guadeloupe
- 02 Martinique
- 03 Guyane
- 04 La Réunion
- 06 Mayotte

Chapitre 26.

Exercice

1. Afficher les données
   - Lire le fichier `population.csv`.
   - Afficher les 5 premières lignes.
   - Afficher les 5 dernières lignes.
   - Afficher les noms des colonnes.
   - Les régions et les départements ne sont désignés que par un code. À l'aide du fichier `20230823-communes-departement-region.csv`, construire par une **jointure** la correspondance code → nom. Attention au format des codes de département : vérifier le nombre de lignes et le nombre de valeurs manquantes après la jointure.
   - Compter le nombre de communes.
   - Afficher les types des colonnes.
2. Nettoyer les colonnes et les données
   - La colonne `CODGEO` ne nous intéresse pas. Supprimer cette colonne.
   - Renommer les étiquettes des colonnes correspondant aux années pour les rendre des entiers (par exemple, `"PMUN2020"` devient `2020`).
   - Nettoyer les valeurs car elles sont de type `str` et contiennent des espaces. Par exemple, `" 1 000 "` devient `1000`.
3. Aggreger/Filtrer les données
   - Afficher les lignes correspondant à la région Provence-Alpes-Côte d'Azur. Combien de communes sont présentes dans cette région ?
   - Afficher les lignes correspondant au département du Var. Combien de communes sont présentes dans ce département ?
   - Choisir une commune et afficher les lignes correspondant à cette commune.
4. Visualiser l'évolution de la population
   - Afficher sur un graphique l'évolution de la population de la commune de 1876 à 2021.
   - Afficher sur un graphique l'évolution de la population de la France metropolitaine de 1876 à 2021.

```
import pandas as pd
import numpy as np

from matplotlib import pyplot as plt # pour les graphiques d'évolution de la population dans le temps

import plotly.express as px # pour les cartes choroplèthes
import requests # pour récupérer les fichiers GeoJSON en ligne pour les cartes


# Pour la génération automatique de graphiques interactifs dans le polycopié
import plotly.io as pio
import os
if os.getenv("QUARTO_OUTPUT_FORMAT") is not None:
    # Export Quarto HTML
    pio.renderers.default = "iframe_connected"
else:
    # Notebook interactif
    pio.renderers.default = "notebook_connected"

from IPython.display import display, Markdown
```

26.1. Afficher les données

```
# 1. Lire le fichier
df = pd.read_csv('base-pop-historiques-1876-2021.csv', sep=';', low_memory=False)
# sep=';' pour lire un csv avec un séparateur ';' au lieu de ','
```

```
#?pd.read_csv
```

Les cinque premières lignes du fichier `population.csv` :

```
# 2. Afficher 5 premières lignes
df.head()
```

	CODGEO	REG	DEP	LIBGEO	PMUN2021	PMUN2020	PMUN2019	PMUN2018	PM
0	55039	44	55	Beaumont-en-Verdunois	0	0	0	0	0
1	55050	44	55	Bezonvaux	0	0	0	0	0
2	55139	44	55	Cumières-le-Mort-Homme	0	0	0	0	0
3	55189	44	55	Fleury-devant-Douaumont	0	0	0	0	0
4	55239	44	55	Haumont-près-Samogneux	0	0	0	0	0

Les cinq dernières lignes du fichier `population.csv` :

```
# 3. Afficher 5 dernières lignes
df.tail()
```

	CODGEO	REG	DEP	LIBGEO	PMUN2021	PMUN2020	PMUN2019	PMUN2018	PMUN2017	P
34942	44109	52	44	Nantes	323 204	320 732	318 808	314 138	309 346	3
34943	06088	93	06	Nice	348 085	343 477	342 669	341 032	340 017	3
34944	31555	76	31	Toulouse	504 078	498 003	493 465	486 828	479 553	4
34945	69123	84	69	Lyon	522 250	522 228	522 969	518 635	516 092	5
34946	13055	93	13	Marseille	873 076	870 321	870 731	868 277	863 310	8

Les noms des colonnes du fichier `population.csv` :

```
# 4. Afficher les noms des colonnes
df.columns
```

```
Index(['CODGEO', 'REG', 'DEP', 'LIBGEO', 'PMUN2021', 'PMUN2020', 'PMUN2019',
       'PMUN2018', 'PMUN2017', 'PMUN2016', 'PMUN2015', 'PMUN2014', 'PMUN2013',
       'PMUN2012', 'PMUN2011', 'PMUN2010', 'PMUN2009', 'PMUN2008', 'PMUN2007',
       'PMUN2006', 'PSDC1999', 'PSDC1990', 'PSDC1982', 'PSDC1975', 'PSDC1968',
       'PSDC1962', 'PTOT1954', 'PTOT1936', 'PTOT1931', 'PTOT1926', 'PTOT1921',
       'PTOT1911', 'PTOT1906', 'PTOT1901', 'PTOT1896', 'PTOT1891', 'PTOT1886',
       'PTOT1881', 'PTOT1876'],
      dtype='object')
```

Combien de régions sont présentes dans le fichier `population.csv` ?

```
df["REG"].nunique()
```

```
17
```

Lesquelles ?

```
df["REG"].unique()
```

```
array([44, 84, 76, 93, 32, 27, 52, 94, 24, 28, 11, 75, 53,  3,  2,  1,  4])
```

26.1.1. D'un code à un nom : une jointure

Le fichier de l'INSEE ne contient que des **codes** : la région vaut `93`, le département vaut `'83'`. Pour produire des résultats lisibles, il nous faut les **noms** correspondants.

On pourrait recopier la correspondance à la main… mais c'est précisément le travail de `merge`. Nous disposons d'un second fichier, `20230823-communes-departement-region.csv`, qui donne pour chaque commune les codes **et** les noms de son département et de sa région.

```
# dtype={'code_departement': str} : sans cela, '04' serait lu comme l'entier 4
reference = pd.read_csv('20230823-communes-departement-region.csv',
                        low_memory = False,
                        dtype      = {'code_departement': str})

reference[['code_departement', 'nom_departement', 'code_region', 'nom_region']].head()
```

	code_departement	nom_departement	code_region	nom_region
0	1	Ain	84.0	Auvergne-Rhône-Alpes
1	1	Ain	84.0	Auvergne-Rhône-Alpes
2	1	Ain	84.0	Auvergne-Rhône-Alpes
3	1	Ain	84.0	Auvergne-Rhône-Alpes
4	1	Ain	84.0	Auvergne-Rhône-Alpes

Cette table a une ligne par **commune**, alors que nous voulons une ligne par **département**. On extrait les colonnes utiles, puis on supprime les doublons :

```
departements = ( reference[['code_departement', 'nom_departement', 'code_region', 'nom_region']]
                   .dropna()
                   .drop_duplicates() )

display(Markdown(f"{len(departements)} départements"))
departements.head()
```

101 départements

	code_departement	nom_departement	code_region	nom_region
0	1	Ain	84.0	Auvergne-Rhône-Alpes
457	2	Aisne	32.0	Hauts-de-France
1287	3	Allier	84.0	Auvergne-Rhône-Alpes
1608	4	Alpes-de-Haute-Provence	93.0	Provence-Alpes-Côte d'Azur
1852	5	Hautes-Alpes	93.0	Provence-Alpes-Côte d'Azur

26.1.1.1. Première tentative : la jointure qui ne joint rien

```
essai = df.merge(departements,
                 left_on  = 'DEP',
                 right_on = 'code_departement',
                 how      = 'left')

display(Markdown(f"communes sans département après la jointure : {essai['nom_departement'].isna().sum()}
↪  / {len(df)}"))
```

communes sans département après la jointure : 3140 / 34947

Plus de **3 000 communes** se retrouvent sans département : et pourtant la jointure n'a produit aucune erreur. C'est le piège classique : **les deux clés n'ont pas le même format.** Regardons-les :

```
display(Markdown(f"DEP, fichier INSEE        : `{sorted(df['DEP'].unique())[:6]}`"))
display(Markdown(f"code_departement, référence :
↪  `{sorted(departements['code_departement'].unique())[:6]}`"))
```

DEP, fichier INSEE : `['01', '02', '03', '04', '05', '06']`

code_departement, référence : `['1', '10', '11', '12', '13', '14']`

`'01'` d'un côté, `'1'` de l'autre : les départements 1 à 9 ne peuvent pas se correspondre. On complète donc les codes de la référence à deux chiffres avec `zfill(2)` : les codes d'outre-mer, qui en comptent déjà trois (`'971'`), ne sont pas affectés.

```
departements['code_departement'] = departements['code_departement'].str.zfill(2)
departements['code_region']      = departements['code_region'].astype(int)

departements.head()
```

	code_departement	nom_departement	code_region	nom_region
0	01	Ain	84	Auvergne-Rhône-Alpes
457	02	Aisne	32	Hauts-de-France
1287	03	Allier	84	Auvergne-Rhône-Alpes
1608	04	Alpes-de-Haute-Provence	93	Provence-Alpes-Côte d'Azur
1852	05	Hautes-Alpes	93	Provence-Alpes-Côte d'Azur

Nouvelle tentative, cette fois en **déclarant** la relation attendue avec `validate` : plusieurs communes pour un même département, donc `many_to_one`.

```
df_complet = df.merge(departements,
                      left_on  = 'DEP',
                      right_on = 'code_departement',
                      how      = 'left',
                      validate = 'many_to_one')

display(Markdown(f"lignes avant : {len(df)}"))
display(Markdown(f"lignes après : {len(df_complet)}"))
display(Markdown(f"communes sans département : {df_complet['nom_departement'].isna().sum()}"))

df_complet[['LIBGEO', 'DEP', 'nom_departement', 'REG', 'nom_region']].head()
```

lignes avant : 34947

lignes après : 34947

communes sans département : 0

	LIBGEO	DEP	nom_departement	REG	nom_region
0	Beaumont-en-Verdunois	55	Meuse	44	Grand Est
1	Bezonvaux	55	Meuse	44	Grand Est
2	Cumières-le-Mort-Homme	55	Meuse	44	Grand Est
3	Fleury-devant-Douaumont	55	Meuse	44	Grand Est
4	Haumont-près-Samogneux	55	Meuse	44	Grand Est

Même nombre de lignes, aucune commune orpheline : la jointure est correcte.

Les trois vérifications à faire après chaque `merge`, que nous venons d'appliquer :

1. le **nombre de lignes** a-t-il changé ?
2. combien de valeurs **manquantes** la jointure a-t-elle introduites ?
3. la **relation** entre les deux tables est-elle bien celle que je crois (`validate`) ?

26.1.1.2. Les tables de correspondance

Nous pouvons maintenant obtenir les dictionnaires code → nom **à partir des données**, sans rien recopier. La suite du notebook les utilise pour afficher des libellés lisibles.

```
# code de région -> nom de région
region_dict = dict(zip(departements['code_region'], departements['nom_region']))

# codes et noms des départements de la région PACA
paca     = departements[ departements['nom_region'] == "Provence-Alpes-Côte d'Azur" ]
dep_PACA = dict(zip(paca['code_departement'], paca['nom_departement']))

display(region_dict)
display(dep_PACA)
```

```
{84: 'Auvergne-Rhône-Alpes',
 32: 'Hauts-de-France',
 93: "Provence-Alpes-Côte d'Azur",
 44: 'Grand Est',
 76: 'Occitanie',
 28: 'Normandie',
 75: 'Nouvelle-Aquitaine',
 24: 'Centre-Val de Loire',
 27: 'Bourgogne-Franche-Comté',
 53: 'Bretagne',
 94: 'Corse',
 52: 'Pays de la Loire',
 11: 'Île-de-France',
 1: 'Guadeloupe',
 2: 'Martinique',
 3: 'Guyane',
 4: 'La Réunion',
 6: 'Mayotte'}
```

```
{'04': 'Alpes-de-Haute-Provence',
 '05': 'Hautes-Alpes',
 '06': 'Alpes-Maritimes',
 '13': 'Bouches-du-Rhône',
 '83': 'Var',
 '84': 'Vaucluse'}
```

```
# Afficher les noms des régions présentes dans le fichier
region_names = [ (code,region_dict[code]) for code in df["REG"].unique()]
sorted(region_names,key=lambda x: x[1])  # trier par nom
```

```
[(84, 'Auvergne-Rhône-Alpes'),
 (27, 'Bourgogne-Franche-Comté'),
 (53, 'Bretagne'),
 (24, 'Centre-Val de Loire'),
 (94, 'Corse'),
 (44, 'Grand Est'),
 (1, 'Guadeloupe'),
 (3, 'Guyane'),
 (32, 'Hauts-de-France'),
 (4, 'La Réunion'),
 (2, 'Martinique'),
 (28, 'Normandie'),
 (75, 'Nouvelle-Aquitaine'),
 (76, 'Occitanie'),
 (52, 'Pays de la Loire'),
 (93, "Provence-Alpes-Côte d'Azur"),
 (11, 'Île-de-France')]
```

Quelle région est absente du fichier `population.csv` ?

```
for cle, val in region_dict.items():
    if cle not in df["REG"].unique():
        print(f"La région absente est : {val} (code {cle})")
```

```
La région absente est : Mayotte (code 6)
```

Combien de communes ?

```
# 5. Nombre de communes
# df['LIBGEO'].count()
# ou
df.shape[0] # donne le nombre de lignes = de communes


# df['LIBGEO'].nunique()
# ne pas utiliser `nunique`` car certains nom des communes sont dupliqués
```

```
34947
```

Les types des colonnes du fichier `population.csv` :

```
df.dtypes
```

```
CODGEO      object
REG          int64
DEP         object
LIBGEO      object
PMUN2021    object
PMUN2020    object
PMUN2019    object
PMUN2018    object
```

```
PMUN2017    object
PMUN2016    object
PMUN2015    object
PMUN2014    object
PMUN2013    object
PMUN2012    object
PMUN2011    object
PMUN2010    object
PMUN2009    object
PMUN2008    object
PMUN2007    object
PMUN2006    object
PSDC1999    object
PSDC1990    object
PSDC1982    object
PSDC1975    object
PSDC1968    object
PSDC1962    object
PTOT1954    object
PTOT1936    object
PTOT1931    object
PTOT1926    object
PTOT1921    object
PTOT1911    object
PTOT1906    object
PTOT1901    object
PTOT1896    object
PTOT1891    object
PTOT1886    object
PTOT1881    object
PTOT1876    object
dtype: object
```

```
# RÉSUMÉ DES INFORMATIONS SUR LE DATAFRAME

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34947 entries, 0 to 34946
Data columns (total 39 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   CODGEO    34947 non-null  object
 1   REG       34947 non-null  int64
 2   DEP       34947 non-null  object
 3   LIBGEO    34947 non-null  object
 4   PMUN2021  34947 non-null  object
 5   PMUN2020  34947 non-null  object
 6   PMUN2019  34947 non-null  object
 7   PMUN2018  34947 non-null  object
 8   PMUN2017  34947 non-null  object
 9   PMUN2016  34947 non-null  object
 10  PMUN2015  34947 non-null  object
 11  PMUN2014  34947 non-null  object
 12  PMUN2013  34947 non-null  object
 13  PMUN2012  34947 non-null  object
 14  PMUN2011  34947 non-null  object
 15  PMUN2010  34947 non-null  object
```

```
 16  PMUN2009  34947 non-null  object
 17  PMUN2008  34947 non-null  object
 18  PMUN2007  34947 non-null  object
 19  PMUN2006  34947 non-null  object
 20  PSDC1999  34947 non-null  object
 21  PSDC1990  34947 non-null  object
 22  PSDC1982  34947 non-null  object
 23  PSDC1975  34947 non-null  object
 24  PSDC1968  34947 non-null  object
 25  PSDC1962  34947 non-null  object
 26  PTOT1954  34921 non-null  object
 27  PTOT1936  34835 non-null  object
 28  PTOT1931  34475 non-null  object
 29  PTOT1926  34475 non-null  object
 30  PTOT1921  34475 non-null  object
 31  PTOT1911  34475 non-null  object
 32  PTOT1906  34475 non-null  object
 33  PTOT1901  34475 non-null  object
 34  PTOT1896  34475 non-null  object
 35  PTOT1891  34475 non-null  object
 36  PTOT1886  34475 non-null  object
 37  PTOT1881  34475 non-null  object
 38  PTOT1876  34475 non-null  object
dtypes: int64(1), object(38)
memory usage: 10.4+ MB
```

Le DataFrame `population` est composé de 39 colonnes et 34947 lignes.
Toutes les colonnes sont de type `object` (chaîne de caractères) sauf la colonne `REG` qui est de type `int64`.

On remarque de plus qu'il y a des données manquantes dans les dernières colonnes (34475 vs 34947).

26.2. Nettoyer les colonnes et les données

26.2.1. Suppression de colonnes inutiles

La colonne `CODGEO` est inutile pour notre analyse. Nous allons la supprimer.

```
df.drop('CODGEO', axis=1, inplace=True, errors='ignore')
df.head(20)
```

	REG	DEP	LIBGEO	PMUN2021	PMUN2020	PMUN2019	PMUN2018	PMUN2017
0	44	55	Beaumont-en-Verdunois	0	0	0	0	0
1	44	55	Bezonvaux	0	0	0	0	0
2	44	55	Cumières-le-Mort-Homme	0	0	0	0	0
3	44	55	Fleury-devant-Douaumont	0	0	0	0	0
4	44	55	Haumont-près-Samogneux	0	0	0	0	0
5	44	55	Louvemont-Côte-du-Poivre	0	0	0	0	0
6	84	26	Rochefourchat	1	1	1	1	1
7	44	54	Leménil-Mitry	2	3	3	3	3
8	84	26	La Bâtie-des-Fonds	2	2	2	3	4
9	44	51	Rouvroy-Ripont	3	3	4	6	8
10	76	31	Caubous	4	4	4	4	4
11	84	26	Pommerol	5	5	6	6	11
12	76	11	Fontanès-de-Sault	5	5	5	5	5
13	93	04	Majastres	5	4	4	4	4

	REG	DEP	LIBGEO	PMUN2021	PMUN2020	PMUN2019	PMUN2018	PMUN2017	PMUN
14	44	52	Charmes-en-l'Angle	6	6	7	9	10	10
15	44	88	Maroncourt	6	6	7	7	9	11
16	32	80	Épécamps	6	6	5	5	5	5
17	27	39	Mérona	7	7	7	8	8	9
18	44	55	Ornes	7	7	6	6	5	6
19	44	57	Molring	7	4	5	5	5	7

26.2.2. Modification des labels des colonnes

Nous allons renommer les étiquettes des colonnes de la troisième jusqu'à la dernière `df.columns[3:]` pour ne garder que les derniers quatre caractères `col[-4:]` (de sort à avoir encore des string mais qui pourrons être transformées en entiers au besoin).

```
# Renommer les colonnes des années
# ================================

# { old_name_of_col : new_name_of_col    for col in df.columns[3:] }
dico_old_new = { col:col[-4:] for col in df.columns[3:] }
df.rename(columns=dico_old_new, inplace=True)
df
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...
0	44	55	Beaumont-en-Verdunois	0	0	0	0	0	0	0	...
1	44	55	Bezonvaux	0	0	0	0	0	0	0	...
2	44	55	Cumières-le-Mort-Homme	0	0	0	0	0	0	0	...
3	44	55	Fleury-devant-Douaumont	0	0	0	0	0	0	0	...
4	44	55	Haumont-près-Samogneux	0	0	0	0	0	0	0	...
...	...	...	...	...	...	...	...	...	...	...	...
34942	52	44	Nantes	323 204	320 732	318 808	314 138	309 346	306 694	303 382	...
34943	93	06	Nice	348 085	343 477	342 669	341 032	340 017	342 637	342 522	...
34944	76	31	Toulouse	504 078	498 003	493 465	486 828	479 553	475 438	471 941	...
34945	84	69	Lyon	522 250	522 228	522 969	518 635	516 092	515 695	513 275	...
34946	93	13	Marseille	873 076	870 321	870 731	868 277	863 310	862 211	861 635	...

26.2.3. Modification des valeurs (nombre d'habitants) : nettoyage et typecasting

Maintenant, nous allons modifier les valeurs corespondant aux nombres d'habitants pour les rendre des entiers (par exemple, `" 1 000 "` devient `1000`).

En effet, les valeurs correspondant à la population sont des chaînes de caractères qui ne peuvent pas être converties en entiers directement. Il faut remplacer le caractère `\xa0` (qui correspond à l'espace insécable) par `''`, puis convertir les chaînes de caractères ainsi nettoyées en entiers.

```
# Ancienne méthode
# df[df.columns[4:]] = df[df.columns[3:]].applymap(lambda x: int(x.replace('\xa0', '')) if isinstance(x,
 ↪  str) else x)
# df

# Méthode avec la fonction clean_value

def clean_value(val):
    try:
        if isinstance(val, str):
            val = val.replace('\xa0', '').strip()
```

```
            if val.isdigit():
                return int(val)
        return val
    except Exception as e:
        return val

# Apply clean_value function to each column individually using map
for col in df.columns[3:]:
    df[col] = df[col].map(clean_value).astype('Int64')

# Int64 est un type Pandas qui permet d'avoir des entiers avec
# des valeurs manquantes (NaN)

df
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	..
0	44	55	Beaumont-en-Verdunois	0	0	0	0	0	0	0	..
1	44	55	Bezonvaux	0	0	0	0	0	0	0	..
2	44	55	Cumières-le-Mort-Homme	0	0	0	0	0	0	0	..
3	44	55	Fleury-devant-Douaumont	0	0	0	0	0	0	0	..
4	44	55	Haumont-près-Samogneux	0	0	0	0	0	0	0	..
...	...	...	...	...	...	...	...	...	...	...	..
34942	52	44	Nantes	323204	320732	318808	314138	309346	306694	303382	..
34943	93	06	Nice	348085	343477	342669	341032	340017	342637	342522	..
34944	76	31	Toulouse	504078	498003	493465	486828	479553	475438	471941	..
34945	84	69	Lyon	522250	522228	522969	518635	516092	515695	513275	..
34946	93	13	Marseille	873076	870321	870731	868277	863310	862211	861635	..

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34947 entries, 0 to 34946
Data columns (total 38 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   REG     34947 non-null  int64
 1   DEP     34947 non-null  object
 2   LIBGEO  34947 non-null  object
 3   2021    34947 non-null  Int64
 4   2020    34947 non-null  Int64
 5   2019    34947 non-null  Int64
 6   2018    34947 non-null  Int64
 7   2017    34947 non-null  Int64
 8   2016    34947 non-null  Int64
 9   2015    34947 non-null  Int64
 10  2014    34947 non-null  Int64
 11  2013    34947 non-null  Int64
 12  2012    34947 non-null  Int64
 13  2011    34947 non-null  Int64
 14  2010    34947 non-null  Int64
 15  2009    34947 non-null  Int64
 16  2008    34947 non-null  Int64
 17  2007    34947 non-null  Int64
 18  2006    34947 non-null  Int64
 19  1999    34947 non-null  Int64
```

```
 20  1990    34947 non-null  Int64
 21  1982    34947 non-null  Int64
 22  1975    34947 non-null  Int64
 23  1968    34947 non-null  Int64
 24  1962    34947 non-null  Int64
 25  1954    34921 non-null  Int64
 26  1936    34835 non-null  Int64
 27  1931    34475 non-null  Int64
 28  1926    34475 non-null  Int64
 29  1921    34475 non-null  Int64
 30  1911    34475 non-null  Int64
 31  1906    34475 non-null  Int64
 32  1901    34475 non-null  Int64
 33  1896    34475 non-null  Int64
 34  1891    34475 non-null  Int64
 35  1886    34475 non-null  Int64
 36  1881    34475 non-null  Int64
 37  1876    34475 non-null  Int64
dtypes: Int64(35), int64(1), object(2)
memory usage: 11.3+ MB
```

26.3. Aggregations

26.3.1. Combien de départements par region ?

```
df.pivot_table(index='REG', values='DEP', aggfunc='nunique')
```

REG	DEP
1	1
2	1
3	1
4	1
11	8
24	6
27	8
28	5
32	5
44	10
52	5
53	4
75	12
76	13
84	12
93	6
94	2

```
# ou avec groupby
# ===================================

df_grby = df.groupby("REG")["DEP"]
nb_dep_par_reg = df_grby.nunique()
# C'est une série:
# - les index sont les codes régionaux,
```

```
# - les valeurs sont le nombre de départements

# On peut alors l'utiliser pour afficher des informations supplémentaires
for reg, count in nb_dep_par_reg.items():
    dep_uniques = df_grby.get_group(reg).unique()  # les départements uniques pour cette région
    print(f"La région {reg} ({region_dict[reg]}) a {count} départements : {dep_uniques}")
```

```
La région 1 (Guadeloupe) a 1 départements : ['971']
La région 2 (Martinique) a 1 départements : ['972']
La région 3 (Guyane) a 1 départements : ['973']
La région 4 (La Réunion) a 1 départements : ['974']
La région 11 (Île-de-France) a 8 départements : ['77' '95' '78' '91' '92' '94' '93' '75']
La région 24 (Centre-Val de Loire) a 6 départements : ['18' '28' '45' '41' '36' '37']
La région 27 (Bourgogne-Franche-Comté) a 8 départements : ['39' '25' '58' '21' '70' '71' '89' '90']
La région 28 (Normandie) a 5 départements : ['50' '76' '61' '14' '27']
La région 32 (Hauts-de-France) a 5 départements : ['80' '62' '60' '02' '59']
La région 44 (Grand Est) a 10 départements : ['55' '54' '51' '52' '88' '57' '08' '10' '67' '68']
La région 52 (Pays de la Loire) a 5 départements : ['72' '53' '49' '85' '44']
La région 53 (Bretagne) a 4 départements : ['22' '35' '56' '29']
La région 75 (Nouvelle-Aquitaine) a 12 départements : ['23' '24' '19' '16' '47' '33' '40' '64' '87' '
La région 76 (Occitanie) a 13 départements : ['31' '11' '09' '65' '66' '34' '30' '32' '12' '81' '48'
La région 84 (Auvergne-Rhône-Alpes) a 12 départements : ['26' '38' '63' '15' '43' '01' '73' '03' '42'
La région 93 (Provence-Alpes-Côte d'Azur) a 6 départements : ['04' '05' '83' '84' '06' '13']
La région 94 (Corse) a 2 départements : ['2B' '2A']
```

26.3.2. Combien de communes par région ?

Attention, avec `nunique()`, on compte le nombre de valeurs uniques, or certains noms de communes sont dupliqués. Il faut donc utiliser une autre méthode comme `pivot_table()` avec `aggfunc='count'`.

```
df_communes_by_region = df.pivot_table(index='REG', values='LIBGEO', aggfunc='count')
df_communes_by_region

# df.groupby('REG')[2021].count()
# df[['REG','DEP','LIBGEO']].pivot_table(index='REG', values='LIBGEO', aggfunc='count')
```

REG	LIBGEO
1	32
2	34
3	22
4	24
11	1287
24	1757
27	3699
28	2651
32	3787
44	5119
52	1233
53	1207
75	4308
76	4453
84	4028
93	946
94	360

Bonus : connexion Internet requise. Les cartes choroplèthes de cette section téléchargent le fond de carte (fichier *GeoJSON* des régions françaises) depuis un dépôt en ligne. Sans accès à Internet, ces cellules échouent : tout le reste de l'exercice fonctionne sans elles.

Cette partie n'est pas au programme et ne sera pas évaluée : elle est là pour vous montrer ce que Pandas permet une fois les données agrégées.

26.3.2.1. Bonus : visualisation des données sur une carte choroplèthe

On peut visualiser le nombre de communes par région sur une carte de France. Cette visualisation nécéssite de récupérer les coordonnées géographiques des régions et de les associer au DataFrame `df_communes_by_region`. Pour cela, nous pouvons utiliser une base de données géographiques telle que GeoJSON. Ensuite, nous pouvons utiliser la bibliothèque Plotly pour créer une **carte choroplèthe** en utilisant les coordonnées géographiques et les données de population par région. Voici un exemple de code pour réaliser cette visualisation :

```
# Préparer les données pour la carte choroplèth
# ==============================================
# Il faut réinitialiser l'index pour que 'REG' devienne une colonne normale
df_communes_by_region = df_communes_by_region.reset_index()
# Renommer les colonnes
df_communes_by_region.columns = ['REG', 'Nombre_communes']
# Ajouter les noms des régions
df_communes_by_region['Nom_Region'] = df_communes_by_region['REG'].map(region_dict)
df_communes_by_region
```

	REG	Nombre_communes	Nom_Region
0	1	32	Guadeloupe
1	2	34	Martinique
2	3	22	Guyane
3	4	24	La Réunion
4	11	1287	Île-de-France
5	24	1757	Centre-Val de Loire
6	27	3699	Bourgogne-Franche-Comté
7	28	2651	Normandie
8	32	3787	Hauts-de-France
9	44	5119	Grand Est
10	52	1233	Pays de la Loire
11	53	1207	Bretagne
12	75	4308	Nouvelle-Aquitaine
13	76	4453	Occitanie
14	84	4028	Auvergne-Rhône-Alpes
15	93	946	Provence-Alpes-Côte d'Azur
16	94	360	Corse

```
# Récupérer les données GeoJSON
geo_url =
 ↪  "https://raw.githubusercontent.com/gregoiredavid/france-geojson/master/regions-version-simplifiee.geojson"
france_regions_geo = requests.get(geo_url).json()
```

```
# Pour comprendre la structure du GeoJSON
# =======================================================

# import json

# print("=== Structure du GeoJSON ===")
# print(f"Type: {type(france_regions_geo)}")
# print(f"Clés principales: {france_regions_geo.keys()}")
```

```
# print(f"\nNombre de régions: {len(france_regions_geo['features'])}")

# print("\n=== Exemple d'une région (première feature) ===")
# first_feature = france_regions_geo['features'][0]
# print(f"Clés de la feature: {first_feature.keys()}")
# print(f"\nPropriétés disponibles:")
# print(json.dumps(first_feature['properties'], indent=2, ensure_ascii=False))

# print("\n=== TOUTES les régions avec leurs propriétés ===")
# for i, feature in enumerate(france_regions_geo['features']):
#     props = feature['properties']
#     print(f"\nRégion {i+1}: {json.dumps(props, indent=2, ensure_ascii=False)}")
```

```
# Créer la carte choroplèth avec plotly
# =======================================

# Appel utilisant les codes des régions
fig = px.choropleth(
    df_communes_by_region,      # DataFrame avec les données
    geojson=france_regions_geo, # Données géographiques au format GeoJSON

    # Appel utilisant les codes des régions
    locations='REG',            # Colonne du DataFrame avec les codes des régions
    featureidkey='properties.code', # Clé du code dans le GeoJSON
    # Appel utilisant les noms des régions
    #locations='Nom_Region',        # Doit correspondre aux noms dans le GeoJSON
    #featureidkey='properties.nom', # Clé dans le GeoJSON pour faire correspondre

    color='Nombre_communes',    # Colonne pour la coloration
    color_continuous_scale='Blues', # Palette de couleurs
    labels={'Nombre_communes': 'Nombre de communes'},   # Étiquette pour la légende
    title='Nombre de communes par région en France',
    hover_data={'REG': True, 'Nom_Region': True, 'Nombre_communes': True}, # Infos au survol
    scope='europe', # Centrer la carte sur l'Europe (optionnel)
    # projection="mercator", # Projection cartographique, par défaut "equirectangular"
)

# Ajuster la vue sur la France et n'afficher que la France
fig.update_geos( fitbounds="locations", visible=False )
# fig.update_layout(width=800, height=600)
fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

26.3.3. Combien de communes par région et département ?

```
df_grbt_REG_DEP_LIBGEO =  df.groupby(['REG', 'DEP'])['LIBGEO'].count()
# On peut utiliser count car dans chaque département le nom des communes est unique

# Pour vérifier on affiché le nombre de communes dans la région PACA (93) selon le département
df_grbt_REG_DEP_LIBGEO[93]
```

```
DEP
04    198
05    162
```

```
06    163
13    119
83    153
84    151
Name: LIBGEO, dtype: int64
```

26.3.4. Combien de personnes par region en 2021 ?

```
df_sum_2021_by_region = df.pivot_table(index='REG', values='2021', aggfunc='sum', margins=True)
# df.groupby('REG')['2021'].sum()
df_sum_2021_by_region
```

REG	2021
1	384315.0
2	360749.0
3	286618.0
4	871157.0
11	12317279.0
24	2573303.0
27	2800194.0
28	3327966.0
32	5995292.0
44	5561287.0
52	3853999.0
53	3394567.0
75	6069352.0
76	6022176.0
84	8114361.0
93	5127840.0
94	347597.0
All	67408052.0

Quelle est la région la plus peuplée en 2021 ?

```
# margins=True ajoute une ligne "All" qu'on doit retirer avant de tracer
df_sum_2021_by_region = df_sum_2021_by_region.drop(index="All")

region_max_2021 = df_sum_2021_by_region.idxmax()
display(region_max_2021)              # indice = code de la région la plus peuplée en 2021
region_dict[region_max_2021.iloc[0]]  # nom de la région la plus peuplée en 2021
```

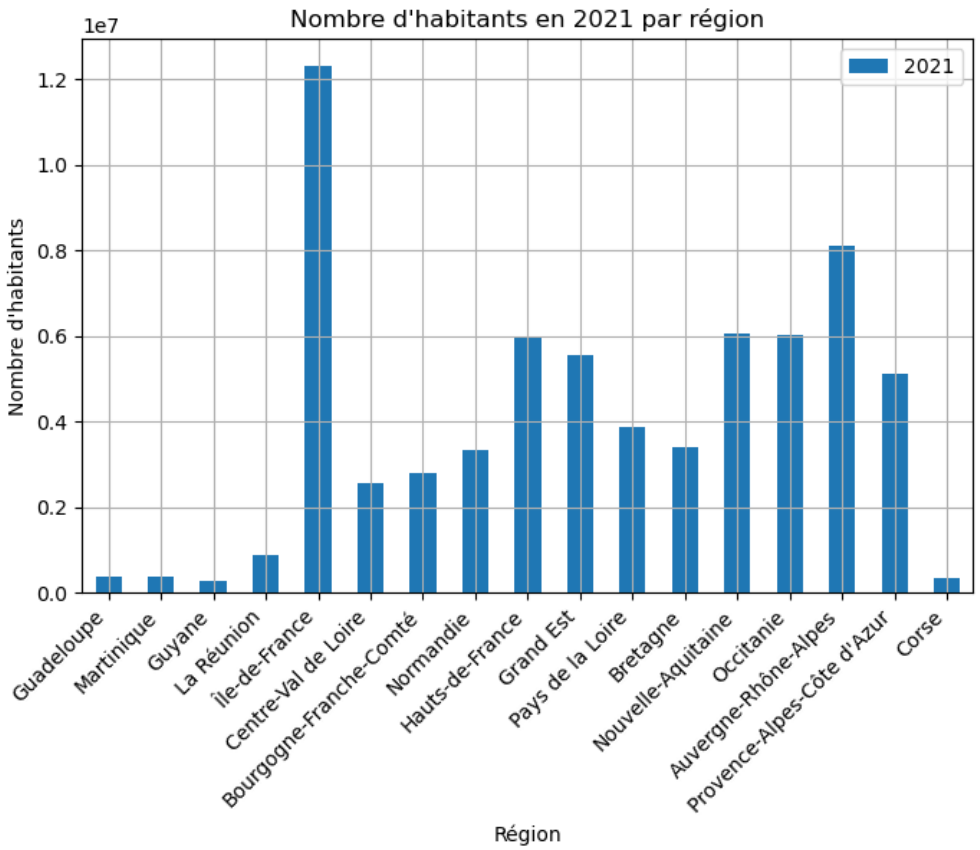
```
2021    11
dtype: object
```

```
'Île-de-France'
```

```
df_sum_2021_by_region.index = ( df_sum_2021_by_region.index.map(region_dict) )

df_sum_2021_by_region.plot(
    kind='bar',
    figsize=(8, 5)
)
plt.xticks(rotation=45, ha='right')
```

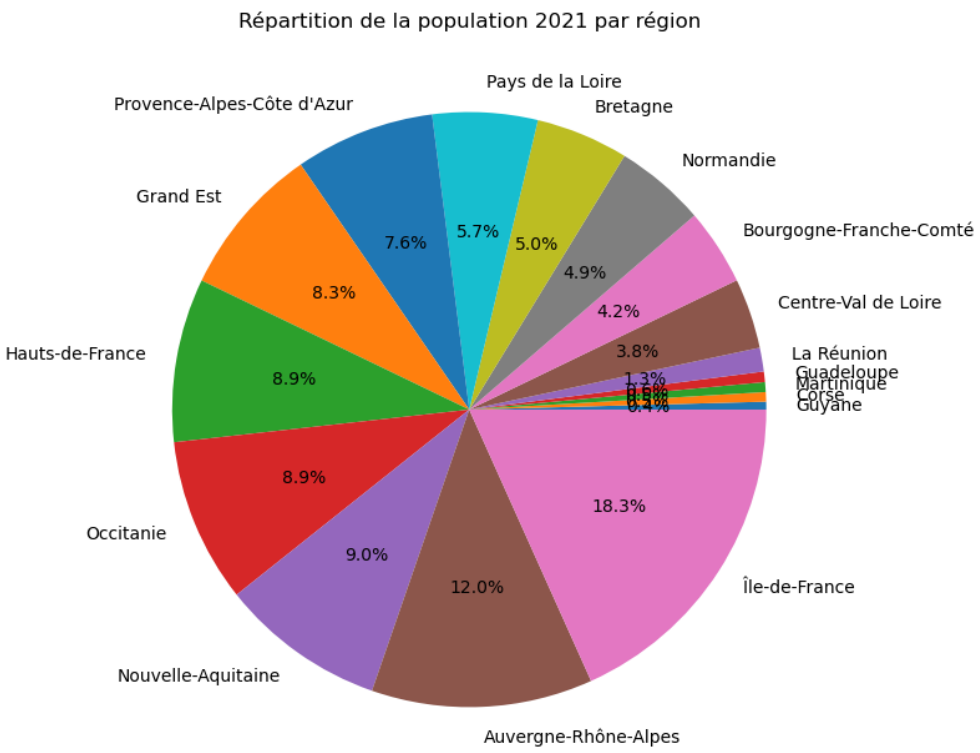
```
plt.grid()
plt.title("Nombre d'habitants en 2021 par région")
plt.xlabel("Région")
plt.ylabel("Nombre d'habitants")
plt.show()
```




```
# Tri en ordre croissant
df_sum_2021_by_region_plot = df_sum_2021_by_region.sort_values(by="2021", ascending=True)


# Pie chart
df_sum_2021_by_region_plot.plot.pie(
    y='2021',
    autopct='%1.1f%%',
    figsize=(8, 8),
    legend=False
)

plt.ylabel("")  # enlève le label y
plt.title("Répartition de la population 2021 par région")
plt.tight_layout()
plt.show()
```



Bonus : afficher la population totale par région dans une carte choroplèthe.

```
# Préparer les données de population par région
df_sum_2021_by_region_reset = df_sum_2021_by_region_plot.reset_index()
df_sum_2021_by_region_reset.columns = ['REG', 'Population']

# Ajouter les noms des régions
df_sum_2021_by_region_reset['Nom_Region'] = df_sum_2021_by_region_reset['REG'].map(region_dict)

# Créer la carte choroplèth avec plotly
fig = px.choropleth(
    df_sum_2021_by_region_reset,
    geojson=france_regions_geo,
    locations='REG', # doit correspondre à "properties.nom" dans le GeoJSON
    featureidkey='properties.nom',
    color='Population',
    color_continuous_scale='YlOrRd',
    labels={'Population': 'Nombre d\'habitants'},
    title='Population par région en France (2021)',
    hover_data={'Nom_Region': True, 'Population': ':,'},
    projection="mercator",
)

# Ajuster la vue sur la France et n'afficher que la France
fig.update_geos( fitbounds="locations", visible=False )
# fig.update_layout(height=600, width=800)
fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

26.4. Étude de la région PACA

Dans la suite nous considérérons les données de la région Provence-Alpes-Côte d'Azur que nous sauvons dans un DataFrame `df_PACA`.

```
# Méthode 1
# mask = (df['REG'] == 93)
# df_PACA = df[mask]

# Méthode 2
df_grbyREG = df.groupby('REG')
df_PACA = df_grbyREG.get_group(name=93)
df_IdF = df_grbyREG.get_group(name=11)

# Affichage
df_PACA
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...
13	93	04	Majastres	5	4	4	4	4	4	3	...
21	93	05	La Haute-Beaume	8	8	8	8	9	9	9	...
25	93	83	Vérignon	9	9	9	9	10	10	10	...
26	93	04	Saint-Martin-lès-Seyne	10	12	13	13	14	13	13	...
34	93	05	Nossage-et-Bénévent	12	14	15	16	15	15	14	...
...	...	...	...	...	...	...	...	...	...	...	...
34890	93	84	Avignon	90330	90597	91143	91729	91921	92378	92130	...
34919	93	13	Aix-en-Provence	147478	147122	145133	143097	142482	143006	142668	...
34932	93	83	Toulon	180452	179659	178745	176198	171953	169634	167479	...
34943	93	06	Nice	348085	343477	342669	341032	340017	342637	342522	...
34946	93	13	Marseille	873076	870321	870731	868277	863310	862211	861635	...

26.4.1. Combien de départements dans la région PACA ?

```
df_PACA["DEP"].nunique()
```

```
6
```

```
for code_dep in df_PACA["DEP"].unique():
    print(code_dep, dep_PACA[code_dep])
```

```
04 Alpes-de-Haute-Provence
05 Hautes-Alpes
83 Var
84 Vaucluse
06 Alpes-Maritimes
13 Bouches-du-Rhône
```

26.4.2. Combien de communes dans la région PACA ?

```
df_PACA["DEP"].shape[0]
```

```
946
```

26.4.3. Combien de commune dans chaque département de la région PACA ?

```
df_PACA.pivot_table(index='DEP', values='LIBGEO', aggfunc='count', margins=True)
```

DEP	LIBGEO
04	198
05	162
06	163
13	119
83	153
84	151
All	946

Est-ce qu'il existe des communes avec le même nom dans la région Provence-Alpes-Côte d'Azur ?

```
df_PACA['LIBGEO'].count() - df_PACA['LIBGEO'].nunique()
```

```
12
```

Il y a bien 12 communes avec le même nom dans la région Provence-Alpes-Côte d'Azur (mais pas le même CODGEO). Lesquelles ?

```
duplicated_names = df_PACA['LIBGEO'].duplicated(keep=False) # c'est un masque, l'option keep=False
↪ signifie que chaque valeur de la colonne qui apparaît plus d'une fois sera marquée comme True. Les
↪ valeurs uniques (apparaissant une seule fois) seront marquées comme False.
communes_duplicated_sorted = df_PACA[duplicated_names].sort_values(by='LIBGEO')
communes_duplicated_sorted
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...	1926	1921	1911	190
27562	93	04	Aiglun	1413	1420	1432	1432	1440	1412	1384	...	174	187	218	220
3063	93	06	Aiglun	93	93	94	93	91	89	86	...	115	100	142	152
30188	93	06	Aspremont	2300	2271	2272	2230	2187	2144	2139	...	424	510	591	611
15215	93	05	Aspremont	366	369	369	363	358	350	341	...	300	330	376	396
19004	93	05	Châteauvieux	528	519	506	496	490	488	485	...	127	141	173	165
2072	93	83	Châteauvieux	75	77	79	81	83	87	85	...	66	66	76	80
15388	93	83	Esparron	371	370	371	364	356	349	345	...	265	267	284	310
1254	93	05	Esparron	58	58	57	56	51	45	40	...	112	107	143	128
34587	93	83	La Garde	25912	25563	25505	25380	25126	25236	25047	...	3048	2827	3037	296
4874	93	04	La Garde	124	121	118	104	89	75	77	...	119	116	122	156
17862	93	05	La Rochette	475	473	474	470	466	469	472	...	265	261	280	274
1915	93	04	La Rochette	72	74	75	72	70	67	66	...	168	190	241	258
32836	93	83	Le Castellet	5285	4578	3873	3887	3886	3875	3947	...	1324	1232	1314	124
13105	93	04	Le Castellet	301	298	299	296	293	289	292	...	177	181	198	186
27587	93	84	Mirabeau	1419	1389	1344	1324	1288	1253	1218	...	302	315	408	420
18559	93	04	Mirabeau	509	512	511	511	511	511	510	...	220	255	314	355
7468	93	05	Rousset	172	177	181	182	183	179	171	...	189	163	176	166
32814	93	13	Rousset	5209	4977	4918	4881	4844	4811	4768	...	705	862	713	683
32350	93	06	Saint-Jeannet	4303	4317	4246	4157	4128	4099	4071	...	857	834	981	939
798	93	04	Saint-Jeannet	47	48	48	50	54	59	63	...	116	112	162	183
3310	93	04	Sigoyer	98	101	103	104	107	105	104	...	120	128	150	151
22301	93	05	Sigoyer	730	727	708	688	665	664	664	...	472	467	526	555

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...	1926	1921	191
9567	93	05	Vitrolles	214	210	209	204	204	205	206	...	206	196	211
34709	93	13	Vitrolles	35532	34418	33333	33101	33310	33880	34089	...	812	794	819

26.4.4. Combien de personnes habitaient en PACA en 2021 ?

```
display( df_PACA['2021'].sum() )
```

```
5127840
```

26.4.5. Combien de personnes habitaient dans chaque département de la région PACA en 2021 ?

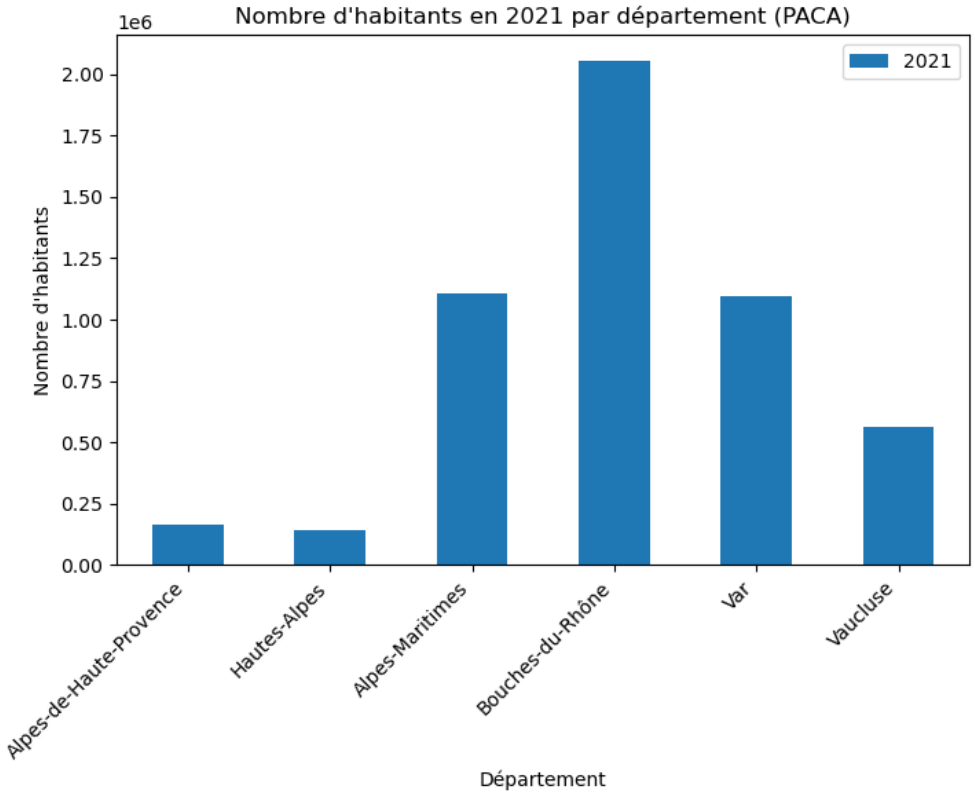
```
PACA_DEP_2021 = df_PACA.pivot_table(index='DEP', values='2021', aggfunc='sum', margins=True)
PACA_DEP_2021
```

DEP	2021
04	166077.0
05	140976.0
06	1103941.0
13	2056943.0
83	1095337.0
84	564566.0
All	5127840.0

```
# margins=True ajoute une ligne "All" qu'on doit retirer avant de tracer
PACA_DEP_2021_plot = PACA_DEP_2021.drop(index="All")

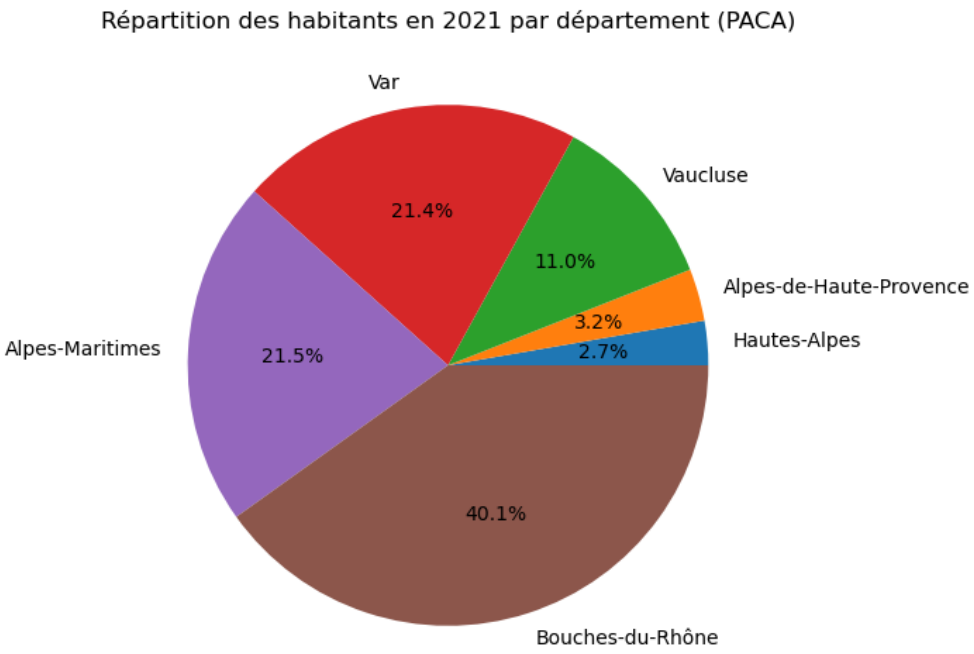
PACA_DEP_2021_plot.index = ( PACA_DEP_2021_plot.index.map(dep_PACA))


PACA_DEP_2021_plot.plot(
    kind='bar',
    figsize=(8, 5)
)
plt.title("Nombre d'habitants en 2021 par département (PACA)")
plt.xlabel("Département")
plt.ylabel("Nombre d'habitants")
plt.xticks(rotation=45, ha='right')
plt.show()
```




```
# Tri en ordre croissant
PACA_DEP_2021_plot = PACA_DEP_2021_plot.sort_values(by="2021", ascending=True)

PACA_DEP_2021_plot.plot.pie(
    y='2021',
    autopct='%1.1f%%',
    figsize=(6, 6),
    legend=False
)
plt.ylabel("")
plt.title("Répartition des habitants en 2021 par département (PACA)")
plt.show()
```



26.5. Étude du département VAR

Dans la suite nous allons considérer les données du département du Var que nous sauvons dans un DataFrame `df_83`.

```
# Methode 1
# df_83 = df[ df['DEP'] == '83' ]
# df_83 = df_PACA[ df_PACA['DEP'] == '83' ]

# Methode 2
df_83 = df_PACA.groupby('DEP').get_group(name='83')

# Affichage
df_83
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...	1926
25	93	83	Vérignon	9	9	9	9	10	10	10	...	60
119	93	83	Brenon	22	23	25	27	29	31	30	...	25
748	93	83	Le Bourguet	45	42	41	38	35	31	30	...	56
975	93	83	Riboux	51	51	49	48	46	44	42	...	21
2072	93	83	Châteauvieux	75	77	79	81	83	87	85	...	66
...	...	...	...	...	...	...	...	...	...	...	...	...
34739	93	83	Draguignan	39745	39434	39433	39106	39340	40053	40278	...	9441
34828	93	83	Hyères	55103	54615	54821	55069	55588	55772	56478	...	19816
34833	93	83	Fréjus	57082	55750	54458	53786	52672	53168	52897	...	9091
34847	93	83	La Seyne-sur-Mer	62763	62232	62987	62888	63936	64620	64903	...	23504
34932	93	83	Toulon	180452	179659	178745	176198	171953	169634	167479	...	11512

Combien de communes sont présentes dans le département du Var ?

```
# df_83['LIBGEO'].count()
df_83['LIBGEO'].shape[0]
```

```
153
```

Est-ce qu'il existe des communes avec le même nom dans le département du VAR ?

```
df_83['LIBGEO'].count() - df_83['LIBGEO'].nunique()
```

```
0
```

Pour choisir une commune, on va d'abord afficher toutes les communes du département du Var (pour vérifier l'orthographe) :

```
# première idée : on affiche toutes les communes mais elles sont trop nombreuses
sorted(df_83['LIBGEO'].tolist())
```

```
['Aiguines',
 'Ampus',
 'Artignosc-sur-Verdon',
 'Artigues',
 'Aups',
 'Bagnols-en-Forêt',
 'Bandol',
 'Bargemon',
 'Bargème',
 'Barjols',
 'Baudinard-sur-Verdon',
 'Bauduen',
 'Belgentier',
 'Besse-sur-Issole',
 'Bormes-les-Mimosas',
 'Bras',
 'Brenon',
 'Brignoles',
 'Brue-Auriac',
 'Cabasse',
 'Callas',
 'Callian',
 'Camps-la-Source',
 'Carcès',
 'Carnoules',
 'Carqueiranne',
 'Cavalaire-sur-Mer',
 'Châteaudouble',
 'Châteauvert',
 'Châteauvieux',
 'Claviers',
 'Cogolin',
 'Collobrières',
 'Comps-sur-Artuby',
 'Correns',
 'Cotignac',
 'Cuers',
 'Draguignan',
```

```
 'Entrecasteaux',
 'Esparron',
 'Fayence',
 'Figanières',
 'Flassans-sur-Issole',
 'Flayosc',
 'Forcalqueiret',
 'Fox-Amphoux',
 'Fréjus',
 'Garéoult',
 'Gassin',
 'Ginasservis',
 'Gonfaron',
 'Grimaud',
 'Hyères',
 'La Bastide',
"La Cadière-d'Azur",
 'La Celle',
 'La Crau',
 'La Croix-Valmer',
 'La Farlède',
 'La Garde',
 'La Garde-Freinet',
 'La Londe-les-Maures',
 'La Martre',
 'La Motte',
 'La Môle',
 'La Roque-Esclapon',
 'La Roquebrussanne',
 'La Seyne-sur-Mer',
 'La Valette-du-Var',
 'La Verdière',
 'Le Beausset',
 'Le Bourguet',
 'Le Cannet-des-Maures',
 'Le Castellet',
 'Le Lavandou',
 'Le Luc',
 'Le Muy',
 'Le Plan-de-la-Tour',
 'Le Pradet',
 'Le Revest-les-Eaux',
 'Le Thoronet',
 'Le Val',
"Les Adrets-de-l'Estérel",
 'Les Arcs',
 'Les Mayons',
 'Les Salles-sur-Verdon',
 'Lorgues',
 'Mazaugues',
 'Moissac-Bellevue',
 'Mons',
 'Montauroux',
 'Montferrat',
 'Montfort-sur-Argens',
 'Montmeyan',
 'Méounes-lès-Montrieux',
 'Nans-les-Pins',
 'Néoules',
```

```
 'Ollioules',
 'Ollières',
 'Pierrefeu-du-Var',
 'Pignans',
 "Plan-d'Aups-Sainte-Baume",
 'Pontevès',
 'Pourcieux',
 'Pourrières',
 'Puget-Ville',
 'Puget-sur-Argens',
 'Ramatuelle',
 'Rayol-Canadel-sur-Mer',
 'Rians',
 'Riboux',
 'Rocbaron',
 'Roquebrune-sur-Argens',
 'Rougiers',
 'Régusse',
 'Saint-Antonin-du-Var',
 'Saint-Cyr-sur-Mer',
 'Saint-Julien',
 'Saint-Mandrier-sur-Mer',
 'Saint-Martin-de-Pallières',
 'Saint-Maximin-la-Sainte-Baume',
 'Saint-Paul-en-Forêt',
 'Saint-Raphaël',
 'Saint-Tropez',
 'Saint-Zacharie',
 'Sainte-Anastasie-sur-Issole',
 'Sainte-Maxime',
 'Salernes',
 'Sanary-sur-Mer',
 'Seillans',
 "Seillons-Source-d'Argens",
 'Signes',
 'Sillans-la-Cascade',
 'Six-Fours-les-Plages',
 'Solliès-Pont',
 'Solliès-Toucas',
 'Solliès-Ville',
 'Tanneron',
 'Taradeau',
 'Tavernes',
 'Toulon',
 'Tourrettes',
 'Tourtour',
 'Tourves',
 'Trans-en-Provence',
 'Trigance',
 'Varages',
 'Vidauban',
 'Villecroze',
 'Vinon-sur-Verdon',
 'Vins-sur-Caramy',
 'Vérignon',
 'Évenos']
```

On cherche alors une commune en particulier avec un filtre sur le nom de la commune avec la méthode `str.contains()` :

```
commune_recherchee = 'Sol'
mask = df_83['LIBGEO'].str.contains(commune_recherchee, case=True, na=False)
# case = False pour ne pas tenir compte de la casse,
# na = False pour ne pas tenir compte des valeurs manquantes
df_83[ mask ]
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...	1926	1921	19
30591	93	83	Solliès-Ville	2529	2523	2526	2488	2450	2412	2391	...	479	376	48
33075	93	83	Solliès-Toucas	5912	5867	5753	5696	5719	5741	5756	...	799	777	88
34110	93	83	Solliès-Pont	12080	11974	11762	11496	11149	11056	10951	...	2666	2689	27

Choisissons une commune et affichons la ligne correspondant à cette commune :

```
Commune = 'Solliès-Toucas'
# Commune = 'Toulon'
mask = df_83['LIBGEO'] == Commune
df_Commune = df_83[mask]
df_Commune
```

	REG	DEP	LIBGEO	2021	2020	2019	2018	2017	2016	2015	...	1926	1921	1911	190
33075	93	83	Solliès-Toucas	5912	5867	5753	5696	5719	5741	5756	...	799	777	882	900

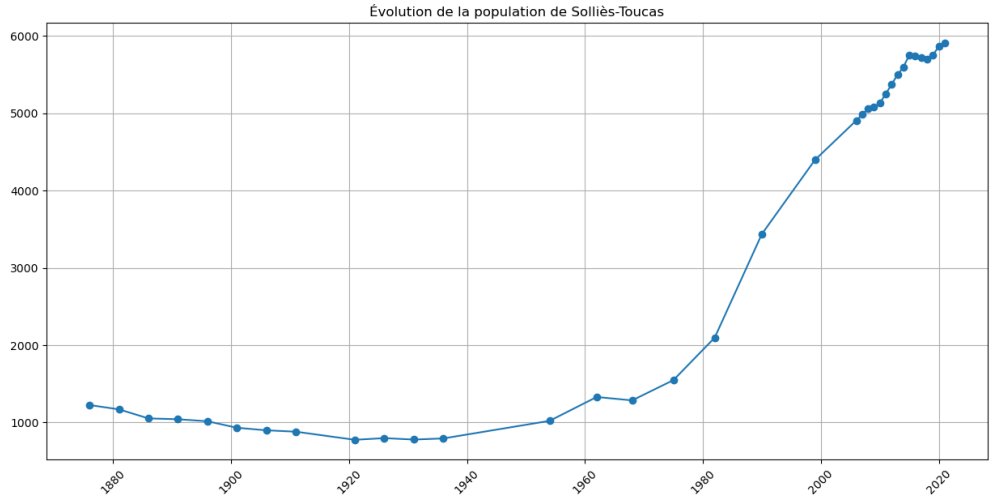
Nous allons afficher l'évolution de la population d'une commune de 1876 à 2021 :

```
# xx = [int(col[-4:]) for col in df_Commune.columns[3:]] # les années
# yy = [int(val.replace('\xa0', '')) for val in df_Commune.values[0][3:]]

# xx = df_Commune.columns[3:].map(int) # les années, on utilise map pour appliquer int à chaque label de
 ↪  colonne
xx = np.array([int(col) for col in df_Commune.columns[3:]])  # les années
yy = df_Commune.values[0][3:] # les valeurs de l'unique ligne correspondant à la commune
```

```
plt.figure(figsize=(15, 7))

plt.plot(xx, yy, 'o-')
plt.xticks(rotation=45)  # Incline les labels des colonnes de 45 degrés
plt.grid()
plt.title(f'Évolution de la population de {Commune}');
```



26.6. Affichages divers

Pour afficher l'évolution de la population de la France métropolitaine de 1876 à 2021, il faut sommer les colonnes.

```
# Copie du DataFrame sans la première ligne
df2 = df.iloc[1:].copy()

xx = df2.columns[3:].map(int)

# Nettoyage des valeurs en supprimant les caractères '\xa0' et conversion en entier dans la copie
#df2[df2.columns[3:]] = df2[df2.columns[3:]].applymap(lambda x: int(x.replace('\xa0', '')) if
 ↪  isinstance(x, str) else x)


# Calcul de la somme des valeurs dans chaque colonne de la copie
yy = df2[df2.columns[3:]].sum()
# Le résulta est une Series : la première colonne est l'indice (les années), la seconde les valeurs

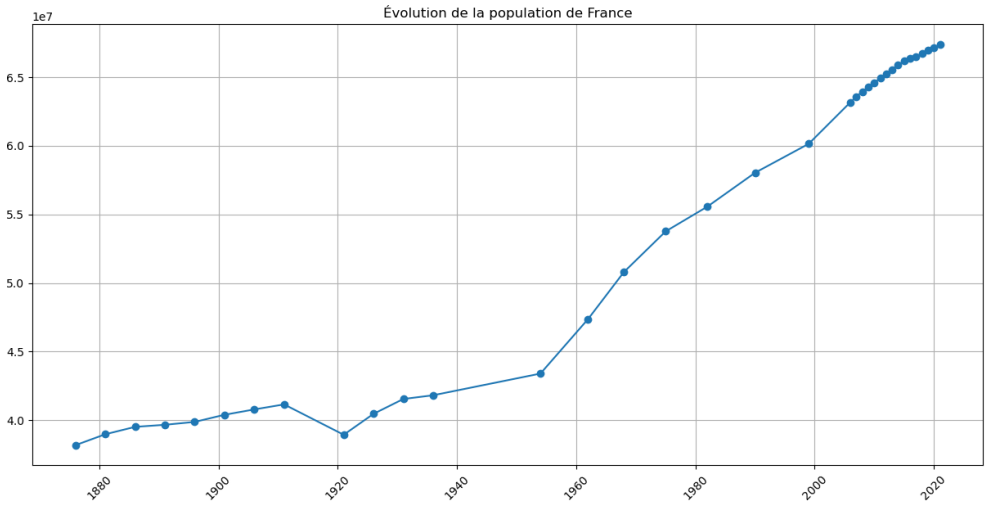
# Vérifions que la somme est correcte : selon l'INSEE, en 2021 en France la population totale était de
 ↪  67,76 millions d'habitants
print(f"En {yy.index[0]}, la population totale de France était de {yy.iloc[0]:e} habitants.")
```

```
En 2021, la population totale de France était de 6.740805e+07 habitants.
```

```
import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(15, 7))

plt.plot(xx,yy, 'o-')
plt.xticks(rotation=45)  # Incline les labels des colonnes de 45 degrés
plt.grid()
plt.title(f'Évolution de la population de France');
```



On affiche l'évolution de la population de la France métropolitaine de 1876 à 2021 par région :

```
import matplotlib.pyplot as plt

# Colonnes des années (de 2021 à 1876)
year_columns = df.columns[3:].map(int)

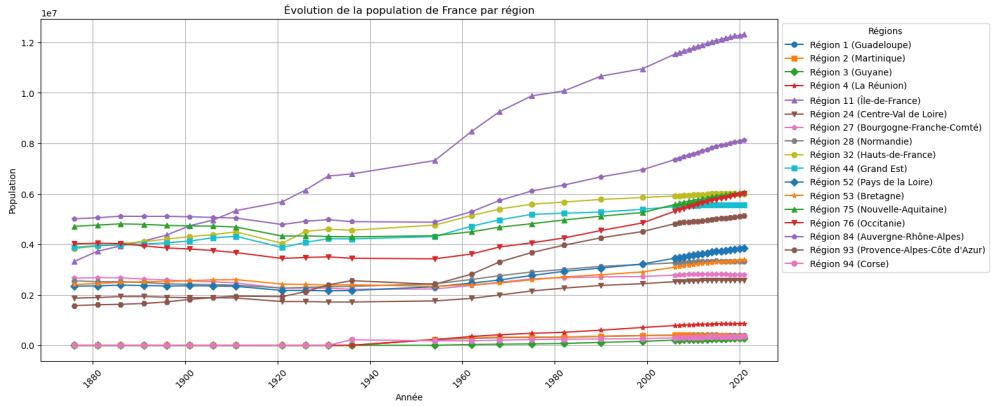
# Groupement des données par région
```

```
df_grbyREG = df.groupby('REG')

# Tracé de l'évolution
plt.figure(figsize=(15, 7))
# Liste des marqueurs
markers = ['o', 's', 'D', '*', '^', 'v', 'p', 'H']  # Ajouter plus si nécessaire
num_markers = len(markers)  # Nombre total de marqueurs

for i, reg in enumerate(df_grbyREG.groups.keys()):
    yy = df_grbyREG.get_group(reg).iloc[:, 3:].sum()
    marker = markers[i % num_markers]  # Sélectionner un marqueur cycliquement
    plt.plot(year_columns,yy, marker=marker, label=f"Région {reg} ({region_dict[reg]})")

# Personnalisation du graphique
plt.xticks(rotation=45)  # Incliner les années
plt.grid()
plt.title("Évolution de la population de France par région")
plt.xlabel("Année")
plt.ylabel("Population")
plt.legend(title="Régions", loc='upper left', bbox_to_anchor=(1, 1))
plt.show()
```



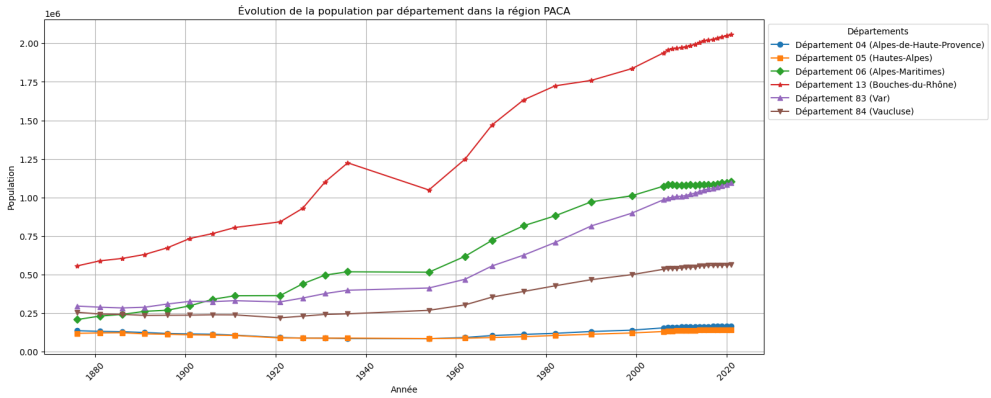

```
# Colonnes des années (de 2021 à 1876)
year_columns_PACA = df_PACA.columns[3:].map(int)

# Groupement des données par région
df_PACA_grbyDEP = df_PACA.groupby('DEP')

# Tracé de l'évolution
plt.figure(figsize=(15, 7))
# Liste des marqueurs
markers = ['o', 's', 'D', '*', '^', 'v', 'p', 'H']  # Ajouter plus si nécessaire
num_markers = len(markers)  # Nombre total de marqueurs

for i, dep in enumerate(df_PACA_grbyDEP.groups.keys()):
    yy = df_PACA_grbyDEP.get_group(dep).iloc[:, 3:].sum()
    marker = markers[i % num_markers]  # Sélectionner un marqueur cycliquement
    plt.plot(year_columns_PACA,yy, marker=marker, label=f"Département {dep} ({dep_PACA[dep]})")

# Personnalisation du graphique
plt.xticks(rotation=45)  # Incliner les années
plt.grid()
plt.title("Évolution de la population par département dans la région PACA")
plt.xlabel("Année")
plt.ylabel("Population")
plt.legend(title="Départements", loc='upper left', bbox_to_anchor=(1, 1))
plt.show()
```




```
dep_IdF = {
    '75': 'Paris',
    '77': 'Seine-et-Marne',
    '78': 'Yvelines',
    '91': 'Essonne',
    '92': 'Hauts-de-Seine',
    '93': 'Seine-Saint-Denis',
    '94': 'Val-de-Marne',
    '95': "Val-d'Oise"
}



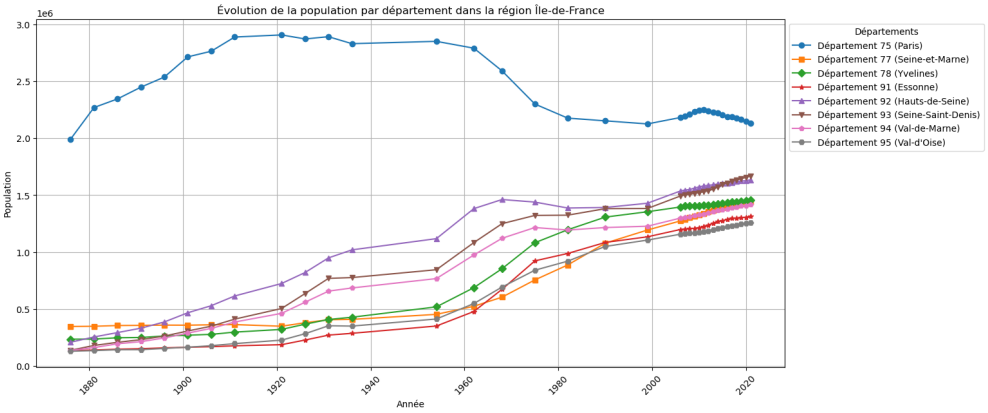
# Colonnes des années (de 2021 à 1876)
year_columns_IdF = df_IdF.columns[3:].map(int)

# Groupement des données par région
df_IdF_grbyDEP = df_IdF.groupby('DEP')

# Tracé de l'évolution
plt.figure(figsize=(15, 7))
# Liste des marqueurs
markers = ['o', 's', 'D', '*', '^', 'v', 'p', 'H']  # Ajouter plus si nécessaire
num_markers = len(markers)  # Nombre total de marqueurs

for i, dep in enumerate(df_IdF_grbyDEP.groups.keys()):
    yy = df_IdF_grbyDEP.get_group(dep).iloc[:, 3:].sum()
    marker = markers[i % num_markers]  # Sélectionner un marqueur cycliquement
    plt.plot(year_columns_IdF,yy, marker=marker, label=f"Département {dep} ({dep_IdF[dep]})")

# Personnalisation du graphique
plt.xticks(rotation=45)  # Incliner les années
plt.grid()
plt.title("Évolution de la population par département dans la région Île-de-France")
plt.xlabel("Année")
plt.ylabel("Population")
plt.legend(title="Départements", loc='upper left', bbox_to_anchor=(1, 1))
plt.show()
```


Évolution de la population par département dans la région Île-de-France
1e6
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Population
1880
1900
1920
1940
1960
1980
2000
2020
Année
Départements
Département 75 (Paris)
Département 77 (Seine-et-Marne)
Département 78 (Yvelines)
Département 91 (Essonne)
Département 92 (Hauts-de-Seine)
Département 93 (Seine-Saint-Denis)
Département 94 (Val-de-Marne)
Département 95 (Val-d'Oise)

Chapitre 27.

Émissions de CO des communes françaises

27.1. Les données

Le CITEPA, pour le compte du ministère chargé de l'Environnement, publie un **inventaire des émissions de gaz à effet de serre commune par commune**, ventilé par secteur d'activité : agriculture, industrie, transport routier, résidentiel, tertiaire, déchets.

Intitulé	Inventaire de gaz à effet de serre territorialisé
Producteur	CITEPA, pour la Direction générale de l'énergie et du climat
Année de référence	**2016**
Couverture	35 798 communes
Unité	tonnes équivalent CO par an
Licence	Licence Ouverte (Etalab)
Source	data.ademe.fr

Le producteur précise que cet inventaire « a vocation à donner des **ordres de grandeur** pertinents » et qu'il « n'a pas le niveau de finesse et de précision des travaux d'inventaire réalisés au niveau territorial ». Autrement dit, ces chiffres servent à comparer des territoires entre eux, pas à établir le bilan exact d'une commune donnée. C'est une limite qu'il faut connaître avant de conclure quoi que ce soit, et que nous rappellerons en fin d'exercice.

Nous croiserons ces émissions avec la **population communale** déjà utilisée dans l'exercice précédent. Les émissions datant de 2016, nous prendrons la population de cette même année, et non la plus récente : diviser des émissions de 2016 par une population de 2021 n'aurait pas de sens.

La question posée est simple à formuler et moins simple à trancher : **quels territoires émettent le plus ?**

27.2. Exercice

27.2.1. 1. Découvrir les données

- Charger `emissions-communes.csv`. Combien de communes, combien de colonnes ?
- Quelles sont les colonnes de secteurs ? Afficher leurs statistiques descriptives.
- La colonne `INSEE commune` est-elle un identifiant unique ? Et la colonne `Commune` ?

27.2.2. 2. Les valeurs manquantes

- Quelle proportion de valeurs manquantes dans chaque colonne ?
- Deux colonnes sont très lacunaires. Afficher les communes où `Autres transports  international` est **renseigné et le plus élevé**. Que reconnaissez-vous ? Qu'en déduire sur la signification du `NaN` ici ?
- Créer une colonne `total` donnant les émissions totales de chaque commune.

27.2.3. 3. Joindre à la population

- Charger `base-pop-historiques-1876-2021.csv` et en extraire le code, le nom, le département et la **population de 2016** de chaque commune. Attention au format des nombres.
- Joindre les deux tables. Combien de communes se raccordent ? Combien restent orphelines ?

27.2.4. 4. Émissions par habitant

- Écarter les communes sans habitant, puis créer `par_habitant`.
- Quelles sont les 5 communes les plus émettrices **au total** ? Et **par habitant** ?
- Agréger par département, des deux façons. Comparer les classements et **commenter**.

27.2.5. 5. Cartographier

- Représenter sur une carte de France les émissions **totales** par département, puis les émissions **par habitant**. Commenter la différence entre les deux cartes.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import json   # pour lire le fond de carte, qui est au format GeoJSON

from IPython.display import display, Markdown

import plotly.express as px

# Pour la génération automatique de graphiques interactifs dans le polycopié
import plotly.io as pio
import os
if os.getenv("QUARTO_OUTPUT_FORMAT") is not None:
    # Export Quarto HTML
    pio.renderers.default = "iframe_connected"
else:
    # Notebook interactif
    pio.renderers.default = "notebook_connected"

sns.set_theme(context="notebook", style="whitegrid", palette="pastel")
plt.rcParams['figure.figsize'] = (11, 5)
```

Chapitre 28.

1. Découvrir les données

Le code INSEE commence par un zéro pour les départements 01 à 09. Si on laisse pandas deviner le type, il le lira comme un entier et `01001` deviendra `1001` : la jointure échouerait plus loin. On impose donc le type texte dès la lecture.

```
# dtype={'INSEE commune': str} force la lecture de cette colonne en texte,
# ce qui préserve le zéro initial des codes 01xxx à 09xxx
emissions = pd.read_csv('emissions-communes.csv', dtype={'INSEE commune': str})

# .shape renvoie le couple (nombre de lignes, nombre de colonnes)
display(Markdown(f"**{emissions.shape[0]}** communes, **{emissions.shape[1]}** colonnes"))

# .head() affiche les cinq premières lignes
emissions.head()
```

35798 communes, **12** colonnes

	INSEE commune	Commune	Agriculture	Autres transports	Autres transports international
0	01001	L'ABERGEMENT-CLEMENCIAT	3711.425991	NaN	NaN
1	01002	L'ABERGEMENT-DE-VAREY	475.330205	NaN	NaN
2	01004	AMBERIEU-EN-BUGEY	499.043526	212.577908	NaN
3	01005	AMBERIEUX-EN-DOMBES	1859.160954	NaN	NaN
4	01006	AMBLEON	448.966808	NaN	NaN

```
# on construit la liste des colonnes de secteurs, c'est-à-dire
# toutes les colonnes sauf les deux colonnes d'identification
secteurs = [c for c in emissions.columns if c not in ('INSEE commune', 'Commune')]

display(Markdown(f"**{len(secteurs)} secteurs** : " + ", ".join(f"`{s}`" for s in secteurs)))
```

10 secteurs : `Agriculture`, `Autres transports`, `Autres transports international`, `CO2 biomasse hors-total`, `Déchets`, `Energie`, `Industrie hors-énergie`, `Résidentiel`, `Routier`, `Tertiaire`

```
# .describe() calcule les statistiques descriptives de chaque colonne numérique
# .T transpose le résultat : un secteur par ligne, plus lisible quand ils sont nombreux
emissions[secteurs].describe().round(0).T
```

	count	mean	std	min	25%	50%	75%	max
Agriculture	35736.0	2460.0	2927.0	0.0	798.0	1559.0	3008.0	98949.0
Autres transports	9979.0	655.0	9233.0	0.0	53.0	107.0	237.0	513141.0
Autres transports international	2891.0	7692.0	113764.0	0.0	10.0	20.0	33.0	3303394.0
CO2 biomasse hors-total	35798.0	1774.0	7871.0	4.0	198.0	425.0	1095.0	576394.0
Déchets	35792.0	411.0	4122.0	0.0	26.0	55.0	111.0	275500.0
Energie	34490.0	663.0	26456.0	2.0	2.0	5.0	52.0	2535858.0

	count	mean	std	min	25%	50%	75%	max
Industrie hors-énergie	34490.0	2423.0	56704.0	1.0	7.0	14.0	152.0	6765119.0
Résidentiel	35792.0	1784.0	8916.0	1.0	96.0	227.0	749.0	410676.0
Routier	35778.0	3536.0	9663.0	1.0	420.0	1071.0	3099.0	586055.0
Tertiaire	35798.0	1105.0	5164.0	0.0	95.0	216.0	576.0	288175.0

Les ordres de grandeur sont très étalés : certaines communes émettent quelques tonnes, d'autres plusieurs millions. C'est attendu pour 35 000 communes allant du hameau à la métropole.

L'identifiant est-il unique ?

```
# .nunique() compte les valeurs distinctes, len() compte les lignes :
# si les deux coïncident, la colonne est un identifiant
display(Markdown(
    f"codes INSEE distincts : **{emissions['INSEE commune'].nunique()}** sur {len(emissions)}  \n"
    f"noms de communes distincts : **{emissions['Commune'].nunique()}** sur {len(emissions)}"))
```

codes INSEE distincts : **35798** sur 35798
noms de communes distincts : **33338** sur 35798

Le **code INSEE** est unique, le **nom** ne l'est pas : plus de deux mille communes françaises portent le même nom qu'une autre. C'est pourquoi toute jointure entre fichiers communaux se fait sur le code, jamais sur le libellé.

Chapitre 29.

2. Les valeurs manquantes

```
# .isnull() renvoie un tableau de booléens de même forme que le DataFrame ;
# .mean() sur des booléens donne la proportion de True, que l'on convertit en pourcentage
(emissions[secteurs].isnull().mean() * 100).round(1).sort_values(ascending=False).to_frame('% manquant')
```

	% manquant
Autres transports international	91.9
Autres transports	72.1
Energie	3.7
Industrie hors-énergie	3.7
Agriculture	0.2
Routier	0.1
CO2 biomasse hors-total	0.0
Déchets	0.0
Résidentiel	0.0
Tertiaire	0.0

Deux colonnes sortent du lot : `Autres transports international` est vide à près de **92 %**, `Autres transports` à **72 %**. Avant de décider quoi en faire, regardons **où** elles sont renseignées.

```
colonne = 'Autres transports international'

# .notna() sélectionne les lignes où la valeur est présente
renseignees = emissions[ emissions[colonne].notna() ]

display(Markdown(f"renseignée pour **{len(renseignees)}** communes sur {len(emissions)}"))

# .nlargest(12, colonne) renvoie les 12 lignes aux plus grandes valeurs de cette colonne
renseignees.nlargest(12, colonne)[['Commune', colonne]].round(0)
```

renseignée pour **2891** communes sur 35798

	Commune	Autres transports international
31099	MAUREGARD	3303394.0
31108	LE MESNIL-AMELOT	3303394.0
31111	MITRY-MORY	2202275.0
30438	LE HAVRE	2109460.0
35561	TREMBLAY-EN-FRANCE	1101146.0
35757	ROISSY-EN-FRANCE	1101131.0
1987	NICE	1003572.0
28017	COLOMBIER-SAUGNIEU	865809.0
35426	PARAY-VIEILLE-POSTE	785878.0
27580	MULHOUSE	615461.0
4397	MARIGNANE	527361.0

	Commune	Autres transports international
35487	WISSOUS	471527.0

Ces noms vous disent quelque chose.

Mauregard, Le Mesnil-Amelot, Mitry-Mory, Tremblay-en-France, Roissy-en-France : ce sont les communes de **l'aéroport Charles-de-Gaulle**. Paray-Vieille-Poste et Wissous bordent **Orly**, Colombier-Saugnieu est **Lyon-Saint-Exupéry**, Marignane **Marseille-Provence**, Blagnac **Toulouse**. Le Havre est un **port**.

La valeur manquante ne signifie donc pas « donnée perdue » : elle signifie **« il n'y a ni aéroport ni port dans cette commune »**. L'absence d'information *est* l'information.

Les conséquences sont directes :

- **remplacer par la moyenne** serait absurde, puisqu'on attribuerait un trafic aérien à des villages ;
- **supprimer les lignes incomplètes** effacerait 92 % de la France ;
- **remplacer par zéro** est ici le seul choix défendable, et c'est exactement ce que fait `sum()` par défaut en ignorant les `NaN`.

```
# .sum(axis=1) additionne sur les colonnes, donc ligne par ligne.
# Par défaut les NaN sont ignorés, ce qui revient ici à les compter pour zéro.
emissions['total'] = emissions[secteurs].sum(axis=1)

emissions[['Commune', 'total']].head()
```

	Commune	total
0	L'ABERGEMENT-CLEMENCIAT	5724.424941
1	L'ABERGEMENT-DE-VAREY	1332.811619
2	AMBERIEU-EN-BUGEY	63259.689119
3	AMBERIEUX-EN-DOMBES	6792.867439
4	AMBLEON	1068.584766

Chapitre 30.

3. Joindre à la population

Le fichier de populations sépare les milliers par un **espace insécable** (`1 000`), que `read_csv` ne sait pas interpréter seul, comme nous l'avons vu dans le cours. On nettoie donc la colonne avant de la convertir.

Nous prenons la colonne `PMUN2016`, population municipale de 2016, pour la faire correspondre à l'année des émissions.

```
# usecols limite la lecture aux colonnes utiles : le fichier en compte une
# par recensement depuis 1876, inutile de tout charger en mémoire
colonnes_pop = ['CODGEO', 'LIBGEO', 'DEP', 'REG', 'PMUN2016']

population = pd.read_csv('base-pop-historiques-1876-2021.csv', sep=';', low_memory=False,
                         dtype={'CODGEO': str, 'DEP': str, 'REG': str},
                         usecols=colonnes_pop)

# le séparateur n'apparaît que sur les valeurs supérieures à mille :
# on repère une telle ligne pour pouvoir l'afficher avant et après conversion
grandes = population['PMUN2016'].astype(str).str.len() > 4
display(Markdown("avant nettoyage : `"
                 + repr(population.loc[grandes, 'PMUN2016'].iloc[0]) + "`"))

# chr(160) est l'espace insécable ; regex=False car on remplace un caractère littéral
population['PMUN2016'] = (population['PMUN2016'].astype(str)
                                                .str.replace(chr(160), '', regex=False)
                                                .astype(int))

display(Markdown("après nettoyage : `"
                 + repr(population.loc[grandes, 'PMUN2016'].iloc[0]) + "`"))
population.head()
```

avant nettoyage : `'1\xa0030'`

après nettoyage : `1030`

	CODGEO	REG	DEP	LIBGEO	PMUN2016
0	55039	44	55	Beaumont-en-Verdunois	0
1	55050	44	55	Bezonvaux	0
2	55139	44	55	Cumières-le-Mort-Homme	0
3	55189	44	55	Fleury-devant-Douaumont	0
4	55239	44	55	Haumont-près-Samogneux	0

```
# how='left' conserve toutes les lignes de l'inventaire, même sans correspondance ;
# validate='one_to_one' fait échouer la jointure si une clé était dupliquée d'un côté
df = emissions.merge(population,
                     left_on='INSEE commune', right_on='CODGEO',
                     how='left', validate='one_to_one')

display(Markdown(
    f"communes dans l'inventaire : **{len(emissions)}**  \n"
```

```
    f"après la jointure : **{len(df)}**  \n"
    f"sans correspondance : **{df['CODGEO'].isna().sum()}**"))
```

communes dans l'inventaire : **35798**
après la jointure : **35798**
sans correspondance : **966**

Environ **un millier de communes** n'ont pas de correspondance. Les deux fichiers ne datent pas du même millésime, et le découpage communal bouge sans cesse : fusions, communes nouvelles. Nous les écartons, en sachant ce que nous écartons.

```
# .notna() garde les lignes où la jointure a trouvé une correspondance
df = df[ df['CODGEO'].notna() ]

display(Markdown(f"**{len(df)}** communes conservées"))
```

34832 communes conservées

Chapitre 31.

4. Émissions par habitant

Quelques communes ont une population nulle, des villages détruits pendant la guerre de 1914-1918 et jamais reconstruits. Il faut les écarter avant toute division.

```
display(Markdown(f"communes sans habitant : **{(df['PMUN2016'] == 0).sum()}**"))
display(df.loc[df['PMUN2016'] == 0, ['Commune', 'DEP', 'PMUN2016']])

# .copy() crée un DataFrame indépendant : sans lui, l'ajout de colonne
# qui suit porterait sur une tranche du tableau d'origine
df = df[ df['PMUN2016'] > 0 ].copy()

df['par_habitant'] = df['total'] / df['PMUN2016']
```

communes sans habitant : **6**

	Commune	DEP	PMUN2016
20731	BEAUMONT-EN-VERDUNOIS	55	0.0
20742	BEZONVAUX	55	0.0
20817	CUMIERES-LE-MORT-HOMME	55	0.0
20861	FLEURY-DEVANT-DOUAUMONT	55	0.0
20898	HAUMONT-PRES-SAMOGNEUX	55	0.0
20957	LOUVEMONT-COTE-DU-POIVRE	55	0.0

Les cinq communes les plus émettrices au total :

```
# .nlargest(5, 'total') : les 5 plus grandes valeurs de la colonne 'total'
df.nlargest(5, 'total')[['Commune', 'DEP', 'PMUN2016', 'total']].round(0)
```

	Commune	DEP	PMUN2016	total
4382	FOS-SUR-MER	13	15608.0	9533813.0
22671	DUNKERQUE	59	88108.0	8643550.0
4398	MARTIGUES	13	48783.0	4009663.0
30560	PORT-JEROME-SUR-SEINE	76	9903.0	3668151.0
31108	LE MESNIL-AMELOT	77	997.0	3471492.0

Et les cinq plus émettrices par habitant :

```
df.nlargest(5, 'par_habitant')[['Commune', 'DEP', 'PMUN2016', 'par_habitant']].round(0)
```

	Commune	DEP	PMUN2016	par_habitant
31099	MAUREGARD	77	340.0	10120.0
31108	LE MESNIL-AMELOT	77	997.0	3482.0

	Commune	DEP	PMUN2016	par_habitant
23189	BAZANCOURT	60	129.0	2380.0
27426	CHALAMPE	68	950.0	1187.0
5682	SAINT-VALLIER	16	137.0	1127.0

Deux listes sans aucun point commun. La première donne de grandes villes industrielles, la seconde de très petites communes abritant une installation lourde : quelques dizaines d'habitants pour une usine, et le rapport s'envole.

31.1. Agréger par département

Attention au piège : la moyenne des rapports n'est pas le rapport des sommes.

Pour obtenir les émissions par habitant d'un département, il ne faut **pas** faire la moyenne des `par_habitant` de ses communes : cela donnerait le même poids à un village de 80 habitants qu'à une ville de 200 000. Il faut diviser le **total des émissions** par le **total des habitants**.

```
# .agg() applique une fonction différente à chaque colonne.
# La syntaxe nom_voulu=('colonne', 'fonction') nomme directement le résultat.
dep = df.groupby('DEP').agg(
    emissions = ('total',    'sum'),
    habitants = ('PMUN2016', 'sum'),
    communes  = ('CODGEO',   'size')
)

dep['par_habitant'] = (dep['emissions'] / dep['habitants']).round(1)             # correct
dep['moyenne_des_communes'] = df.groupby('DEP')['par_habitant'].mean().round(1)  # faux

dep.sort_values('par_habitant', ascending=False).head(5)
```

DEP	emissions	habitants	communes	par_habitant	moyenne_des_communes
13	2.421180e+07	1157506.0	118	20.9	15.5
23	2.327727e+06	119502.0	256	19.5	29.2
76	2.154345e+07	1255503.0	707	17.2	14.3
15	2.506101e+06	145969.0	246	17.2	32.7
55	3.199781e+06	189055.0	493	16.9	26.8

Comparons les rangs obtenus par les deux méthodes :

```
# .rank(ascending=False) donne le rang de chaque département, 1 pour le plus élevé
comparaison = pd.DataFrame({
    'rapport des sommes'   : dep['par_habitant'].rank(ascending=False).astype(int),
    'moyenne des communes' : dep['moyenne_des_communes'].rank(ascending=False).astype(int),
})
comparaison['écart de rang'] = (comparaison['rapport des sommes']
                                - comparaison['moyenne des communes']).abs()

comparaison.sort_values('rapport des sommes').head(10)
```

DEP	rapport des sommes	moyenne des communes	écart de rang
13	1	37	36
23	2	3	1

DEP	rapport des sommes	moyenne des communes	écart de rang
15	3	2	1
76	3	45	42
55	5	8	3
53	6	18	12
03	7	6	1
52	7	3	4
48	9	5	4
12	10	9	1

Les deux méthodes ne classent pas de la même façon : le choix de l'agrégation n'est pas neutre.

31.2. Total ou par habitant ?

```
cote_a_cote = pd.DataFrame({
    'total (kt CO )'  : (dep['emissions'] / 1000).round(0),
    'par habitant (t)': dep['par_habitant'],
})

display(Markdown("**Classement par émissions totales**"))
display(cote_a_cote.nlargest(5, 'total (kt CO )'))

display(Markdown("**Classement par émissions par habitant**"))
display(cote_a_cote.nlargest(5, 'par habitant (t)'))

# .nsmallest() est le symétrique de .nlargest()
display(Markdown("**Les cinq départements les plus sobres par habitant**"))
display(cote_a_cote.nsmallest(5, 'par habitant (t)'))
```

Classement par émissions totales

DEP	total (kt CO)	par habitant (t)
59	26075.0	10.0
13	24212.0	20.9
76	21543.0	17.2
77	18161.0	13.0
62	12069.0	8.2

Classement par émissions par habitant

DEP	total (kt CO)	par habitant (t)
13	24212.0	20.9
23	2328.0	19.5
15	2506.0	17.2
76	21543.0	17.2
55	3200.0	16.9

Les cinq départements les plus sobres par habitant

DEP	total (kt CO)	par habitant (t)
75	6730.0	3.1
92	5796.0	3.6
94	5254.0	3.8
93	6254.0	3.9
83	4301.0	4.1

Deux questions, deux réponses.

En **volume**, les départements industriels et peuplés dominent : ce sont eux qui pèsent sur le bilan national, et c'est là que des réductions en valeur absolue sont possibles.

En **intensité**, le classement se renverse. Les départements ruraux arrivent en tête, avec peu d'habitants mais de l'agriculture et des trajets longs, tandis que **Paris et la petite couronne sont les plus sobres de France par habitant**, d'un facteur proche de cinq. La densité urbaine réduit les transports et le chauffage par personne.

Aucun des deux classements n'est « le bon ». Ils répondent à deux questions différentes, et confondre les deux est la façon la plus commune de faire mentir des données climatiques.

Chapitre 32.

5. Cartographier

Un tableau de 96 lignes se lit mal ; une carte montre immédiatement la géographie du phénomène. Nous utilisons le fond de carte `departements.geojson`, un fichier **local** : aucune connexion n'est nécessaire.

Chaque département y est décrit par une propriété `code` (`'01'`, `'02'`, jusqu'à `'2A'` et `'2B'` pour la Corse), qui correspond exactement à notre colonne `DEP`.

```
# un fichier GeoJSON est un fichier JSON : on le lit avec le module json
with open('departements.geojson', encoding='utf-8') as f:
    france = json.load(f)

display(Markdown(f"**{len(france['features'])}** départements dans le fond de carte"))
display(Markdown("propriétés disponibles : `" + str(france['features'][0]['properties']) + "`"))
```

96 départements dans le fond de carte

propriétés disponibles : `{'code': '02', 'nom': 'Aisne'}`

```
# .reset_index() ramène le code département, qui était l'index, dans une colonne :
# plotly a besoin d'une colonne pour faire la correspondance avec le fond de carte
carte = dep.reset_index()

# on ne garde que les départements présents dans le fond de carte
codes_du_fond = {f['properties']['code'] for f in france['features']}
carte = carte[ carte['DEP'].isin(codes_du_fond) ]

display(Markdown(f"**{len(carte)}** départements cartographiables "
                 "(l'outre-mer n'est pas dans ce fond de carte)"))
```

96 départements cartographiables (l'outre-mer n'est pas dans ce fond de carte)

Les émissions totales :

```
# locations indique la colonne du DataFrame qui identifie chaque zone,
# featureidkey indique où trouver l'identifiant correspondant dans le GeoJSON
fig = px.choropleth(carte,
                    geojson=france, locations='DEP', featureidkey='properties.code',
                    color='emissions', color_continuous_scale='Reds',
                    labels={'emissions': 't CO '},
                    title="Émissions totales par département (2016)")

# fitbounds recentre la carte sur les zones affichées, visible=False masque le globe
fig.update_geos(fitbounds="locations", visible=False)
fig.update_layout(height=550, margin={"r":0, "t":40, "l":0, "b":0})
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

Et les émissions par habitant :

```
fig = px.choropleth(carte,
                    geojson=france, locations='DEP', featureidkey='properties.code',
                    color='par_habitant', color_continuous_scale='Reds',
                    labels={'par_habitant': 't CO  / hab'},
                    title="Émissions par habitant (2016)")
fig.update_geos(fitbounds="locations", visible=False)
fig.update_layout(height=550, margin={"r":0, "t":40, "l":0, "b":0})
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Les deux cartes n'ont presque rien en commun.

La première dessine la France industrielle et peuplée : vallée de la Seine, Nord, pourtour de l'étang de Berre, grandes agglomérations.

La seconde dessine une France rurale, et fait apparaître **l'Île-de-France en creux** : les départements les plus denses du pays sont les plus sobres par habitant. Une carte des émissions totales de l'Île-de-France aurait suggéré l'inverse.

C'est la même leçon que plus haut, rendue visible d'un coup d'œil : **la carte que l'on publie est déjà une réponse à une question que l'on a choisie.**

Chapitre 33.

Conclusion

Ce que cet exercice voulait montrer

- Une **valeur manquante peut être une information** : ici, `NaN` signifiait « pas d'aéroport ». Avant de combler les trous, il faut comprendre pourquoi ils existent.
- Une **jointure se fait sur un identifiant**, jamais sur un libellé, puisque plus de deux mille communes françaises sont homonymes.
- Les deux tables doivent porter sur **la même année**, sans quoi le rapport calculé n'a pas de sens.
- La **moyenne des rapports n'est pas le rapport des sommes.**
- Et surtout : **total** et **par habitant** répondent à deux questions distinctes et donnent deux classements opposés. Choisir l'un ou l'autre, c'est déjà prendre parti.

Une dernière prudence. Le producteur de ces données écrit lui-même qu'elles donnent « des ordres de grandeur » et qu'elles n'ont pas « le niveau de finesse et de précision des travaux d'inventaire réalisés au niveau territorial ». Nos comparaisons entre départements sont donc légitimes ; affirmer le bilan exact d'une commune précise à partir de ce fichier ne le serait pas.

Savoir ce qu'une source permet de dire, et ce qu'elle ne permet pas, fait partie du travail d'analyse.

Chapitre 34.

Netflix User Data

L'ensemble de données fu fichier `Netflix_Userbase.csv` fournit un aperçu d'un échantillon d'utilisateurs de Netflix.

Chaque ligne représente un utilisateur unique, identifié par son ID utilisateur.

L'ensemble de données comprend les colonnes suivantes :

- le type d'abonnement de l'utilisateur (Basic, Standard ou Premium),
- le revenu mensuel généré par son abonnement,
- la date d'inscription à Netflix,
- la date de son dernier paiement et
- le pays dans lequel il se trouve.

Des colonnes supplémentaires ont été incluses pour fournir des informations sur le comportement et les préférences des utilisateurs, comme le type d'appareil.

Source : Kaggle
Disclaimer : l'ensemble de données sert de représentation synthétique et ne reflète pas les données réelles des utilisateurs de Netflix.

34.1. Exercice

1. Affichages
   - Charger le fichier `Netflix_Userbase.csv` dans un DataFrame.
   - Afficher les 20 premières lignes.
   - Combien de ligne et de colonnes contient ce fichier ?
   - Quels sont les types des colonnes ?
   - Pour chaque colonnes, combien de valeurs différentes contient-elle ?
   - Y a-t-il des valeurs manquantes ?
2. Modification des données
   - Ne garder que les utilisateurs europeens.
3. Aggregations/Visualisations
   - Afficher le nombre d'utilisateurs par pays.
   - Afficher le nombre d'utilisateurs par sexe.
   - Afficher le nombre d'utilisateurs par devices utilisés.
   - Continuer l'analyse des données en affichant des visualisations qui vous semblent pertinentes en combinant les différentes caractéristiques des utilisateurs.
4. Conclusion
   - Quels sont les insights que vous avez pu tirer de cette analyse ?

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
# plt.rcParams['figure.figsize'] = (15, 7) # set default size of plots

import seaborn as sns
```

```
import plotly.express as px

from IPython.display import display, Markdown
```

34.1.1. Affichages

```
# Importer le jeu de données
df_complet = pd.read_csv('Netflix_Userbase.csv', sep=',', low_memory=False)

# Afficher les 20 premières lignes
df_complet.head(20)
```

	User ID	Subscription Type	Monthly Revenue	Join Date	Last Payment Date	Country	Age	Ge
0	1	Basic	10	15-01-22	10-06-23	United States	28	Ma
1	2	Premium	15	05-09-21	22-06-23	Canada	35	Fe
2	3	Standard	12	28-02-23	27-06-23	United Kingdom	42	Ma
3	4	Standard	12	10-07-22	26-06-23	Australia	51	Fe
4	5	Basic	10	01-05-23	28-06-23	Germany	33	Ma
5	6	Premium	15	18-03-22	27-06-23	France	29	Fe
6	7	Standard	12	09-12-21	25-06-23	Brazil	46	Ma
7	8	Basic	10	02-04-23	24-06-23	Mexico	39	Fe
8	9	Standard	12	20-10-22	23-06-23	Spain	37	Ma
9	10	Premium	15	07-01-23	22-06-23	Italy	44	Fe
10	11	Basic	10	16-05-22	22-06-23	United States	31	Fe
11	12	Premium	15	23-03-23	28-06-23	Canada	45	Ma
12	13	Standard	12	30-11-21	27-06-23	United Kingdom	48	Fe
13	14	Basic	10	01-08-22	26-06-23	Australia	27	Ma
14	15	Standard	12	09-05-23	28-06-23	Germany	38	Fe
15	16	Premium	15	07-04-22	27-06-23	France	36	Ma
16	17	Basic	10	24-01-22	25-06-23	Brazil	30	Fe
17	18	Standard	12	18-10-21	24-06-23	Mexico	43	Ma
18	19	Premium	15	15-02-23	23-06-23	Spain	32	Fe
19	20	Basic	10	27-05-23	22-06-23	Italy	41	Ma

```
# display(df_complet.shape)
df_complet.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2500 entries, 0 to 2499
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   User ID            2500 non-null   int64
 1   Subscription Type  2500 non-null   object
 2   Monthly Revenue    2500 non-null   int64
 3   Join Date          2500 non-null   object
 4   Last Payment Date  2500 non-null   object
 5   Country            2500 non-null   object
 6   Age                2500 non-null   int64
 7   Gender             2500 non-null   object
 8   Device             2500 non-null   object
 9   Plan Duration      2500 non-null   object
dtypes: int64(3), object(7)
memory usage: 195.4+ KB
```

Notre jeu de données contient 2500 lignes (= utilisateurs) et 12 colonnes. 3 colonnes sont de type `int`, 7 colonnes sont de type `object`. Notons que les colonnes `date` sont de type `object` et devront être converties en `datetime` pour être exploitées correctement le cas échéant.

Statistiques descriptives pour les colonnes numériques :

```
df_complet.describe()
```

	User ID	Monthly Revenue	Age
count	2500.00000	2500.000000	2500.000000
mean	1250.50000	12.508400	38.795600
std	721.83216	1.686851	7.171778
min	1.00000	10.000000	26.000000
25%	625.75000	11.000000	32.000000
50%	1250.50000	12.000000	39.000000
75%	1875.25000	14.000000	45.000000
max	2500.00000	15.000000	51.000000

Statistiques descriptives pour les colonnes catégorielles :

```
df_complet.describe(include='object')

# df_complet.nunique()
```

	Subscription Type	Join Date	Last Payment Date	Country	Gender	Device	Plan Duration
count	2500	2500	2500	2500	2500	2500	2500
unique	3	300	26	10	2	4	1
top	Basic	05-11-22	28-06-23	United States	Female	Laptop	1 Month
freq	999	33	164	451	1257	636	2500

Pour les colonnes dont les valeurs uniques sont inférieures à 20, afficher les possibilités de ces valeurs.

```
masque = [col for col in df_complet.columns if df_complet[col].nunique() < 20]

for c in masque:
    if df_complet[c].nunique() < 20:
        print(f"{c:.<20}", sorted(df_complet[c].unique()))
```

```
Subscription Type... ['Basic', 'Premium', 'Standard']
Monthly Revenue..... [10, 11, 12, 13, 14, 15]
Country............. ['Australia', 'Brazil', 'Canada', 'France', 'Germany', 'Italy', 'Mexico', 'Spain', 'Uni
Gender.............. ['Female', 'Male']
Device.............. ['Laptop', 'Smart TV', 'Smartphone', 'Tablet']
Plan Duration....... ['1 Month']
```

```
for col in masque:
    display(df_complet[col].value_counts().to_frame().sort_index())
```

Subscription Type	count
Basic	999

Subscription Type	count
Premium	733
Standard	768

Monthly Revenue	count
10	409
11	388
12	455
13	418
14	431
15	399

Country	count
Australia	183
Brazil	183
Canada	317
France	183
Germany	183
Italy	183
Mexico	183
Spain	451
United Kingdom	183
United States	451

Gender	count
Female	1257
Male	1243

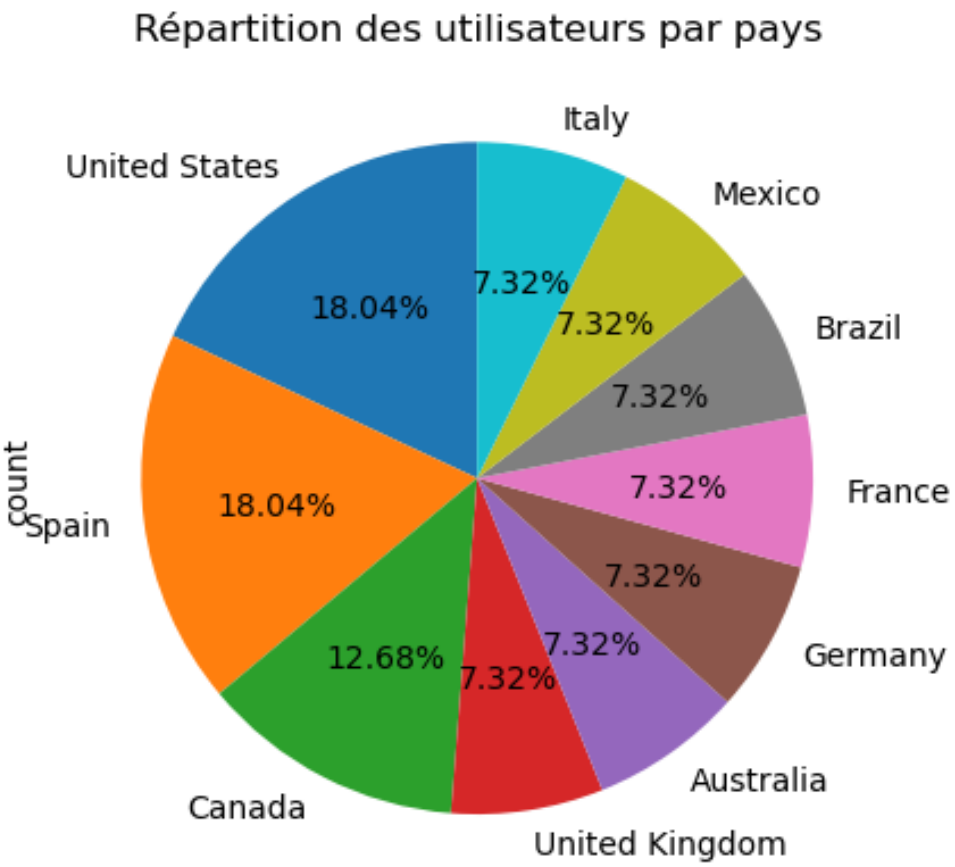
Device	count
Laptop	636
Smart TV	610
Smartphone	621
Tablet	633

Plan Duration	count
1 Month	2500

```
# Avec Pandas
df_complet['Country'].value_counts().plot(kind='pie', autopct='%1.2f%%', startangle=90, shadow=False,
 ↪  legend=False)
plt.title('Répartition des utilisateurs par pays');
plt.show()
```

```
# Avec Seaborn
# Note: Seaborn n'a pas de fonction native pour les diagrammes circulaires (pie charts)
# sns.countplot(data=df_complet, x='Country');
# plt.title('Répartition des utilisateurs par pays');
# plt.show()

# Avec Plotly
# fig = px.pie(df_complet,
#              names='Country',
#              title='Répartition des utilisateurs par pays'
#              ).update_traces(texttemplate='%{label}<br>%{percent:.2%}',
#                              textposition='outside').show()
```



34.1.2. Modification des données

User ID est une colonne qui ne sert pas à grand chose, on peut la supprimer.

```
df_complet.drop('User ID', axis=1, inplace=True)
```

```
filtre = ['United Kingdom', 'Germany', 'France', 'Spain', 'Italy']

# mask =  (df_complet['Country'] in filtre) # erreur
# mask = df_complet['Country']==filtre[0] | df_complet['Country']==filtre[1] |
 ↪  df_complet['Country']==filtre[2] | df_complet['Country']==filtre[3] |
 ↪  df_complet['Country']==filtre[4] # erreur
mask = df_complet['Country'].isin(filtre)
```

```
df = df_complet[mask].copy()  # .copy() : on va ajouter des colonnes
df
```

	Subscription Type	Monthly Revenue	Join Date	Last Payment Date	Country	Age	Gender	D
2	Standard	12	28-02-23	27-06-23	United Kingdom	42	Male	Sr
4	Basic	10	01-05-23	28-06-23	Germany	33	Male	Sr
5	Premium	15	18-03-22	27-06-23	France	29	Female	Sr
8	Standard	12	20-10-22	23-06-23	Spain	37	Male	Sr
9	Premium	15	07-01-23	22-06-23	Italy	44	Female	Sr
...	...	...	...	...	...	...	...	...
2490	Premium	13	18-07-22	11-07-23	France	41	Female	Sr
2493	Premium	12	21-07-22	15-07-23	Spain	36	Male	Sr
2494	Basic	15	23-07-22	12-07-23	Italy	43	Female	La
2495	Premium	14	25-07-22	12-07-23	Spain	28	Female	Sr
2496	Basic	15	04-08-22	14-07-23	Spain	33	Female	Sr

34.1.3. Aggregations/Visualisations

34.1.4. Répartition selon chaque caractéristique

Nous voulons le même tableau : effectifs et pourcentages : pour quatre colonnes différentes. Plutôt que de recopier quatre fois les mêmes instructions, écrivons-les **une fois** dans une fonction. C'est plus court, et surtout : si l'on veut changer la présentation, il n'y a qu'un seul endroit à corriger.

```
categorielles = ['Country', 'Gender', 'Device', 'Subscription Type']

def repartition(colonne):
    """Effectifs et pourcentages des modalités d'une colonne."""
    effectifs = df[colonne].value_counts()
    return pd.DataFrame({
        'Nombre'          : effectifs,
        'Pourcentage (%)' : (effectifs / effectifs.sum() * 100).round(1)
    })

for colonne in categorielles:
    display(Markdown(f"**{colonne}**"))
    display(repartition(colonne))
```

Country

Country	Nombre	Pourcentage (%)
Spain	451	38.1
United Kingdom	183	15.5
Germany	183	15.5
France	183	15.5
Italy	183	15.5

Gender

Gender	Nombre	Pourcentage (%)
Female	602	50.9

Gender	Nombre	Pourcentage (%)
Male	581	49.1

Device

Device	Nombre	Pourcentage (%)
Laptop	316	26.7
Smart TV	297	25.1
Smartphone	286	24.2
Tablet	284	24.0

Subscription Type

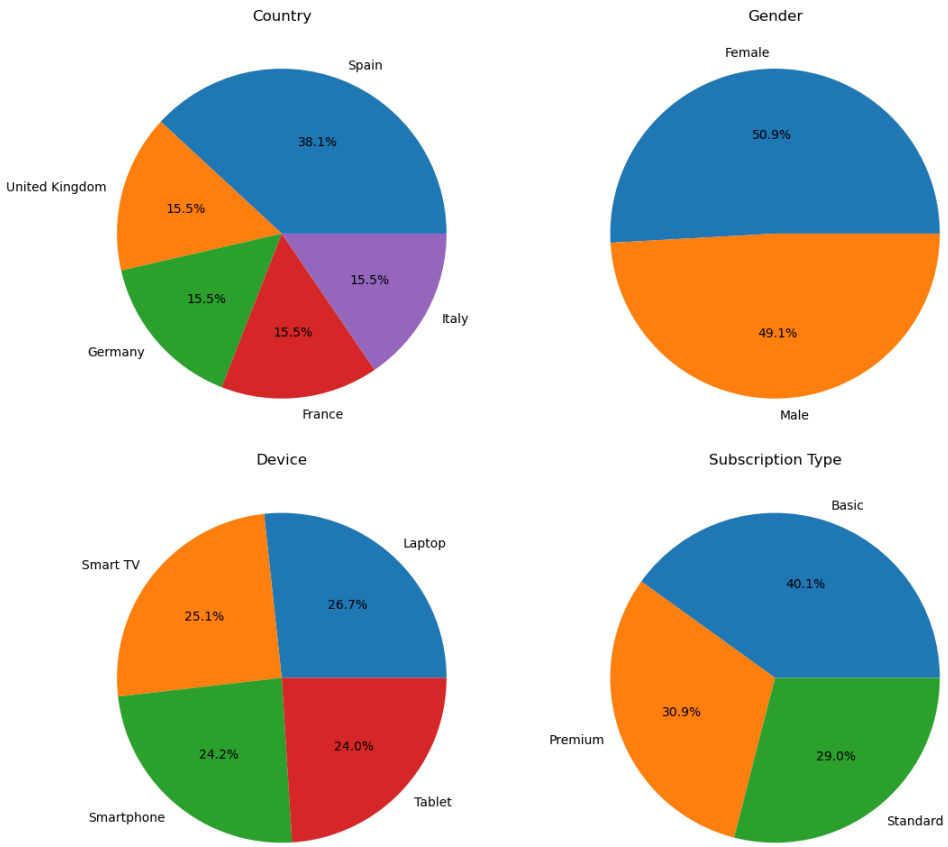
Subscription Type	Nombre	Pourcentage (%)
Basic	474	40.1
Premium	366	30.9
Standard	343	29.0

Et les quatre camemberts d'un seul coup :

```
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

for ax, colonne in zip(axes.flat, categorielles):
    df[colonne].value_counts().plot(kind='pie', autopct='%1.1f%%', ax=ax, ylabel='')
    ax.set_title(colonne)

plt.tight_layout()
plt.show()
```



L'Espagne concentre à elle seule 38 % des abonnés européens, les quatre autres pays se partageant le reste à parts presque égales. En revanche, sexe, appareil et formule d'abonnement sont proches de l'équilibre.

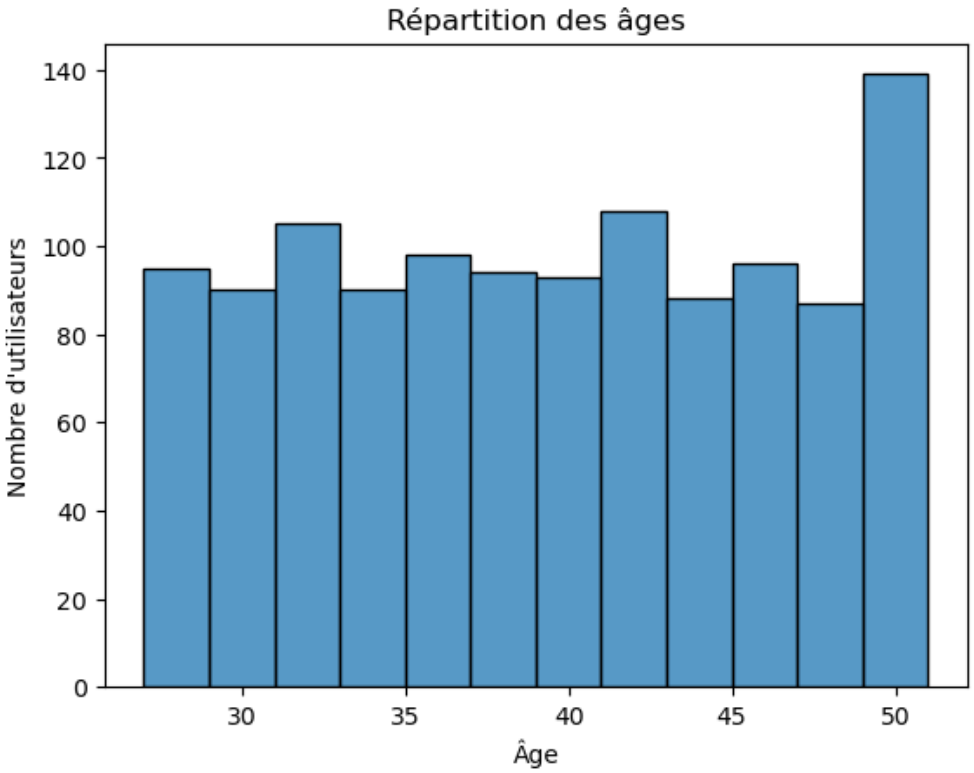
34.1.5. L'âge : une variable numérique

L'âge se traite **autrement**. `value_counts()` produirait ici un tableau d'une ligne par âge distinct : 25 lignes illisibles, où l'on ne verrait rien. Pour une variable numérique, on résume par ses statistiques et on visualise par un **histogramme**.

```
df['Age'].describe().round(1)
```

```
count    1183.0
mean       38.9
std         7.2
min        27.0
25%        33.0
50%        39.0
75%        45.0
max        51.0
Name: Age, dtype: float64
```

```
sns.histplot(data=df, x='Age', bins=12)
plt.title("Répartition des âges")
plt.xlabel("Âge"); plt.ylabel("Nombre d'utilisateurs")
plt.show()
```



Si l'on tient à un tableau, il faut d'abord **regrouper les âges en tranches** avec `pd.cut` :

```
df['tranche_age'] = pd.cut(df['Age'], bins=[25, 30, 35, 40, 45, 50, 55], right=False)

df['tranche_age'].value_counts().sort_index().to_frame('Nombre')
```

tranche_age	Nombre
[25, 30)	134
[30, 35)	246
[35, 40)	241
[40, 45)	240
[45, 50)	235
[50, 55)	87

Les abonnés ont entre 27 et 51 ans, assez uniformément répartis entre 30 et 50. Le jeu de données ne contient ni adolescents ni personnes âgées : encore un indice de son caractère synthétique.

34.1.6. Les dates d'inscription et de paiement

Les colonnes `Join Date` et `Last Payment Date` ont été lues comme du **texte** : telles quelles, on ne peut ni les comparer, ni en extraire un mois. Il faut les convertir.

```
display(Markdown(f"avant conversion : **{df['Join Date'].dtype}**"))
display(Markdown(f"exemples : **{df['Join Date'].head(3).tolist()}**"))
```

avant conversion : **object**

exemples : [**'28-02-23', '01-05-23', '18-03-22'**]

Le format `15-01-22` est ambigu : jour-mois-année ou mois-jour-année ? Ici c'est le premier, que l'on indique explicitement avec `format` : il vaut toujours mieux le préciser que laisser pandas deviner.

```
df['Join Date']         = pd.to_datetime(df['Join Date'],         format='%d-%m-%y')
df['Last Payment Date'] = pd.to_datetime(df['Last Payment Date'], format='%d-%m-%y')

display(Markdown(f"après conversion : **{df['Join Date'].dtype}**"))
df[['Join Date', 'Last Payment Date']].head(3)
```

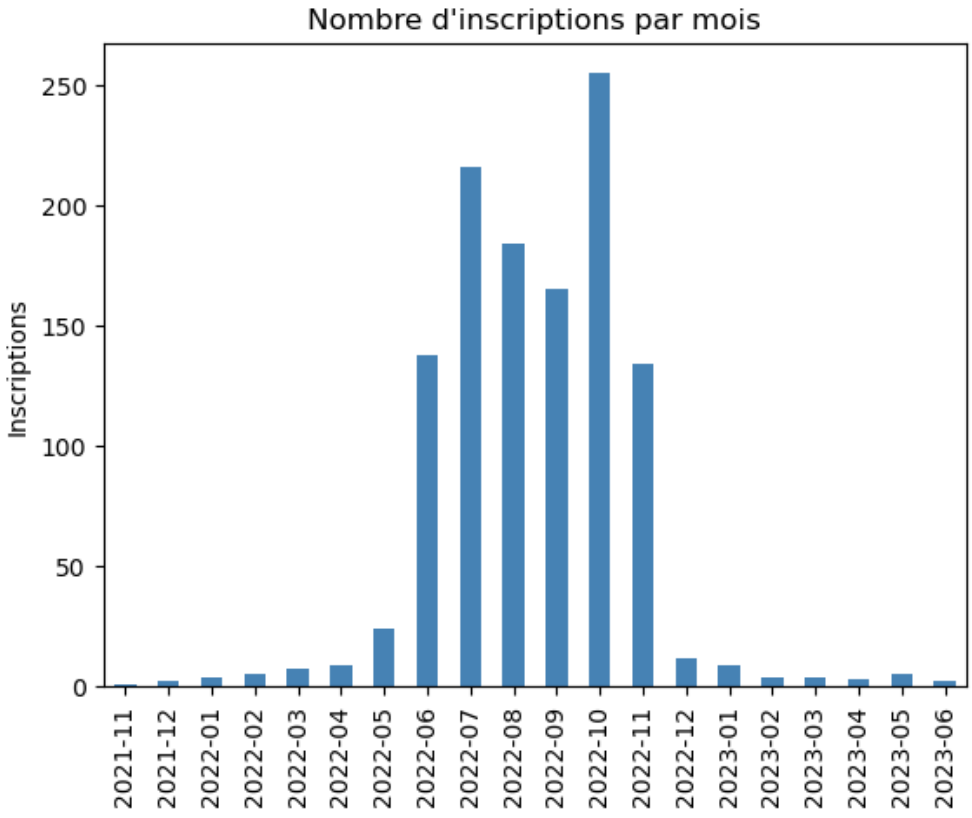
après conversion : **datetime64[ns]**

	Join Date	Last Payment Date
2	2023-02-28	2023-06-27
4	2023-05-01	2023-06-28
5	2022-03-18	2023-06-27

Une fois la colonne en type date, l'accesseur `.dt` donne accès à l'année, au mois, au jour de la semaine :

```
inscriptions = df['Join Date'].dt.to_period('M').value_counts().sort_index()

inscriptions.plot(kind='bar', color='steelblue')
plt.title("Nombre d'inscriptions par mois")
plt.xlabel(""); plt.ylabel("Inscriptions")
plt.show()
```



Les inscriptions se concentrent massivement entre juin et novembre 2022 ; avant et après, il n'y en a presque aucune. Là encore, aucune plateforme réelle ne recrute de cette façon : c'est la fenêtre de génération des données.

Enfin, la différence entre deux dates donne une durée, dont on extrait le nombre de jours avec `.dt.days` :

```
df['anciennete_jours'] = (df['Last Payment Date'] - df['Join Date']).dt.days

df['anciennete_jours'].describe().round(0)
```

```
count    1183.0
mean      307.0
std        65.0
min         8.0
25%       260.0
50%       307.0
75%       356.0
max       574.0
Name: anciennete_jours, dtype: float64
```

Nombre d'utilisateurs par type d'abonnement et genre

```
# pd.crosstab(df['Subscription Type'], df['Gender']) # tableau de contingence
# pd.crosstab(df['Subscription Type'], df['Gender'], normalize='index') # pourcentage par ligne
# pd.crosstab(df['Subscription Type'], df['Gender'], normalize='columns') # pourcentage par colonne
pd.crosstab(df['Subscription Type'], df['Gender'], normalize='all') # pourcentage total
```

Gender Subscription Type	Female	Male
Basic	0.207101	0.193576
Premium	0.161454	0.147929
Standard	0.140321	0.149620

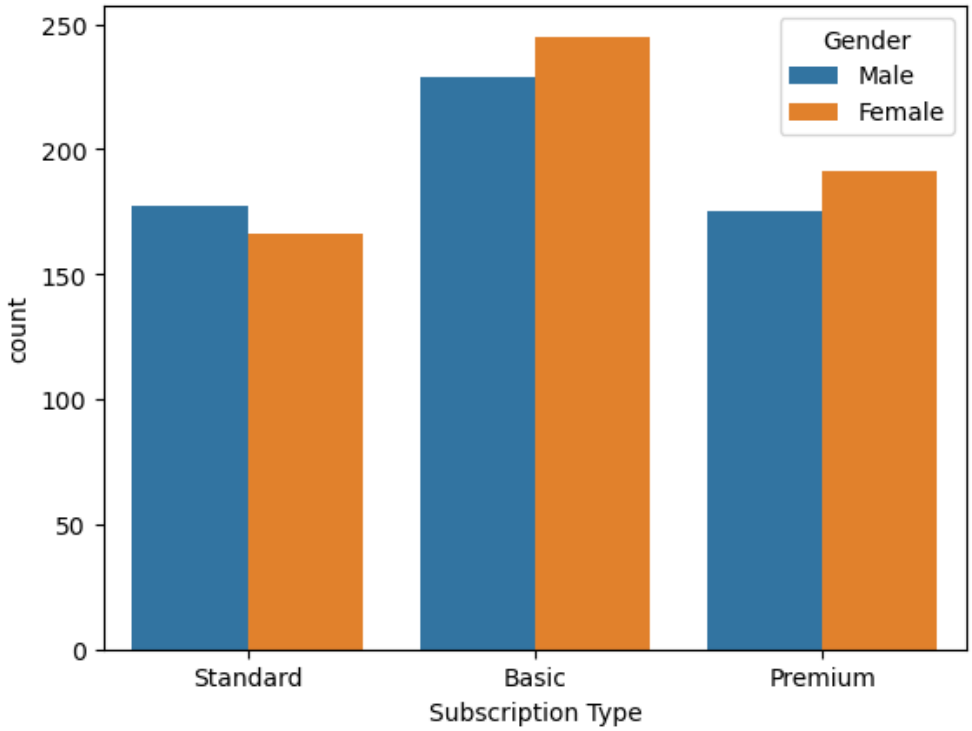
Type d'abonnement par pays, genre et device

```
pd.pivot_table(df, index='Country', columns=['Gender','Device'], values='Subscription Type',
↪  aggfunc='count')
```

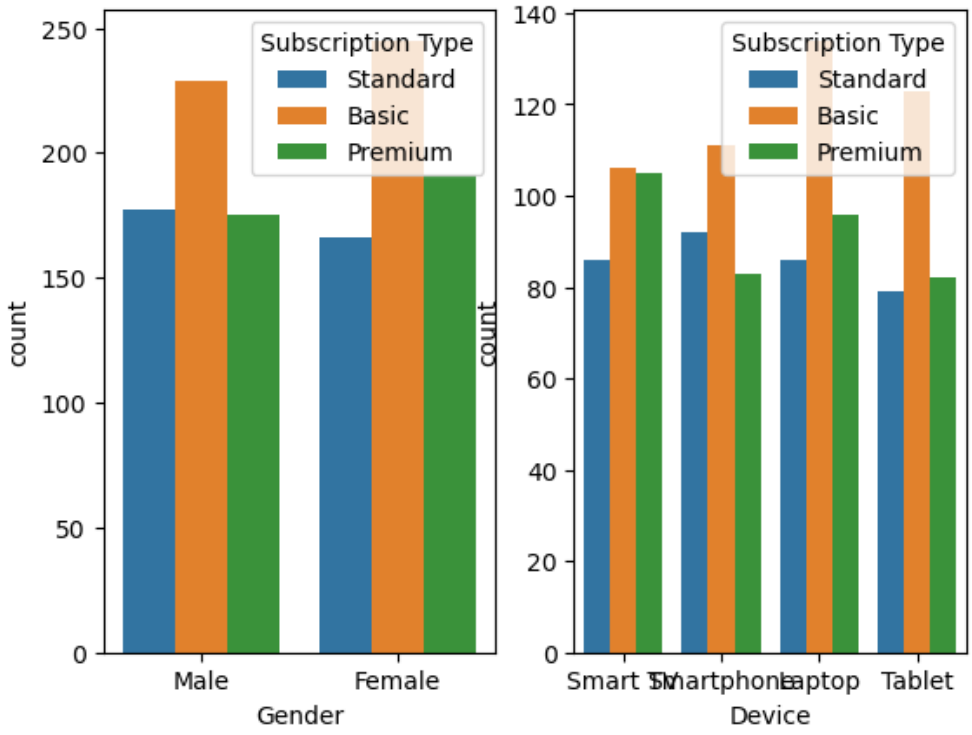
Gender Device Country	Female Laptop	Smart TV	Smartphone	Tablet	Male Laptop	Smart TV	Smartphone	Tablet
France	29	25	24	13	23	18	23	28
Germany	27	26	17	24	36	16	19	18
Italy	20	31	20	20	30	15	27	20
Spain	56	63	49	65	51	63	53	51
United Kingdom	22	16	27	28	22	24	27	17

Type d'abonnement par pays et genre

```
sns.countplot(data=df, x='Subscription Type', hue='Gender');
```




```
for i,c in enumerate(['Gender','Device']):
    plt.subplot(1,2,i+1)
    sns.countplot(data=df, x=c, hue='Subscription Type');
```



34.1.7. Peut-on tout corréler ? Un contre-exemple

Une idée séduisante pour repérer d'un coup d'œil quelles caractéristiques « vont ensemble » : convertir toutes les colonnes textuelles en nombres, puis calculer la matrice de corrélation. Essayons : et surtout, vérifions ce que le

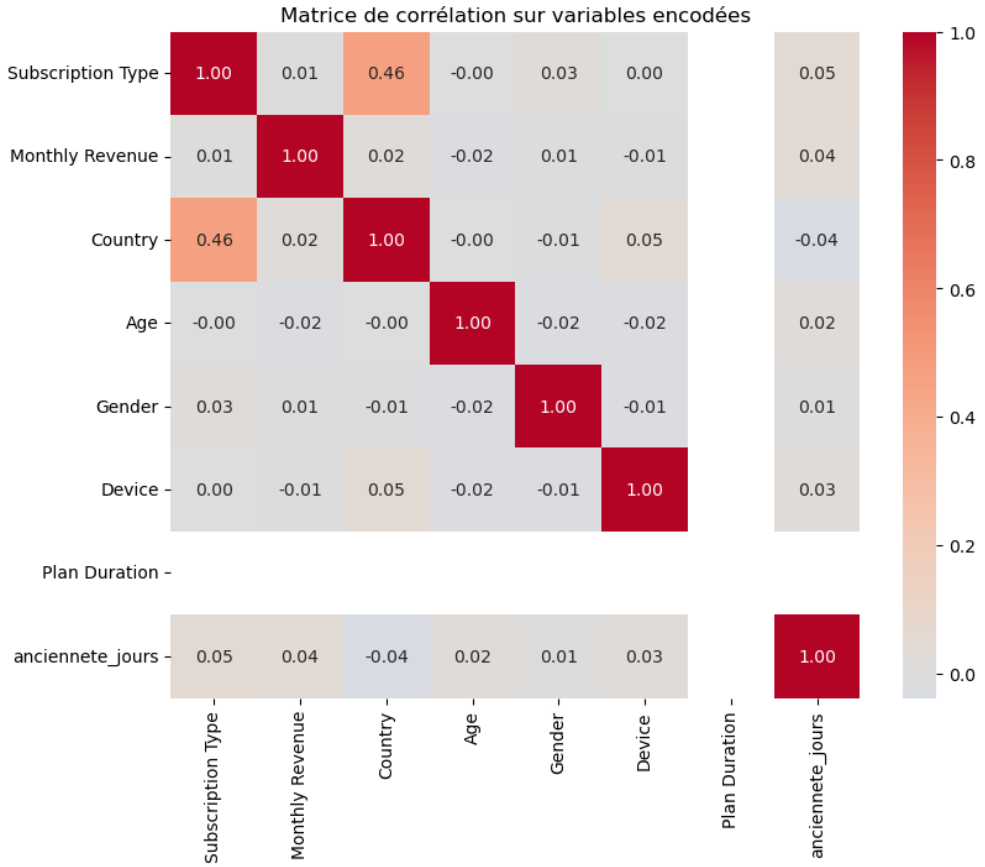
résultat vaut.

```
# On travaille sur une COPIE : l'encodage détruirait les libellés dont nous avons encore besoin
df_code = df.copy()

for col in df_code.select_dtypes(include='object').columns:
    df_code[col] = df_code[col].astype('category').cat.codes

# on ne garde que les colonnes numériques : les dates et la tranche d'âge
# (de type « catégorie ») n'entrent pas dans une matrice de corrélation
corr = df_code.select_dtypes(include='number').corr()

plt.figure(figsize=(9, 7))
sns.heatmap(corr, annot=True, fmt='.2f', cmap='coolwarm', center=0)
plt.title("Matrice de corrélation sur variables encodées")
plt.show()
```



Tous les coefficients sont proches de zéro. La conclusion semble sans appel : **aucune caractéristique n'est liée à une autre.**

Vérifions cette conclusion par un tableau croisé, en reprenant les libellés d'origine.

```
pd.crosstab(df['Subscription Type'], df['Country'], normalize='columns').mul(100).round(1)
```

Country Subscription Type	France	Germany	Italy	Spain	United Kingdom
Basic	19.7	81.4	96.2	24.4	1.6
Premium	80.3	1.6	2.2	47.0	0.0
Standard	0.0	16.9	1.6	28.6	98.4

La matrice se trompait complètement.

Le tableau croisé montre une association massive : **96 %** des abonnés italiens ont une formule *Basic*, **98 %** des britanniques une formule *Standard* et aucun une *Premium*, tandis qu'en France **80 %** sont en *Premium* et aucun en *Standard*. L'écart atteint **98 points de pourcentage** : le pays détermine presque entièrement le type d'abonnement.

Et la corrélation correspondante valait 0,02.

Pourquoi ? Le coefficient de Pearson mesure si deux *nombres* varient ensemble de façon **linéaire**. En codant les pays `France=0, Germany=1, Italy=2, Spain=3, United Kingdom=4`, on lui demande si le numéro de l'abonnement croît proportionnellement au numéro du pays : question dépourvue de sens, puisque l'ordre des pays est celui de l'alphabet. L'association est bien réelle, mais elle n'est pas linéaire dans un ordre arbitraire : le coefficient ne peut pas la voir.

Le piège est d'autant plus dangereux que le calcul **n'émet aucun avertissement** et produit un graphique d'apparence professionnelle.

Le problème touche même les variables **ordinales**, dont les modalités ont pourtant un ordre naturel. `cat.codes` classe par ordre alphabétique, ce qui donne ici `Basic=0, Premium=1, Standard=2` : la formule la plus chère se retrouve **au milieu**.

```
alphabetique = df['Subscription Type'].astype('category').cat.codes
naturel      = df['Subscription Type'].map({'Basic': 0, 'Standard': 1, 'Premium': 2})

display(Markdown(f"codage alphabétique (Basic, Premium, Standard) : r = {alphabetique.corr(df['Monthly
↪  Revenue']): .4f}"))
display(Markdown(f"ordre naturel       (Basic, Standard, Premium) : r = {naturel.corr(df['Monthly
↪  Revenue']): .4f}"))
```

codage alphabétique (Basic, Premium, Standard) : r = 0.0145

ordre naturel (Basic, Standard, Premium) : r = 0.0458

Le coefficient triple simplement parce qu'on a rangé les modalités dans le bon ordre. Si vous encodez une variable ordinale, faites-le **explicitement** avec un dictionnaire, jamais avec `cat.codes`.

34.1.8. La bonne façon de faire

Pour deux variables catégorielles, l'outil adapté est le **tableau croisé normalisé**, que nous avons déjà utilisé plus haut. Il se lit directement et ne repose sur aucun codage arbitraire.

```
for colonne in ['Device', 'Gender']:
    display(Markdown(f"**Type d'abonnement selon `{colonne}` (% par colonne)"))
    display(pd.crosstab(df['Subscription Type'], df[colonne], normalize='columns').mul(100).round(1))
```

Type d'abonnement selon `Device` (% par colonne)

Device Subscription Type	Laptop	Smart TV	Smartphone	Tablet
Basic	42.4	35.7	38.8	43.3
Premium	30.4	35.4	29.0	28.9
Standard	27.2	29.0	32.2	27.8

Type d'abonnement selon `Gender` (% par colonne)

Gender Subscription Type	Female	Male
Basic	40.7	39.4
Premium	31.7	30.1
Standard	27.6	30.5

Contrairement au pays, ni l'appareil ni le sexe ne structurent le type d'abonnement : d'une colonne à l'autre, le profil reste le même, environ 40 % de *Basic*, 30 % de *Premium*, 29 % de *Standard*. C'est précisément cette **stabilité d'une colonne à l'autre** qui signale l'absence de lien, et non le fait que les trois formules soient à égalité. Ici, l'absence de lien est une vraie conclusion, lisible sans ambiguïté.

Un mot sur ces données. Une répartition aussi tranchée : 98 % d'une seule formule dans un pays, 0 % d'une autre : ne s'observe jamais sur des abonnés réels. C'est un **artefact de génération** : le jeu de données est synthétique, comme l'indique l'avertissement en tête de ce notebook.

Conservez ce réflexe pour vos propres analyses : un résultat trop propre doit déclencher une vérification, pas une conclusion.

Chapitre 35.

Loi de Benford et suspicion de fraudes aux présidentielles des USA en 2016

Source : problème 9 de *"Python & Pandas et les 36 problèmes de Data Science"*, Frédéric Bro et Chantal Remy

La **loi de Benford** est une loi empirique qui concerne la distribution des premiers chiffres significatifs dans de nombreux ensembles de données réels. Elle prédit que **le premier chiffre significatif suit une distribution logarithmique,** c'est-à-dire que le chiffre 1 apparaît en premier 30% du temps, le chiffre 2 apparaît 17,6% du temps, le chiffre 3 apparaît 12,5% du temps, etc. Une loi analogue est valable pour le second chiffre significatif. De nos jours, pour vérifier l'authenticité des données, on peut vérifier si elles suivent la loi de Benford. Si ce n'est pas le cas, cela peut être un signe de fraude.

Lors des **élections présidentielles américaines de 2016,** des chercheurs ont appliqué la loi de Benford aux résultats des élections dans les différents États. Ils ont constaté que les résultats des élections dans certains États ne suivaient pas la loi de Benford et ont suggéré que cela pourrait être dû à des fraudes électorales. Ceci incita les candidats Hilary Clinton et Jill Stein à demander un comptage manuel dans certains comtés.

Dans cet exercice, vous allez vérifier si les résultats des élections dans les différents États des États-Unis en 2016 suivent la loi de Benford. Pour cela, vous allez utiliser les données des élections présidentielles américaines de 2016, disponibles dans le fichier `benford_usa.csv`.

Quelques liens pour en savoir plus sur la loi de Benford :

- benfordonline
- testingbenfordslaw
- wikipedia

35.1. Exercice

1. **Loi de Benford**
   Tracer la fréquence relative du **deuxième** chiffre significatif $c \in \{0, 1, \dots, 9\}$ dans la représentation en base 10 prédite par la loi de Benford suivante :

$$p(c) = \sum_{i=1}^{9} \log_{10}\left(1 + \frac{1}{10i + c}\right).$$

2. **Ouverture et sélection des données**

- Le fichier `benford_usa.csv` contient le nombre de voix remportées par trois candidats dans chaque **district** des États-Unis. Importer le fichier et analyser brièvement les données.
  Bonus : retrouver le **résultat** des élections en combinant avec le **nombre** de grands électeurs par **État** (cf. Wikipedia).
- Ne garder ensuite que les lignes correspondant aux **comtés** du **Michigan**, de la **Pennsylvanie** et du **Wisconsin**. Parmi ces lignes, ne garder que celles où chaque candidat **a obtenu** plus de 10 voix. Ce sont ces données que nous allons analyser.

3. **Analyse des données**
   En théorie, la répartition des fréquences du **deuxième** chiffre des nombres de voix suit la **loi de Benford**. Pour chaque candidat, vérifier si c'est le cas.

   - Calculer la fréquence du **deuxième** chiffre des nombres de voix pour chaque candidat et comparer avec la loi de Benford (créer un **DataFrame** où chaque ligne correspond à un chiffre et chaque colonne à un candidat ; ajouter une colonne pour les **fréquences théoriques**).

   - Afficher/visualiser la distribution de chaque colonne.
4. **Bonus : simulation pour la détection d'éventuelles anomalies**
   Les graphes peuvent susciter des doutes quant à la **régularité** des résultats. Pour aller plus loin, introduire un indicateur **synthétique** pour résumer l'analyse.

   - Calculer la **distance euclidienne** entre la colonne **théorique** et la colonne **observée** pour chaque candidat.

   - Simuler 10^4 votations qui suivent la **distribution théorique** de Benford et vérifier si ce résultat est effectivement **rare** ou non.

Importation des bibliothèques nécessaires.

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
plt.style.use('bmh')
import seaborn as sns
import plotly.express as px

from IPython.display import display, Markdown

# Pour la génération automatique de graphiques interactifs dans le polycopié
import plotly.io as pio
import os
if os.getenv("QUARTO_OUTPUT_FORMAT") is not None:
    # Export Quarto HTML
    pio.renderers.default = "iframe_connected"
else:
    # Notebook interactif
    pio.renderers.default = "notebook_connected"
```

35.2. Loi de Benford

La fonction théorique p qui donne la fréquence du second chiffre significatif dans la représentation en base 10 prédite par la loi de Benford est donnée par :

$$p(c) = \sum_{i=1}^{9} \log_{10}\left(1 + \frac{1}{10i + c}\right).$$

```
# Fonction pour calculer la probabilité
p = lambda c : sum( [ np.log10( 1+1/(10*i+c)) for i in range( 1 , 10 ) ] )

# Générer les chiffres de 0 à 9
cc = np.arange(0,10)

# Calculer les probabilités pour chaque valeur de cc
FB = [ p(c) for c in cc ]

# Affichage des résultats dans un DataFrame, utiliser la première colonne comme index
df = pd.DataFrame({'Chiffre':cc,'Probabilité':FB})
display(df)
```

```
# Vérification que FB est normalisée
display(df['Probabilité'].sum())

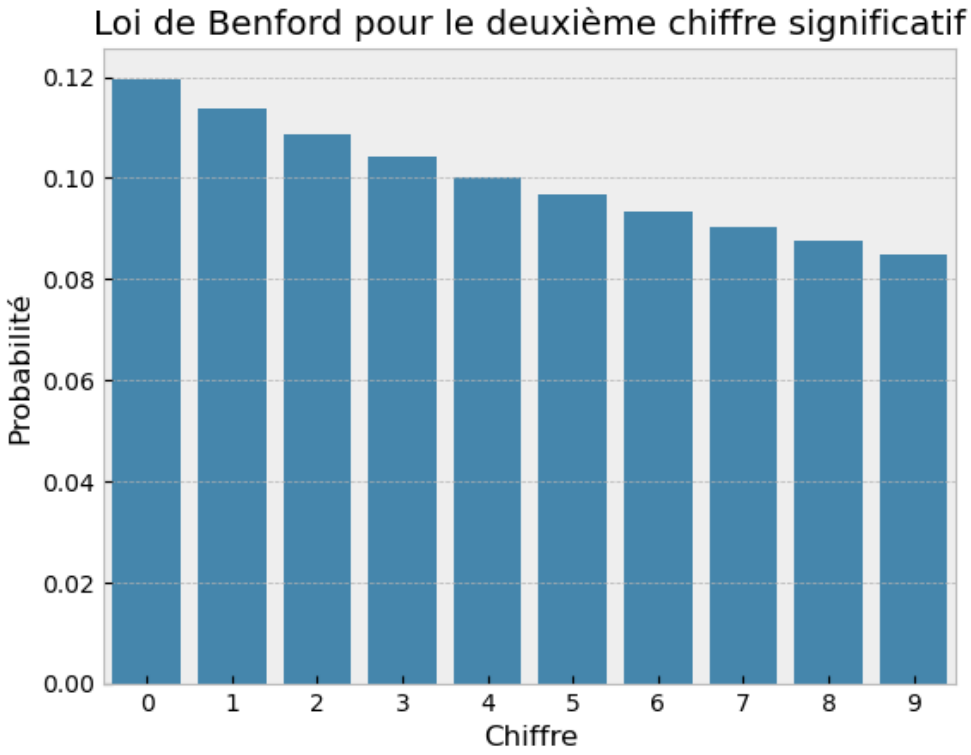


# Tracer les résultats avec seaborn
sns.barplot(x='Chiffre',y='Probabilité',data=df)
plt.title("Loi de Benford pour le deuxième chiffre significatif")
plt.show()

# Tracer les résultats avec plotly
fig = px.bar(df, x='Chiffre', y='Probabilité', title="Loi de Benford pour le deuxième chiffre
 ↪ significatif")
fig.show()
```

	Chiffre	Probabilité
0	0	0.119679
1	1	0.113890
2	2	0.108821
3	3	0.104330
4	4	0.100308
5	5	0.096677
6	6	0.093375
7	7	0.090352
8	8	0.087570
9	9	0.084997

```
0.9999999999999997
```




```
Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): text/html
```

35.3. Ouverture et séléction des données

On importe les données à partir d'un fichier `csv`. Par défaut, le séparateur d'éléments est `,` . Dans ce fichier c'est `;`, il faut donc l'indiquer explicitement.

```
Tot = pd.read_csv('benford_usa.csv', sep=';')
display(Tot.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3143 entries, 0 to 3142
Data columns (total 7 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   State    3143 non-null   object
 1   ST       3143 non-null   object
 2   Fips     3143 non-null   int64 
 3   County   3143 non-null   object
 4   Trump    3143 non-null   int64 
 5   Clinton  3143 non-null   int64 
 6   Stein    3143 non-null   int64 
dtypes: int64(4), object(3)
memory usage: 172.0+ KB

None
```

```
Tot.head(10)
```

	State	ST	Fips	County	Trump	Clinton	Stein
0	Georgia	GA	13089	DeKalb County, Georgia	47531	239131	0
1	Texas	TX	48487	Wilbarger County, Texas	3166	807	13
2	Virginia	VA	51111	Lunenburg County, Virginia	3206	2226	25
3	Georgia	GA	13297	Walton County, Georgia	31093	8279	0
4	North Carolina	NC	37011	Avery County, North Carolina	6226	1670	0
5	North Carolina	NC	37069	Franklin County, North Carolina	16320	12811	0
6	Michigan	MI	26037	Clinton County, Michigan	21635	16490	379
7	Arkansas	AR	5045	Faulkner County, Arkansas	29170	14525	507
8	North Carolina	NC	37025	Cabarrus County, North Carolina	53224	35048	0
9	Idaho	ID	16025	Camas County, Idaho	410	110	15

Combien de valeurs uniques y a-t-il dans la colonne `State` ? Et dans la colonne `County` ? Ou encore dans la colonne `Fips` ?

```
for col in ['State', 'County', 'Fips']:
    display(Markdown(f"**Nombre de valeurs uniques dans la colonne `{col}` :** {Tot[col].nunique()}"))
```

Nombre de valeurs uniques dans la colonne `State` : 51

Nombre de valeurs uniques dans la colonne `County` : 3143

Nombre de valeurs uniques dans la colonne `Fips` : 3143

Les colonnes "Fips" et "County" ne sont pas utiles pour notre analyse, on peut les supprimer.

```
Tot.drop(columns=['Fips', 'County'],inplace=True)
Tot
```

	State	ST	Trump	Clinton	Stein
0	Georgia	GA	47531	239131	0
1	Texas	TX	3166	807	13
2	Virginia	VA	3206	2226	25
3	Georgia	GA	31093	8279	0
4	North Carolina	NC	6226	1670	0
...	...	...	...	...	...
3138	Tennessee	TN	9525	2425	36
3139	Michigan	MI	224589	176238	3886
3140	West Virginia	WV	3516	1315	47
3141	North Carolina	NC	6225	1585	0
3142	Maryland	MD	7938	5695	94

Combien d'états sont dans le fichier ? Lesquels ? Affichons la liste des états (sans doublons) :

```
# Résumé statistique de la colonne 'State'
# ================================================================

colonne = Tot['State']

display(Markdown("**Nombre de valeurs uniques = nombre d'États**"))
display(colonne.nunique())

# display(Markdown("**Valeurs uniques = noms des États**"))
# display(colonne.unique())

display(Markdown("**Combien d'occurrences de chaque valeur unique = combien de fois chaque État
 ↪  apparaît**"))
display(pd.DataFrame({
    'Occurrences': colonne.value_counts(),
    'Proportions': colonne.value_counts(normalize=True)
}))
```

Nombre de valeurs uniques = nombre d'États

```
51
```

Combien d'occurrences de chaque valeur unique = combien de fois chaque État apparaît

State	Occurrences	Proportions
Texas	254	0.080815
Georgia	159	0.050589
Virginia	134	0.042634
Kentucky	120	0.038180
Missouri	115	0.036589
Kansas	105	0.033408
Illinois	102	0.032453
North Carolina	100	0.031817
Iowa	99	0.031499
Tennessee	95	0.030226
Nebraska	93	0.029590
Indiana	92	0.029271
Ohio	88	0.027999
Minnesota	87	0.027681
Michigan	83	0.026408

State	Occurrences	Proportions
Mississippi	82	0.026090
Oklahoma	77	0.024499
Arkansas	75	0.023863
Wisconsin	72	0.022908
Pennsylvania	67	0.021317
Florida	67	0.021317
Alabama	67	0.021317
South Dakota	66	0.020999
Louisiana	64	0.020363
Colorado	64	0.020363
New York	62	0.019726
California	58	0.018454
Montana	56	0.017817
West Virginia	55	0.017499
North Dakota	53	0.016863
South Carolina	46	0.014636
Idaho	44	0.013999
Washington	39	0.012409
Oregon	36	0.011454
New Mexico	33	0.010500
Utah	29	0.009227
Alaska	29	0.009227
Maryland	24	0.007636
Wyoming	23	0.007318
New Jersey	21	0.006682
Nevada	17	0.005409
Maine	16	0.005091
Arizona	15	0.004773
Massachusetts	14	0.004454
Vermont	14	0.004454
New Hampshire	10	0.003182
Connecticut	8	0.002545
Hawaii	5	0.001591
Rhode Island	5	0.001591
Delaware	3	0.000955
District of Columbia	1	0.000318

Combien de voix ont été obtenues par chaque candidat au total ?

```
# On affiche le nombre total de votes pour chaque candidat, i.e. la somme des colonnes Trump, Clinton et
↪ Stein
cols = ['Trump','Clinton','Stein']

pour_affichage = pd.DataFrame( Tot[cols].sum().sort_values(ascending=False),columns=['Nombre de votes'])
pour_affichage['Pourcentage'] = pour_affichage / pour_affichage.sum() * 100

# pour un affichage plus joli
pour_affichage = pour_affichage.style.format({'Pourcentage':'{:.2f} %'})
pour_affichage
```

Table 35.5.

	Nombre de votes	Pourcentage
Clinton	62426228	50.02 %

	Nombre de votes	Pourcentage
Trump	61064602	48.93 %
Stein	1311595	1.05 %

35.4. Bonus : pourquoi une fraude éventuelle dans ces trois états aurait changé le résultat de l'élection ?

Aux USA, ce n'est pas la majorité qui gagne mais le candidat qui obtient le plus de **grands électeurs**. Clinton a obtenu plus de voix que Trump mais c'est ce dernier qui a été élu. En effet, chaque **État** a un nombre de grands électeurs qui lui est attribué et le candidat qui remporte la majorité des voix dans un État obtient **tous** les grands électeurs de cet État.

Voici le bilan des voix puis des grands électeurs obtenus par chaque candidat :

```
# Agréger les votes par état
# votes_by_state = Tot.pivot_table(index='State', values=['Trump', 'Clinton', 'Stein'], aggfunc='sum')

# Pour garder la colonne 'ST' (code état) on utilise groupby avec agg
votes_by_state = Tot.groupby('State')[['Trump', 'Clinton', 'Stein', 'ST']].agg({
    'Trump': 'sum',
    'Clinton': 'sum',
    'Stein': 'sum',
    'ST': 'first'
})

votes_by_state
```

State	Trump	Clinton	Stein	ST
Alabama	1306925	718084	9287	AL
Alaska	0	0	0	AK
Arizona	1021154	936250	25255	AZ
Arkansas	677904	378729	9837	AR
California	3916209	7362490	220312	CA
Colorado	1137455	1212209	33147	CO
Connecticut	668266	884432	22793	CT
Delaware	185103	235581	6100	DE
District of Columbia	11553	260223	3995	DC
Florida	4605515	4485745	64019	FL
Georgia	2068623	1837300	0	GA
Hawaii	128815	266827	12727	HI
Idaho	407199	189677	8464	ID
Illinois	2118179	2977498	74112	IL
Indiana	1556220	1031953	0	IN
Iowa	798923	650790	11119	IA
Kansas	656009	414788	22698	KS
Kentucky	1202942	628834	13913	KY
Louisiana	1178004	779535	14018	LA
Maine	334838	354873	14075	ME
Maryland	873646	1497951	31839	MD
Massachusetts	1083069	1964768	46910	MA
Michigan	2279805	2268193	50700	MI
Minnesota	1322891	1366676	36957	MN
Mississippi	678457	462001	3580	MS
Missouri	1585753	1054889	25086	MO

State	Trump	Clinton	Stein	ST
Montana	274120	174521	7669	MT
Nebraska	485819	273858	8346	NE
Nevada	511319	537753	0	NV
New Hampshire	345789	348521	6416	NH
New Jersey	1535513	2021756	35949	NJ
New Mexico	315875	380724	9729	NM
New York	2640570	4143874	99895	NY
North Carolina	2339603	2162074	0	NC
North Dakota	216133	93526	3769	ND
Ohio	2771984	2317001	44310	OH
Oklahoma	947934	419788	0	OK
Oregon	742506	934631	45132	OR
Pennsylvania	2912941	2844705	48912	PA
Rhode Island	179421	249902	6155	RI
South Carolina	1143611	849469	12917	SC
South Dakota	227460	114938	0	SD
Tennessee	1517402	867110	15919	TN
Texas	4681590	3867816	71307	TX
Utah	452086	274188	7695	UT
Vermont	95053	178179	6748	VT
Virginia	1731156	1916845	27272	VA
Washington	1129120	1610524	51066	WA
West Virginia	486198	187457	8000	WV
Wisconsin	1403694	1380823	30934	WI
Wyoming	174248	55949	2512	WY

On constate que, pour l'**Alaska**, les résultats des trois candidats (Trump, Clinton et Stein) sont tous à 0 ; cela signifie qu'il n'y a pas de votes enregistrés pour **cet État** dans le fichier. Y a-t-il d'autres **États** pour lesquels les résultats sont manquants ? Si oui, lesquels ? Pour détecter ces États, on peut filtrer les lignes où les **totaux** de votes pour ces candidats sont **égaux à zéro**.

```
# Filtrer les lignes où les votes sont tous à zéro pour Trump, Clinton et Stein
# On utilise sum(axis=1) qui calcule la somme des votes pour chaque ligne (chaque état)
mask = votes_by_state[['Trump', 'Clinton', 'Stein']].sum(axis=1) == 0
states_with_no_votes = votes_by_state[mask]
# Afficher les états avec 0 votes
states_with_no_votes
```

State	Trump	Clinton	Stein	ST
Alaska	0	0	0	AK

Nous constatons que les résultats pour 1 état sont absents dans notre jeu de données. Cependant, nous poursuivrons l'analyse en supposant que toutes les données nécessaires sont présentes, sans tenir compte de cette omission.

Ajoutons maintenant le nombre de grands électeurs par état. Pour cela, nous allons utiliser un dictionnaire qui contient le nombre de grands électeurs par état.

```
# Dictionnaire des grands électeurs par état en 2016
electoral_votes = {
    "Georgia": 16,
    "Texas": 38,
    "Virginia": 13,
    "North Carolina": 15,
```

```
    "Michigan": 16,
    "Arkansas": 6,
    "Idaho": 4,
    "Florida": 29,
    "Tennessee": 11,
    "Kentucky": 8,
    "Alabama": 9,
    "Colorado": 9,
    "Missouri": 10,
    "Ohio": 18,
    "Pennsylvania": 20,
    "South Dakota": 3,
    "Louisiana": 8,
    "Wisconsin": 10,
    "Minnesota": 10,
    "Indiana": 11,
    "Maryland": 10,
    "Kansas": 6,
    "New York": 29,
    "Mississippi": 6,
    "North Dakota": 3,
    "Iowa": 6,
    "California": 55,
    "Wyoming": 3,
    "New Mexico": 5,
    "Montana": 3,
    "Utah": 6,
    "Oklahoma": 7,
    "Alaska": 3,
    "Oregon": 7,
    "Illinois": 20,
    "Hawaii": 4,
    "South Carolina": 9,
    "West Virginia": 5,
    "Arizona": 11,
    "Maine": 4,
    "New Jersey": 14,
    "Connecticut": 7,
    "Nebraska": 5,
    "Washington": 12,
    "Nevada": 6,
    "Massachusetts": 11,
    "Rhode Island": 4,
    "New Hampshire": 4,
    "Delaware": 3,
    "Vermont": 3,
    "District of Columbia": 3,
}


# Ajouter les grands électeurs au DataFrame
votes_by_state['ElectoralVotes'] = electoral_votes
votes_by_state
```

State	Trump	Clinton	Stein	ST	ElectoralVotes
Alabama	1306925	718084	9287	AL	9
Alaska	0	0	0	AK	3
Arizona	1021154	936250	25255	AZ	11
Arkansas	677904	378729	9837	AR	6
California	3916209	7362490	220312	CA	55

State	Trump	Clinton	Stein	ST	ElectoralVotes
Colorado	1137455	1212209	33147	CO	9
Connecticut	668266	884432	22793	CT	7
Delaware	185103	235581	6100	DE	3
District of Columbia	11553	260223	3995	DC	3
Florida	4605515	4485745	64019	FL	29
Georgia	2068623	1837300	0	GA	16
Hawaii	128815	266827	12727	HI	4
Idaho	407199	189677	8464	ID	4
Illinois	2118179	2977498	74112	IL	20
Indiana	1556220	1031953	0	IN	11
Iowa	798923	650790	11119	IA	6
Kansas	656009	414788	22698	KS	6
Kentucky	1202942	628834	13913	KY	8
Louisiana	1178004	779535	14018	LA	8
Maine	334838	354873	14075	ME	4
Maryland	873646	1497951	31839	MD	10
Massachusetts	1083069	1964768	46910	MA	11
Michigan	2279805	2268193	50700	MI	16
Minnesota	1322891	1366676	36957	MN	10
Mississippi	678457	462001	3580	MS	6
Missouri	1585753	1054889	25086	MO	10
Montana	274120	174521	7669	MT	3
Nebraska	485819	273858	8346	NE	5
Nevada	511319	537753	0	NV	6
New Hampshire	345789	348521	6416	NH	4
New Jersey	1535513	2021756	35949	NJ	14
New Mexico	315875	380724	9729	NM	5
New York	2640570	4143874	99895	NY	29
North Carolina	2339603	2162074	0	NC	15
North Dakota	216133	93526	3769	ND	3
Ohio	2771984	2317001	44310	OH	18
Oklahoma	947934	419788	0	OK	7
Oregon	742506	934631	45132	OR	7
Pennsylvania	2912941	2844705	48912	PA	20
Rhode Island	179421	249902	6155	RI	4
South Carolina	1143611	849469	12917	SC	9
South Dakota	227460	114938	0	SD	3
Tennessee	1517402	867110	15919	TN	11
Texas	4681590	3867816	71307	TX	38
Utah	452086	274188	7695	UT	6
Vermont	95053	178179	6748	VT	3
Virginia	1731156	1916845	27272	VA	13
Washington	1129120	1610524	51066	WA	12
West Virginia	486198	187457	8000	WV	5
Wisconsin	1403694	1380823	30934	WI	10
Wyoming	174248	55949	2512	WY	3

Éliminons la ligne correspondant à l'Alaska, car elle n'a pas de données de vote.

```
votes_by_state = votes_by_state.drop('Alaska', errors='ignore')
```

```
# Trouver le gagnant pour chaque état
votes_by_state['Winner'] = votes_by_state[['Trump', 'Clinton', 'Stein']].idxmax(axis=1)
votes_by_state
```

State	Trump	Clinton	Stein	ST	ElectoralVotes	Winner
Alabama	1306925	718084	9287	AL	9	Trump
Arizona	1021154	936250	25255	AZ	11	Trump
Arkansas	677904	378729	9837	AR	6	Trump
California	3916209	7362490	220312	CA	55	Clinton
Colorado	1137455	1212209	33147	CO	9	Clinton
Connecticut	668266	884432	22793	CT	7	Clinton
Delaware	185103	235581	6100	DE	3	Clinton
District of Columbia	11553	260223	3995	DC	3	Clinton
Florida	4605515	4485745	64019	FL	29	Trump
Georgia	2068623	1837300	0	GA	16	Trump
Hawaii	128815	266827	12727	HI	4	Clinton
Idaho	407199	189677	8464	ID	4	Trump
Illinois	2118179	2977498	74112	IL	20	Clinton
Indiana	1556220	1031953	0	IN	11	Trump
Iowa	798923	650790	11119	IA	6	Trump
Kansas	656009	414788	22698	KS	6	Trump
Kentucky	1202942	628834	13913	KY	8	Trump
Louisiana	1178004	779535	14018	LA	8	Trump
Maine	334838	354873	14075	ME	4	Clinton
Maryland	873646	1497951	31839	MD	10	Clinton
Massachusetts	1083069	1964768	46910	MA	11	Clinton
Michigan	2279805	2268193	50700	MI	16	Trump
Minnesota	1322891	1366676	36957	MN	10	Clinton
Mississippi	678457	462001	3580	MS	6	Trump
Missouri	1585753	1054889	25086	MO	10	Trump
Montana	274120	174521	7669	MT	3	Trump
Nebraska	485819	273858	8346	NE	5	Trump
Nevada	511319	537753	0	NV	6	Clinton
New Hampshire	345789	348521	6416	NH	4	Clinton
New Jersey	1535513	2021756	35949	NJ	14	Clinton
New Mexico	315875	380724	9729	NM	5	Clinton
New York	2640570	4143874	99895	NY	29	Clinton
North Carolina	2339603	2162074	0	NC	15	Trump
North Dakota	216133	93526	3769	ND	3	Trump
Ohio	2771984	2317001	44310	OH	18	Trump
Oklahoma	947934	419788	0	OK	7	Trump
Oregon	742506	934631	45132	OR	7	Clinton
Pennsylvania	2912941	2844705	48912	PA	20	Trump
Rhode Island	179421	249902	6155	RI	4	Clinton
South Carolina	1143611	849469	12917	SC	9	Trump
South Dakota	227460	114938	0	SD	3	Trump
Tennessee	1517402	867110	15919	TN	11	Trump
Texas	4681590	3867816	71307	TX	38	Trump
Utah	452086	274188	7695	UT	6	Trump
Vermont	95053	178179	6748	VT	3	Clinton
Virginia	1731156	1916845	27272	VA	13	Clinton
Washington	1129120	1610524	51066	WA	12	Clinton
West Virginia	486198	187457	8000	WV	5	Trump
Wisconsin	1403694	1380823	30934	WI	10	Trump
Wyoming	174248	55949	2512	WY	3	Trump

Qui a obtenu le plus d'états indépendamment du nombre total de voix ou de grands électeurs ?

```
votes_by_state['Winner'].value_counts()
```

```
Winner
Trump      29
Clinton    21
Name: count, dtype: int64
```

Bonus : on peut afficher sur une carte choroplèthe le résultat des élections. On colore chaque État selon le candidat qui a remporté le plus de voix (en effet, aux États-Unis, le président est élu par un collège électoral où chaque État dispose d'un certain nombre de grands électeurs, en général proportionnel au nombre d'habitants).

```
# Transformer l'index en colonne pour plotly
votes_map = votes_by_state.reset_index()

# Fonction formatage français pour l'affichage des nombres avec des espaces comme séparateurs de milliers
fmt = lambda n: f"{n:,}".replace(",", " ")

# Carte choroplèthe
fig = px.choropleth(
    votes_map,
    locations='ST',
    locationmode='USA-states',
    color='Winner',
    hover_name='State',
    hover_data={
        'Winner': True,
        'Grandes Electeurs': votes_map['ElectoralVotes'].apply(fmt),
        'Trump votes': votes_map['Trump'].apply(fmt),
        'Clinton votes': votes_map['Clinton'].apply(fmt),
        'Stein votes': votes_map['Stein'].apply(fmt),
        'ST': False,
    },
    color_discrete_map={'Trump':'red','Clinton':'blue','Stein':'green'},
    scope='usa',
    title='Résultats des élections US 2016 par État'
)

fig.show()
```

```
Unable to display output for mime type(s): text/html
```

On regarde juste les trois lignes associées aux états du Michigan, de Pennsylvanie et du Wisconsin.

```
votes_by_state.loc[['Michigan', 'Wisconsin', 'Pennsylvania']]
```

State	Trump	Clinton	Stein	ST	ElectoralVotes	Winner
Michigan	2279805	2268193	50700	MI	16	Trump
Wisconsin	1403694	1380823	30934	WI	10	Trump
Pennsylvania	2912941	2844705	48912	PA	20	Trump

```
# Résultats finaux
final_results = votes_by_state.pivot_table(index='Winner', values='ElectoralVotes', aggfunc='sum')

# Ajouter le nombre de votes pour chaque candidat, le pourcentage de grands électeurs et le pourcentage
 ↪ de votes
final_results.insert(0, 'Votes', votes_by_state[['Trump', 'Clinton', 'Stein']].sum())
final_results.insert(1, 'PercentageVotes' , final_results['Votes'] / final_results['Votes'].sum() * 100)
final_results['PercentageElectoralVotes'] =  final_results['ElectoralVotes'] /
 ↪ final_results['ElectoralVotes'].sum() * 100
final_results
```

Winner	Votes	PercentageVotes	ElectoralVotes	PercentageElectoralVotes
Clinton	62426228	50.551307	233	43.551402
Trump	61064602	49.448693	302	56.448598

Sur Wikipedia on peut trouver les résultats suivants pour les élections :

- Hillary Clinton : 65 853 514 voix, 232 grands électeurs
- Donald Trump : 62 984 828 voix, 306 grands électeurs

Nos données ne sont donc pas complètes.

On voit que Trump a obtenu 305 grands électeurs contre 233 pour Clinton. Il a donc été élu président des USA. Cependant, si les résultats dans les états du Michigan, de Pennsylvanie et du Wisconsin avaient été différents, Clinton aurait obtenu plus de grands électeurs (et Trump moins). Aurait-elle été élue ?

```
# Calcul de la somme des grands électeurs des états impliqués dans la triche éventuelle
si_triche_verifiee = electoral_votes['Michigan'] + electoral_votes['Wisconsin'] +
 ↪  electoral_votes['Pennsylvania']

# Ajuster les résultats de Trump et Clinton
new_Trump = final_results.loc['Trump','ElectoralVotes'] - si_triche_verifiee  # Déduire les grands
 ↪  électeurs pour Trump
new_Clinton = final_results.loc['Clinton','ElectoralVotes'] + si_triche_verifiee  # Ajouter les grands
 ↪  électeurs à Clinton

# Afficher les résultats ajustés
display(Markdown("**Résultats ajustés si la triche est confirmée :**"))
display(Markdown(f"- Trump : {new_Trump} grands électeurs"))
display(Markdown(f"- Clinton : {new_Clinton} grands électeurs"))
```

Résultats ajustés si la triche est confirmée :

- Trump : 256 grands électeurs
- Clinton : 279 grands électeurs

35.5. Sélection des données dans les États du Michigan, de la Pennsylvanie et du Wisconsin

Nous allons examiner les données relatives aux **États** du **Michigan**, du **Wisconsin** et de la **Pennsylvanie**. Nous nous intéressons aux **comtés** où **chaque candidat** a obtenu **plus de 10 voix**.

```
mask1 = (Tot['State'] == 'Michigan') | (Tot['State'] == 'Wisconsin') | (Tot['State'] == 'Pennsylvania')
# mask1 = Tot['State'].isin(['Michigan', 'Wisconsin', 'Pennsylvania'])

mask2 = (Tot['Trump'] >= 10) & (Tot['Clinton'] >= 10) & (Tot['Stein'] >= 10)

# Nouveau DataFrame avec les lignes sélectionnées
Sel = Tot[mask1 & mask2].copy()
Sel
```

	State	ST	Trump	Clinton	Stein
6	Michigan	MI	21635	16490	379
12	Michigan	MI	2158	1156	49
20	Pennsylvania	PA	22676	6849	136
26	Wisconsin	WI	17310	18524	583

	State	ST	Trump	Clinton	Stein
33	Wisconsin	WI	31044	17391	449
...	...	...	...	...	...
3057	Wisconsin	WI	39010	26476	641
3090	Pennsylvania	PA	257488	363017	5021
3105	Wisconsin	WI	46620	42506	834
3125	Michigan	MI	7228	3973	116
3139	Michigan	MI	224589	176238	3886

Combien de bureaux de vote sont concernés par ces critères ?

```
nb_bureaux = Sel.shape[0]
display(Markdown(f"**Nombre de bureaux de vote sélectionnés :** {nb_bureaux}"))
```

Nombre de bureaux de vote sélectionnés : 220

35.6. Analyse des données

Nous allons ajouter, pour **chaque candidat**, une colonne contenant le **deuxième chiffre** du **nombre de voix** :

```
for candidat in ["Trump", "Clinton", "Stein"]:
    Sel.loc[:, candidat + '_2chiffre'] = Sel[candidat].apply(lambda x: int(str(x)[1]))

Sel
```

	State	ST	Trump	Clinton	Stein	Trump_2chiffre	Clinton_2chiffre	Stein_2chiffre
6	Michigan	MI	21635	16490	379	1	6	7
12	Michigan	MI	2158	1156	49	1	1	9
20	Pennsylvania	PA	22676	6849	136	2	8	3
26	Wisconsin	WI	17310	18524	583	7	8	8
33	Wisconsin	WI	31044	17391	449	1	7	4
...	...	...	...	...	...	...	...	...
3057	Wisconsin	WI	39010	26476	641	9	6	4
3090	Pennsylvania	PA	257488	363017	5021	5	6	0
3105	Wisconsin	WI	46620	42506	834	6	2	3
3125	Michigan	MI	7228	3973	116	2	9	1
3139	Michigan	MI	224589	176238	3886	2	7	8

On crée un DataFrame avec l'histogramme des secondes chiffres par candidat. Autrement dit, on compte le nombre de fois où chaque chiffre est apparu en deuxième position dans les nombres de votes.

Pour cela on utilise `apply` qui applique une fonction donnée (ici, `pd.Series.value_counts`) à chaque colonne du DataFrame résultant. La fonction `pd.Series.value_counts` compte les occurrences uniques des valeurs dans une série (ici une colonne) et renvoie une série dont les indices sont les valeurs uniques présentes dans la colonne d'origine. Ces valeurs sont triées par défaut en ordre décroissant de fréquence.

```
# Première méthode
# Trump_histo = Sel['Trump_2chiffre'].value_counts()
# Clinton_histo = Sel['Clinton_2chiffre'].value_counts()
# Stein_histo = Sel['Stein_2chiffre'].value_counts()

Trump_histo = Sel['Trump_2chiffre'].value_counts(normalize=True)
Clinton_histo = Sel['Clinton_2chiffre'].value_counts(normalize=True)
```

```
Stein_histo = Sel['Stein_2chiffre'].value_counts(normalize=True)

His = pd.DataFrame({'Trump_histo': Trump_histo,
                    'Clinton_histo': Clinton_histo,
                    'Stein_histo': Stein_histo})
His = His.rename_axis('Chiffre')
display(His)

# Deuxième méthode
# cols = ['Trump_2chiffre', 'Clinton_2chiffre', 'Stein_2chiffre']
# # His = Sel[cols].apply(pd.Series.value_counts) / nb_bureaux
# His = Sel[cols].apply(lambda x: x.value_counts(normalize=True))
# His = His.rename_axis('Chiffre')
# His = His.rename(columns={"Trump_2chiffre": "Trump_histo", "Clinton_2chiffre": "Clinton_histo",
 ↪  "Stein_2chiffre": "Stein_histo"})
# display(His)
```

Chiffre	Trump_histo	Clinton_histo	Stein_histo
0	0.100000	0.086364	0.109091
1	0.140909	0.113636	0.136364
2	0.104545	0.122727	0.063636
3	0.109091	0.090909	0.095455
4	0.113636	0.104545	0.163636
5	0.104545	0.090909	0.027273
6	0.109091	0.122727	0.109091
7	0.095455	0.109091	0.063636
8	0.063636	0.095455	0.104545
9	0.059091	0.063636	0.127273

On ajoute la colonne `Benford` avec les effectifs théoriques des seconde chiffres attendus par la loi de Benford :

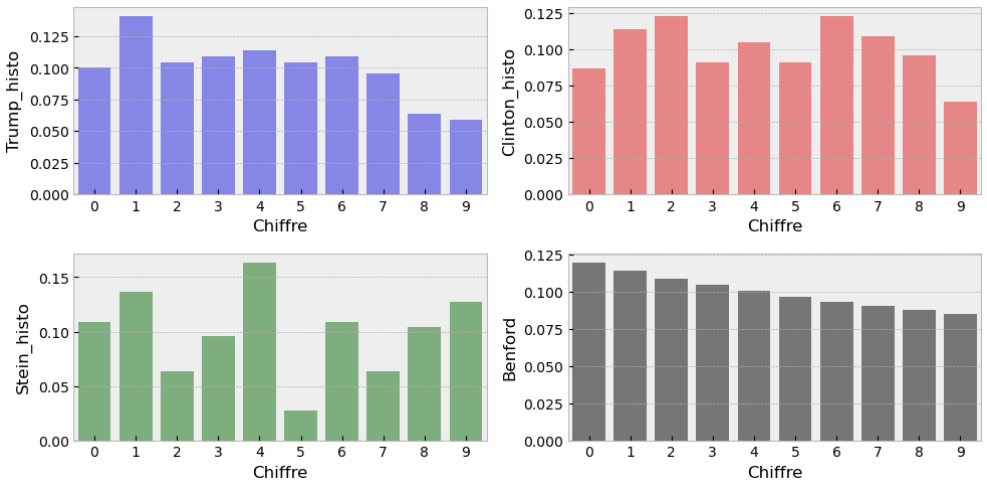
```
# On ajoute la colonne Benford avec les effectifs théoriques
# His['Benford'] = { Chiffre : round(p(Chiffre)*nb_bureaux) for Chiffre in His.index}
His['Benford'] = { Chiffre : p(Chiffre) for Chiffre in His.index}
His
```

Chiffre	Trump_histo	Clinton_histo	Stein_histo	Benford
0	0.100000	0.086364	0.109091	0.119679
1	0.140909	0.113636	0.136364	0.113890
2	0.104545	0.122727	0.063636	0.108821
3	0.109091	0.090909	0.095455	0.104330
4	0.113636	0.104545	0.163636	0.100308
5	0.104545	0.090909	0.027273	0.096677
6	0.109091	0.122727	0.109091	0.093375
7	0.095455	0.109091	0.063636	0.090352
8	0.063636	0.095455	0.104545	0.087570
9	0.059091	0.063636	0.127273	0.084997

On affiche la distribution des seconds chiffres par candidat.

```
# His.plot(figsize=(15,5))
# plt.xticks(range(10));
# His.plot.bar(figsize=(15,5));
```

```
plt.figure(figsize=(10,5))
plt.subplot(2,2,1)
sns.barplot(data=His, x=His.index, y='Trump_histo', color='blue', alpha=0.5)
plt.subplot(2,2,2)
sns.barplot(data=His, x=His.index, y='Clinton_histo', color='red', alpha=0.5)
plt.subplot(2,2,3)
sns.barplot(data=His, x=His.index, y='Stein_histo', color='green', alpha=0.5)
plt.subplot(2,2,4)
sns.barplot(data=His, x=His.index, y='Benford', color='black', alpha=0.5)
plt.tight_layout();
```



Pour chaque candidat, on calcule la **distance euclidienne entre les effectifs théoriques et les effectifs de chaque candidats**. Si A et B sont deux listes de même longueur, la distance euclidienne entre A et B est donnée par :

$$\text{dist}(A,B) = ||A-B||_2 = \sqrt{\sum_{c=0}^{9}(A_c - B_c)^2}.$$

Dans numpy, cela se calcule simplement par `np.linalg.norm(A-B)`.

```
# Dictionnaire pour stocker les distances
D = {}

# Pour chaque candidat, calculer la norme euclidienne entre les colonnes
for candidat in ["Trump_histo", "Clinton_histo", "Stein_histo"]:
    D[candidat] = np.linalg.norm(His[candidat] - His['Benford'])

D
```

```
{'Trump_histo': 0.05398891120857391,
 'Clinton_histo': 0.05714784909840473,
 'Stein_histo': 0.12083530757488783}
```

Selon ces valeurs, le candidat Stein semble légitime à contester les résultats. Mais est-ce effectivement si rare d'obtenir une telle distance ?

35.7. Bonus : Simulation pour la détection d'éventuelles anomalies

Pour commencer, on simule 10 votations qui suivent la loi de Benford.

```
# Créer une copie de Sel et ajouter une colonne avec des chiffres simulés selon FB
Sel_copy = Sel.copy()

for test in ["test"+str(i) for i in range(1,11)]:
    # Générer nb_bureaux chiffres aléatoires selon la distribution FB
    Sel_copy[test] = np.random.choice(range(10), size=nb_bureaux, p=FB)
    # Calculer les effectifs pour la colonne simulée
    # His[test] = Sel_copy[test].value_counts()
    His[test] = Sel_copy[test].value_counts(normalize=True)
    # Calculer la norme entre les effectifs simulés et ceux de Benford
    D[test] = np.linalg.norm(His[test] - His['Benford'])

# Affichage des distances un peu plus joli
pd.DataFrame( index=D.keys(), data=D.values(), columns=['Distance']).sort_values(by='Distance')
```

	Distance
test8	0.033691
test6	0.044985
Trump_histo	0.053989
test7	0.054228
test4	0.056988
test3	0.057119
Clinton_histo	0.057148
test5	0.057707
test10	0.058235
test1	0.066595
test2	0.076312
test9	0.107039
Stein_histo	0.120835

Bien que la valeur semble effectivement élevée pour Stein, il est possible que cette valeur ne soit pas si rares. En effet, dans nos 10 simulations, nous avons parfois obtenu des valeurs de distance euclidienne presque aussi élevées que celles observées pour Stein. Pour comprendre s'il s'agit d'evenement rares, on simule 10^4 fois pour voir combien de fois `D['test']>D[candidat]` :

```
distance_Stein = D['Stein_histo']
distance_Clinton = D['Clinton_histo']
distance_Trump = D['Trump_histo']


cpt_Stein = 0
cpt_Clinton = 0
cpt_Trump = 0

N = 10_000
for i in range(N):
    test = pd.Series(np.random.choice(range(10), size=nb_bureaux, p=FB)) # le chiffre i apparait avec une
 ↪  probabilité FB[i] et ce pour 220 bureaux de vote
    # test_histo = test.value_counts()
    test_histo = test.value_counts(normalize=True)
    D_test = np.linalg.norm( test_histo - His['Benford'])
    if D_test > distance_Stein:
        cpt_Stein += 1
    if D_test > distance_Clinton:
        cpt_Clinton += 1
    if D_test > distance_Trump:
        cpt_Trump += 1

print(f"""Sur {N} simulations,
```

```
• {cpt_Stein} fois on a obtenu une distance supérieure à celle de Stein, soit {cpt_Stein/N*100:.2f}% des
↪  fois;
• {cpt_Clinton} fois on a obtenu une distance supérieure à celle de Clinton, soit
↪  {cpt_Clinton/N*100:.2f}% des fois;
• {cpt_Trump} fois on a obtenu une distance supérieure à celle de Trump, soit {cpt_Trump/N*100:.2f}% des
↪  fois.
 """)
```

```
Sur 10000 simulations,
```
- `4 fois on a obtenu une distance supérieure à celle de Stein, soit 0.04% des fois;`
- `6178 fois on a obtenu une distance supérieure à celle de Clinton, soit 61.78% des fois;`
- `6953 fois on a obtenu une distance supérieure à celle de Trump, soit 69.53% des fois.`

35.8. Conclusion

La visualisation de la distribution des effectifs du second chiffre des résultats de vote dans les bureaux des états du pour chaque candidat à montré une anomalie. Les résultats ne sont pas conforme à la loi théorique de Benford. De plus, très peu de simulations ont donné des écarts encore plus importants pour le candidat Stein, autrement dit c'est très rare de voir une telle distribution.

Quatrième partie

Downloads

Chapitre 36.

Notebook Originaux et fichiers de données à télécharger

Dans cette partie vous trouverez les notebooks originaux et les fichiers de données utilisés dans le cours et les exercices. Ils sont disponibles en téléchargement pour que vous puissiez les exécuter sur votre propre machine.

Pour **télécharger un fichier**, cliquez sur le bouton correspondant ci-dessous (le poids du fichier est indiqué entre parenthèses).

36.1. Cours

- **Premiers pas : Numpy → Pandas**

  Download M55-0.ipynb (130 Ko)

  Download voitures.csv (1 Ko)
- **Series**

  Download M55-1.ipynb (145 Ko)
- **DataFrames**

  Download M55-2.ipynb (404 Ko)
- **Titanic : Importing and Analyzing Data**

  Download M55-3.ipynb (3.7 Mo) Download titanic.csv (65 Ko)

36.2. Exercices corrigés

- **Premières manipulations**

  Download M55-Exercice-00.ipynb (167 Ko) Download grillons.csv (206 octets) Download cigales.csv (249 octets)
- **Notes L3 Mathématiques**

  Download M55-Exercice-10.ipynb (22 Ko)
- **Ventes et pluie : agréger, c'est choisir**

  Download M55-Exercice-15.ipynb (160 Ko) Download ventes.csv (641 octets) Download pluviometrie.csv (745 octets)
- **Population des communes en France**

  Download M55-Exercice-20.ipynb (1.6 Mo) Download base-pop-historiques-1876-2021.csv (6.3 Mo) Download 20230823-communes-departement-region.csv (4.8 Mo)
- **Émissions de CO des communes françaises**

  Download M55-Exercice-25.ipynb (2.2 Mo) Download emissions-communes.csv (5.5 Mo) Download departements.geojson (1.0 Mo)
- **Netflix User**

  Download M55-Exercice-30.ipynb (520 Ko) Download Netflix_Userbase.csv (167 Ko)

- **Élections présidentielles USA 2016**

  Download M55-Exercice-40.ipynb (292 Ko) Download benford_usa.csv (173 Ko)