vBook livre & vidéo

Python pour la Data Science

livre

Analysez vos données avec NumPy, Pandas, Matplotlib et Seaborn

vidéo

Visualisation de données

1h15 de vidéo

Madjid KHICHANE
Amandine VELT

eni Editions

Python
pour la Data Science

Analysez vos données avec NumPy, Pandas, Matplotlib et Seaborn

Visualisation de données

1h15 de vidéo

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **VKEIPYTDAT** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Chapitre 1
Avant-propos et introduction

1. Avant-propos 11
2. Python et l'analyse de données 12
   2.1 L'explosion des données 12
   2.2 L'analyse de données 13
   2.3 R et Python pour l'analyse de données 14
3. Connaître les sources de données libres 15
   3.1 Kaggle 15
   3.2 Les données gouvernementales 17
4. Déroulement du livre 18

Chapitre 2
Mise en place de l'environnement de travail

1. Introduction : pourquoi utiliser Python pour la Data Science ? 19
2. Introduction à IPython et Jupyter 20
   2.1 Introduction à IPython 20
   2.2 Le projet Jupyter 22
3. Qu'est-ce qu'Anaconda ? 23
4. Installation d'Anaconda 24
   4.1 Installation sur Windows 24
   4.2 Installation sur MacOS 26
   4.3 Installation sur Linux 26

5. Découverte d'Anaconda Navigator 28
5.1 Applications disponibles dans Anaconda Navigator 28
5.2 Gestion des packages et environnements 29
6. Prise en main de Jupyter Notebook 32
6.1 Tableau de bord de Jupyter Notebook 32
6.2 Premiers pas avec les notebooks 34
6.3 Comprendre l'interface des notebooks 38
6.3.1 La barre de menus 38
6.3.2 La barre d'outils 40
6.3.3 Les cellules 41
6.3.4 Les modes Commande et Edition et les raccourcis-clavier 42
6.3.5 Les bases du langage Markdown pour écrire dans un notebook 45
6.3.6 Partager son notebook 51
7. Les packages Python essentiels pour la Data Science 52
7.1 NumPy 52
7.2 Pandas 53
7.3 Matplotlib 53
7.4 Seaborn 53

Chapitre 3
Rappels sur le langage Python

1. Introduction sur le langage de programmation Python 55
2. Les variables 56
3. Les différents types de données (int, float, bool, str) 58
3.1 Les nombres réels et entiers 58
3.2 Les booléens 59
3.3 Les chaînes de caractères 60

4. Les structures de données basiques (listes, tuples et dictionnaires) . 62
4.1 Les listes 62
4.1.1 Créer une liste 62
4.1.2 Accéder aux éléments d'une liste 63
4.1.3 Ajouter et supprimer des éléments à une liste. 66
4.2 Les tuples. 70
4.3 Les dictionnaires 72
4.3.1 Introduction aux dictionnaires. 72
4.3.2 Ajouter, modifier et supprimer des éléments d'un dictionnaire 74
4.3.3 Parcourir un dictionnaire 76
5. Les opérateurs arithmétiques, relationnels et logiques 78
5.1 Les opérateurs arithmétiques. 78
5.2 Les opérateurs relationnels et logiques 80
5.2.1 Les opérateurs relationnels 80
5.2.2 Les opérateurs logiques 81
6. Vocabulaire en Python : fonctions, méthodes, attributs, modules et librairies (packages) 82
6.1 Fonctions. 82
6.2 Méthodes. 84
6.3 Attributs 85
6.4 Modules. 85
6.5 Librairies (packages) 87
7. Instructions de condition if et boucles for 88
7.1 Instruction de condition if. 88
7.2 Boucle for 90

Chapitre 4
Maîtriser la librairie NumPy

1. Introduction à NumPy 93
2. Les tableaux NumPy 94
   2.1 Créer un ndarray 94
      2.1.1 Créer un ndarray à partir de listes 94
      2.1.2 Créer un ndarray grâce à des fonctions NumPy 97
      2.1.3 Créer un ndarray à partir d'un fichier 100
   2.2 Indexation 103
      2.2.1 Indexation simple 103
      2.2.2 Indexation booléenne 105
      2.2.3 Fancy indexing 108
   2.3 Accéder aux éléments par tranche (slicing) 110
      2.3.1 Slicing sur un tableau NumPy à 1 dimension 110
      2.3.2 Slicing sur un tableau NumPy à 2 dimensions 112
   2.4 Notion de vue et copie 114
3. Les opérations mathématiques avec NumPy 116
   3.1 Les opérations arithmétiques 116
   3.2 Les fonctions d'agrégations 119
4. Inspecter un tableau grâce aux attributs de NumPy 122
5. Manipuler des tableaux NumPy 124
   5.1 Ajouter et supprimer des éléments dans un tableau 124
      5.1.1 Ajouter des éléments dans un tableau 124
      5.1.2 Supprimer des éléments d'un tableau 127
   5.2 Diviser un tableau NumPy (split, hsplit et vsplit) 128
      5.2.1 Sur un tableau à une dimension 129
      5.2.2 Sur un tableau à deux dimensions 130
   5.3 Concaténer/combiner des tableaux 131
      5.3.1 La fonction concatenate() 131
      5.3.2 Les fonctions vstack() et hstack() 133
6. Introduction aux matrices avec NumPy 135

Chapitre 5
Maîtriser la librairie Pandas

1. Introduction 137
   1.1 Introduction à la librairie Pandas 137
   1.2 Introduction au jeu de données utilisé pour les exemples 140
2. Lire et écrire des fichiers avec Pandas 143
   2.1 Lecture de fichiers texte (CSV ou TXT) 143
      2.1.1 Lecture basique d'un fichier 143
      2.1.2 Gestion de l'en-tête 146
      2.1.3 Gestion des index 148
      2.1.4 Création d'un tableau à une dimension à partir du fichier 150
      2.1.5 Filtrage des colonnes lors de la lecture du fichier 150
      2.1.6 Les types des différentes colonnes 152
      2.1.7 Gestion des dates lors de la lecture du fichier 152
   2.2 Lecture de fichiers Excel 153
   2.3 Importation des données à partir d'une base de données 156
   2.4 Lecture de fichiers au format JSON 158
   2.5 Écriture de fichiers ou exportation de données 160
3. Structure de données Pandas : les Series (Séries) 162
   3.1 Introduction 162
   3.2 Créer des séries 163
      3.2.1 À partir de valeurs aléatoires 163
      3.2.2 À partir d'une liste Python 165
      3.2.3 À partir d'un tableau NumPy (ndarray) 168
      3.2.4 À partir d'un fichier texte 168
   3.3 Choisir l'index d'une série 169
   3.4 Accéder aux valeurs d'une série 171
      3.4.1 Indexing via la position des valeurs 171
      3.4.2 Indexing via l'étiquette des valeurs 172
      3.4.3 Les indexeurs loc et iloc 173
      3.4.4 Indexing via une expression booléenne 175

3.4.5 Slicing : découpage de valeurs successives 178
3.5 Les attributs et les méthodes des objets de classe Series 184
3.5.1 Les attributs des objets de classe Series 184
3.5.2 Les méthodes des objets de classe Series 185
3.6 Ajouter, supprimer et modifier les valeurs d'une série 187
3.6.1 Ajouter des valeurs à une série 187
3.6.2 Supprimer une valeur d'une série 188
3.6.3 Modifier les valeurs d'une série 189
4. Structure de données Pandas : les objets de type DataFrame 191
4.1 Introduction 191
4.2 Indexing : sélectionner des valeurs d'un dataframe 193
4.2.1 Indexing et slicing avec l'attribut loc 194
4.2.2 Indexing et slicing avec l'attribut iloc 198
4.2.3 Indexing avec une expression booléenne 199
4.3 Ajout, suppression et modification sur un dataframe 201
4.3.1 Ajouter une ou plusieurs colonnes à un dataframe 201
4.3.2 Ajouter une ligne à un dataframe 202
4.3.3 Supprimer des lignes ou colonnes d'un dataframe 205
4.3.4 Modifier des valeurs dans un dataframe 207
4.4 Nettoyage et préparation des données avec Pandas 209
4.4.1 Gestion des données manquantes 210
4.4.2 Gestion des données dupliquées 215
4.5 Exploration préliminaire d'un dataframe 219
4.5.1 Principaux attributs 219
4.5.2 Définition des termes variable, variable quantitative et variable qualitative et découverte de la méthode describe() 222
4.5.3 Méthodes de tri d'un dataframe 225
5. Structure de données Pandas : les panels 227
6. Manipulation avancée des données avec Pandas 228
6.1 Les opérations groupby 228
6.1.1 groupby sur une colonne 228
6.1.2 groupby sur plusieurs colonnes 231

6.1.3 Appliquer plusieurs fonctions avec la méthode groupby et la méthode aggregate 232
6.2 Appliquer une fonction à un dataframe avec la méthode apply 234
6.3 Remodeler/réorganiser des dataframes 236
6.3.1 Pivotage : la méthode pivot_table 236
6.3.2 Les méthodes stack (empiler) et unstack (désempiler) . 238

Chapitre 6
Maîtriser la librairie Matplotlib

1. Introduction 241
2. Le fonctionnement de Matplotlib 242
2.1 Architecture de Matplotlib 242
2.2 Organisation des figures avec Matplotlib 244
3. La création d'un premier graphique simple 246
3.1 Préparer son jeu de données 246
3.2 Créer un nuage de points 250
3.3 Ajouter un titre principal et des labels aux axes du nuage de points 254
3.4 Enregistrer son graphique 256
3.5 Changer la taille de la fenêtre graphique et la résolution de son graphique 258
3.6 Tracer plusieurs courbes sur un même graphique (sur un même objet axes) 260
3.7 Ajouter une légende à son graphique 263
3.8 Annoter son graphique avec du texte 265
3.9 Combiner plusieurs graphiques grâce à subplot et subplots . . 268
3.9.1 Tracer des sous-graphiques (subplot) sur une ligne ou une colonne 268
3.9.2 Tracer des sous-graphiques sur plusieurs lignes et plusieurs colonnes 271
3.9.3 Incruster un objet axes dans un autre 274

4. Les différents types de graphes 278
4.1 Types de graphiques selon les types de variables (quantitatives et qualitatives) 278
4.2 Scatterplot 279
4.3 Graphique à barres (bargraph) 282
4.3.1 Graphique à barres simple 282
4.3.2 Graphique à barres groupées 287
4.3.3 Graphique à barres empilées 294
4.4 Boxplots 296

Chapitre 7
Maîtriser la librairie Seaborn

1. Introduction 301
2. L'esthétique des figures avec Seaborn (Aesthetic) 303
2.1 Paramétrer les styles Seaborn (thèmes) 304
2.2 Supprimer les axes 306
2.3 Paramétrer les contextes avec Seaborn 309
2.4 Les palettes de couleur avec Seaborn 314
2.4.1 Choisir une palette de couleurs existante 314
2.4.2 Créer sa propre palette de couleurs 317
3. Les différents types de graphiques 318
3.1 Préparation du jeu de données 318
3.2 Nuage de points (scatterplot) 318
3.3 Graphiques de régression 323
3.4 Pointplot 327
3.5 Nuage de points avec une variable qualitative : stripplot 331
3.6 Boxplots 334
3.7 Graphique à barres : countplot 336
3.8 Histogrammes 339
3.9 Jointplot 344
3.10 Pairplot 346
3.11 Heatmap 349

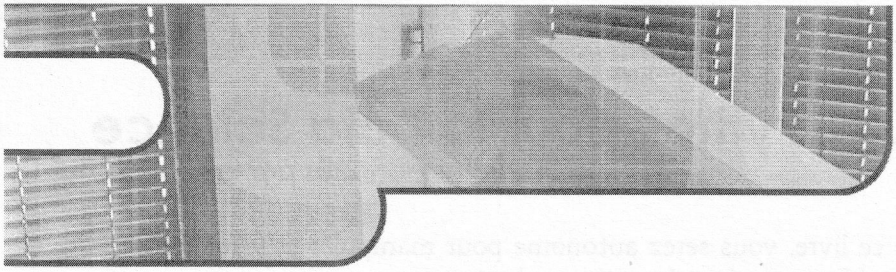
4. Les graphiques multi-grilles 352
4.1 FacetGrid 352
4.2 PairGrid 354
4.3 JointGrid 359
5. Conclusion 361

Chapitre 8
Exercice complet sur jeu de données réel

1. Introduction 363
2. Présentation du jeu de données 365
3. Énoncé de l'exercice 366
3.1 Lire le fichier 366
3.2 Afficher les dimensions du dataframe 367
3.3 Compter les films et les séries 367
3.4 Générer le résumé statistique du dataframe 367
3.5 Compter les valeurs manquantes 368
3.6 Explorer les valeurs manquantes 368
3.6.1 Sur la colonne des directeurs de production 368
3.6.2 Sur la colonne des acteurs 368
3.7 Supprimer les lignes dupliquées 369
3.8 Compter les films/séries produits par les États-Unis et par la France 369
3.9 Afficher le contenu le plus vieux disponible sur Netflix 370
3.10 Afficher le film avec la durée la plus longue sur Netflix 370
3.10.1 Nouvelle notion : les méthodes str 370
3.10.2 Énoncé 371
3.11 Étudier les catégories avec le plus de contenu 372
3.12 Afficher les directeurs qui ont produit le plus de films/séries disponibles sur Netflix 375
3.13 Voir si Jan Suter travaille souvent avec les mêmes acteurs . . . 375

3.14 Représenter les dix pays qui ont produit le plus de contenus disponibles sur Netflix, avec le nombre de contenus par pays 376
3.15 Tracer un graphe à barres du nombre de films/séries par classement de contenu (rating) 377
3.16 Afficher l'évolution du nombre de films/séries disponibles sur Netflix au cours du temps 377
3.16.1 Notions supplémentaires sur les dates 377
3.16.2 Énoncé 377
3.17 Afficher la distribution de la durée des films disponibles sur Netflix 378
3.18 Tracer un graphique représentant le nombre de séries par modalité de nombre de saisons 379

Index 381



Chapitre 1
Avant-propos et introduction

1. Avant-propos

Ce livre de la collection vBook se compose d'un **livre de référence** sur la Data Science avec le langage Python, alliant théorie et pratique, pour apprendre à analyser des données avec Python et d'un **approfondissement sous forme de vidéo** qui présente Numpy et Pandas, deux bibliothèques dédiées à la Data Science et à la visualisation de données. Il est destiné aux personnes souhaitant avoir une première immersion dans l'analyse de données avec le langage de programmation Python et les librairies dédiées à l'analyse de données : NumPy, Pandas, Matplotlib et Seaborn. Il est conseillé d'avoir une première expérience avec le langage Python, même si cela n'est pas indispensable car un chapitre de ce livre traite des rappels sur ce langage. Dans ce livre, nous verrons ensemble comment explorer, manipuler et visualiser des données structurées, c'est-à-dire des tableaux contenant des lignes et des colonnes, que vous pouvez comparer à un tableau Excel. En revanche, dans cet ouvrage, nous n'introduirons pas les méthodologies d'analyse statistique avec Python, ni le Machine Learning : un livre à part, plus théorique, pourrait être dédié à cette fin. Le réel but de ce livre est de prendre en main et de maîtriser Python **techniquement**, c'est-à-dire de maîtriser la programmation, les librairies dédiées à l'analyse de données, mais aussi les outils disponibles pour ces analyses. Cet ouvrage vous permettra de maîtriser les différentes structures de données disponibles afin de les manipuler, les explorer, les nettoyer et les visualiser pour en faire ressortir des tendances et des idées.

À la fin de ce livre, vous serez autonome pour manipuler et visualiser l'ensemble des tableaux de données qui vous intéressent.

En téléchargement avec ce livre, vous avez accès à l'ensemble du code sous forme de notebooks, présentés dans le prochain chapitre. N'hésitez pas à les récupérer et à lancer les lignes de code de votre côté, pour mieux les comprendre. Dans ce livre, nous utiliserons également un jeu de données principal, regroupant un ensemble d'informations très intéressantes sur les athlètes des Jeux olympiques, depuis le début des jeux jusqu'en 2016. Nous explorerons ensemble ce tableau et y détecterons des tendances et des informations parfois étonnantes ! À la fin de la lecture de ce livre, n'hésitez pas à récupérer un nouveau jeu de données via les sources qui seront présentées dans ce chapitre, pour à votre tour détecter des tendances et informations sur des sujets qui vous intéressent. Nous avons la chance, depuis quelques années, d'avoir un accès libre à une grande quantité de données, et ce dans des domaines extrêmement variés, alors n'hésitez pas à travailler sur des données provenant de domaines qui vous passionnent. En effet, plus vous analyserez de données, plus vous acquerrez d'expérience et plus vous serez à l'aise dans cette discipline.

2. Python et l'analyse de données

2.1 L'explosion des données

Depuis quelques années, une explosion du volume des données a commencé. À présent, nous vivons dans un monde totalement digitalisé, avec l'ensemble de nos appareils connectés, l'existence des médias sociaux, et la puissance d'Internet et des données libres, ou open data : l'analyse de ces données est à la portée de tous, à condition de savoir programmer et d'utiliser les bons outils.

Les données ouvertes, plus communément appelées Open Data, permettent d'accéder librement à un ensemble de données qu'il est possible d'analyser pour en tirer de nouvelles connaissances, de nouvelles stratégies ou encore de nouveaux services. Par exemple, il est possible d'accéder à un nombre conséquent de données gouvernementales rien qu'en allant sur le site du gouvernement français : https://www.data.gouv.fr.

© Editions ENI - All rights reserved

Cela permet à tout un chacun d'accéder à des jeux de données liés à notre pays, sur l'agriculture, l'alimentation, la culture, l'économie, l'environnement, la santé, les transports et tant d'autres domaines plus intéressants les uns que les autres.

Il existe aussi des sites web regroupant des jeux de données sur des domaines très divers, déjà nettoyés et structurés, prêts à être analysés. Le site web le plus connu est Kaggle (https://www.kaggle.com/), qui sera présenté plus tard dans ce chapitre. Enfin, un volume de données immense est stocké en permanence via les e-mails, les tweets, les photos, les vidéos YouTube ou encore les données de géolocalisation via nos smartphones et leurs applications.

À présent, nos ordinateurs de bureau sont puissants, les outils d'analyse sont pour la plupart libres et l'analyse de données est ainsi ouverte à tous. Il existe deux types de données : les données structurées sous forme de tableaux (lignes et colonnes) et les données non structurées : par exemple des pages web, emails, tweets, images, audio, vidéo. Dans ce livre, nous traiterons uniquement des données structurées : à partir de données non structurées, l'analyste essaiera généralement de créer des données structurées. En effet, les tableaux représentent la structure de données la plus populaire en analyse et la plus simple à comprendre par l'Homme. Par exemple, à partir d'e-mails ou de tweets, il sera courant de créer un tableau contenant les fréquences d'un ensemble de mots, puis de travailler sur ces fréquences pour faire ressortir des tendances. C'est pour cela que ce livre ne traitera que de l'analyse de données structurées avec Python. Si vous possédez un ordinateur et une connexion Internet, alors vos possibilités sont immenses et nous allons les voir ensemble dans ce livre.

2.2 L'analyse de données

Le but de l'analyse de données est de détecter des tendances à partir de données et ainsi d'en extraire de la connaissance, car il est alors possible de raconter une histoire, de comprendre ce qu'il se passe et quelles en sont les conséquences. À partir de grands jeux de données, structurés ou non, en explorant, visualisant et analysant ces données, il est aussi possible de recueillir des informations utiles pour résoudre un problème.

L'analyse de données est utilisée notamment par les entreprises qui se servent des données pour détecter des tendances qu'elles utiliseront ensuite pour prendre des décisions stratégiques. Notamment, en analysant leurs données, les entreprises vont pouvoir augmenter leurs revenus, réduire leurs coûts, ou encore mieux comprendre leurs clients pour améliorer leurs stratégies marketing.

L'analyse de données est utilisée partout, dans tous les domaines : les possibilités derrière les données sont énormes et plus personne ne les ignore.

De même que les possibilités sont grandes, les méthodes d'analyses sont aussi très diverses, tant en matière de méthodologies statistiques que de manipulation et visualisation des données. Toutefois, pour commencer à analyser des données, il est nécessaire de maîtriser un langage de programmation dédié à l'analyse. Il existe des outils interfacés pour analyser des données, tels que le célèbre outil Tableau (https://www.tableau.com/fr-fr) : même si cet outil est complet, performant et simple d'utilisation, son très grand désavantage est d'être payant. Le meilleur moyen d'être libre de travailler sur les données, et ce gratuitement, est d'apprendre à programmer dans un langage dédié à l'analyse de données.

2.3 R et Python pour l'analyse de données

Lorsqu'on parle de programmation pour l'analyse de données, deux outils principaux ressortent : R et Python. Effectivement, ces deux outils permettent de faire de l'analyse de données, et ce de manière tout aussi efficace : c'est pour cela qu'aucun choix n'a été fait sur le langage à utiliser préférentiellement. Cela dépend plutôt de l'utilisateur ou de l'entreprise.

R est un projet qui a vu le jour en 1993 et est un langage de programmation open source (gratuit) qui a été développé pour des statisticiens. C'est un langage très orienté statistique, mais qui, jusqu'à récemment, n'était pas très performant sur de grands jeux de données. Toutefois, depuis quelques années et à l'aide de packages très populaires comme dplyr et ggplot, ce langage a su rester dans l'air du temps. En effet, jusqu'à très récemment, R était utilisé dans les enseignements de nombreux domaines qui nécessitaient de l'analyse de données et notamment les statistiques. Cette tendance change doucement pour laisser place à Python dans ces enseignements.

© Editions ENI - All rights reserved

Python est adapté à l'analyse de données depuis quelques années maintenant, mais cela n'a pas toujours été le cas, contrairement à R. Python a vu le jour en 1991 et est un langage de programmation open source qui a été créé pour faire du développement logiciel, en premier lieu. Mais depuis, et par le développement de nombreuses librairies, Python est utilisé dans des domaines variés et pour des applications très larges, beaucoup plus larges que les applications possibles du langage R. En 1994, Python possédait une librairie dédiée au calcul scientifique, NumPy, mais n'était pas utilisé en premier choix pour l'analyse de données. Avec l'apparition des célèbres librairies Pandas (2008) ou encore Matplotlib (2003), que nous verrons dans ce livre, Python s'est fait une place dans le domaine de l'analyse de données, jusqu'à faire de l'ombre au langage R.

Aujourd'hui, Python est un langage bien plus populaire et plus largement utilisé que R, du fait que c'est un langage de programmation plus généraliste et permettant des applications dans de nombreux domaines. Toutefois, R reste populaire auprès des statisticiens et dans le milieu académique. Mais dans le vaste domaine de la science des données et de leur analyse, la popularité de Python par rapport à R ne cesse d'augmenter : Python est donc le langage de référence à apprendre pour travailler sur les données.

3. Connaître les sources de données libres

3.1 Kaggle

Kaggle est une plateforme très connue dans le domaine de la science des données (*Data Science*). Cette plateforme propose plus de 19000 jeux de données dans des domaines très variés. Ces jeux de données sont généralement bien structurés, pour pouvoir être directement analysés.

En plus de proposer des jeux de données très variés, Kaggle permet à ses utilisateurs d'analyser leurs données via ce qu'on appelle des notebooks, que nous verrons dans le chapitre suivant, puis de rendre leurs analyses publiques. Ainsi, pour chaque jeu de données de la plateforme, un ou plusieurs rapports d'analyse sont disponibles, avec le code et les résultats associés.

Lorsqu'on débute dans l'analyse de données, il est très intéressant d'aller regarder les différents rapports d'analyses pour augmenter ses compétences et voir ce que les autres analystes font pour analyser leurs données.

Enfin, Kaggle propose tout un système de compétitions : de temps en temps, Kaggle propose des jeux de données à analyser sous forme de compétition, et l'analyste qui génère la meilleure analyse de données remporte un prix. Certaines entreprises passent même par Kaggle pour effectuer leurs recrutements d'analystes, en embauchant la personne ayant gagné la compétition. Bref, vous l'aurez compris, cette plateforme est très complète et vraiment très utilisée par les analystes en général. Toutefois, l'interface est en anglais : ne vous découragez pas, l'anglais technique du domaine s'acquiert vite.

Dans ce livre, nous travaillerons beaucoup sur un jeu de données récupéré via cette plateforme, dont voici le lien :
https://www.kaggle.com/heesoo37/
120-years-of-olympic-history-athletes-and-results

Sur l'image ci-dessus se trouve un aperçu du lien vers le jeu de données que nous utiliserons dans ce livre. Il s'agit d'informations sur l'ensemble des athlètes ayant participé aux Jeux olympiques, du début des jeux en 1896 jusqu'aux derniers Jeux olympiques en 2016.

© Editions ENI - All rights reserved

Il y a quatre informations importantes ici :

- 1 : La licence du jeu de données est importante. Ici, la licence est CCO : Public Domain. Cela signifie que le jeu de données est libre et utilisable à des fins commerciales ou non.
- 2 : Les kernels contiennent ce qu'on appelle les notebooks : les rapports d'analyse. Pour ce jeu de données, 155 analyses ont été effectuées.
- 3 : En cliquant sur **Download**, vous téléchargerez le jeu de données.
- 4 : Il est possible d'effectuer votre propre analyse et de la rendre publique en cliquant sur **New Notebook**.

Kaggle est réellement la meilleure plateforme participative dans le domaine de l'analyse des données et il est conseillé d'aller l'explorer.

Remarque

Le jeu de données sur les Jeux olympiques est disponible en téléchargement avec ce livre.

3.2 Les données gouvernementales

En Open Data, le site web le plus populaire pour récupérer des données libres est le site www.data.gov : il s'agit des données gouvernementales des États-Unis, avec plus de 200000 jeux de données qui vous ouvrent les portes de la connaissance, à condition de savoir analyser des données, bien sûr. Son équivalent français est le site : https://www.data.gouv.fr/

Une autre source de jeux de données très intéressante est le portail de données ouvertes européennes, disponible sur ce site : https://data.europa.eu/euodp/fr/data/. Ces données gouvernementales sont utilisées par les particuliers intéressés par un domaine précis, à des fins de recherches, mais aussi par les entreprises qui analysent ces données pour créer de nouveaux services pour leurs clients. Les possibilités de ces données sont illimitées. Par exemple, récemment, l'Open Data a révolutionné les applications mobiles, qui utilisent ces données ouvertes pour proposer de nouveaux services, que ce soit en transport, en loisirs, et dans beaucoup de domaines où la demande est présente.

Les données ouvertes sont aussi utilisées pour trouver des tendances, comprendre et expliquer l'actualité, ou juste pour acquérir de nouvelles connaissances humaines.

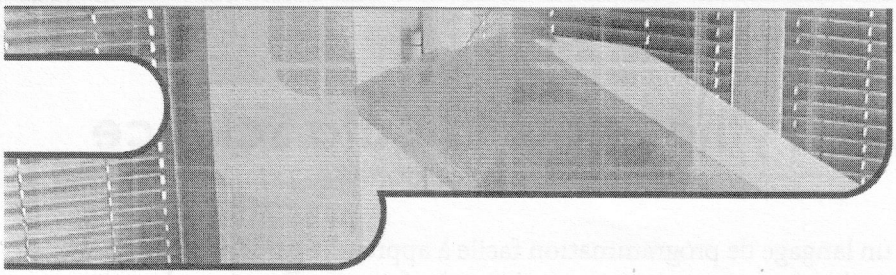
4. Déroulement du livre

Dans ce livre, nous allons apprendre à analyser des données avec Python et les librairies dédiées à l'analyse. Nous verrons ensemble comment importer un jeu de données structuré pour pouvoir ensuite le nettoyer et le filtrer, l'explorer et le visualiser. Pour pouvoir faire l'ensemble de ces étapes, nous verrons en détail les librairies principales dédiées à l'analyse de données que sont NumPy, Pandas, Matplotlib et Seaborn. Nous apprendrons à importer des fichiers sous forme de structures Python adaptées, à manipuler ces structures et à les explorer pour en faire ressortir des tendances.

L'ensemble de ces pratiques seront vues théoriquement, puis seront agrémentées d'exemples sur un jeu de données concret contenant des informations sur l'ensemble des athlètes ayant participé aux Jeux olympiques. Enfin, dans un dernier chapitre, un nouveau jeu de données vous sera proposé, qu'il vous faudra analyser par vous-même. Ce chapitre, court et sous forme de questions auxquelles vous devrez répondre, vous permettra de tester vos connaissances acquises sur l'ensemble du livre. La correction complète de cet exercice, sous forme d'un notebook, est bien sûr disponible en téléchargement avec ce livre, en plus de l'ensemble du code utilisé tout au long de l'ouvrage.

À la fin de votre lecture, vous serez capable d'aller récupérer des données structurées sous forme de tableau et de les explorer pour en faire ressortir des tendances et répondre à vos propres questions.

© Editions ENI - All rights reserved



Chapitre 2
Mise en place de l'environnement de travail

1. Introduction : pourquoi utiliser Python pour la Data Science ?

Python est un langage de programmation libre de droits, populaire et utilisé dans énormément de domaines. Il est utilisé dans le domaine scientifique pour l'analyse de données, et plus généralement la Data Science, mais aussi pour développer des applications web, des jeux, des interfaces graphiques d'outils, ou encore pour faire du développement logiciel en général. Il propose un ensemble complet de librairies dédiées à la Data Science et donc des fonctions facilitant la manipulation de données, les analyses statistiques sur ces données et leur visualisation, notamment.

Son concurrent direct est le langage R, langage dédié à l'analyse statistique pensé par les statisticiens, contrairement à Python qui est un langage très généraliste et applicable à de larges domaines. Ces deux langages, R et Python, sont très utilisés en Data Science, mais depuis quelques années, Python est devenu plus populaire auprès du Data Scientist, en fait, depuis l'apparition des librairies dédiées à la Data Science.

Python est un langage de programmation facile à apprendre car il possède une syntaxe très simple et une documentation très fournie. De plus, dans le domaine de la Data Science, une interface graphique est disponible pour l'analyse de données avec Python et le partage de ces analyses : les notebooks Jupyter – des documents permettant d'écrire du texte et du code Python pour créer un rapport d'analyse. Enfin, Python possède des librairies et méthodes qui permettent, en quelques lignes de code, d'effectuer des traitements complexes sur les données.

Avec une utilisation dans de nombreux domaines, Python est continuellement maintenu et amélioré, il possède une très grande communauté et une documentation très complète, en anglais comme en français.

Toutes ces raisons font de Python un langage de choix pour tout projet concernant l'analyse de données en général.

2. Introduction à IPython et Jupyter

2.1 Introduction à IPython

iPython (pour *Interactive Python* ou Python interactif en français) est apparu avant le projet Jupyter et avait pour but de fournir un interpréteur interactif pour le langage de programmation Python. Il simplifiait le développement en Python avec des fonctionnalités avancées comme la coloration syntaxique, la complétion des noms de fonctions, mots-clés, variables, etc.

Plus tard, d'autres composants ont été ajoutés à IPython, comme les notebooks. Les notebooks sont des pages web interactives permettant de représenter et interpréter le code Python et donc d'afficher les résultats de ce code, cela simplifiant notamment l'analyse de données.

© Editions ENI - All rights reserved

Plus tard, le projet Jupyter a vu le jour et tous les composants non spécifiques du langage Python, comme par exemple les notebooks, ont été ajoutés au projet Jupyter. Le but de cette manœuvre était de pouvoir se concentrer sur le développement des notebooks et de les adapter à d'autres langages de programmation que Python. L'autre objectif était de recentrer IPython sur son rôle premier : faire du Python interactif en développant toujours plus le shell interactif Python plutôt que les notebooks.

Remarque

Un shell, ou terminal interactif est un interpréteur de commandes dont le but est d'interpréter les commandes que nous écrivons pour que notre système d'exploitation comprenne ce que nous voulons faire et d'exécuter ces commandes pour générer le résultat attendu. Il existe un interpréteur pour chaque langage interprété, dont Python.

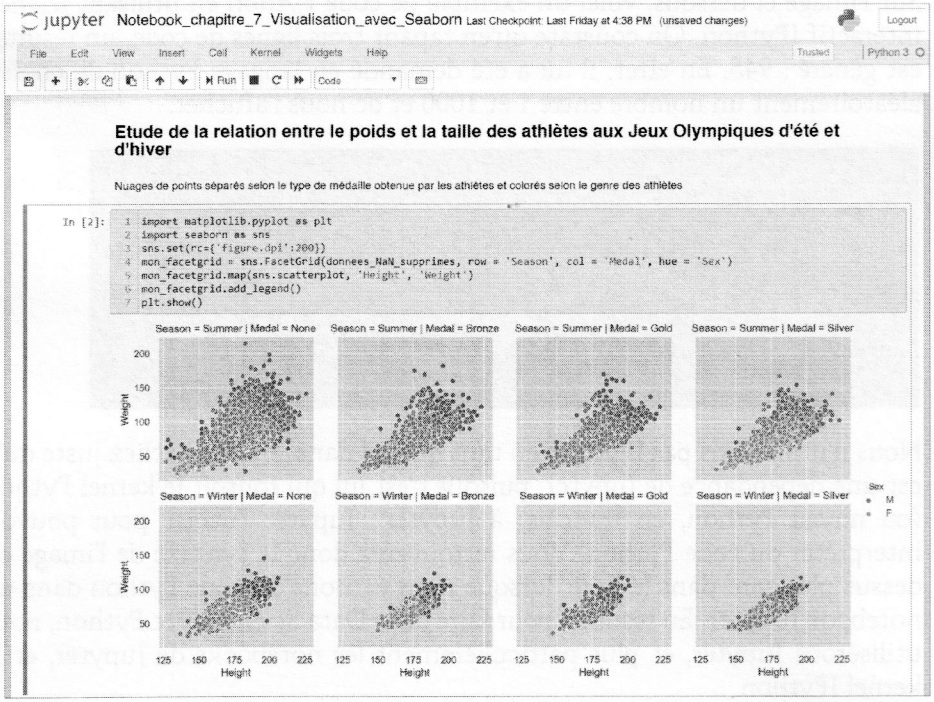
Sur l'image ci-dessous, voici un exemple de code Python en utilisant le shell interactif IPython. On constate qu'en tapant trois lignes de code, un résultat est généré : 945. En effet, il lui a été demandé en langage Python de choisir aléatoirement un nombre entre 1 et 1000 et de nous l'afficher.

```
                :~$ ipython
Python 3.7.3 | packaged by conda-forge | (default, Jul  1 2019, 21:52:21)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import random

In [2]: nombre = random.randint(1,1000)

In [3]: print(nombre)
945
```

Nous n'utiliserons pas IPython en tant que tel dans ce livre, sachez juste qu'il est une dépendance de Jupyter, puisque c'est lui qui fournit le kernel Python (ou noyau Python, en français) à Jupyter : Jupyter l'utilise pour pouvoir interpréter du code Python. Vous retrouverez donc la syntaxe de l'image ci-dessus plus tard dans le livre, lorsque nous écrirons du code Python dans un notebook Jupyter. En résumé, pour faire de la Data Science avec Python, nous utiliserons Jupyter, et plus particulièrement les notebooks de Jupyter, et le kernel IPython.

2.2 Le projet Jupyter

Le nom de Jupyter provient des trois langages de programmation qui ont été inclus dans le projet au début, à savoir Julia, Python et R. Le but du projet Jupyter était de développer des outils et services pour faire de la programmation interactive avec plusieurs langages de programmation. À présent, Jupyter supporte plus de quarante langages de programmation, et permet de programmer de manière interactive avec ces langages, ce qui est très important pour faire de la Data Science, d'où sa popularité dans ce domaine. La force de Jupyter, ce sont sans conteste ses notebooks. Un notebook est un document exécutable, qui contient du code (Python, R, ou autre) qui va pouvoir être interprété et dont les résultats seront intégrés dans le document. Le notebook peut aussi contenir des éléments de texte comme des paragraphes de texte, des figures, des liens, des équations mathématiques, etc.



© Editions ENI - All rights reserved

Sur l'image ci-dessus, vous pouvez observer un exemple de notebook que nous générerons avec Jupyter Notebook, avec du texte permettant de décrire ce qui est fait ou encore du code Python et une figure présentant les résultats de ce code. La force des notebooks est qu'ils se comportent comme un rapport d'analyse, que l'on peut partager pour présenter une analyse de données et ses résultats. De plus, ils se comportent comme un environnement interactif dans lequel on va pouvoir lancer une analyse de données, c'est-à-dire qu'au sein d'un notebook, on va pouvoir exécuter du code et générer le résultat de ce code en temps réel, directement dans l'interface du notebook.

Pour travailler avec les notebooks, il est nécessaire d'avoir l'application Jupyter Notebook, qui permettra de visualiser, modifier et exécuter les notebooks à travers un navigateur web (type Firefox ou Chrome). Nous verrons plus en détail cette application dans la section Découverte d'Anaconda Navigator de ce chapitre. Une chose est sûre, pour pouvoir commencer à faire de la Data Science avec Python, le premier outil essentiel est Jupyter Notebook, il faut donc l'installer. Nous installerons Jupyter Notebook au travers de l'application Anaconda, présentée dans la section à suivre.

3. Qu'est-ce qu'Anaconda ?

Anaconda est tout simplement la distribution Python et R la plus connue et la plus utilisée en Data Science car cette plateforme est dédiée à ce domaine précisément. En effet, c'est une distribution libre qui contient non seulement les environnements Python et R mais aussi toutes les librairies essentielles pour pouvoir faire de la Data Science (par exemple pour Python : NumPy, Pandas, Matplotlib, pour ne citer qu'elles). De plus, la distribution Anaconda inclut Jupyter Notebook, RStudio et beaucoup d'autres applications essentielles à la Data Science. Nous utiliserons uniquement Jupyter Notebook et le kernel IPython sous Anaconda, mais il est bon de savoir que cette distribution inclut les deux langages de programmation les plus populaires en Data Science.

Cette distribution est très simple à installer, elle est supportée sur Windows, MacOS et Linux et possède une large communauté d'utilisateurs ainsi qu'une documentation très complète (en anglais). Encore mieux, une interface graphique a été développée pour cette distribution ; elle s'appelle Anaconda Navigator et permet en quelques clics d'installer de nouveaux packages (ou librairies), de créer de nouveaux environnements, d'accéder aux applications, etc.

En vous espérant convaincus, nous allons maintenant installer cette distribution Anaconda ainsi que l'interface graphique qui l'accompagne : Anaconda Navigator.

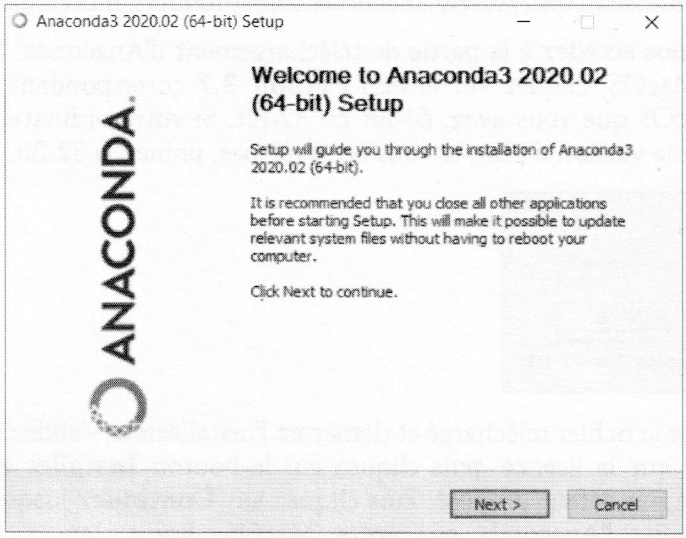
4. Installation d'Anaconda

4.1 Installation sur Windows

- Rendez-vous à l'adresse https://www.anaconda.com/products/individual.
- En fin de page, vous accédez à la partie de téléchargement d'Anaconda. Si vous êtes sous Windows, cliquez sur le lien **Python 3.7** correspondant à l'architecture Windows que vous avez, 64-Bit ou 32-Bit. Si votre ordinateur est récent, prenez la version 64-Bit, si vous ne savez pas, prenez la 32-Bit. Un fichier à l'extension .exe se téléchargera.

© Editions ENI - All rights reserved

- Une fois téléchargé, exécutez le fichier **.exe** en cliquant dessus. La fenêtre d'installation d'Anaconda s'ouvre, comme sur l'image suivante :

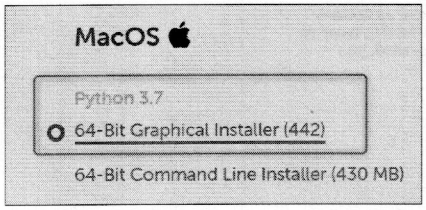


- Cliquez sur **Next**, puis **I Agree** pour accepter la licence, laissez cochée l'option **Just Me** et cliquez sur **Next**. Laissez le chemin d'installation par défaut et cliquez sur **Next**. Laissez la case cochée par défaut et cliquez sur **Install**. Une fois l'installation terminée, cliquez sur **Next**, **Next** et **Finish**.

- Si vous cliquez sur le bouton **Démarrer** de Windows et que vous recherchez le mot *Anaconda*, vous verrez une icône appelée **Anaconda Navigator**. Vous pouvez cliquer dessus pour ouvrir Anaconda Navigator, c'est par là que tout va commencer, et cela vous sera expliqué juste après la partie Installation.

4.2 Installation sur MacOS

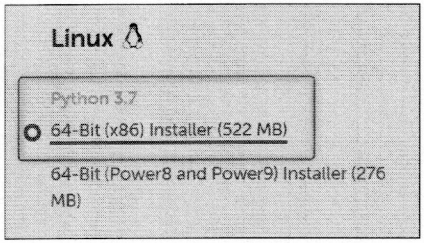
- Rendez-vous à l'adresse https://www.anaconda.com/products/individual.
- En fin de page, vous accédez à la partie de téléchargement d'Anaconda. Si vous êtes sous MacOS, cliquez sur le lien **Python 3.7** correspondant à l'architecture MacOS que vous avez, 64-Bit ou 32-Bit. Si votre ordinateur est récent, prenez la version 64-Bit, si vous ne savez pas, prenez la 32-Bit.



- Double cliquez sur le fichier téléchargé et démarrez l'installation. Validez le "**Read Me**" ainsi que la licence, puis cliquez sur le bouton **Installer** en laissant le chemin par défaut proposé. Puis cliquez sur **Continuer** jusqu'à l'installation réussie d'Anaconda sur votre MacOS. Enfin, lancez le programme nommé **Anaconda Navigator** : vous êtes prêt pour commencer à apprendre à utiliser Python pour la Data Science.

4.3 Installation sur Linux

- Rendez-vous à l'adresse https://www.anaconda.com/products/individual.
- En fin de page, vous accédez à la partie de téléchargement d'Anaconda. Si vous êtes sous Linux, cliquez sur le lien **Python 3.7** correspondant à l'architecture Linux que vous avez, 64-Bit (x86) ou 64-Bit (Power8 and Power9).



© Editions ENI - All rights reserved

- Sous votre terminal Linux, allez dans le dossier où a été téléchargé le script `.sh`. Tapez la commande suivante, en remplaçant le caractère "*" par le numéro de version si vous en avez plusieurs dans le dossier.

Code

```
chmod +x Anaconda3-*-Linux-x86_64.sh
bash Anaconda3-*-Linux-x86_64.sh
```

Une fois le script exécuté, les étapes à suivre sont les suivantes :

- Appuyez sur [Entrée] pour valider la licence, écrivez **yes** et appuyez sur [Entrée]. Laissez le chemin d'installation par défaut proposé par Anaconda en appuyant sur [Entrée]. Anaconda s'installe. Enfin, il est demandé si vous voulez que l'installateur initialise Anaconda3 au démarrage du terminal. Je vous conseille d'écrire **yes** et d'appuyer sur [Entrée].
- Fermez et rouvrez votre terminal, l'installation est terminée et la configuration est faite.

Remarque

Si vous n'avez pas demandé à initialiser Anaconda3 par défaut, alors vous devez le faire manuellement avant de pouvoir ouvrir l'interface Anaconda Navigator, avec la commande suivante : source /home/utilisateur/anaconda3/bin/activate."/home/utilisateur/anaconda3" étant le chemin que vous avez choisi lors de l'installation d'Anaconda.

- Vous pouvez à présent ouvrir l'interface graphique Anaconda Navigator en tapant la commande suivante :

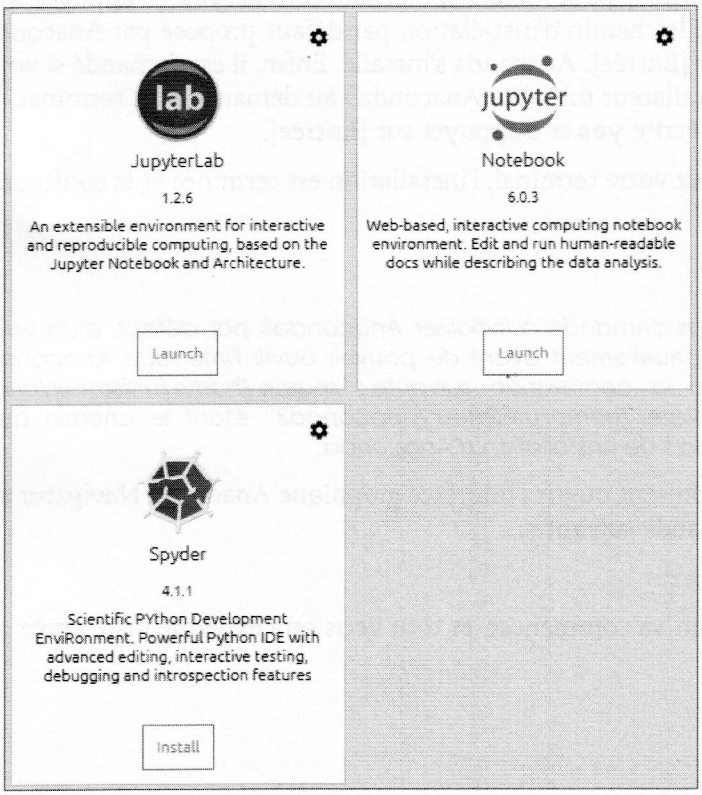
```
anaconda-navigator
```

C'est par là que tout va commencer, et tout vous sera expliqué juste après.

5. Découverte d'Anaconda Navigator

5.1 Applications disponibles dans Anaconda Navigator

Anaconda Navigator propose différentes applications disponibles directement lors de l'ouverture de l'outil. Sur l'image ci-dessous, voici les trois outils dont nous parlerons dans cette section, car tous les trois concernent Python et la Data Science.



Parlons tout d'abord de Jupyter Notebook, l'application que nous utiliserons tout au long de ce livre. Cette application permet donc de créer des notebooks avec Python, notamment, et dans notre cas, de créer des rapports d'analyse.

© Editions ENI - All rights reserved

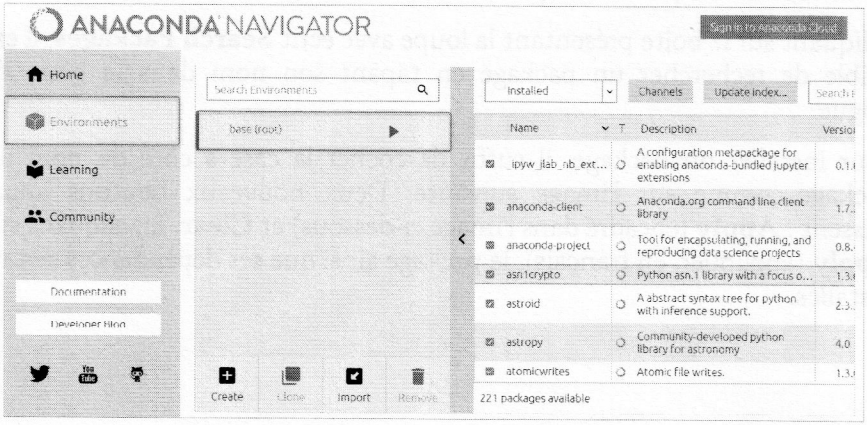
Jupyter Lab intègre Jupyter Notebook mais aussi d'autres fonctionnalités et est, dans un sens, plus compliquée à prendre en main et ne nous apporterait pas grand-chose par rapport à Jupyter Notebook. Ainsi, nous ne l'utiliserons pas dans ce livre car son interface est plus complexe que celle de Jupyter Notebook et parce que Jupyter Notebook est largement suffisant pour apprendre à utiliser Python pour le domaine de la Data Science.

Ensuite vient l'application Spyder (*Scientific Python Development EnviRonment*), qui est un environnement de développement intégré (abrégé IDE) pour Python adapté à l'usage scientifique, avec des librairies très utilisées en Data Science directement intégrées (Matplotlib, Numpy, etc.). Il a d'ailleurs été développé pour des scientifiques et ingénieurs analysant des données. Cet IDE offre de nombreuses fonctionnalités facilitant le développement d'applications avec Python.

Dans ce livre, nous utiliserons Jupyter Notebook pour apprendre à utiliser Python et ses librairies dédiées à la Data Science.

5.2 Gestion des packages et environnements

En plus de faciliter le lancement des applications, Anaconda Navigator facilite l'installation et la gestion des packages et environnements. Cette partie de gestion est disponible sous l'onglet **Environments**, encadré dans l'image ci-dessous.



Un environnement sous Anaconda correspond à un ensemble de packages installés, avec des versions particulières. En effet, certains packages nécessitent des versions spécifiques d'autres packages pour pouvoir fonctionner. Il est alors possible de créer des environnements différents, avec des versions de packages différentes, et de choisir son environnement avant de commencer son analyse. Ce seront les versions de packages de l'environnement chargé qui seront utilisées. Par défaut, un environnement existe sous Anaconda Navigator, qui est appelé **base (root)** (encadré sur l'image ci-dessus). Nous n'en créerons pas d'autres dans ce livre car cela n'est pas utile.

Remarque

Nous introduirons plus tard les notions de packages (ou librairies, les deux termes se valent), modules et fonctions en Python. Sachez juste qu'un package regroupe des modules, qui eux-mêmes regroupent des fonctions. Une fonction en Python est un bout de code permettant de faire un traitement. Python est populaire grâce à l'ensemble des packages qu'il fournit et qui facilitent grandement la programmation.

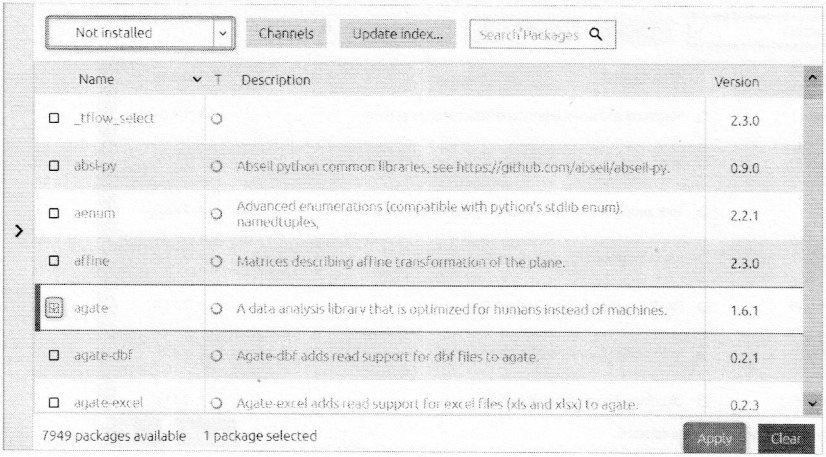
En cliquant sur l'environnement **base (root)**, vous verrez apparaître à droite la liste des packages disponibles dans l'environnement sélectionné. Par défaut, les packages installés (**Installed**) sont affichés, avec leur nom, une description de ce qu'ils permettent de faire, ainsi que leur version.

En cliquant sur le menu déroulant affichant **Installed**, il est alors possible d'afficher les packages non installés (**Not installed**) ou encore ceux pour lesquels une mise à jour est disponible (**Updatable**).

En cliquant sur la boîte présentant la loupe avec écrit **Search Packages**, il est possible de rechercher un package en tapant son nom dans la boîte de dialogue.

- Pour installer un package, il suffit de cocher la case à côté du nom du package, comme sur l'image suivante. Deux nouveaux boutons apparaissent : **Apply** (encadré dans l'image ci-dessous) et **Clear**. En cliquant sur **Apply** (appliquer, en français), le package ainsi que ses dépendances seront installés.

© Editions ENI - All rights reserved

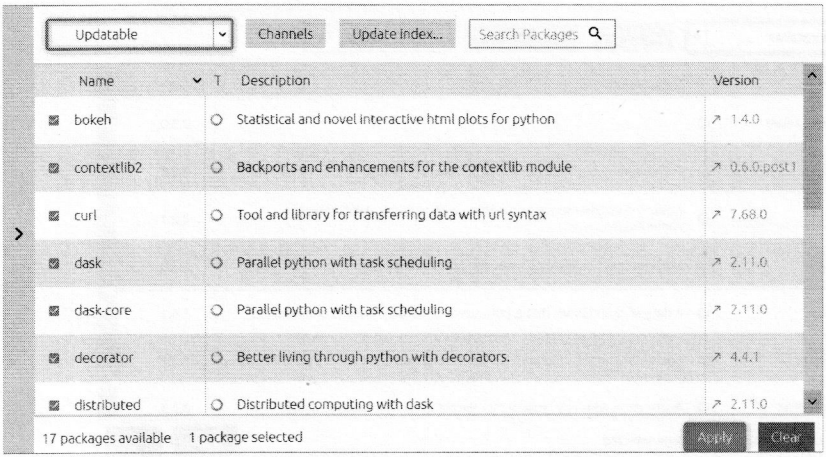


Enfin, à côté de la liste déroulante contenant le terme **Not installed** sur l'image ci-dessus, vous pouvez voir le terme **Channels**. Les *channels* (ou canaux en français) sont les localisations où Anaconda Navigator va chercher les packages. Il existe plusieurs channels contenant chacun des packages. Il est parfois nécessaire, pour installer un package, de devoir ajouter un channel dans son Anaconda Navigator, afin qu'il puisse le trouver. Nous n'en parlerons pas plus car nous garderons les channels par défaut que possède Anaconda Navigator et qui sont largement suffisants.

Par défaut, tous les packages essentiels à la Data Science avec Python, que nous utiliserons tout au long de ce livre, sont déjà installés. Votre environnement par défaut, **base (root)**, est déjà prêt pour apprendre à utiliser Python pour la Data Science.

Pour finir, actuellement, l'ensemble de vos packages doivent être à jour suite à l'installation de la dernière version d'Anaconda. Toutefois, il est conseillé de les mettre à jour manuellement de temps en temps.

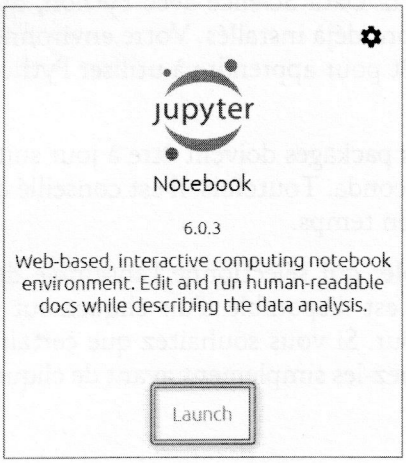
- Pour cela, il suffit de choisir **Updatable**, qui sélectionne l'ensemble des packages pour lesquels une mise à jour est disponible. Puis cliquez sur le bouton **Apply**, qui les mettra tous à jour. Si vous souhaitez que certains packages ne soient pas mis à jour, décochez-les simplement avant de cliquer sur le bouton **Apply**.



6. Prise en main de Jupyter Notebook

6.1 Tableau de bord de Jupyter Notebook

Pour ouvrir Jupyter Notebook, il faut ouvrir Anaconda Navigator, puis cliquer sur le bouton **Launch** situé sous l'application Jupyter Notebook.

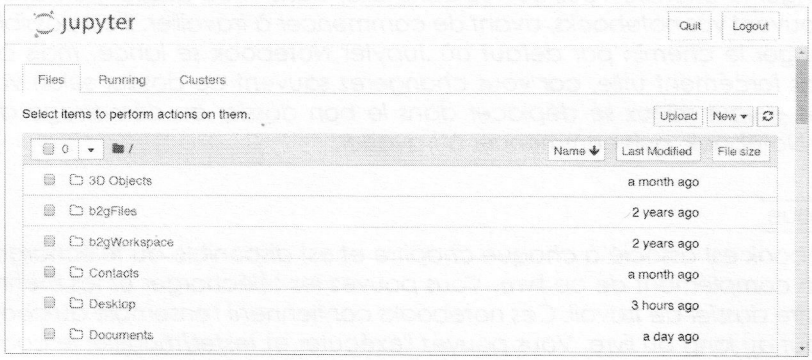


© Editions ENI - All rights reserved

L'application démarre et un navigateur web s'ouvre. En regardant l'adresse de l'application, vous pouvez voir qu'elle se lance sur le port 8888 de votre machine (ou éventuellement sur le port 8889 ou 8890).

Sur l'exemple de l'auteur, le lien est le suivant : http://localhost:8890/tree

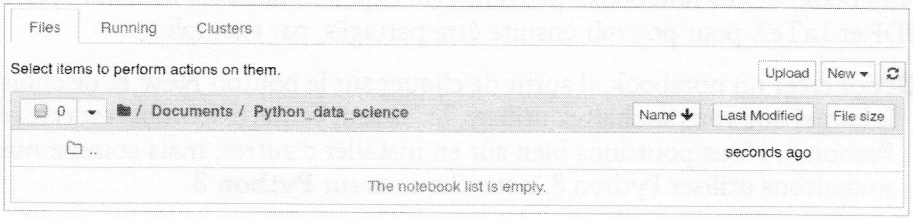
En lançant Jupyter Notebook, vous êtes renvoyé automatiquement sur son tableau de bord, que nous allons découvrir ensemble à présent.



Vous pouvez voir qu'il y a trois onglets disponibles sur le tableau de bord : **Files** (fichiers), **Running** (en cours d'exécution) et **Clusters** (grilles de calcul).

Le premier onglet **Files**, affiche les dossiers et fichiers du dossier dans lequel vous êtes, par défaut. Il est conseillé de vous créer un dossier de travail pour ce cours.

- Par exemple, nous pouvons créer un dossier nommé **Python_data_science** dans le dossier **Documents**, puis y accéder via l'interface Jupyter Notebook en cliquant sur le dossier **Documents**, puis **Python_data_science** dans l'onglet **Files**.



Nous voyons bien que l'arborescence a changé, nous sommes maintenant dans **Documents/Python_data_science**. Ce dossier est pour l'instant vide, mais plus tard, nous y créerons notre premier Notebook.

Remarque

Lorsque vous fermerez Jupyter Notebook et le relancerez, vous serez à nouveau dans votre dossier courant par défaut, que vous aviez au début avant de changer le chemin. Vous devrez donc aller dans votre dossier de travail, où se trouvent vos notebooks, avant de commencer à travailler. Il est possible de changer le chemin par défaut où Jupyter Notebook se lance, mais ce n'est pas forcément utile, car vous changerez souvent de dossier selon vos analyses. Il vaut mieux se déplacer dans le bon dossier au démarrage de Jupyter Notebook, puis commencer à travailler.

Remarque

Un notebook est associé à chaque chapitre et est disponible au téléchargement en complément de ce livre. Vous pouvez les télécharger et les mettre dans votre dossier de travail. Ces notebooks contiennent l'ensemble du code utilisé tout au long du livre. Vous pouvez l'exécuter et tester/modifier le code au fur et à mesure du chapitre que vous êtes en train de lire : c'est comme ça qu'on apprend le mieux.

L'onglet **Running** permet de visualiser les notebooks en cours d'exécution, nous verrons cela ensemble lorsque nous lancerons notre premier notebook.

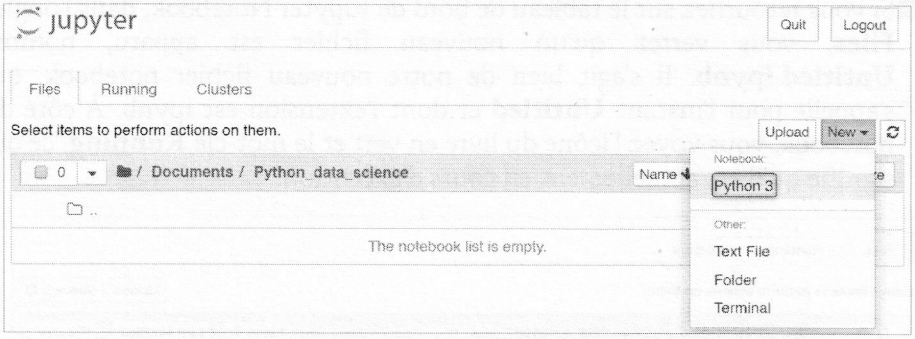
Enfin, l'onglet **Clusters**, qui ne sera pas traité dans ce livre, permet d'utiliser la parallélisation avec le package IPython parallel.

6.2 Premiers pas avec les notebooks

Un notebook est un fichier enregistré sous le format **ipynb** (pour IPython Notebook) et ces notebooks peuvent être exportés dans les formats HTML, PDF et LaTeX pour pouvoir ensuite être partagés, par exemple.

- Pour créer un notebook, il suffit de cliquer sur le bouton **New** et de choisir le kernel que vous souhaitez utiliser. Ici, le seul kernel disponible est celui de Python 3. Nous pourrions bien sûr en installer d'autres, mais comme nous souhaitons utiliser Python 3, nous cliquons sur **Python 3**.

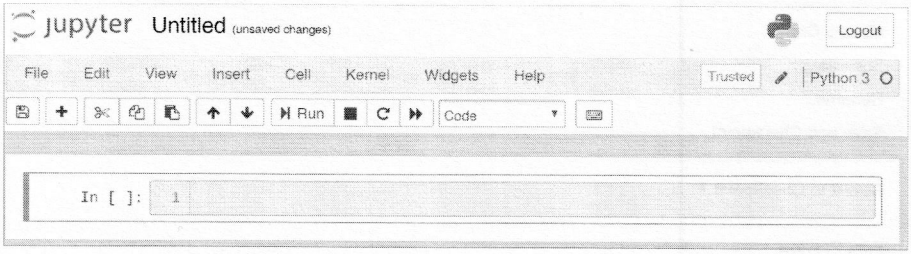
© Editions ENI - All rights reserved



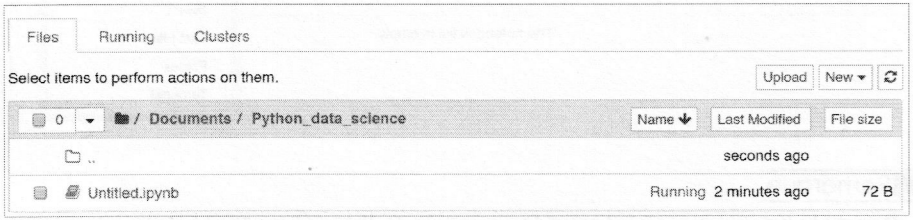
Remarque

Rappel : pour pouvoir interpréter le code Python, Jupyter Notebook a besoin d'un environnement d'exécution spécifique au langage de programmation utilisé. Cet environnement d'exécution s'appelle kernel, ou noyau en français, et est spécifique au langage. Pour Python, comme nous en avons déjà parlé en introduction, il s'agit du noyau IPython, qui est un programme permettant d'exécuter et interpréter du code Python dans Jupyter Notebook. Ainsi, en cliquant sur ***Python 3****, Jupyter Notebook va utiliser le noyau IPython.*

- Après avoir cliqué sur **Python 3**, un nouvel onglet s'ouvre dans votre navigateur web ; il s'agit de votre premier notebook. Pour l'instant, celui-ci n'a pas de nom particulier, il est appelé **Untitled** par défaut, ce qui signifie "sans titre".

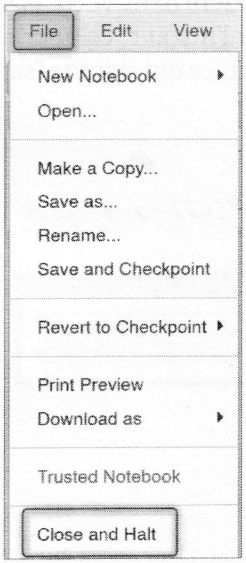


- Si vous retournez sur le tableau de bord de Jupyter Notebook, dans l'onglet **Files**, vous verrez qu'un nouveau fichier est apparu, nommé **Untitled.ipynb**. Il s'agit bien de notre nouveau fichier notebook, qui s'appelle pour l'instant **Untitled** et dont l'extension est ipynb. À côté du notebook, vous voyez l'icône du livre en vert et le mot-clé **Running**, ce qui signifie qu'il est actuellement en cours d'exécution.



De plus, si vous cliquez sur l'onglet **Running** du tableau de bord, vous verrez le nom de votre notebook apparaître.

- Si vous souhaitez éteindre proprement votre notebook, retournez sur l'onglet où il est en cours d'exécution, puis cliquez sur le bouton **File**. Une liste déroulante apparaît, cliquez sur le dernier champ **Close and Halt**.

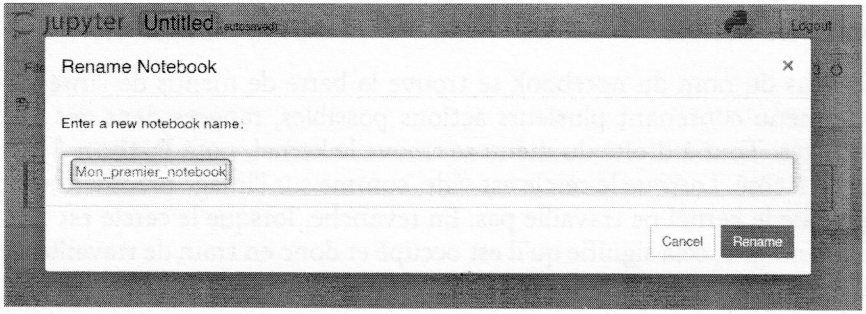


© Editions ENI - All rights reserved

L'onglet se ferme automatiquement et vous voyez que l'icône du livre à côté de votre notebook est devenue grise et que le mot **Running** n'est plus là. Votre notebook n'est donc plus en cours d'exécution.

- Pour le redémarrer, cliquez simplement sur le nom de fichier du notebook dans votre tableau de bord Jupyter Notebook, dans l'onglet **Files**, et le notebook se réexécutera.
- Enfin, pour renommer votre notebook, la façon la plus simple est de le faire via l'interface de votre notebook lorsque celui-ci est lancé, en cliquant sur le mot **Untitled**. Une fenêtre s'ouvre, vous entrez le nom de votre notebook souhaité puis vous cliquez sur **Rename** (renommer).

  Voir l'image ci-dessous :

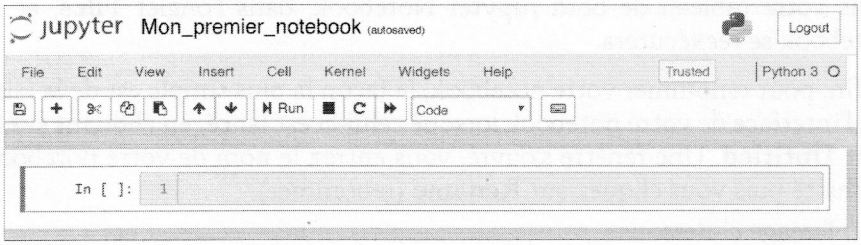


Remarque

Il existe d'autres manières d'éteindre les notebooks et de les renommer, mais vous ont été montrées celles qui paraissent les plus simples. Par ailleurs, pour le nommage des notebooks, nous vous conseillons de respecter une des règles principales en informatique : éviter les espaces. Cela ne pose pas de problème en soi pour Jupyter Notebook, mais préférez tout de même le symbole underscore ("_") aux espaces.

6.3 Comprendre l'interface des notebooks

Sur l'image ci-dessous se trouve un exemple d'interface utilisateur des notebooks Jupyter. Nous allons la parcourir ensemble.



6.3.1 La barre de menus

En dessous du nom du notebook se trouve la barre de menus de l'interface, chaque menu contenant plusieurs actions possibles, rangées dans des listes déroulantes. Tout à droite du menu se trouve le kernel, noté **Python 3**, avec un cercle à côté. Lorsque le cercle est vide, comme sur l'image précédente, cela signifie que le kernel ne travaille pas. En revanche, lorsque le cercle est plein, entièrement gris, cela signifie qu'il est occupé et donc en train de travailler (en train d'exécuter/interpréter du code par exemple).

Avec ces menus, vous pouvez agir sur le notebook, les cellules et le kernel. Voici une courte description de ce que permet de faire chaque menu.

- **File** : ce menu permet d'effectuer des actions sur le notebook. Il permet de créer un nouveau notebook ou d'en ouvrir un existant, de sauvegarder le notebook actuel ou encore de l'exporter dans un autre format, par exemple en PDF ou en HTML, si vous souhaitez partager votre analyse.
- **Edit** : ce menu permet d'effectuer diverses actions sur la (ou les) cellule sélectionnée, comme par exemple la couper (**cut**), la copier (**copy**), coller des cellules au-dessus de la cellule sélectionnée (**paste above**), en dessous (**paste below**) ou encore supprimer la cellule sélectionnée. Il permet aussi de fusionner des cellules (**merge**), de déplacer les cellules (**move**), etc.

© Editions ENI - All rights reserved

Remarque

Une cellule est représentée par un rectangle précédé d'un chiffre, par exemple 1 sur l'image précédente. Nous verrons plus en détails les cellules plus tard, sachez juste que se sont elles qui contiendront textes et code.

- **View** : ce menu permet de modifier l'affichage du notebook, par exemple, de cacher le titre du notebook en appuyant sur **Toggle Header** ou de le faire réapparaître en cliquant à nouveau sur **Toggle Header**. Il permet aussi de faire apparaître les numéros de ligne dans les cellules de code, en cliquant sur **Toggle Line Numbers**.
- **Insert** : ce menu permet d'insérer une nouvelle cellule avant (**above**) ou après (**below**) la cellule sélectionnée.
- **Cell** : ce menu permet d'exécuter les cellules de différentes manières. Par exemple, exécuter la ou les cellule(s) sélectionnée(s), exécuter l'ensemble des cellules du notebook, etc. Il permet aussi de modifier le type de la ou les cellules(s) sélectionnée(s), en allant dans **Cell Type** puis en choisissant le type d'intérêt. Nous verrons ces types de cellules dans la section Les cellules.
- **Kernel** : ce menu permet d'agir sur le kernel lui-même. Vous pouvez interrompre le kernel, le redémarrer, le redémarrer et exécuter toutes les cellules ou encore l'éteindre. Quand vous êtes sur le point de partager un notebook, il est conseillé de redémarrer le kernel et de réexécuter l'ensemble des cellules, pour être sûr qu'il n'y a pas de soucis et que l'ensemble des résultats sont bien générés (**Kernel -> Restart & Run All**).
- **Help** : le menu **Help** permet d'accéder à l'aide des différents raccourcis-clavier ou encore à l'aide (en anglais) des différents packages que nous verrons dans ce livre (NumPy, SciPy, Matplotlib ou encore Pandas).

À présent, vous avez un aperçu du menu. Toutefois, vous le verrez dans la section sur les modes Commande (**Command Mode**) et Edition (**Edit Mode**), il existe beaucoup de raccourcis-clavier qui permettent d'effectuer ce que propose la barre de menus. Une fois ces derniers maîtrisés, vous passerez rarement par cette barre.

6.3.2 La barre d'outils

La barre d'outils se situe sous la barre de menus et propose des raccourcis pour effectuer des actions courantes.

Le bouton de la disquette permet d'enregistrer les modifications apportées au notebook.

Le bouton + permet d'ajouter une nouvelle cellule en dessous de la cellule sélectionnée.

Le bouton ciseaux permet de couper la cellule sélectionnée.

Le bouton des deux fichiers superposés permet de copier la ou les cellules sélectionnées.

Le bouton à droite de celui-ci permet de coller ce qui a été copié en dessous de la ou les cellule(s) copiée(s).

Les flèches du haut et du bas permettent de naviguer entre les cellules du notebook, la flèche du haut permettant de remonter dans les cellules, la flèche du bas permettant de descendre.

Le bouton **Run** permet d'exécuter la ou les cellule(s) sélectionnée(s).

Le bouton avec le carré plein permet de stopper le kernel (et les exécutions en cours).

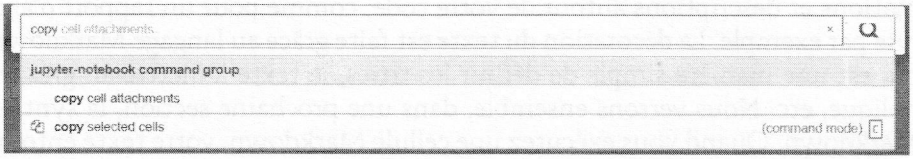
Le bouton à droite de celui-ci permet de redémarrer le kernel.

Le bouton des deux flèches permet de redémarrer le kernel et réexécuter l'ensemble des cellules du notebook.

Enfin, la liste déroulante à droite permet de sélectionner le type de la cellule.

© Editions ENI - All rights reserved

Et pour finir, la dernière icône correspond à la palette de commandes, ou **Command Palette** : elle permet de trouver une commande par son nom et de l'exécuter, voire de trouver le raccourci correspondant. Par exemple, si vous tapez le terme "copy" dans la barre de recherche de la palette de commandes, voici le résultat :



Deux commandes sont possibles, dont celle pour copier les cellules sélectionnées, **copy selected cells**. De plus, on obtient le raccourci pour copier les cellules sélectionnées : la lettre C dans le mode **Command**, que nous verrons plus tard dans ce chapitre.

6.3.3 Les cellules

Lorsque vous créez votre notebook, une cellule de code vide est créée automatiquement. Il existe différents types de cellules, ceux majoritairement utilisés étant les types Code et Markdown. Par défaut, lorsque vous créez une cellule, il s'agit d'une cellule de code. Pour changer le type de cette cellule, il faudra utiliser un raccourci-clavier que nous verrons plus tard, ou sélectionner la cellule et changer son type via la liste déroulante de la barre d'outils.

- Allez dans la barre d'outils sur la liste déroulante contenant le mot **Code** et changez ce mot-clé à **Markdown** : cela vous permettra d'écrire du texte dans cette cellule, qui ne sera pas interprété comme du code Python par le kernel.

Une cellule de type Code permet d'écrire du code, ici du code Python puisque nous avons choisi le kernel IPython. Les cellules de code présentent des fonctionnalités spécifiques au code, comme la coloration syntaxique de celui-ci. Lorsqu'on écrit dans une cellule, celle-ci n'est pas automatiquement exécutée.

- Pour exécuter une cellule, il suffit d'utiliser le raccourci [Ctrl][Entrée] ou de cliquer sur le bouton **Run** de la barre d'outils. En l'exécutant, le code contenu dans la cellule sera envoyé au kernel, qui l'interprétera et retournera le résultat à afficher sous le code contenu dans cette cellule.

Une cellule de type Markdown permet d'écrire du texte et d'ajouter des informations et descriptions autour de votre code, comme pour un rapport d'analyse par exemple. La décoration du texte est faite grâce au langage Markdown, qui est une manière simple de définir les titres, le texte à mettre en gras, en italique, etc. Nous verrons ensemble, dans une prochaine section, la syntaxe Markdown. Quand vous exécutez une cellule Markdown, votre texte entouré de balises Markdown devient du texte joliment formaté.

Ce qui permet de différencier le code de la sortie, c'est le texte In suivi des symboles [ ] à gauche de chaque cellule, qui reste même une fois le rapport généré, ce qui permet efficacement de différencier le code des résultats obtenus à partir de ce code. Le In signifie Input, pour "Entrée" en français. Quand vous transformez votre cellule de code en Markdown, vous verrez que ces crochets disparaissent, car il ne s'agit plus de code à exécuter.

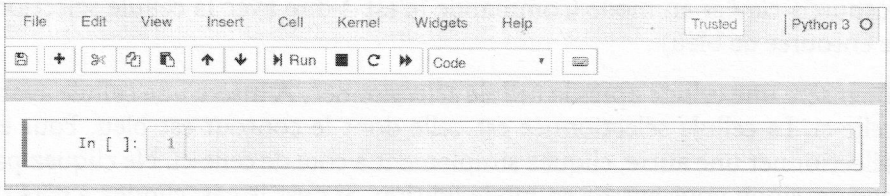
Une fois la cellule exécutée, on constate qu'un chiffre apparaît entre les crochets, cela permet d'avoir l'ordre d'exécution des commandes dans le notebook. Par exemple, la première commande qu'on exécute dans le notebook contient le chiffre 1 entre crochets, la dixième commande contient le nombre 10 entre crochets et ainsi de suite. Lorsque le symbole * apparaît entre les crochets, c'est que la cellule est en cours d'exécution dans le kernel. Vous rencontrerez cette * lorsque vous commencerez à travailler sous Jupyter Notebook.

6.3.4 Les modes Commande et Edition et les raccourcis-clavier

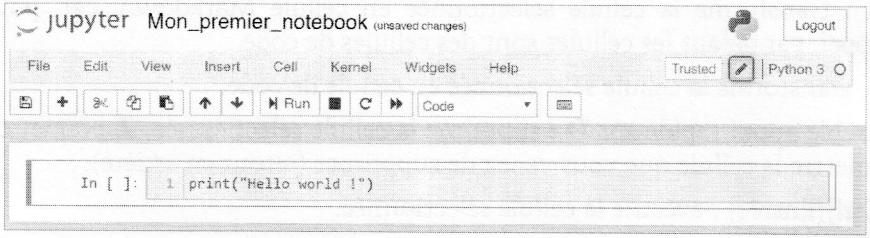
Il faut savoir que le notebook possède deux modes : le mode de commande et le mode d'édition.

Par défaut, en ouvrant votre notebook, vous êtes dans le mode de commande, qui permet de faire différentes actions sur le notebook et les cellules. Des raccourcis-clavier spécifiques à ce mode sont disponibles, et cela fait toute la force des notebooks Jupyter. Lorsque vous êtes en mode de commande, le contour de la cellule sélectionnée est bleu.

© Editions ENI - All rights reserved



Pour passer en mode Edition, appuyez sur [Entrée] ou cliquez sur la cellule avec votre souris et vous verrez un petit crayon apparaître à côté de votre Kernel. De plus, le contour de la cellule sélectionnée devient vert. Ce mode permet d'écrire dans la cellule sélectionnée ; vous pouvez par exemple y écrire le code que vous souhaitez envoyer au kernel.



- Pour repasser en mode Commande, appuyez sur [Echap] ou cliquez en dehors de la cellule avec votre souris.

Chaque mode a ses raccourcis-clavier qui sont disponibles dans l'aide du notebook en cliquant sur **Help** dans la barre des menus puis sur **Keybord Shortcuts**, qui signifie raccourcis-clavier. Une nouvelle fenêtre s'ouvre avec l'ensemble des raccourcis disponibles, dont nous allons voir les plus utiles ensemble. Prenez l'habitude d'utiliser les raccourcis-clavier, cela vous rendra beaucoup plus efficace dans votre manipulation des notebooks de Jupyter.

Comme vous pouvez le voir, les raccourcis-clavier sont spécifiques à chaque mode.

Raccourcis-clavier en mode Commande (c'est-à-dire avec la cellule sélectionnée entourée de bleu) :

- **B** : insère une cellule après la cellule sélectionnée, **A** insère une cellule avant celle-ci. La cellule sélectionnée est celle dont le contour est bleu. Pour en sélectionner une autre, cliquez simplement à côté de celle-ci. Ne cliquez pas dessus, sinon vous entrez en mode Edition. Si c'est le cas, cliquez à côté de la cellule pour repasser en mode Commande.
- **F** : ouvre une fenêtre qui permet de chercher-remplacer du texte dans l'ensemble de votre notebook. Dans le cas où vous souhaitez simplement chercher une occurrence dans votre notebook, sans la remplacer, utilisez le raccourci [Ctrl] **F**.
- **M** : transforme la cellule sélectionnée en cellule Markdown, car, pour rappel, par défaut les cellules sont des cellules de code.
- **Y** : transforme la cellule sélectionnée en cellule de code.
- Double appui rapide sur **D** : supprime la cellule sélectionnée. **Z** permet de remettre la cellule que vous venez de supprimer (retour en arrière).
- [Ctrl][Entrée] : exécute la cellule sélectionnée.
- [Shift][Entrée] : exécute la cellule sélectionnée et sélectionne la cellule du dessous, ce qui est pratique pour exécuter plusieurs cellules à la suite.
- [Flèches en haut] et [Flèche en bas] : naviguent entre les cellules et en sélectionnent une autre, par exemple.
- [Ctrl] **S** : enregistre les modifications faites dans le notebook.

Raccourcis en mode Edition :

- [Ctrl] **A** : sélectionne tout le code/texte dans une cellule.
- [Tab] : indente votre code ou utilise la complétion de code.
- [Ctrl] **Z** : retour en arrière.
- [Ctrl] / : commente votre code (fonctionne dans les cellules de code uniquement). Commenter son code signifie que celui-ci ne sera plus interprété/exécuté par le kernel.
- [Ctrl] **S** enregistre les modifications.

© Editions ENI - All rights reserved

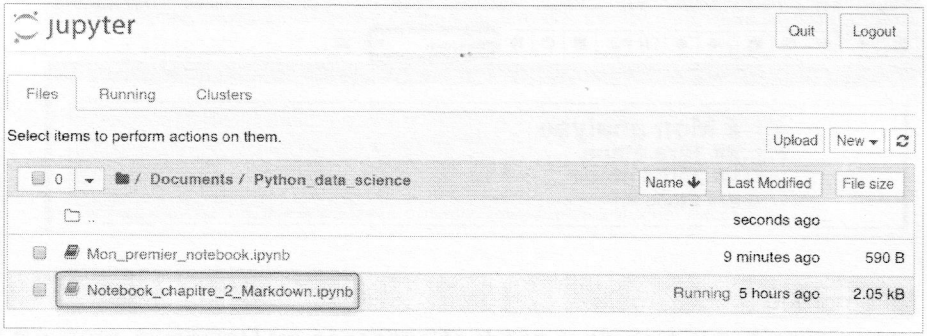
– Pour sélectionner plusieurs cellules, sélectionnez une cellule et appuyez sur [Shift] **J** autant de fois que le nombre de cellules en dessous que vous souhaitez sélectionner.

6.3.5 Les bases du langage Markdown pour écrire dans un notebook

Tout au long de ce cours, il vous sera expliqué comment manipuler et visualiser des données dans un notebook grâce à du code Python, mais il est aussi important de décrire ce que font ces commandes et pourquoi vous les lancez. C'est pour cela qu'un notebook permet d'exécuter du code dans les cellules de code, mais aussi d'écrire du texte dans les cellules Markdown, grâce au langage Markdown. Le langage Markdown permet de formater du texte ; en voici les bases.

Ce chapitre possède un notebook disponible en téléchargement, que vous pouvez copier-coller dans votre dossier de travail, puis sur lequel vous pouvez cliquer via le tableau de bord Jupyter Notebook afin de l'exécuter et suivre les commandes.

Voici un exemple visuel :



Remarque

À présent, pour chaque nouveau chapitre, vous pouvez télécharger et copier-coller le notebook correspondant puis l'exécuter pour suivre les différentes commandes. Chaque notebook contient le numéro du chapitre correspondant dans son nom.

En Markdown, le # permet de définir les titres et doit être placé avant le texte. Un unique # définit un titre de niveau 1. Deux # (##) permettent de définir un titre de niveau 2, etc.

Syntaxe générale

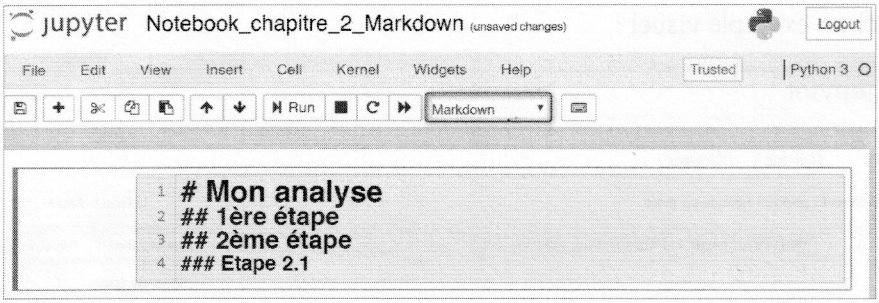
```
# mon titre 1
## mon titre 2
### mon titre 3
```

Écrivons nos premières lignes de texte en Markdown.

Code dans une cellule de type Markdown

```
# Mon analyse
## 1ère étape
## 2ème étape
### Etape 2.1
```

Si on n'exécute pas cette cellule, celle-ci ressemble à l'image ci-dessous :

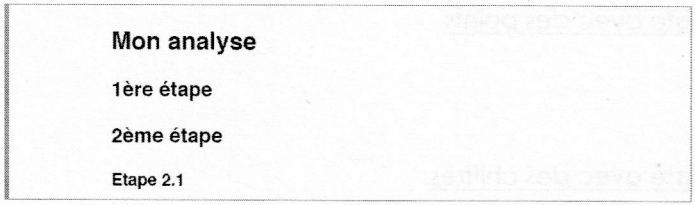


Remarque

Notez bien que le type de la cellule sélectionnée est Markdown, pour que le code puisse fonctionner.

© Editions ENI - All rights reserved

En cliquant sur [Ctrl][Entrée] pour exécuter la cellule, le texte devient du texte formaté et joliment présenté, comme sur l'image ci-dessous :



On constate que le texte de la première ligne est plus grand que celui de la deuxième, nous avons bien mis en place des titres de niveaux différents.

Remarque

Lorsque vous exécutez la cellule Markdown, celle-ci disparaît et laisse place au texte formaté. Si vous souhaitez revenir sur la cellule pour modifier le texte, cliquez deux fois sur celui-ci avec la souris.

Pour écrire un paragraphe, le texte n'a pas besoin d'être précédé d'un symbole. Vous avez juste besoin d'écrire le texte que vous souhaitez. Pour passer à un autre paragraphe, il faut aller deux fois à la ligne, c'est-à-dire laisser une ligne vide entre vos paragraphes, pour que cela soit pris en compte par Markdown.

Syntaxe

```
Mon premier paragraphe

Mon deuxième paragraphe

Mon troisième paragraphe
```

Pour que le texte apparaisse en gras, il faut l'encadrer de deux symboles étoiles : **.

Syntaxe

```
** mon texte en gras **
```

Pour qu'il apparaisse en italique, il faut encadrer le texte du symbole *.

Syntaxe

```
* mon texte en italique *
```

Pour créer une liste, il faut précéder le début du texte par une *, ou par des chiffres pour avoir une liste numérique.

Syntaxe pour une liste avec des points

```
* élément 1
* élément 2
* élément 3
```

Syntaxe pour une liste avec des chiffres

```
1. élément 1
2. élément 2
3. élément 3
```

Pour écrire du code Python, il faut l'entourer de trois accents graves.

Syntaxe

````
```
print("Hello")
a= 3+2
print(a)
```
````

Il est aussi possible d'ajouter des images. Par exemple, en mettant une image appelée `22.PNG` dans le même dossier que votre notebook, la syntaxe sera :

```
<img src="22.PNG"/>
```

Il est possible d'intégrer des liens avec la syntaxe suivante :

```
[blue_text](https://www.mon_lien.fr/)
```

Le texte `blue_text` apparaîtra en bleu et, en cliquant dessus, un navigateur web s'ouvrira et renverra vers l'adresse https://www.mon_lien.fr/.

Il est enfin possible d'écrire des formules mathématiques en utilisant la notation LaTeX.

Syntaxe permettant décrire une formule mathématique

```
$ formule $
```

Exemple de code

```
$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$
```

© Editions ENI - All rights reserved

Ces bases sont suffisantes pour écrire du texte dans votre notebook.

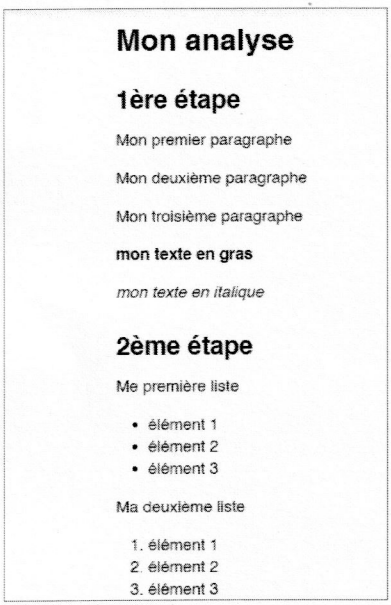
Voici le code Markdown des exemples que nous avons vus ensemble.

Code

```
# Mon analyse

## 1ère étape

Mon premier paragraphe

Mon deuxième paragraphe

Mon troisième paragraphe


*mon texte en gras**
*mon texte en italique*

## 2ème étape

Ma première liste

* élément 1
* élément 2
* élément 3

Ma deuxième liste

1. élément 1
2. élément 2
3. élément 3

### Etape 2.1

Mon code Python :

```
print("Hello")
a= 3+2
print(a)
```
```

```
Mon lien :

[Mon lien]( https://www.mon_lien.fr/)

Ma formule mathématique

$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$

Mon image :

<img style="float: left;" src="22.PNG"/>
```

Sur l'image ci-dessous, voici le rendu du résultat.



Mon analyse

1ère étape

Mon premier paragraphe

Mon deuxième paragraphe

Mon troisième paragraphe

mon texte en gras

mon texte en italique

2ème étape

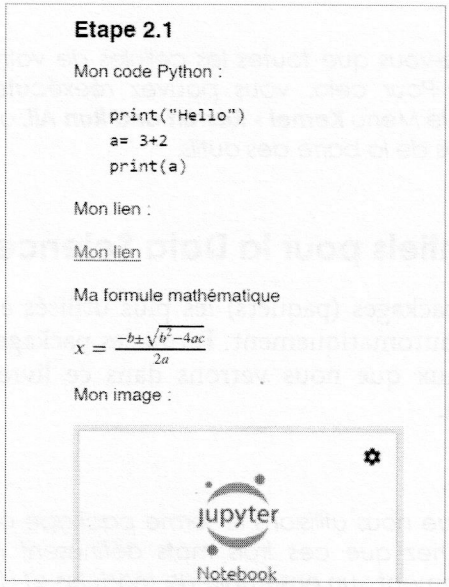
Me première liste

- élément 1
- élément 2
- élément 3

Ma deuxième liste

1. élément 1
2. élément 2
3. élément 3

© Editions ENI - All rights reserved



Vous connaissez à présent les bases du langage Markdown pour pouvoir écrire du texte joliment formaté autour de votre code Python, afin de créer un rapport d'analyse.

6.3.6 Partager son notebook

Pour partager votre notebook, vous pouvez l'exporter au format HTML, PDF, LATEX, etc.

- Pour cela, il vous faut cliquer sur le menu **File** du notebook, puis **Download As**, et choisir le format qui vous intéresse. Bien sûr, en faisant cela, votre notebook n'est plus interactif. Les personnes avec qui vous le partagez peuvent donc en lire le contenu mais ne peuvent pas le modifier.

Remarque

Il est possible que vous rencontriez des problèmes lors de la conversion de votre notebook en PDF. Assurez-vous bien d'avoir l'outil LaTeX installé sur votre ordinateur. Sinon, utilisez le format HTML pour ne pas rencontrer ce genre de problème.

Remarque

Attention, avant de partager, assurez-vous que toutes les cellules de votre notebook ont bien été exécutées. Pour cela, vous pouvez réexécuter l'ensemble des cellules en cliquant sur le Menu ***Kernel - Restart and Run All****, ou en cliquant sur l'icône des deux flèches de la barre des outils.*

7. Les packages Python essentiels pour la Data Science

Quand vous installez Anaconda, les packages (paquets) les plus utilisés en Data Science sont installés également automatiquement. Parmi ces packages figurent les quatre packages principaux que nous verrons dans ce livre : NumPy, Pandas, Matplotlib et Seaborn.

Remarque

Tout au long du livre, il est possible que nous utilisions le terme package ou librairie ou encore bibliothèque. Sachez que ces trois mots définissent la même chose en Python, conceptuellement : un ensemble de modules et de fonctions. Ces termes seront expliqués plus tard dans le livre.

7.1 NumPy

NumPy est le package le plus utilisé en calcul scientifique et sert de base pour de nombreux autres packages qui dépendent de lui. Ce package a été optimisé pour traiter de grands tableaux ou matrices multidimensionnelles très rapidement et propose de nombreuses méthodes clé en main pour effectuer des opérations diverses sur ces tableaux ou matrices.

© Editions ENI - All rights reserved

7.2 Pandas

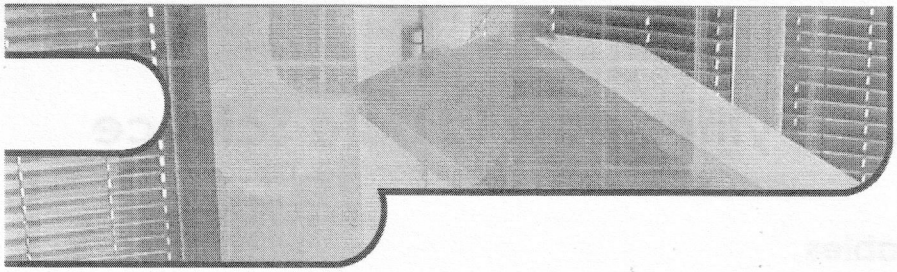
Pandas est un package dédié à l'analyse de données et propose une structure originale appelée **DataFrame**, très utilisée en analyse de données. Le package Pandas est basé sur NumPy. Cette librairie est fournie avec un nombre impressionnant de méthodes qui facilitent grandement le travail du Data Scientist. Le but de ce package est vraiment de permettre d'explorer, filtrer, et manipuler des données très facilement.

7.3 Matplotlib

Matplotlib est le package Python le plus connu pour faire de la visualisation de données. Il permet, à partir de tableaux de données, de créer divers graphiques comme des nuages de points, des boîtes à moustache, etc. Ce package est très complet et permet de paramétrer finement chaque composant du graphique.

7.4 Seaborn

Seaborn est aussi un package Python très connu pour la visualisation de données. Il est basé sur Matplotlib et ajoute des fonctionnalités qui en font un package de plus haut niveau, permettant de créer des graphiques plus complexes.



Chapitre 3
Rappels sur le langage Python

1. Introduction sur le langage de programmation Python

Python est un langage de programmation dont la version publique a été rendue disponible en 1991. Depuis, ce langage de programmation a fait son chemin, actuellement nous en sommes à la version 3.7. La version 2 de Python (2.7) n'est plus maintenue depuis janvier 2020, il n'est donc plus conseillé de l'utiliser.

Python est un langage de programmation gratuit disponible sous Windows, Mac et Linux, notamment. Il s'agit d'un langage de haut niveau, permettant de faire abstraction du fonctionnement de l'ordinateur (allocation mémoire, pointeurs, etc.), ce qui le rend plus accessible qu'un langage bas niveau comme C par exemple.

C'est un langage dit interprété, c'est-à-dire qu'il n'y a pas besoin de compiler son script pour pouvoir l'utiliser. Le code du script est lu ligne par ligne et traduit à l'ordinateur par ce qu'on appelle un interpréteur.

Ce chapitre proposera quelques rappels sur le langage Python avant d'enchaîner sur les librairies essentielles en Data Science et leur utilisation.

2. Les variables

Il est inconcevable d'écrire un programme sans utiliser de variable. Une variable consiste à attribuer un nom à une information. On range cette information dans une variable que l'on nomme, ce qui permet ensuite d'accéder à cette information en appelant simplement la variable.

Syntaxe pour assigner une information à une variable

```
a = 11
```

Une fois le nombre `11` assigné à la variable `a`, on peut y accéder en tapant simplement `a` dans l'interpréteur Python, ce qui affichera le contenu de la variable, `11`.

Si nous écrivons juste `11` dans la console Python, le résultat est affiché directement, mais ensuite nous n'avons aucun moyen de rappeler cette information. Ceci est dû au fait que la valeur `11` n'est stockée nulle part, on ne pourra pas la réutiliser, alors que si elle est stockée dans une variable, elle est accessible à n'importe quel moment.

Le nom d'une variable peut contenir des lettres, des chiffres ou encore des underscores _, mais il ne peut pas commencer par un chiffre. Attention, Python est sensible à la casse, c'est-à-dire qu'il fait une différence entre minuscules et majuscules.

Ainsi, si nous écrivons :

```
1a=11
```

On obtient une erreur, Python ne veut pas de chiffre devant un nom de variable. Mais :

```
a1=11
```

va fonctionner.

© Editions ENI - All rights reserved

De plus si nous écrivons le code suivant :

Code

```
m=3
M=4
print(M)
print(m)
```

Résultat

```
4
3
```

On a bien 4 puis 3 qui s'affichent ; ce sont deux variables différentes car Python différencie les majuscules et minuscules.

Alors que si nous écrivons :

Code

```
m=2
print(m)
```

Résultat

```
2
```

Python affiche 2, nous avons donc ainsi modifié la valeur de la variable m.

Remarque

La fonction `print()` de Python permet d'afficher quelque chose comme par exemple du texte : `print("Hello")`, ou encore le contenu d'une variable : `print(ma_variable)`.

3. Les différents types de données (int, float, bool, str)

3.1 Les nombres réels et entiers

Le type de données ou le type d'une variable indique à l'interpréteur Python ce qu'il peut faire ou ne pas faire avec cette variable.

Par exemple, précédemment, nous manipulions des chiffres ou nombres entiers, c'est ce qu'on appelle le type `int` pour *integer* (entier, en français). La fonction `type()` permet d'afficher le type d'une valeur.

Code

```
type(11)
```

Résultat

```
int
```

Si nous mettons une virgule à la valeur, on obtient un nombre réel (un nombre à virgule) et le type sera `float`.

Code

```
type(11.1)
```

Résultat

```
float
```

Il existe donc deux types de nombres en Python ; les nombres réels (`float`) et les nombres entiers (`int`).

Il est possible de transformer un integer en float et vice versa, grâce aux fonctions `int()` et `float()`.

Code

```
int(3.5)
```

© Editions ENI - All rights reserved

Résultat

```
3
```

Code

```
float(3)
```

Résultat

```
3.0
```

Il est possible de faire des opérations arithmétiques sur les types `int` et `float`, nous le verrons dans une future section de ce chapitre.

3.2 Les booléens

Les booléens résultent d'opérations logiques et ont deux valeurs possibles : `TRUE` ou `FALSE`, soit VRAI ou FAUX. Plus simplement, ce type de données permet de savoir si une condition est vraie ou fausse. C'est assez intuitif, voici un exemple :

Code

```
3 < 5
```

Résultat

```
True
```

Code

```
3 > 5
```

Résultat

```
False
```

Ici, le premier code demande à Python si 3 est inférieur à 5 et il renvoie `True`, donc "vrai". Ensuite, on lui demande si 3 est supérieur à 5 et il répond `False`, donc "faux".

C'est ce qu'on appelle des opérateurs de comparaison, qui vont retourner des booléens selon que le résultat est vrai ou faux. Nous verrons en détail les opérateurs relationnels et logiques plus tard. Mais gardez en tête que ce type existe et qu'en programmation, selon la valeur qu'il retourne, des décisions différentes sont prises par le programme informatique.

Pour afficher le type de 3 > 5, on utilisera le code suivant :

```
type(3>5)
```

Résultat

```
bool
```

3.3 Les chaînes de caractères

Le type "chaîne de caractères" permet d'écrire une suite finie de caractères, qu'on peut aussi définir comme du texte. Pour créer une chaîne de caractères, il faut mettre du texte entre doubles guillemets ou simples guillemets.

Syntaxe

```
"mon texte"
```

Code

```
print("mon texte")
```

Résultat

```
mon texte
```

On voit que c'est bien la phrase `mon texte` qui est affichée.

Regardons le type de `mon texte`.

Code

```
type("mon texte")
```

Résultat

```
str
```

© Editions ENI - All rights reserved

`str` signifie *string* en anglais, ou encore phrase en français.

Un nombre peut aussi être considéré comme du texte, s'il est mis entre guillemets.

Code

```
type("15")
```

Résultat

```
str
```

Mais il ne se comporte plus comme un nombre ici.

Par exemple, avec deux nombres, on peut utiliser Python comme une calculatrice et les additionner.

Code

```
5 + 15
```

Résultat

```
20
```

En revanche, on ne peut plus additionner la chaîne de caractères "15" avec un nombre comme précédemment, car il n'est plus considéré comme un nombre, mais bien comme du texte.

Code

```
"15" + 2
```

Résultat

```
TypeError: can only concatenate str (not "int") to str
```

On obtient une erreur, car on ne peut pas additionner du texte et un nombre.

Enfin, si on veut transformer un nombre en chaîne de caractères, on utilisera la fonction `str()`.

Code

```
str(15)
```

Résultat

```
'15'
```

4. Les structures de données basiques (listes, tuples et dictionnaires)

Nous avons vu précédemment comment ranger de l'information dans une variable. À présent, nous allons voir comment structurer les informations, c'est-à-dire la ranger dans des structures qui permettent de correctement les séparer, les stocker et les appeler au besoin, ensemble ou séparément.

4.1 Les listes

Les types que nous avons vus dans la section précédente, nombres ou caractères, ne sont pas suffisants pour traiter des données complexes. Il faut que nous puissions créer des objets pour stocker l'information.

Une première manière de stocker de l'information en Python est d'utiliser les listes. Une liste est un nouveau type de variable, qui permet de stocker plusieurs éléments, plusieurs valeurs, qui sont ordonnés, modifiables et peuvent être de type différent (`int`, `float`, `str` ou `bool`). Ainsi, on peut avoir une liste contenant uniquement des nombres entiers, uniquement des nombres réels, uniquement des chaînes de caractères, uniquement des booléens ou encore les quatre mélangés. De plus, une liste peut contenir elle-même des listes.

4.1.1 Créer une liste

Visuellement, on peut voir les listes comme des vecteurs, ou encore des tableaux à une dimension.

Pour créer une liste, nous allons utiliser des crochets, entre lesquels nous mettrons l'ensemble de nos valeurs, séparées par une virgule.

© Editions ENI - All rights reserved

Syntaxe

```
ma_liste = [ma_premiere_valeur, ma_deuxieme_valeur,
ma_troisieme_valeur, ...]
```

Créons notre première liste avec le code suivant :

```
ma_liste = [1,2,3,4,5,6,7,8,"hello" ,10.5]
print(type(ma_liste))
print(ma_liste)
```

Résultat

```
<class 'list'>
[1, 2, 3, 4, 5, 6, 7, 8, 'hello', 10.5]
```

Comme la liste est une structure de données ordonnée, les valeurs sont rangées dans l'ordre où on les donne. Ici, nos valeurs sont stockées dans une liste (grâce aux crochets), elle-même assignée à la variable `ma_liste` (grâce au symbole d'assignation =). En affichant la liste avec la fonction `print()`, on voit bien qu'on a une suite de valeurs, séparées par des virgules, et qu'il n'y a aucun problème à mélanger les types. Oui, mais comment accéder à ces valeurs ? C'est ce que nous allons voir tout de suite.

4.1.2 Accéder aux éléments d'une liste

Pour accéder aux éléments d'une liste, il faut que nous introduisions le terme index, ou indice en français. En fait, pour afficher une valeur particulière d'une liste, on donnera l'index auquel se trouve la valeur qu'on souhaite afficher, c'est-à-dire la position de la valeur dans la liste.

Syntaxe

```
ma_liste[index]
```

Il faut savoir que, comme beaucoup de langages de programmation, Python commence à compter à partir de `0`. Donc pour afficher la première valeur de la liste, il faut donner l'index `0`, pour afficher la deuxième valeur de la liste, il faut donner l'index `1`, etc.

Code

```
print(ma_liste[1])
```

Résultat

```
2
```

Ici, nous lui donnons le nom de notre liste avec, entre crochets, l'index auquel nous souhaitons accéder. Python voit qu'à l'index `1` de notre liste, donc à la deuxième position de `ma_liste`, il y a la valeur `2` et il nous l'affiche grâce à la fonction `print()`.

Plutôt que de donner un seul index, Python permet d'effectuer du slicing sur les listes, c'est-à-dire du découpage en français.

Syntaxe

```
ma_liste[start:stop:step]
```

Ici, plutôt que d'accéder aux valeurs une par une grâce à l'indexage simple (où on ne donne qu'un index), on va donner l'index de départ, l'index de fin et le pas (par défaut le pas est de 1, si on met 2, par exemple, Python saute une valeur sur 2).

Code

```
ma_liste[2:4]
```

Résultat

```
[3, 4]
```

Python retourne une liste contenant les valeurs des index `2` et `3`, c'est-à-dire les valeurs `3` et `4` de `ma_liste`. Instinctivement, on pourrait penser qu'en écrivant `2:4`, Python donnerait les valeurs aux index `2`, `3` et `4`, mais non, il donne les valeurs jusqu'à l'index `n-1`. C'est une gymnastique à prendre.

Si on ne donne pas d'index au début, Python part du début de la liste.

Code

```
ma_liste[:4]
```

Résultat

```
[1, 2, 3, 4]
```

Python affiche `1`, `2`, `3` et `4`, les valeurs des index `0`, `1`, `2`, `3`.

© Editions ENI - All rights reserved

Si on ne met rien comme dernier index, Python affichera la liste à partir de l'index qu'on lui donne jusqu'à la fin de la liste.

Code

```
ma_liste[3:]
```

Résultat

```
[4, 5, 6, 7, 8, 'hello', 10.5]
```

On peut aussi donner une valeur de début, une valeur de fin et une valeur de pas. Le pas est de 1 par défaut, mais si nous le mettons à 2, Python saute une valeur sur deux.

Code

```
ma_liste[0:3:2]
```

Résultat

```
[1, 3]
```

Python affiche 1 et 3 car en partant de 0 jusqu'à 2, il y a 3 valeurs, 1, 2, 3 mais comme on lui a demandé un pas de 2, donc d'afficher les valeurs tous les deux pas dans cet intervalle, il affichera seulement 1 et 3.

Voici quelques astuces de slicing supplémentaires.

Afficher les valeurs des index pairs, en partant de l'index 0 (premier index pair) jusqu'à la fin de la liste, avec un pas de 2.

Code

```
ma_liste[::2]
```

Résultat

```
[1, 3, 5, 7, 'hello']
```

Afficher les valeurs des index impairs, en partant de l'index 1 (premier index impair) jusqu'à la fin de la liste, avec un pas de 2.

Code

```
ma_liste[1::2]
```

Résultat

```
[2, 4, 6, 8, 10.5]
```

Si on souhaite partir de la fin de la liste, il suffit d'utiliser des nombres négatifs.

Code

```
ma_liste[-2]
```

Résultat

```
'hello'
```

Enfin, on peut aussi partir d'un entier positif pour arriver jusqu'à un entier négatif.

Code

```
ma_liste[1:-4]
```

Résultat

```
[2, 3, 4, 5, 6]
```

4.1.3 Ajouter et supprimer des éléments à une liste

Comme nous l'avons vu, la liste est une structure modifiable, on peut donc ajouter, supprimer et modifier les valeurs de la liste.

Ajout

La méthode `append()` permet d'ajouter des éléments à une liste.

© Editions ENI - All rights reserved

Remarque

Une section de ce chapitre explique ce que sont les fonctions, méthodes, attributs, modules, packages, librairies, etc., et les différences entre eux. Reportez-vous-y pour plus de clarté. Sachez juste qu'une méthode comme `append()` *va effectuer un traitement lorsqu'on l'appelle ; ici,* `append()` *ajoute un élément à la liste.*

Syntaxe

```
ma_liste.append(valeur)
```

Ajoutons la valeur 2 à notre liste précédente.

Code

```
ma_liste.append(2)
print(ma_liste)
```

Résultat

```
[1, 2, 3, 4, 5, 6, 7, 8, 'hello', 10.5, 2]
```

La méthode `append()` ajoute l'élément en fin de liste. Si on souhaite ajouter un élément à un endroit précis de la liste, on utilise la méthode `insert()`, qui signifie insérer en français.

Ajoutons la chaîne de caractères `"nouvelle"` à la position 5, l'index `5` de `ma_liste`.

Code

```
ma_liste.insert(5,"nouvelle")
print(ma_liste)
```

Résultat

```
[1, 2, 3, 4, 5, 'nouvelle', 6, 7, 8, 'hello', 10.5, 2]
```

On voit bien que la chaîne de caractères a été insérée à la position 5, en comptant à partir de 0, toujours.

Modification

Pour modifier un élément, on peut se servir de l'indexage simple, avec un index entre crochets, ou du slicing. Puis on attribue une (ou des) nouvelle(s) valeur(s) à la position (ou aux positions) donnée(s).

Syntaxe

```
ma_liste[index]=nouvelle_valeur
ma_liste[start:stop:step]=[liste de nouvelles valeurs]
```

Remplaçons la première valeur de `ma_liste` par la chaîne de caractères `a`.

Code

```
ma_liste[0]="a"
print(ma_liste)
```

Résultat

```
['a', 2, 3, 4, 5, 'nouvelle', 6, 7, 8, 'hello', 10.5, 2]
```

Lorsque l'on souhaite modifier plusieurs éléments en utilisant le slicing, il faut donner autant de valeurs que le nombre de valeurs qu'on veut remplacer, sinon Python générera une erreur.

Code

```
ma_liste[3:5]=[8,10]
print(ma_liste)
```

Résultat

```
['a', 2, 3, 8, 10, 'nouvelle', 6, 7, 8, 'hello', 10.5, 2]
```

Ici, on demande de remplacer les valeurs aux index `3` et `4`, Python attend donc deux valeurs, qu'on donne entre crochets, ce qui correspond à une liste, donc. Et, dans l'ordre, Python met la valeur `8` à l'index `3` et la valeur `10` à l'index `4`.

© Editions ENI - All rights reserved

Suppression

Pour supprimer des valeurs avec le slicing, c'est le même principe que la modification : on donne les index auxquels on veut supprimer les valeurs et on donne une liste vide.

Syntaxe

```
ma_liste[start:stop:step] = []
```

Imaginons que l'on veuille supprimer les valeurs aux index `0` et `1`.

Code

```
ma_liste[0:2] = []
print(ma_liste)
```

Résultat

```
[3, 8, 10, 'nouvelle', 6, 7, 8, 'hello', 10.5, 2]
```

Les deux premiers éléments de la liste ont bien été supprimés.

Quand on veut supprimer un élément unique, on ne peut pas donner un index puis une liste vide, sinon la valeur à cet index est remplacée par une liste. En effet, les listes peuvent contenir des listes, et Python considère qu'on veut modifier la valeur par une liste vide.

Code

```
ma_liste[3]=[]
print(ma_liste)
```

Résultats

```
[3, 8, 10, [], 6, 7, 8, 'hello', 10.5, 2]
```

Pour réussir à supprimer un seul élément avec l'indexage simple (un seul index entre crochets), on peut utiliser le mot-clé `del`.

Remarque

En Python, un mot-clé est un nom réservé. Ce mot ne peut pas être utilisé comme nom de variable par exemple, car ce mot est dédié à sa fonction particulière définie dans Python. Ces mots-clés permettent de faire des traitements spéciaux. Par exemple, `del` prend en entrée un argument et le supprime.

Syntaxe

```
del ma_liste[mon_index]
```

Supprimons la valeur à l'index 3 de `ma_liste`.

Code

```
del ma_liste[3]
print(ma_liste)
```

Résultat

```
[3, 8, 10, 6, 7, 8, 'hello', 10.5, 2]
```

La liste vide précédemment ajoutée a bien été supprimée de `ma_liste`.

4.2 Les tuples

Nous allons voir succinctement ce qu'est un tuple. Nous ne les utiliserons pas tels quels, mais il est possible de les rencontrer au cours de ce livre.

Un tuple n'est rien d'autre qu'une liste, à la seule différence que le tuple est une liste qu'on ne peut pas modifier. Pour créer un tuple, cela se fait de la même manière que pour une liste, sauf que les crochets sont remplacés par des parenthèses.

Syntaxe

```
mon_tuple=(ma_premiere_valeur,ma_deuxieme_valeur,
ma_troisieme_valeur,...)
```

© Editions ENI - All rights reserved

Créons un tuple contenant cinq valeurs.

Code

```
mon_tuple=(1,2,3,4,5)
print(type(mon_tuple))
print(mon_tuple)
```

Résultat

```
<class 'tuple'>
(1, 2, 3, 4, 5)
```

Pour accéder à un élément d'un tuple, on utilise l'indexage simple, comme pour les listes.

Syntaxe

```
mon_tuple[index]
```

Code pour afficher le premier élément d'un tuple

```
mon_tuple[0]
```

Résultat

```
1
```

Mais quelle est l'utilité des tuples par rapport aux listes ? Il y en a plusieurs ; les deux plus importantes, de l'avis de l'auteur, sont l'assignation multiple et la protection en écriture.

Il est, en effet, possible de faire de l'assignation multiple grâce aux tuples.

Imaginons, par exemple, que l'on veuille assigner des valeurs à plusieurs variables, on utilisera alors les tuples.

Syntaxe de l'assignation multiple

```
(a,b) = (1,2)
```

La syntaxe peut également être simplifiée comme suit :

```
a, b = 1, 2
```

On peut donc créer deux variables sur la même ligne grâce aux tuples.

Code

```
ma_variable1, ma_variable2 = 1, 2
print(ma_variable1)
print(ma_variable2)
```

Résultat

```
1
2
```

On a bien créé deux variables auxquelles on peut accéder indépendamment.

Par ailleurs, comme nous le disions en introduction des tuples, ceux-ci ne sont pas modifiables, contrairement aux listes. On ne peut pas ajouter, supprimer ou modifier des éléments. Testons-le sous Python et essayons de remplacer la première valeur du tuple par la valeur `10`.

Code

```
mon_tuple[0]=10
print(mon_tuple)
```

Résultat

```
TypeError: 'tuple' object does not support item assignment
```

Lorsque nous souhaitons que des données restent inchangées dans notre programme, nous les assignons à un tuple, ce qui garantit leur protection et leur non-modification.

4.3 Les dictionnaires

4.3.1 Introduction aux dictionnaires

Un dictionnaire est une structure qui n'est pas ordonnée (contrairement aux listes), qui peut être modifiée et qui est indexée, tout comme les listes. Il est aussi possible de mélanger les types de valeurs dans un dictionnaire (caractères, entiers...).

© Editions ENI - All rights reserved

Le principe des dictionnaires est de lier une clé à une valeur et de pouvoir accéder aux valeurs grâce à ces clés. Ainsi, plutôt que de travailler avec des numéros d'index pour accéder à nos valeurs, nous pouvons travailler avec des clés, des mots. La syntaxe de création d'un dictionnaire est la suivante :

```
mon_dictionnaire = {
"clé1" : valeur1,
"clé2" : valeur2,
"clé3" : valeur3
}
```

Pour créer une liste, on utilisait les crochets. Pour créer un dictionnaire, on utilise les accolades { }, puis on donne un ensemble de couples clé-valeur, séparés par une virgule.

Imaginons qu'on veuille créer un dictionnaire avec des notes d'élèves, le code serait le suivant.

Code

```
notes_eleves = {
  "Marie": 15,
  "Thomas": 12,
  "Julien": "absent",
  "Elise": 9,
  "Samuel": 17
}
print(notes_eleves)
```

Résultat

```
{'Marie': 15, 'Thomas': 12, 'Julien': 'absent', 'Elise': 9,
'Samuel': 17}
```

Ici, on assigne à la variable `notes_eleves` les notes de cinq élèves. Puis on crée le dictionnaire : on écrit `"Marie"` entre guillemets, c'est la clé, généralement entre guillemets, puis le caractère deux-points `:` et on lui assigne la note `15`, et ainsi de suite pour chaque élève. Quand on affiche le dictionnaire, on voit qu'une note est assignée à chaque élève.

Comme pour les listes, on peut mélanger les types de données, mettre des nombres et des caractères : par exemple, Julien qui était absent. Les valeurs sont à présent retrouvables grâce aux clés. Pour afficher une valeur grâce à sa clé, la syntaxe est la suivante.

Syntaxe

```
mon_dictionnaire["clé1"]
```

On va donner le nom de la variable contenant le dictionnaire, puis appeler la clé correspondant à la valeur à laquelle on souhaite accéder, en la mettant entre crochets. Plus puissant que les listes, le dictionnaire permet d'accéder aux valeurs en donnant la clé, plutôt que l'index. Tout comme pour les listes, la clé est entourée par des crochets. C'est de cette manière que Python comprend qu'on lui donne des index ou des clés et qu'il doit afficher les valeurs correspondantes.

Affichons la note de Samuel, via notre dictionnaire `notes_eleves`.

Code

```
notes_eleves["Samuel"]
```

Résultat

```
17
```

Ici, on sait que Samuel a eu 17.

4.3.2 Ajouter, modifier et supprimer des éléments d'un dictionnaire

Ajout

Pour ajouter un nouveau couple clé-valeur dans un dictionnaire existant, la syntaxe est la suivante.

Syntaxe

```
mon_dictionnaire["clé4"]=valeur4
```

On définit le nom de la clé entre crochets, puis on assigne la valeur liée à cette clé grâce au symbole d'assignation =.

© Editions ENI - All rights reserved

Imaginons que l'on ait oublié de rentrer la note de Julie, qui a eu 9. Pour ajouter cette note à notre dictionnaire `notes_eleves`, le code est le suivant.

Code

```
notes_eleves["Julie"]=9
print(notes_eleves)
```

Résultat

```
{'Marie': 15, 'Thomas': 12, 'Julien': 'absent', 'Elise': 9,
'Samuel': 17, 'Julie': 9}
```

Julie apparaît maintenant à la fin du dictionnaire.

Modification

Pour modifier une valeur, il faut dire à Python pour quelle clé il doit modifier la valeur associée.

Syntaxe

```
mon_dictionnaire["clé2"]=valeur2_bis
```

Par exemple, imaginons que l'on veuille modifier la note de Julien, qui a rattrapé son contrôle et qui a donc obtenu une note.

Code

```
notes_eleves["Julien"]=13
print(notes_eleves)
```

Résultat

```
{'Marie': 15, 'Thomas': 12, 'Julien': 13, 'Elise': 9,
'Samuel': 17, 'Julie': 9}
```

La nouvelle valeur associée à la clé `'Julien'` n'est plus `'Absent'` mais bien `13`.

Remarque

Attention : un dictionnaire ne peut pas contenir deux fois la même clé, sinon la valeur pour cette clé est remplacée par la dernière valeur assignée à cette clé. Exemple : un nouvel élève arrive dans la classe avec le prénom Julien. Le dictionnaire ne permettra pas de stocker les deux notes des deux Julien. Il faudra alors ajouter les noms de famille aux clés pour pouvoir le faire et avoir ainsi des clés uniques.

Suppression

Si on souhaite supprimer un élément d'un dictionnaire, on pourra encore une fois utiliser le mot-clé `del`.

Syntaxe

```
del mon_dictionnaire["clé1"]
```

Ici, on utilise le mot-clé `del` et on donne le nom de la clé qu'on souhaite supprimer du dictionnaire. La valeur associée sera supprimée elle aussi.

Imaginons que Julie change de classe, on peut la supprimer du dictionnaire `notes_eleves` grâce au mot-clé `del`.

Code

```
del notes_eleves["Julie"]
print(notes_eleves)
```

Résultat

```
{'Marie': 15, 'Thomas': 12, 'Julien': 13, 'Elise': 9, 'Samuel': 17}
```

4.3.3 Parcourir un dictionnaire

Parcourir un dictionnaire, c'est-à-dire itérer sur chaque élément de celui-ci, est possible soit par clés-valeurs, soit juste par les clés, soit juste par les valeurs. Pour cela, on utilisera une boucle `for`. Pour rappel, la boucle `for` permet d'itérer sur des éléments, comme une liste, un dictionnaire et tant d'autres possibilités.

© Editions ENI - All rights reserved

Pour les dictionnaires, il existe une méthode, `keys()`, qui retourne la liste de toutes les clés d'un dictionnaire. Il suffit ensuite d'y appliquer une boucle `for` pour parcourir cette liste.

Syntaxe

```
for cle in mon_dictionnaire.keys():
    print(cle)
```

De même, il existe une méthode, `values()`, qui retourne la liste de toutes les valeurs d'un dictionnaire. Il suffit ensuite d'y appliquer une boucle `for` pour parcourir cette liste de valeurs.

Syntaxe

```
for valeur in mon_dictionnaire.values():
    print(valeur)
```

Et enfin, si on veut parcourir simultanément les clés et les valeurs, on utilise la méthode `items()`, qui va renvoyer une liste contenant des tuples de couples clé-valeur.

```
for cle,valeur in mon_dictionnaire.items():
    print(cle,valeur)
```

Voici un exemple avec le dictionnaire `notes_eleves`.

Affichons les résultats de la méthode `items()`.

Code

```
notes_eleves.items()
```

Résultat

```
dict_items([('Marie', 15), ('Thomas', 12), ('Julien', 13),
('Elise', 9), ('Samuel', 17)])
```

On voit que c'est une liste (définie avec des crochets) qui contient des tuples. Comme chaque élément de la liste contient un tuple qui lui-même contient un couple clé-valeur, on va pouvoir donner deux noms de variables à la boucle `for`. La première variable prendra la valeur de la première valeur du tuple, la clé. La deuxième variable prendra comme valeur la valeur associée à la clé.

Parcourons le dictionnaire avec une boucle `for` et deux variables, `cle` et `valeur`.

Code

```
for cle,valeur in notes_eleves.items():
    print(cle,valeur)
```

Résultat

```
Marie 15
Thomas 12
Julien 13
Elise 9
Samuel 17
```

Vous savez maintenant comment parcourir un dictionnaire. Cela permet aussi de visualiser l'imbrication possible des différentes structures de données, dictionnaires, listes et tuples.

5. Les opérateurs arithmétiques, relationnels et logiques

5.1 Les opérateurs arithmétiques

Python peut se comporter comme une calculatrice grâce aux opérateurs arithmétiques que l'on connaît bien : l'addition, la soustraction, la multiplication, la division.

Syntaxe de l'addition :

```
5 + 3
```

Syntaxe de la soustraction :

```
5 - 3
```

Syntaxe de la multiplication :

```
5 * 3
```

© Editions ENI - All rights reserved

Syntaxe de la division :

```
5 / 3
```

Syntaxe pour récupérer le résultat entier d'une division :

```
5 // 3
```

Syntaxe pour connaître le reste d'une division :

```
5 % 3
```

Écrivons ensemble un code exemple pour visualiser les résultats de ces opérations. Nous allons ainsi en profiter pour introduire les commentaires en Python : il est possible d'écrire du texte dans le code Python qui ne sera pas interprété. Cela permet, lorsque vous revenez sur votre code, de vous rappeler plus rapidement ce que celui-ci fait. Commenter son code fait partie des bonnes pratiques en programmation. Pour commenter du texte, il faut utiliser le caractère #.

Code

```
# Une addition
print("Résultat de l'addition 5+3 :")
print(5+3)
# Une soustraction
print("Résultat de la soustraction 5-3 :")
print(5-3)
# Une division
print("Résultat de la division 10/3 :")
print(10/3)
print("Le reste de la division 10/3 :")
print(10 % 3)
```

Résultat

```
Résultat de l'addition 5+3 :
8
Résultat de la soustraction 5-3 :
2
Résultat de la division 10/3 :
3.3333333333333335
Le reste de la division 10/3 :
1
```

Le texte précédé d'un # n'est pas affiché, ce qui était attendu.

5.2 Les opérateurs relationnels et logiques

5.2.1 Les opérateurs relationnels

Les opérateurs relationnels sont aussi appelés opérateurs de comparaison, car ils permettent de comparer des expressions. Ils sont fréquemment utilisés en Data Science lorsqu'on explore des données.

Voici les opérateurs relationnels.

Il y a l'opérateur ==, qui permet de comparer deux valeurs, afin de tester si elles sont égales ou non. Il y a aussi l'opérateur !=, qui est le contraire de == et permet de tester si deux valeurs sont différentes.

Remarque

Attention : ne confondez pas l'opérateur d'assignation = avec l'opérateur de comparaison ==.

Ces comparaisons renvoient un type de données que nous avons vu précédemment ensemble : le booléen. Si la comparaison est vraie, la valeur `True` est renvoyée, si elle est fausse, la valeur `False` est renvoyée.

Code

```
print(1==1)
print(1!=1)
print(5>3)
print(10>=10)
print(3>25)
```

Résultat

```
True
False
True
True
False
```

© Editions ENI - All rights reserved

Chaque expression nous renvoie bien une valeur `True` ou `False` ici. 1 est bien égal à 1, donc la valeur retournée et `True`. 1 n'est pas différent de 1, donc la valeur retournée est `False`, etc.

Ces opérateurs sont souvent utilisés dans les structures conditionnelles, `if ... else`, qui permettront d'exécuter des blocs de code si et seulement si les conditions énoncées, avec les opérateurs relationnels, sont respectées.

5.2.2 Les opérateurs logiques

Les opérateurs logiques permettent de comparer des expressions booléennes, et non plus des valeurs en tant que telles. Globalement, les opérateurs logiques sont associés aux opérateurs relationnels pour écrire des expressions de comparaisons plus complexes.

Syntaxe générale

```
(valeur == valeur2) and (valeur3 != valeur4)
```

Il y a deux opérateurs logiques, `and` (qui signifie et) et `or` (qui signifie ou). Dans l'expression écrite ci-dessus, la valeur `True` sera renvoyée si et seulement si `valeur` est égale à `valeur2` **ET** si `valeur3` est différente de `valeur4`.

Syntaxe générale avec `or`

```
(valeur == valeur2) or (valeur3 != valeur4)
```

Ici, si une des comparaisons seulement est vraie, alors cette expression renvoie `True`. Avec le `or`, il n'est pas nécessaire que les deux comparaisons soient vraies, contrairement au `and`.

Code

```
print((3 ==3) and (4 == 4))
print((3 ==3) and (4 == 5))
print((3 ==3) or (4 == 5))
```

Résultat

```
True
False
True
```

Cet exemple permet de voir la différence de résultat entre deux expressions comparées avec `and` ou avec `or`.

Remarque

Si vous souhaitez le contraire de l'expression que vous énoncez, par exemple, si vous souhaitez que lorsque le résultat de votre expression booléenne est `False`, Python vous renvoie `True`, alors vous utiliserez le dernier opérateur logique, qui est `not`, pour "non" ou "pas" en français. Exemple : `not((3 ==4) or (4 == 5))` renvoie `True`.

Remarque

Ici nous utilisons les opérateurs logiques car chaque expression booléenne génère une unique valeur `True` ou `False`. Par exemple, `3==3` génère `True` et `4==4` génère `True` aussi, donc les opérateurs logiques fonctionnent. Nous verrons dans le chapitre sur la librairie Pandas, notamment, que lorsque les expressions booléennes renvoient des objets contenant plusieurs booléens, nous utilisons les opérateurs `bitwise` (comparaison bit à bit), plutôt que les opérateurs logiques.

6. Vocabulaire en Python : fonctions, méthodes, attributs, modules et librairies (packages)

6.1 Fonctions

Dans ce chapitre, nous avons vu quelques fonctions, comme `print()` ou `type()`. Une fonction est une portion de code à l'intérieur d'un fichier qui permet d'effectuer des tâches bien précises et ce de manière répétée.

Une fonction attend généralement quelque chose en entrée, cela peut être un nombre, une chaîne de caractères, ou une structure de données plus complexe comme une liste, un dictionnaire, etc. Une fois que cette fonction est appelée par Python, le code qu'elle contient est exécuté sur ce que Python lui donne en entrée et, en sortie, on a généralement un résultat différent de ce qui a été donné en entrée. Le nom de la fonction définit généralement ce qu'elle fait.

© Editions ENI - All rights reserved

Python possède un nombre conséquent de fonctions "built-in", c'est-à-dire de fonctions intégrées au langage Python, qui sont présentes dès l'installation du langage. Par exemple, la fonction `print()`, qui veut dire afficher en français, est disponible automatiquement avec Python et permet d'afficher ce qu'on lui donne en entrée. Une fonction a donc un nom, puis elle est suivie de parenthèses, à l'intérieur desquelles on lui donne les arguments, les objets, que l'on veut qu'elle prenne en entrée pour pouvoir faire son traitement.

Syntaxe

```
ma_fonction(objet1, objet2, ...)
```

Par exemple, nous assignons une valeur à une variable `a=12`. Nous tapons `print(a)` sous Python et `print()` va nous afficher le contenu de notre variable, la valeur `12`.

Une fonction peut prendre plusieurs arguments, plusieurs options. Il y a les arguments obligatoires, nécessaires à son bon fonctionnement, et les arguments optionnels, si on souhaite que la fonction se comporte différemment de son comportement par défaut.

Pour savoir ce qu'une fonction prend comme arguments, vous pouvez utiliser la fonction `help()`, une autre fonction "built-in" de Python. Il suffit de lui donner le nom de la fonction et elle affiche l'aide de la fonction donnée.

Syntaxe

```
help(ma_fonction)
```

Affichons l'aide de la fonction `print()`.

Code

```
help(print)
```

Résultat

```
Help on built-in function print in module builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current
```

```
sys.stdout.
    Sep:   string inserted between values, default a space.
    End:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

On voit divers arguments, dont `end`, qui permet de spécifier le caractère à la fin du texte affiché par `print()` : c'est par défaut un `\n`, qui correspond à un retour à la ligne. Pour l'exemple, mettons le caractère `b` comme fin de ligne.

Code

```
print(12, end="b")
```

Résultat

```
12b
```

Les différents arguments sont séparés par des virgules, ce qui fait qu'une fonction peut être très finement paramétrée grâce à différents arguments.

Les fonctions sous Python nous facilitent la vie car nous n'avons pas besoin de les coder nous-mêmes, mais juste de les utiliser. Il existe énormément de fonctions spécifiques à la Data Science et vous verrez la puissance de leur utilisation au cours de ce livre.

6.2 Méthodes

Une méthode est une fonction qui appartient à une classe. Une fonction n'est pas liée à une classe d'objet (liste, tuple, dictionnaire...), alors qu'une méthode, si. Par exemple, un objet de type `list` en Python possède ses méthodes, de même que les dictionnaires possèdent leurs méthodes. Ces méthodes permettent de manipuler facilement les objets de la classe à laquelle elles appartiennent. Par exemple, pour les listes, il existe la méthode `insert()`, qui permet d'insérer un élément dans la liste, la méthode `count()`, qui permet de compter le nombre d'éléments dans une liste, etc.

Syntaxe générale d'utilisation d'une méthode

```
ma_liste.ma_methode()
```

© Editions ENI - All rights reserved

Contrairement à la fonction, où on donne l'objet entre parenthèses comme argument, ici, on précède le nom de la méthode par le nom de l'objet sur lequel on souhaite effectuer le traitement. Et si on souhaite ajouter des options, celles-ci seront données entre parenthèses.

6.3 Attributs

Les attributs sont des variables associées à une classe d'objets. Tout comme pour les méthodes, les classes d'objets possèdent un ensemble d'attributs spécifiques. Les attributs permettent d'accéder à des informations concernant un objet : une liste, un tuple... Par exemple, les listes ont un ensemble d'attributs.

Pour utiliser un attribut, la syntaxe est la suivante :

```
ma_liste.mon_attribut
```

Contrairement à la méthode, qui est suivie de parenthèses permettant de donner des arguments supplémentaires, l'attribut, lui, ne l'est pas. Il s'agit juste d'un nom de variable qui permet de récupérer des informations sur un objet. Par exemple, la classe `ndarray` (les tableaux NumPy, que nous verrons plus tard) possède un ensemble d'attributs très pratiques permettant d'avoir des informations sur le tableau. Par exemple, `mon_ndarray.size` permet d'afficher le nombre d'éléments dans le tableau, grâce à l'attribut `size`.

6.4 Modules

Dans la section sur les fonctions, nous avons évoqué des fonctions built-in, disponibles de base lors de l'installation de Python. Ces fonctions sont fournies automatiquement car ce sont des fonctions essentielles et très utilisées, comme la fonction `print()`, pour ne citer qu'elle. Les fonctions moins courantes sont regroupées dans des modules qu'il faut importer pour pouvoir utiliser ces fonctions.

Un module est un ensemble de fichiers .py (extension de Python) contenant des fonctions, des classes ou encore des variables pouvant être appelées dans votre code. Ainsi, lorsque vous développez votre programme, vous pouvez importer des modules afin de pouvoir utiliser les fonctions qu'ils contiennent.

De même, si votre programme devient conséquent, il est recommandé de ranger vos fonctions dans des fichiers `.py` séparés, que vous appellerez dans votre programme principal. Ces fichiers `.py` dans lesquels vous rangez vos fonctions sont des modules Python. Un module a un nom, et ce nom n'est autre que le texte qui se trouve avant l'extension `.py`. Ainsi, si on veut importer dans son programme le code contenu dans le fichier `mon_module.py`, on importera le module avec le nom `mon_module`.

Pour accéder à ces modules, il faut utiliser le mot-clé `import`.

Syntaxe

```
import mon_module
```

Généralement, les importations de modules se font en début de programme, et une fois qu'ils sont importés, l'ensemble des fonctions sont disponibles et utilisables dans votre programme. Comme pour les fonctions, il existe un ensemble de modules "built-in", disponibles de base avec Python. Les plus populaires sont les modules `math`, `random`, etc.

Ces modules contiennent donc des fonctions. Une fois les modules importés, si vous souhaitez utiliser une fonction de ce module, il suffit d'utiliser la syntaxe suivante :

```
import mon_module
mon_module.ma_fonction()
```

Si le nom du module est trop long, il est possible de créer un alias de ce module pour raccourcir son nom.

Syntaxe

```
import mon_module as m
m.ma_fonction()
```

© Editions ENI - All rights reserved

Enfin, si on souhaite ne plus utiliser le nom du module avant le nom de la fonction, il faut utiliser la syntaxe suivante :

```
from mon_module import ma_fonction
ma_fonction()
```

De cette manière, on définit `ma_fonction` dans l'espace de noms de notre programme et à partir de ce moment-là, il n'est plus nécessaire de mettre le nom du module avant le nom de la fonction.

6.5 Librairies (packages)

Une librairie (ou package) peut être vue comme un dossier contenant un ensemble de modules, un ensemble de fichiers `.py`. Une librairie a une application particulière et regroupe donc l'ensemble des modules permettant de répondre à cette application. Par exemple, on peut créer une librairie pour faire du calcul scientifique, des graphiques, de la manipulation de données, etc. Ces librairies contiendront un ensemble de modules, et donc de fonctions, méthodes, classes, etc. permettant d'effectuer ces tâches précises.

Ces librairies sont créées et mises à disposition pour faciliter la programmation en Python, mais aussi pour nous éviter d'avoir à réécrire du code utilisé par un grand nombre de personnes. Au début de ce livre, nous avons installé la distribution Anaconda, qui permet d'installer Python et un ensemble de librairies dédiées à la Data Science, comme Pandas pour la manipulation de tableaux (dataframes) ou encore Matplotlib pour la visualisation de données. Vous le verrez tout au long de ce livre, ces librairies vont nous faciliter grandement la vie. Elles nous permettront d'utiliser des fonctions qui font des choses très complexes, en leur donnant juste nos objets en entrée et en récupérant le résultat à la fin, sans rien avoir à faire de plus !

Remarque

Vous pourrez trouver différentes définitions de librairies et packages, l'auteur les utilise de la même manière avec la même signification, car, conceptuellement, ils correspondent à la même chose : un ensemble de modules/fonctionnalités à utiliser pour faciliter la vie de l'analyste.

7. Instructions de condition if et boucles for

7.1 Instruction de condition if

Pour terminer avec le rappel sur le langage de programmation Python, parlons des instructions de condition `if` et surtout, de l'utilité des opérateurs relationnels et logiques vus précédemment.

Un des piliers de la programmation informatique, ce sont les instructions de condition `if`. Celles-ci vous permettent de faire des comparaisons et de réaliser différentes actions selon le résultat de votre comparaison, c'est-à-dire selon que le résultat est `True` ou `False`.

La syntaxe est la suivante :

```
if condition:
    action1
else:
    action2
```

Si la condition est vraie, c'est-à-dire `True`, le code de `l'action1` s'exécute, si la condition est fausse, `False`, c'est le code de `l'action2`.

Essayons avec un exemple concret. Créons d'abord trois variables.

Code

```
ma_variable1=2
ma_variable2=6
ma_variable3="hello"
```

Nous allons maintenant faire des comparaisons entre ces variables et exécuter du code différent selon le résultat de ces comparaisons.

Dans la plupart des langages de programmation, il existe des symboles particuliers pour délimiter le `if` (si, en français) et le `else` (sinon, en français), par exemple l'accolade. En Python, il n'y a pas de délimiteur, mais il y a un `:` en début d'instruction puis une indentation particulière, qu'il faut impérativement respecter.

© Editions ENI - All rights reserved

Code

```
if (ma_variable1 == 2) and (ma_variable3 == "hello"):
    print("Ta comparaison est vraie !")
else:
    print("Ta comparaison est fausse !")
```

Résultat

```
Ta comparaison est vraie !
```

Ici, le texte `Ta comparaison est vraie !` s'affiche car, effectivement, `ma_variable1=2` et `ma_variable3="hello"`, donc le résultat de la comparaison est `True`. Ainsi, on entre dans le premier bloc de code et on affiche `Ta comparaison est vraie !`.

Remarque

Vous constatez une indentation lorsqu'on entre dans les blocs de code. Cette indentation est importante et doit être la même dans l'ensemble de votre programme. Comme bonne pratique, il est conseillé d'utiliser une indentation de quatre espaces. Mais en réalité, le nombre importe peu, tant qu'il est le même dans tout votre programme.

Essayons un exemple avec l'opérateur logique `or`.

Code

```
if (ma_variable1 > ma_variable2) or (ma_variable3 == "hello"):
    print("Ta comparaison est vraie !")
else:
    print("Ta comparaison est fausse !")
```

Résultat

```
Ta comparaison est vraie !
```

Ici, on exécute le premier bloc de code car une des comparaisons est vraie. Même si `ma_variable1` n'est pas supérieure à `ma_variable2`, `ma_variable3` est bien égale à `0`.

7.2 Boucle for

Pour finir, nous allons voir la boucle `for`, qui veut dire "pour" en français. Nous l'avons vue très rapidement lors du parcours du dictionnaire, mais voyons-la à présent plus en détail.

La boucle `for` permet d'itérer, on parle d'itérations. Elle itère sur un ensemble de valeurs, c'est-à-dire qu'elle les parcourt et effectue des actions sur ces valeurs.

L'utilité de la boucle `for` est qu'elle permet de parcourir les valeurs d'une liste, ou encore d'un dictionnaire, et d'effectuer des traitements sur l'ensemble de ces valeurs. On ne peut pas faire de comparaison des valeurs au sein d'une liste si on n'est pas capable de parcourir ses valeurs, et c'est ce que permet de faire la boucle `for`.

Syntaxe

```
for valeur in liste:
    actions
```

Créons une liste pour comprendre comment fonctionnent les boucles `for`.

Code

```
ma_liste = [1,2,3,4,5,6,7,8,9,10]
print(ma_liste)
```

Résultat

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Parcourons cette liste avec une boucle `for` et multiplions les valeurs par 2.

Code

```
for ma_valeur in ma_liste:
    print(ma_valeur*2)
```

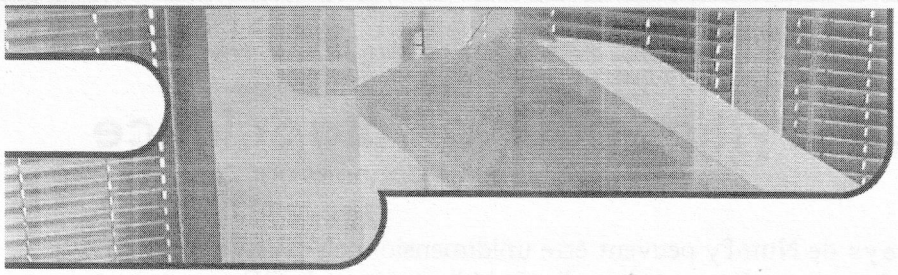
© Editions ENI - All rights reserved

Résultat

```
2
4
6
8
10
12
14
16
18
20
```

Lors d'un premier tour, Python affecte le chiffre `1` à la variable `ma_valeur` et exécute le code sur cette valeur. Puis, au deuxième tour, la variable `ma_valeur` vaut `2`, puis `3`, puis `4`, jusqu'à arriver à la dernière valeur de la liste, ce qui arrête la boucle `for`. En effet, une fois que la boucle `for` a parcouru toute la liste, elle s'arrête.

Le mot `ma_valeur` est un nom de variable. Vous pouvez la nommer comme vous le souhaitez : `ma_valeur`, `valeur`, `hello`, etc. Ensuite, c'est ce nom de variable là que vous utilisez dans votre bloc de code.



Chapitre 4
Maîtriser la librairie NumPy

1. Introduction à NumPy

NumPy est la librairie Python dédiée au calcul scientifique fournissant des fonctions très performantes de calcul, mais aussi des structures de données, tout aussi performantes.

En Data Science, il est essentiel d'avoir des structures adaptées pour stocker et manipuler de grandes quantités de données. C'est là qu'intervient NumPy, qui intègre une nouvelle structure de données en Python, les `ndarrays` (tableaux à N dimensions, en français, N représentant un chiffre), qui sont des tableaux multidimensionnels ou matrices. Il est important de noter que cette structure de données permet de stocker des données uniquement de même type (uniquement des nombres entiers par exemple).

Les `ndarrays` sont optimisés pour le stockage de données, mais aussi pour leur manipulation et plus encore pour les calculs, car ceux-ci peuvent être vectorisés. NumPy peut gérer de très gros tableaux et est très performante en temps de calcul sur ces tableaux : les `ndarrays` prennent en effet moins de place mémoire que d'autres objets Python, comme par exemple les listes. C'est pour cela que cette librairie a été développée et qu'elle est si utilisée : elle est très performante. C'est également pour cette raison que de nombreuses librairies ont été développées au-dessus de celle-ci, telle que Pandas, que nous verrons plus tard dans ce livre.

Les `ndarrays` de NumPy peuvent être unidimensionnels (aussi appelés 1D array, ce qu'on peut voir comme une liste), bidimensionnels (2D array) donc un tableau avec des lignes et des colonnes, ou encore des tableaux à plus de deux dimensions (3D array, 4D array, 5D array...), que nous ne verrons pas dans ce livre. Nous travaillerons exclusivement avec les `ndarrays` à deux dimensions dans ce chapitre, qui est la structure de données la plus utilisées en Data Science pour manipuler de grands jeux de données.

Lorsque vous souhaitez effectuer des opérations mathématiques ou logiques rapides sur un grand jeu de données, NumPy est votre allié. Pour pouvoir utiliser NumPy sous Python, il suffit de charger la librairie sous Jupyter, celle-ci étant déjà installée dans la distribution Anaconda.

Syntaxe

```
import numpy as np
```

Remarque

Il est courant d'utiliser l'alias "np" pour la librairie NumPy.

Remarque

Pour la suite de ce chapitre, n'hésitez pas à tester les codes que nous proposerons directement dans le notebook lié à ce chapitre et disponible en téléchargement.

2. Les tableaux NumPy

2.1 Créer un ndarray

2.1.1 Créer un ndarray à partir de listes

On peut créer un tableau NumPy à partir d'une liste en utilisant la fonction `array()` afin de convertir cette liste en tableau.

Syntaxe

```
import numpy as np
notre_tableau=np.array([élément1,élément2,élément3,élément4])
```

© Editions ENI - All rights reserved

Exemple de code

```
import numpy as np
notre_tableau=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20])
print(notre_tableau)
type(notre_tableau)
```

Résultat

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
numpy.ndarray
```

Ici, on importe la librairie NumPy, on crée un tableau à partir d'une liste, on affiche ce tableau avec `print()` et on affiche le type de l'objet `notre_tableau`. En résultat, on a notre premier array NumPy, qui est un tableau à une dimension ici, qu'on pourrait considérer comme une liste, mais qui est bien de type `ndarray`.

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

Syntaxe

```
mon_array_bidimensionnel=np.array([[élément ligne 1 colonne 1,
élément ligne 1 colonne 2,élément ligne 1 colonne 3],
[élément ligne 2 colonne 1, élément ligne 2 colonne 2,
élément ligne 2 colonne 3]])
```

Code

```
import numpy as np
mon_array_bidimensionnel=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(mon_array_bidimensionnel)
```

Résultat

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

On remarque tout de suite que, visuellement, cela ressemble à un tableau à deux dimensions, dont la première ligne contient trois colonnes avec les chiffres 1,2,3, la deuxième ligne (la deuxième liste de la liste principale) également trois colonnes avec les chiffres 4, 5, 6, etc.

Il s'agit d'un tableau d'entiers ici, et on ne pourrait pas changer une valeur par un type caractère par exemple, cela générerait une erreur.

Code

```
mon_array_bidimensionnel[0,0]="texte"
```

Résultat

```
ValueError: invalid literal for int() with base 10: 'texte'
```

Ici, nous disons à Python que nous souhaitons modifier la valeur de la première ligne à la première colonne par la chaîne de caractères "texte". Python nous retourne une erreur en nous disant que "texte" n'est pas de type int (entier), donc ça ne fonctionne pas. Comme nous le disions, un ndarray ne peut contenir qu'un seul type de données. Pour connaître le type des données contenues dans un ndarray, il suffit d'utiliser l'attribut dtype.

Syntaxe

```
mon_array.dtype
```

Regardons le type de données de notre tableau d'exemple.

Exemple de code

```
mon_array_bidimensionnel.dtype
```

Résultat

```
dtype('int32')
```

Ici, le type est int32, notre tableau ne peut donc contenir que des nombres entiers, d'où l'erreur précédente.

© Editions ENI - All rights reserved

Remarque

Il existe deux types `int` *:* `int32` *et* `int64`*. La différence entre* `int32` *et* `int64`*, c'est globalement la capacité de stockage du type.* `int32` *peut stocker des nombres entiers entre -2 147 483 648 et 2 147 483 647 et* `int64` *peut stocker des nombres entiers entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 808.*

Il est possible de spécifier le type des données du tableau qu'on souhaite créer, avec l'option `dtype` dans la fonction `array()` de NumPy.

Code

```
mon_array_bidimensionnel=np.array([[1.5,2,3],[4,5.2,6],[7,8,9.1]],
dtype = float)
print(mon_array_bidimensionnel)
mon_array_bidimensionnel.dtype
```

Ici, on crée un tableau en spécifiant le type, `float`, puis on affiche le contenu de ce tableau. Enfin, on affiche le type des données contenues dans ce tableau.

Résultat

```
[[1.5 2.  3. ]
 [4.  5.2 6. ]
 [7.  8.  9.1]]
Out[9]:
dtype('float64')
```

Il s'agit bien d'un tableau contenant des données de type `float`, donc des nombres à virgule. Tous les entiers qu'on a donnés lors de la création du tableau (par exemple 2, 3, 4...) sont transformés en réels, `float`, et sont donc suivis de ".0".

2.1.2 Créer un ndarray grâce à des fonctions NumPy

Il arrive que vous sachiez d'avance la taille de votre `ndarray` mais pas encore son contenu, car vous souhaitez remplir ce tableau au fur et à mesure de votre code. NumPy fournit des fonctions permettant de créer et initialiser des `ndarrays` de taille connue et de contenu inconnu.

La première fonction est `np.zeros()`. Cette fonction permet de créer un `ndarray` dont l'ensemble des éléments correspondent à la valeur zéro.

Syntaxe

```
np.zeros((nombre de lignes, nombre de colonnes))
```

Il suffit de donner le nombre de lignes et de colonnes que vous souhaitez voir apparaître dans votre tableau.

Code

```
import numpy as np
mon_array_zeros=np.zeros((4,6))
print(mon_array_zeros)
```

Résultat

```
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

On obtient un tableau de type `float`, par défaut, rempli de zéros et de dimensions quatre lignes et six colonnes. Si on veut, on peut spécifier que le tableau est de type `int`.

Code

```
import numpy as np
mon_array_zeros=np.zeros((4,6), dtype=int)
print(mon_array_zeros)
```

Résultat

```
[[0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]]
```

Remarque

Il est aussi possible de créer un `ndarray` à une dimension contenant uniquement des zéros. Pour cela, plutôt que de donner un tuple contenant le nombre de lignes et de colonnes, il suffit de donner un seul nombre à la fonction `np.zeros` : `np.zeros(5)` créera un tableau à une dimension contenant 5 valeurs 0.

© Editions ENI - All rights reserved

Il existe ensuite la fonction `np.ones()`. Cette fonction permet de créer un `ndarray` dont l'ensemble des éléments correspondent au chiffre 1.

Syntaxe

```
np.ones((nombre de lignes, nombre de colonnes))
```

Exemple de code

```
import numpy as np
mon_array_ones=np.ones((4,6), dtype=int)
print(mon_array_ones)
```

Résultat

```
[[1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]]
```

Enfin, il existe la fonction `np.emptys()`. Cette fonction permet de créer un `ndarray` dont les valeurs des éléments sont aléatoires.

Syntaxe

```
np.empty((nombre de lignes, nombre de colonnes))
```

Exemple de code

```
import numpy as np
mon_array_empty=np.empty((4,6), dtype=int)
print(mon_array_empty)
```

Résultat

```
[[-1805877096         611          64           0           0           0]
 [          0           0           0           0   929446194   912471142]
 [ 1667510839   909654373   909193776  1664104498  1630941233   929260599]
 [  959657314   878982705  1650615603   929457204  1697985846   842152292]]
```

2.1.3 Créer un ndarray à partir d'un fichier

Il est aussi possible de créer un `ndarray` en lisant des données dans un fichier txt ou encore csv. Pour cela, nous avons besoin d'un fichier d'exemple. Il existe de nombreuses sources fournissant des jeux de données pour s'entraîner avec les langages R et Python. Une ressource que nous utilisons est le site web : https://vincentarelbundock.github.io/Rdatasets/datasets.html qui répertorie un ensemble de jeux de données très variés, disponibles au format CSV.

Pour l'exemple, nous allons créer un tableau à partir d'un fichier de données CSV disponible via le lien suivant :
https://vincentarelbundock.github.io/Rdatasets/csv/datasets/iris.csv

Ce fichier est disponible en téléchargement avec ce livre, sous le nom iris.csv.

Nous avons modifié ce fichier afin de pouvoir l'importer dans un `ndarray`, qui ne doit contenir qu'un seul type de données car, à l'origine, ce fichier contient des nombres entiers et des chaînes de caractères. Ce fichier contient des informations sur la longueur et la largeur de pétales et de sépales de 150 iris, sous forme de nombres réels, `float`. Les longueurs et largeurs de pétales et sépales permettent de différencier trois espèces d'Iris, représentées dans la cinquième colonne par des réels plutôt que le nom de l'espèce, étant donné que les `ndarrays` n'acceptent pas des types de données différents. L'espèce 1.0 correspond à l'espèce Setosa, l'espèce 2.0 à l'espèce Versicolor et l'espèce 3.0 correspond à l'espèce Virginica. Ce jeu de données est très populaire en Data Science, car il fonctionne très bien sur de nombreux modèles statistiques. Ici, nous l'utiliserons pour apprendre à créer un `ndarray` à partir d'un fichier, mais aussi pour apprendre à explorer et filtrer un `ndarray` avec NumPy.

© Editions ENI - All rights reserved

Sur l'image ci-dessous, voici un aperçu du fichier ouvert dans Excel. Le but va être de le lire et de stocker les informations qu'il contient dans un `ndarray`.

	A	B	C	D	E
1	Longueur des sépales	Largeur des sépales	Longueur des pétales	Largeur des pétales	Espèce
2	5.1	3.5	1.4	0.2	1
3	4.9		3 1.4	0.2	1
4	4.7	3.2	1.3	0.2	1
5	4.6	3.1	1.5	0.2	1
6		5 3.6	1.4	0.2	1
7	5.4	3.9	1.7	0.4	1
8	4.6	3.4	1.4	0.3	1

Pour lire un fichier, il est possible d'utiliser la fonction `genfromtxt()` de NumPy. Il en existe d'autres, mais celle-ci est la fonction qui possède le plus d'options, qui permettent de l'adapter à des fichiers de nombreux formats différents.

Syntaxe

```
mon_array=np.genfromtxt("mon_fichier.txt", delimiter=";",
skip_header=True)
```

Dans cette commande, nous donnons le nom du fichier que nous souhaitons lire. Si le fichier est dans le dossier dans lequel vous travaillez, il suffit de donner son nom. Sinon, il faut donner le chemin vers ce fichier en plus de son nom.

L'option `delimiter` permet de spécifier à la fonction `genfromtxt()` quel est le séparateur de colonne ; ici, on spécifie que les colonnes sont séparées par un point-virgule ; car il s'agit d'un fichier au format `CSV`.

Enfin, l'option `skip_header` permet de spécifier à la fonction `genfromtxt()` que le fichier contient un en-tête et qu'on souhaite l'ignorer, en spécifiant la valeur `True`. Par exemple, dans l'image précédente, l'en-tête est "Longueur des sépales", "Largeur des sépales", etc. Ce sont les noms de colonnes, l'en-tête ou header en anglais. Si le fichier ne contient pas de header, vous n'êtes pas obligé de donner l'option `skip_header`. Ici, cette option permet d'ignorer la première ligne lors de la lecture du fichier. Nous verrons plus tard, avec la librairie Pandas et les dataframes, l'utilité des headers.

Une fois le fichier lu et stocké dans un `ndarray`, il est accessible via la variable `mon_array` dans cet exemple de syntaxe.

À présent, exécutons le code pour lire le fichier iris.csv :

```
import numpy as np
iris=np.genfromtxt("../datasets/iris.csv", delimiter=";",
skip_header=True)
print(iris)
```

Résultat

```
[[5.1 3.5 1.4 0.2 1. ]
 [4.9 3.  1.4 0.2 1. ]
 [4.7 3.2 1.3 0.2 1. ]
 ...
 [6.2 3.4 5.4 2.3 3. ]
 [5.9 3.  5.1 1.8 3. ]]
```

La structure du dossier de travail dans cet exemple est la suivante :

```
|- datasets
  |- iris.csv
|- notebooks
  |- Notebook_chapitre_4_Numpy.ipynb
```

Nous travaillons dans le notebook `Notebook_chapitre_4_Numpy.ipynb` (disponible en téléchargement) et celui-ci se trouve dans le dossier `notebooks` ; ainsi, lorsque nous travaillons avec ce notebook, nous nous trouvons dans le dossier `notebooks`. Nous rangeons notre fichier iris.csv (disponible en téléchargement aussi) dans un dossier nommé `datasets`. Pour pouvoir lire le fichier `iris.csv` dans notre notebook, il faut aller le chercher dans le dossier `datasets`, donc nous utilisons la syntaxe `../datasets` pour dire à Python de revenir dans le dossier au-dessus du dossier `notebooks` puis d'aller dans le dossier `datasets`.

Remarque

Si nous avions mis le fichier `iris.csv` dans le même dossier que le notebook `Notebook_chapitre_4_Numpy.ipynb`, nous aurions juste eu à donner le nom du fichier. Comme ce n'est pas le cas, nous devons spécifier à Python le chemin pour y accéder.

© Editions ENI - All rights reserved

En résultat de la fonction `genfromtxt()`, on obtient un objet de type `ndarray` contenant 5 colonnes et 150 lignes. On a bien l'ensemble de nos données d'iris stockées dans un `ndarray`.

Enfin, imaginons que vous souhaitiez enregistrer un `ndarray` dans un fichier sur votre ordinateur. Pour cela, vous pourrez utiliser la fonction `savetxt()` de NumPy.

Syntaxe

```
np.savetxt("monarray.txt", mon_array, delimiter=";")
```

Ici, il faut donner le nom du fichier dans lequel vous souhaitez écrire votre `ndarray`, le nom de la variable contenant ce `ndarray` et le séparateur de colonne qui doit être utilisé.

Exemple de code pour enregistrer le tableau `iris` sur son ordinateur :

```
import numpy as np
np.savetxt("iris_2.csv", iris, delimiter=";")
```

Si vous ouvrez votre dossier de travail, vous y verrez un nouveau fichier, nommé `iris_2.csv` et contenant les données du tableau `iris`.

2.2 Indexation

2.2.1 Indexation simple

Il est possible d'accéder aux éléments d'un tableau grâce à l'indexation, mise en place avec des symboles crochets `[]` entre lesquels on donnera les indices des valeurs qu'on souhaite récupérer. Cela se fait de la même manière avec NumPy qu'avec les listes en Python, que nous avons vues dans le chapitre précédent. L'indexation simple permet d'accéder à un élément unique d'un tableau.

Syntaxe pour un `ndarray` à une dimension

```
mon_array[2]
```

Ce code nous permet d'accéder à la troisième valeur du tableau `mon_array`. Il ne faut pas oublier qu'on commence à compter à partir de 0 en Python, ainsi 2 correspond à la troisième valeur.

Syntaxe pour un `ndarray` à deux dimensions

```
mon_array[2,1]
```

Ce code permet d'accéder à la valeur située à la troisième ligne, deuxième colonne du tableau `mon_array`.

Enfin, pour modifier une valeur d'un `ndarray`, il suffit d'utiliser la syntaxe suivante :

```
mon_array[2,1]=10
```

On sélectionne la position où se trouve la valeur à modifier et on lui assigne une nouvelle valeur avec le symbole d'assignation =. Avec cette syntaxe, cela modifie la valeur à la troisième ligne, deuxième colonne, par la valeur `10`.

Testons cela sur notre tableau `iris`, créé précédemment. Modifions la valeur à la troisième ligne, deuxième colonne.

Code

```
import numpy as np
iris=np.genfromtxt("../datasets/iris.csv",
delimiter=";",skip_header=True)
iris[2,1]
```

Résultat

```
3.2
```

Avec ce code, nous sélectionnons la valeur à la ligne 3, colonne 2 du tableau `iris` et nous l'affichons. À présent, modifions-la et mettons la valeur `4` à la place de `3.2`.

Code

```
iris[2,1]=4
iris[2,1]
```

Résultat

```
4.0
```

© Editions ENI - All rights reserved

On a bien modifié la valeur grâce à l'indexation. De plus, vous pouvez constater que la valeur n'est pas `4` mais `4.0`, puisque le tableau `iris` est un `ndarray` contenant des nombres de type `float`. Ainsi, NumPy convertit directement l'entier 4 en son équivalent réel.

2.2.2 Indexation booléenne

Contrairement à l'indexation simple, où on va utiliser les positions des valeurs pour les récupérer, ici, nous allons utiliser des masques booléens (*masks*, en anglais) pour sélectionner une portion du tableau. Comme nous l'avons vu dans les rappels sur Python, les opérateurs relationnels vont pouvoir être utilisés ici pour faire ce qu'on appelle une indexation booléenne.

Remarque

Rappels : les opérateurs relationnels, permettent de comparer des valeurs : ==, !=, >, <, >=, <=.

Imaginons que l'on souhaite garder uniquement les valeurs d'un tableau qui sont supérieures à la valeur `10`, la syntaxe serait la suivante :

```
mon_array[mon_array > 10]
```

La condition `mon_array > 10` va générer un `ndarray`, de même dimension que le tableau qu'on cherche à indexer, contenant des valeurs booléennes. Ce tableau de valeurs booléennes permettra à Python de savoir quelles valeurs de `mon_array` garder : celles aux positions où la valeur est à `True` dans le `ndarray` généré par `mon_array > 10`. En revanche, il ne gardera pas les valeurs de `mon_array` aux positions où les valeurs sont à `False` dans le tableau de booléens.

Nous allons voir ensemble un exemple pas à pas pour comprendre comment fonctionne cette indexation booléenne. Reprenons notre tableau `iris` et imaginons que nous ne voulions garder que les valeurs supérieures à `2` dans ce tableau, peu importe la colonne.

Code

```
iris[iris >2]
```

Aperçu du résultat

```
array([5.1, 3.5, 4.9, 3. , 4.7, 3.2, 4.6, 3.1, 5. , 3.6, 5.4, 3.9, 4.6,
       3.4, 5. , 3.4, 4.4, 2.9, 4.9, 3.1, 5.4, 3.7, 4.8, 3.4, 4.8, 3. ,
       4.3, 3. , 5.8, 4. , 5.7, 4.4, 5.4, 3.9, 5.1, 3.5, 5.7, 3.8, 5.1,
       3.8, 5.4, 3.4, 5.1, 3.7, 4.6, 3.6, 5.1, 3.3, 4.8, 3.4, 5. , 3. ,
       5. , 3.4, 5.2, 3.5, 5.2, 3.4, 4.7, 3.2, 4.8, 3.1, 5.4, 3.4, 5.2,
       4.1, 5.5, 4.2, 4.9, 3.1, 5. , 3.2, 5.5, 3.5, 4.9, 3.6,
])
```

Ici, on obtient un tableau, toutes colonnes confondues, avec uniquement les valeurs supérieures à `2`. Il n'y a plus de notions de colonnes, il s'agit d'un tableau à une dimension.

Pour que vous compreniez comment cela fonctionne, affichons juste ce que génère le masque booléen `iris >2`.

Code

```
iris >2
```

Aperçu du résultat

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

Cette expression booléenne génère un nouveau tableau, de même dimension que le tableau `iris`, mais qui contient uniquement des booléens, `True` ou `False`. Python sait qu'il doit garder uniquement les valeurs du tableau `iris` aux index pour lesquels les valeurs du tableau généré par l'expression booléenne sont à `True`. Ainsi, grâce à ce tableau de booléens, Python gardera la valeur à la première ligne, première colonne, à l'indice `[0,0]`, la valeur à la première ligne, deuxième colonne, à l'indice `[0,1]`, mais ne gardera pas la valeur à la première ligne, troisième colonne, à l'indice `[0,3]`, car elle est à `False`, etc.

© Editions ENI - All rights reserved

Remarque

L'expression booléenne génère un `ndarray` à 2 dimensions contenant des booléens, que l'on donne entre crochets à notre `ndarray` à 2 dimensions d'origine (Iris). Pourtant, en sortie, on obtient un `ndarray` à 1 dimension. Cela est dû au fait que certaines colonnes sont à False, ce qui déstructure totalement le `ndarray`, et Python sort un tableau à une dimension contenant l'ensemble des valeurs étant à True : il ne peut pas faire autrement.

Sur ce tableau `iris`, il n'y a pas vraiment d'intérêt à appliquer cette condition sur l'ensemble du tableau. Mais, par exemple, la cinquième colonne correspond à l'espèce, donc en sélectionnant l'ensemble des valeurs supérieures à 2, on supprime d'office tous les individus d'Iris de l'espèce 1.

Pour appliquer la condition sur une colonne particulière du tableau, il faut la sélectionner en écrivant le masque booléen.

Syntaxe

```
mon_array[mon_array[:,index_colonne] == 2]
```

Ici, on sélectionne la colonne en donnant sa position à droite de la virgule et on sélectionne l'ensemble des lignes de cette colonne en spécifiant le caractère `:` à gauche de la virgule (slicing).

Pour l'exemple, nous allons sélectionner uniquement les valeurs où la longueur des sépales est supérieure à 5 centimètres, c'est-à-dire où les valeurs de la première colonne sont supérieures à 5.

Code

```
iris[iris[:,0] >5]
```

Nous prenons le tableau `iris` et, entre crochets, nous écrivons notre condition : nous voulons que la première colonne (indice 0) soit supérieure à 5.

Remarque

L'expression booléenne "`iris[:,0] >5`" génère un `ndarray` à 1 dimension contenant 150 valeurs (car il y a 150 lignes dans le tableau). Ainsi, Python sait qu'il doit garder uniquement les lignes à l'indice où la valeur est à True dans (iris[:,0] >5).

Aperçu du tableau initial

```
[[5.1 3.5 1.4 0.2 1. ]
 [4.9 3.  1.4 0.2 1. ]
 [4.7 3.2 1.3 0.2 1. ]
 [4.6 3.1 1.5 0.2 1. ]
 [5.  3.6 1.4 0.2 1. ]
 [5.4 3.9 1.7 0.4 1. ]
...]
```

Aperçu du tableau final, après indexation booléenne

```
array([[5.1, 3.5, 1.4, 0.2, 1. ],
       [5.4, 3.9, 1.7, 0.4, 1. ],
       [5.4, 3.7, 1.5, 0.2, 1. ],
       [5.8, 4. , 1.2, 0.2, 1. ],
       [5.7, 4.4, 1.5, 0.4, 1. ]
...
])
```

On constate que les lignes pour lesquelles la valeur de la première colonne est strictement supérieure à 5 sont présentes, mais les autres ne sont plus là.

2.2.3 Fancy indexing

Difficile de savoir comment traduire fancy indexing (littéralement indexation fantaisie...). Toutefois, le but est simple : donner un tableau d'indices (ou une liste d'indices) entre crochets à notre `ndarray` pour sélectionner plusieurs éléments non consécutifs. Alors qu'avec l'indexation simple on n'accède qu'à un seul élément, ici on peut en sélectionner autant qu'on veut. Ce qui est intéressant aussi, c'est que le format du tableau en sortie est le même que celui du tableau d'indices qu'on donne. Nous allons voir ça sous forme d'exemples.

Tout d'abord, voici la syntaxe générale du fancy indexing :

```
mon_array[[5,2,8,4]]
```

On voit que l'ordre des indices peut ne pas être respecté, et qu'on donne autant d'indices qu'on veut. Ici, on va récupérer les valeurs du tableau `mon_array` aux indices 5, 2, 8 et 4, dans cet ordre. Cela génère un tableau à une dimension puisqu'on donne une unique liste de valeurs.

© Editions ENI - All rights reserved

Pour générer un tableau à deux dimensions, on peut structurer manuellement les indices qu'on donne entre crochets.

Syntaxe

```
import numpy as np
array_indices=np.array([[4,10,5],[12,8,9]])
mon_array[array_indices]
```

Ici, on crée un tableau à deux dimensions de type `ndarray` avec la fonction `array()` de NumPy. Ce tableau appelé `array_indices`, contient les indices et on donne ce tableau entre crochets à `mon_array`. On obtient en sortie un tableau à deux dimensions (trois colonnes et deux lignes), avec une première ligne contenant les valeurs de `mon_array` aux indices `4`, `10` et `5` et une deuxième ligne contenant les valeurs de `mon_array` aux indices `12`, `8` et `9`.

Pour l'exemple, créons un tableau à une dimension contenant des nombres de 1 à 20. À partir de ce tableau à une dimension, créons un tableau à deux dimensions avec deux lignes et trois colonnes grâce au fancy indexing.

Code

```
notre_tableau=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20])
print("Contenu de notre_tableau :")
print(notre_tableau)
print("Nouveau tableau après fancy indexing :")
print(notre_tableau[np.array([[4,10,5],[12,8,9]])])
```

Résultat

```
Contenu de notre_tableau :
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
Nouveau tableau après fancy indexing :
[[ 5 11  6]
 [13  9 10]]
```

À partir d'un tableau à une dimension, `notre_tableau`, on obtient bien un tableau à deux dimensions avec lignes et colonnes, grâce au fancy indexing.

Il existe de nombreuses possibilités pour le fancy indexing et il est très facile de trouver de la documentation autour de ce sujet sur Internet. Il faut retenir que l'intérêt du fancy indexing est réellement de pouvoir sélectionner plusieurs valeurs dans un ordre quelconque, mais aussi de pouvoir structurer directement la sortie comme on le souhaite.

2.3 Accéder aux éléments par tranche (slicing)

Contrairement à l'indexation simple, qui utilise des indices simples et permet d'accéder à un seul élément, le slicing permet de sélectionner des tranches de tableaux (*slices*, en anglais) et donc de sélectionner plusieurs éléments du tableau. Avec le slicing, on récupérera ainsi un sous-ensemble du tableau.

Voici la syntaxe du slicing avec NumPy :

```
mon_array[start:stop:step]
```

La syntaxe du slicing utilise le symbole **:** pour séparer les différentes valeurs d'index. `start` correspond à l'index auquel on souhaite commencer la sélection. Si elle n'est pas spécifiée, sa valeur par défaut est `0`, soit le début du tableau. `stop` correspond à la valeur d'index à laquelle on veut arrêter la sélection, le numéro d'index donné n'étant pas compris dans la sélection. Exemple : si on donne l'index 5 comme valeur de `stop`, Python sélectionnera les valeurs jusqu'à l'index 4 (l'index 5 n'étant pas inclus dans la sélection). Si elle n'est pas spécifiée, la valeur par défaut est le dernier index du tableau. Enfin, `step` correspond au pas ; par défaut le pas est de 1.

2.3.1 Slicing sur un tableau NumPy à 1 dimension

Commençons par faire du slicing sur un tableau NumPy à une dimension.

On reprend notre tableau à une dimension, créé dans la section précédente, `notre_tableau` :

```
notre_tableau=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20])
```

On affiche les trois premiers éléments de ce tableau grâce au slicing.

© Editions ENI - All rights reserved

Code

```
notre_tableau[:3]
```

Résultat

```
array([1, 2, 3])
```

Ici, comme on ne donne pas de `start`, la valeur est `0`, puis on donne le `stop`, qui est 3, donc Python affiche la valeur jusqu'à l'index 3 non inclus, donc jusqu'à l'index 2. Enfin, le `step` n'est pas spécifié, donc il y a un pas de 1. Ainsi, les valeurs affichées sont celles aux index `0`, `1` et `2`.

À présent, on va afficher toutes les valeurs à partir de l'index `10` jusqu'à la fin du tableau.

Code

```
notre_tableau[10:]
```

Résultat

```
array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
```

L'index `10` correspond à la valeur `11` et la valeur `20` correspond à la dernière valeur du tableau, comme le `stop` n'est pas spécifié. Le `step` est toujours par défaut à 1.

Maintenant, on va afficher les éléments des index 3 à 6.

Code

```
notre_tableau[3:7]
```

Comme nous voulons afficher la valeur à l'index 6, nous spécifions `7` comme index `stop`, puisque Python n'inclut pas le `stop` dans sa sélection.

Résultat

```
array([4, 5, 6, 7])
```

Enfin, on peut spécifier un pas, `step`. Si nous souhaitons afficher tous les éléments du tableau avec un pas de 2, cela est possible en spécifiant un `step`.

Code

```
notre_tableau[::2]
```

Ici, on ne spécifie pas le `start`, donc par défaut c'est `0`, on ne spécifie pas le `stop`, donc par défaut c'est le dernier élément du tableau, et on spécifie le `step`, qui est de 2, donc une valeur sur deux est affichée.

Résultat

```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

2.3.2 Slicing sur un tableau NumPy à 2 dimensions

Le slicing sur un tableau à deux dimensions se fait comme précédemment sur le tableau à une dimension, sauf qu'on va écrire deux slices (tranches) séparées par une virgule : cela permet de spécifier les slices à faire sur les colonnes et celles à faire sur les lignes.

Syntaxe

```
mon_array[start:stop:step, start:stop:step]
```

Comme pour l'indexing simple, à gauche la slice concerne les lignes, et à droite les colonnes.

Pour les exemples, nous allons reprendre notre tableau à deux dimensions `iris`.

Affichons les deux premières lignes et les quatre premières colonnes du tableau `iris`.

Code

```
iris[:2,:4]
```

Ici, le `start` est à `0` pour les deux, on commence à la première ligne, première colonne. On ne spécifie pas le `step`, donc il est de 1. Comme on veut les deux premières lignes, on met `2` à gauche de la virgule, pour afficher les lignes aux index `0` et `1`. Comme on veut les quatre premières colonnes, on met `4` à droite de la virgule, pour afficher les colonnes aux index `0`, `1`, `2` et `3`.

© Editions ENI - All rights reserved

Résultat

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2]])
```

Affichons les lignes à partir de l'index `145`, jusqu'à la dernière ligne, et les colonnes à partir l'index `2`, jusqu'à la dernière colonne.

Code

```
iris[145:,2:]
```

Résultat

```
array([[5.2, 2.3, 3. ],
       [5. , 1.9, 3. ],
       [5.2, 2. , 3. ],
       [5.4, 2.3, 3. ],
       [5.1, 1.8, 3. ]])
```

Pour afficher une ligne précise avec toutes ses colonnes, sans spécifier les colonnes à afficher, on peut mettre le symbole `:` à droite de la virgule, ce qui signifie "affiche-moi toutes les colonnes".

Code

```
iris[0,:]
```

Résultat

```
array([5.1, 3.5, 1.4, 0.2, 1. ])
```

La première ligne (et l'ensemble des colonnes) du tableau `iris` est affichée.

De même, si on veut afficher toutes les lignes de la première colonne, nous mettrons cette fois-ci le symbole `:` à gauche de la virgule.

Code

```
iris[:,0]
```

On peut aussi utiliser des chiffres négatifs, pour réfléchir à l'envers et afficher le tableau à partir de la fin.

Code

```
iris[-1,:]
```

Résultat

```
array([5.9, 3. , 5.1, 1.8, 3. ])
```

La dernière ligne du tableau est affichée.

À présent, vous avez un bel aperçu des possibilités du slicing. Nous vous conseillons de vous entraîner avec, il y a plein de possibilités. Le but est vraiment d'afficher un sous-tableau d'intérêt à partir d'un grand tableau.

2.4 Notion de vue et copie

Attention avec les tableaux NumPy lorsque vous faites du slicing pour créer un sous-tableau ! Expliquons par l'exemple.

Imaginons qu'on crée un sous-ensemble du tableau `notre_tableau` contenant les cinq premières valeurs du tableau.

Code

```
sous_tableau=notre_tableau[:5]
sous_tableau
```

Résultat

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

À partir de ce moment-là, on se dit que nos deux tableaux, `notre_tableau` et `sous_tableau`, sont distincts. Regardons si c'est vrai.

Modifions la valeur du premier élément de `sous_tableau`.

Code

```
sous_tableau[0]=10
sous_tableau
```

© Editions ENI - All rights reserved

Résultat

```
array([10,  2,  3,  4,  5])
```

La première valeur est bien devenue `10`.

Regardons maintenant l'objet `notre_tableau`.

Code

```
notre_tableau
```

Résultat

```
array([10,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20])
```

La première valeur est devenue `10` aussi ! Pourtant, nous avons modifié l'objet `sous_tableau` et non pas `notre_tableau` !

Lorsqu'on fait du slicing sur un tableau, ce qu'il retourne, c'est une **vue** du tableau et non pas une **copie** du tableau. Ainsi, une modification de `sous_tableau` entraîne la modification du tableau initial. Quand on fait une extraction de cette manière (slicing), on crée une vue de `notre_tableau` qu'on range dans `sous_tableau`. `sous_tableau` n'est pas un nouvel `ndarray`, mais bien une vue du `ndarray notre_tableau`. C'est un peu comme si on zoomait sur le tableau `notre_tableau`.

Remarque

Les slices de liste, par exemple, sont bien des copies de la liste originale, c'est-à-dire que si on modifie la sous-liste, la liste initiale n'est pas modifiée. La notion de vue et copie est réellement spécifique à la librairie NumPy.

Vous vous demandez sûrement la raison pour laquelle NumPy crée une vue par défaut. En fait, NumPy a été développée pour travailler sur de très gros tableaux et pour être très performante en calcul scientifique. Ainsi, il faut savoir que créer une copie prend deux fois plus de temps que de créer une vue, mais aussi que quand on crée une copie du tableau, cela prend de la mémoire supplémentaire, alors que la vue partage le même espace mémoire (sur votre ordinateur) que le tableau original. Étant donné que la mémoire est souvent ce qui est limitant dans le cas de gros calculs, cet avantage des vues n'est pas négligeable.

Si toutefois vous voulez vraiment travailler sur une copie du tableau original, c'est-à-dire sur un sous-ensemble du tableau original sans modifier l'original, vous pouvez explicitement créer une copie.

Code

```
notre_tableau=np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
18,19,20])
sous_tableau=notre_tableau[:5].copy()
sous_tableau[0]=10
print(sous_tableau)
print(notre_tableau)
```

Ici, nous recréons le tableau initial. Ensuite, nous créons un sous-ensemble du tableau original contenant les cinq premières valeurs de `notre_tableau` que nous rangeons dans `sous_tableau`, en spécifiant qu'il s'agit d'une copie grâce à la méthode `copy()` de NumPy. Enfin, nous modifions la première valeur de `sous_tableau`.

Résultat

```
[10  2  3  4  5]

[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```

Aucune modification de `notre_tableau` n'a été faite avec ce code, la première valeur est restée `1`, alors que la première valeur de `sous_tableau` est bien devenue `10`. Ainsi, vous comprenez bien la différence entre une vue et une copie.

3. Les opérations mathématiques avec NumPy

3.1 Les opérations arithmétiques

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

© Editions ENI - All rights reserved

Syntaxe

```
import numpy as np
np.add(mon_array,nombre)
np.add(mon_array,mon_array2)
```

Ceci est la syntaxe générale de la fonction `add()` ; celle des autres fonctions est la même, seul le résultat change.

On peut additionner l'ensemble des éléments d'un tableau avec un `nombre`, ou encore additionner deux tableaux.

Très simplement, créons un tableau à deux dimensions, comme suit.

Code

```
mon_tableau=np.array([[10,11,12],[13,14,15],[16,17,18]])
mon_tableau
```

Résultat

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

C'est un tableau à deux dimensions : trois lignes et trois colonnes.

Si on veut ajouter la valeur `10` à l'ensemble des valeurs de ce tableau, on utilisera la fonction `add()` de la même manière que la syntaxe présentée précédemment.

Code

```
import numpy as np
np.add(mon_tableau,10)
```

Résultat

```
array([[20, 21, 22],
       [23, 24, 25],
       [26, 27, 28]])
```

Si on veut multiplier l'ensemble des valeurs par 20, on utilisera cette fois-ci la fonction `multiply()`.

Code

```
import numpy as np
np.multiply(mon_tableau,20)
```

Et de même pour soustraire et diviser, avec les fonctions respectives `subtract()` et `divide()`.

Comme nous l'avons montré dans la syntaxe, il est possible de faire ces calculs entre deux tableaux `ndarray`. Créons un nouveau tableau pour illustrer les exemples.

Code

```
mon_tableau2=np.array([10,20,30])
mon_tableau
```

Résultat

```
array([10, 20, 30])
```

Il s'agit d'un tableau à une dimension, contenant une ligne et trois colonnes. Le tableau `mon_tableau`, lui, est un tableau à deux dimensions de trois lignes et trois colonnes. Nous allons soustraire ces deux tableaux de dimensions différentes.

Code

```
import numpy as np
np.subtract(mon_tableau,mon_tableau2)
```

Résultat

```
array([[  0,  -9, -18],
       [  3,  -6, -15],
       [  6,  -3, -12]])
```

Avec ce code, on enlève 10 à la première colonne du tableau `mon_tableau`, on enlève 20 à la deuxième colonne de `mon_tableau` et on enlève 30 à la troisième colonne de `mon_tableau`. Cela est dû au fait que le tableau `mon_tableau2` contient le même nombre de "colonnes" que l'objet `mon_tableau`, ainsi NumPy soustrait à chaque colonne de `mon_tableau` la valeur à la position correspondante dans l'objet `mon_tableau2`.

© Editions ENI - All rights reserved

Il est aussi possible de faire ça avec des tableaux de mêmes dimensions. Créons un tableau de mêmes dimensions que `mon_tableau`.

Code

```
mon_tableau3=np.array([[1,2,3],[4,5,6],[7,8,9]])
mon_tableau3
```

Résultat

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

Maintenant, multiplions les tableaux de mêmes dimensions.

Code

```
np.multiply(mon_tableau,mon_tableau3)
```

Résultat

```
array([[ 10,  22,  36],
       [ 52,  70,  90],
       [112, 136, 162]])
```

Ici, chaque valeur d'un tableau est multipliée par la valeur à la même position dans l'autre tableau. Ainsi, la valeur de `mon_tableau` à la ligne 1, colonne 1 est multipliée par la valeur de `mon_tableau3` à la ligne 1, colonne 1. La valeur de `mon_tableau` à la ligne 1, colonne 2 est multipliée à la valeur de `mon_tableau3` à la ligne 1, colonne 2, etc.

3.2 Les fonctions d'agrégations

Lorsqu'on a un grand tableau de données, il est important de pouvoir explorer un résumé de ce tableau. La classe `ndarray` de NumPy possède des méthodes d'agrégation : `sum()` (somme), `median()` (médiane), `mean()` (moyenne), `min()` (minimum), `max()` (maximum), `corrcoef()` (coefficient de corrélation), `cumsum()` (somme cumulative) et `std()` (écart-type).

Ces méthodes peuvent être appliquées soit sur l'ensemble des éléments du tableau, soit par colonne ou par ligne, dans le cas de tableaux à deux dimensions. Pour cela, il suffit de préciser l'axe qu'on veut traiter avec l'option `axis` : `axis = 0` signifie qu'on veut effectuer le calcul par colonne et `axis = 1` signifie qu'on veut effectuer le calcul par ligne.

Syntaxe

```
import numpy as np
np.sum(array)
np.sum(array, axis=0)
np.sum(array, axis=1)
```

Reprenons le tableau `iris` pour les exemples et appliquons la fonction `sum()` sur l'ensemble des éléments, puis par colonne et enfin par ligne.

Code

```
import numpy as np
print("Somme sur l'ensemble des éléments:")
print(np.sum(iris))
print("Somme par colonne:")
print(np.sum(iris, axis=0))
print("Somme par ligne:")
print(np.sum(iris, axis=1))
```

Résultat

```
Somme sur l'ensemble des éléments:
2378.7
Somme par colonne:
[876.5 458.6 563.7 179.9 300. ]
Somme par ligne:[11.2 10.5 10.4 10.4 11.2 12.4 10.7 11.1 9.9 10.6
11.8 11. 10.3  9.5
12.2 13.  12.  11.3 12.5 11.7 11.7 11.7 10.4 11.6 11.3 10.8 11.4 11.4
11.2 10.7 10.7 11.7 11.9 12.3 10.7 10.6 11.5 11.   9.9 11.2 11.1  9.4
10.1 11.7 12.2 10.5 11.7 10.4 11.7 10.9 18.3 17.6 18.4 15.1 17.4 16.3
17.9 13.6 17.4 15.2 13.5 16.6 15.2 17.1 15.4 17.6 16.6 15.6 16.4 15.1
17.7 16.2 17.2 16.8 16.9 17.4 17.8 18.4 16.9 14.8 14.8 14.6 15.6 17.4
16.4 17.5 18.  16.3 16.  15.3 15.7 17.1 15.6 13.6 15.8 16.1 16.1 16.7
13.7 15.9 21.1 18.5 21.1 19.6 20.5 22.3 16.6 21.3 19.8 22.4 19.8 19.3
20.4 18.2 19.1 20.2 19.8 23.4 22.5 17.7 21.1 18.3 22.2 18.7 20.8 21.2
18.6 18.8 19.9 20.6 21.2 23.1 20.  18.7 18.7 22.1 20.7 19.8 18.6 20.5
20.8 20.4 18.5 21.2 21.2 20.2 18.7 19.7 20.3 18.8]
```

© Editions ENI - All rights reserved

Quand on ne spécifie pas d'option `axis` à la fonction `sum()`, elle additionne l'ensemble des valeurs du tableau `iris` et retourne donc une valeur unique, ici `2378.7`. Si on précise l'option `axis=0` la somme sera effectuée par colonne et on aura une valeur par colonne, donc 5 valeurs ici. Enfin, si on précise l'option `axis=1`, la somme sera faite par ligne et on obtient une valeur par ligne, donc 150 valeurs, dans cet exemple.

Les autres fonctions s'utilisent de la même manière. Par exemple, si on veut la moyenne par colonne ainsi que l'écart-type correspondant, on utilisera respectivement les fonctions `mean()` et `std()`.

Code

```
import numpy as np
print(np.mean(iris, axis=0))
print(np.std(iris, axis=0))
```

Résultat

```
[5.84333333 3.05733333 3.758      1.19933333 2.        ]
[0.82530129 0.43441097 1.75940407 0.75969263 0.81649658]
```

Grâce à ces deux fonctions, on sait qu'en moyenne, la longueur des sépales des 150 iris du tableau est de 5.84 cm avec un écart-type de 0.8 cm, la largeur des sépales de 3.05 cm avec un écart-type de 0.43 cm, la longueur des pétales de 3.75 cm avec un écart-type de 1.75 cm et la largeur des pétales de 1.19 cm avec un écart-type de 0.81 cm.

Remarque

Définition de l'écart-type : permet de mesurer la dispersion d'un ensemble de valeurs. Plus la distribution des valeurs est dispersée, plus l'écart-type sera élevé. Par rapport à la moyenne, plus l'écart-type est grand, plus les valeurs sont éloignées de la moyenne (donc moins la moyenne est représentative de ces valeurs).

Enfin, nous allons effectuer un dernier exemple où nous allons afficher les valeurs maximales de chaque colonne du tableau `iris`.

Code

```
import numpy as np
print(np.max(iris, axis=0))
```

Résultat

```
[7.9 4.4 6.9 2.5 3. ]
```

La longueur de sépale la plus grande est de 7.9 cm, la largeur de sépale la plus grande est de 4.4 cm, la longueur de pétale la plus grande est de 6.9 cm et la largeur de pétale la plus grande est de 2.5 cm.

Il est possible de se faire une première idée sur les données grâce aux fonctions d'agrégation. Dans la prochaine section, nous allons voir comment inspecter un tableau NumPy grâce aux attributs fournis par NumPy.

4. Inspecter un tableau grâce aux attributs de NumPy

Les objets de classe `ndarray` (les tableaux NumPy) possèdent un ensemble d'attributs qui permettent de les inspecter. La liste de ces attributs est disponible dans la documentation de NumPy, dans la partie *Attributes* :
https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html

Nous allons en voir quelques-uns ensemble :

- Il est possible de connaître le type des éléments d'un tableau avec l'attribut `dtype`.
- Il est possible de connaître le nombre d'éléments contenus dans un tableau grâce à l'attribut `size`.
- Il est possible de faire la transposée d'un tableau, c'est-à-dire de permuter les dimensions (les lignes devenant les colonnes et les colonnes les lignes), grâce à l'attribut `T`.
- Il est possible de connaître la dimension d'un tableau avec l'attribut `ndim`, afin de savoir s'il s'agit d'un tableau à une dimension, deux dimensions, trois dimensions, etc.
- Il est possible de connaître la taille des dimensions d'un tableau, c'est-à-dire le nombre de lignes et de colonnes, grâce à l'attribut `shape`.

© Editions ENI - All rights reserved

Syntaxe d'utilisation des attributs

```
mon_array.attribut
```

Remarque

Pour rappel, l'attribut est une variable liée à une classe d'objets. Il permet d'accéder aux informations d'un objet de cette classe en faisant précéder l'attribut par le nom de cet objet.

Reprenons le tableau `iris` et utilisons les différents attributs listés précédemment.

Code

```
print("Attribut size:")
print(iris.size)
print("Attribut shape:")
print(iris.shape)
print("Attribut ndim:")
print(iris.ndim)
print("Attribut dtype:")
print(iris.dtype)
```

Résultat

```
Attribut size:
750
Attribut shape:
(150, 5)
Attribut ndim:
2
Attribut dtype:
float64
```

Grâce aux attributs, on obtient plusieurs informations sur notre tableau `iris`. On sait qu'il contient 750 éléments, qu'il contient 150 lignes et 5 colonnes, que c'est un tableau à deux dimensions et que le type des données est `float64` (nombres réels). Les attributs sont donc pratiques pour avoir un premier aperçu de la structure du tableau qu'on doit analyser.

Pour la transposée, le code est le suivant.

Code

```
iris.T
```

Avec ce code, les lignes deviennent des colonnes et les colonnes des lignes, on peut le vérifier avec le code suivant.

Code

```
(iris.T).shape
```

Résultat

```
(5, 150)
```

On a bien 5 lignes et 150 colonnes à présent.

5. Manipuler des tableaux NumPy

5.1 Ajouter et supprimer des éléments dans un tableau

5.1.1 Ajouter des éléments dans un tableau

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

Syntaxe d'utilisation de la fonction `append()`

```
import numpy as np
np.append(mon_array, elements_a_ajouter)
```

Le cas d'utilisation le plus simple concerne les tableaux à une dimension, pour lesquels il suffit de donner une liste de valeurs. Pour l'exemple, reprenons l'objet `notre_tableau`, qui contient une suite de valeurs allant de 1 à 20, et ajoutons cinq nouvelles valeurs à la suite du tableau.

© Editions ENI - All rights reserved

Code

```
np.append(notre_tableau,[21,22,23,24,25])
```

Résultat

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25])
```

On a bien ajouté les valeurs 21, 22, 23, 24, 25 en fin de tableau avec la fonction `append()`.

Avec un tableau à deux dimensions, on peut préciser selon quel axe on souhaite ajouter les valeurs : on peut spécifier si on souhaite ajouter une colonne, une ligne, ou encore ne rien spécifier et les éléments seront ajoutés à la suite. Attention, si on souhaite ajouter une ligne ou une colonne, il faut que ce qu'on donne comme éléments à `append` soit de même structure que le tableau auquel on ajoute la ligne ou la colonne.

Exemple avec `iris`, imaginons qu'on veuille ajouter une ligne à la fin du tableau.

Code

```
np.append(iris,[[10,10,10,10,10]], axis=0)
```

Résultat

```
array([[ 5.1,  3.5,  1.4,  0.2,  1. ],
...
[ 5.9,  3. ,  5.1,  1.8,  3. ],
[10. , 10. , 10. , 10. , 10. ]])
```

Ici, les doubles crochets permettent de donner un tableau à deux dimensions en entrée à `append()`, contenant cinq valeurs : cela est nécessaire puisqu'il y a cinq colonnes dans le tableau. En donnant le bon nombre de valeurs entre doubles crochets, on respecte la structure du tableau `iris` et la fonction `append()` fonctionne. En revanche, si on donnait ces cinq valeurs entre simples crochets ou si on donnait sept valeurs entre doubles crochets, Python générerait une erreur.

Pour insérer une ou plusieurs valeurs à un endroit précis du tableau, il faut utiliser la fonction `insert()` de NumPy, dont la syntaxe est la suivante :

```
import numpy as np
np.insert(mon_array, index, element)
```

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

Exemple sur un tableau à une dimension, avec l'objet `notre_tableau`.

Code

```
import numpy
np.insert(notre_tableau, 1, 42)
```

Résultat

```
array([ 1, 42, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20])
```

L'élément `42` a été ajouté à l'index `1`, donc à la place de la valeur `2`, et la valeur `2` se retrouve alors à l'index `2`.

On peut aussi le faire avec une liste d'éléments.

Code

```
import numpy
np.insert(notre_tableau, 1, [42,43,44,45,46])
```

Résultat

```
array([ 1, 42, 43, 44, 45, 46, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
       13, 14, 15, 16, 17, 18, 19, 20])
```

Pour utiliser cette fonction sur un tableau à deux dimensions, il faudra encore une fois respecter le bon nombre de valeurs si on précise l'axe sur lequel on souhaite insérer les nouvelles valeurs. Par exemple, si on souhaite ajouter une ligne à l'index 1 du tableau `iris`, il faudra respecter le nombre de colonnes de ce tableau et donc donner cinq valeurs. Tout comme nous l'avons fait précédemment avec la fonction `append()`.

© Editions ENI - All rights reserved

Code

```
import numpy
np.insert(iris, 1, [10,10,10,10,10], axis = 0)
```

Résultat

```
array([[ 5.1,  3.5,  1.4,  0.2,  1. ],
       [10. , 10. , 10. , 10. , 10. ],
       [ 4.9,  3. ,  1.4,  0.2,  1. ],
       ...
       [ 5.9,  3. ,  5.1,  1.8,  3. ]])
```

On vient d'insérer une ligne en deuxième position du tableau `iris`, contenant cinq valeurs `10`.

Remarque

Pour `append()` comme pour `insert()`, si on ne précise pas d'axe sur lequel ajouter les valeurs, NumPy transformera le tableau à deux dimensions en tableau à une dimension. Dans le cas de la fonction `insert()`, NumPy ajoutera les valeurs à partir de l'index 1 de ce tableau à une dimension. Cet exemple est disponible dans le notebook lié au chapitre et disponible en téléchargement.

5.1.2 Supprimer des éléments d'un tableau

Pour supprimer des éléments dans un tableau, il suffit d'utiliser la fonction `delete()` de NumPy.

Syntaxe

```
import numpy as np
np.delete(mon_array, index, axe)
```

Si on ne précise pas l'axe, les éléments aux index donnés seront supprimés. Si on précise l'axe, par exemple `axis=1`, ce sont les colonnes entières, aux numéros d'index spécifiés, qui sont supprimées.

Testons cette fonction avec le tableau `iris`.

Code

```
np.delete(iris, 1, axis = 1)
```

Résultat

```
array([[5.1, 1.4, 0.2, 1. ],
       [4.9, 1.4, 0.2, 1. ],
       [4.7, 1.3, 0.2, 1. ],
       [4.6, 1.5, 0.2, 1. ],
... ])
```

On constate que la colonne 2 (à l'index 1) a bien été supprimée.

Remarque

Il en va de même pour les lignes en spécifiant `axis=0`. Si vous souhaitez supprimer plusieurs lignes d'un tableau, il suffit de donner une liste d'index et de spécifier `axis=0`. L'exemple est disponible dans le notebook.

Assez logiquement, pour des tableaux à une dimension comme `notre_tableau`, il n'y a pas besoin de spécifier l'axe. Vous pouvez supprimer une valeur à un index unique ou plusieurs valeurs en donnant une liste d'index à la fonction `delete()`.

Code

```
np.delete(notre_tableau,1)
```

Résultat

```
array([ 1,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
       19, 20])
```

La valeur 2 qui se trouvait à l'index 1 a bien été supprimée.

5.2 Diviser un tableau NumPy (split, hsplit et vsplit)

Il existe trois fonctions pour diviser un tableau avec NumPy :

- la fonction `split()`, utilisée pour les tableaux à une dimension,
- la fonction `hsplit()`, qui permet de diviser les tableaux à deux dimensions horizontalement,
- la fonction `vsplit()`, qui permet de diviser les tableaux à deux dimensions verticalement.

© Editions ENI - All rights reserved

Voici la syntaxe générale de l'utilisation de ces fonctions :

```
import numpy as np
np.split(mon_array,index)
np.hsplit(mon_array,index)
np.vsplit(mon_array,index)
```

Le but ici est de "splitter", c'est-à-dire "casser/diviser" le tableau en plusieurs morceaux. L'index permet de dire à partir de quel index du tableau on souhaite le diviser.

5.2.1 Sur un tableau à une dimension

Imaginons que l'on veuille couper en trois morceaux notre tableau `notre_tableau`.

Aperçu de `notre_tableau`

```
notre_tableau
```

Résultat

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20])
```

Division du tableau en trois sous-tableaux.

Code

```
import numpy as np
notre_tableau1, notre_tableau2, notre_tableau3 =
np.split(notre_tableau, [2, 6])
print(notre_tableau1, notre_tableau2, notre_tableau3)
```

Résultat

```
[1 2] [3 4 5 6] [ 7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```

De cette manière, on obtient trois tableaux. Un tableau avec les éléments avant l'index 2, un tableau avec les éléments entre l'index 2 et l'index 5 de `notre_tableau` et enfin un tableau avec les éléments à partir de l'index 6 jusqu'à la fin de `notre_tableau`. Ici, 2 et 6 sont les points de cassure donc on obtient trois tableaux. Si on avait mis un index unique et non pas une liste d'index, on aurait eu un unique point de cassure et donc deux tableaux.

On peut diviser le tableau en autant de sous-tableaux qu'on le souhaite, en spécifiant autant de points de cassure que nécessaire.

5.2.2 Sur un tableau à deux dimensions

Sur un tableau à deux dimensions, on peut diviser le tableau verticalement (`vsplit()`) ou horizontalement (`hsplit()`). `vsplit()` et `hsplit()` ne fonctionnent que sur les tableaux à deux dimensions.

Pour l'exemple, créons un tableau à deux dimensions simple et divisons-le verticalement en deux sous-tableaux, à l'index 2.

Code

```
array=np.array([[2,5,6],[9,10,11],[5,6,9]], dtype="float")
print("Avant division")
print(array)
array1, array2 = np.vsplit(array, [2])
print("Après division")
print(array1)
print(array2)
```

Résultat

```
Avant division
[[ 2.  5.  6.]
 [ 9. 10. 11.]
 [ 5.  6.  9.]]
Après division
[[ 2.  5.  6.]
 [ 9. 10. 11.]]
 [[5. 6. 9.]]
```

Ici, on coupe verticalement en lui donnant l'index 2, donc on lui demande que les deux premières lignes soient mises dans un tableau et la dernière dans un autre tableau. En lui donnant l'index 2 en option, cela signifie que de 0 à 2 (non compris), NumPy met les lignes dans un tableau, puis le reste du tableau dans un autre tableau.

Enfin, visualisons ensemble une division horizontale.

© Editions ENI - All rights reserved

Code

```
array= np.array([[2,5,6],[9,10,11],[5,6,9]], dtype="float")
print("Avant division")
print(array)
array1, array2 = np.hsplit(array, [2])
print("Après division")
print(array1)
print(array2)
```

Résultat

```
Avant division
[[ 2.  5.  6.]
 [ 9. 10. 11.]
 [ 5.  6.  9.]]
Après division
[[ 2.  5.]
 [ 9. 10.]
 [ 5.  6.]]
[[ 6.]
 [11.]
 [ 9.]]
```

Là, on obtient les deux premières colonnes dans un tableau et la dernière dans un autre.

5.3 Concaténer/combiner des tableaux

Trois fonctions NumPy sont disponibles pour combiner des tableaux : `concatenate()`, `vstack()` et `hstack()`.

5.3.1 La fonction concatenate()

La fonction `concatenate()` permet de faire de la concaténation ou de la jointure de deux tableaux en NumPy.

Syntaxe

```
import numpy as np
np.concatenate([mon_array1,mon_array2])
```

Il suffit de donner une liste de tableaux NumPy à la fonction, comme argument, pour qu'elle les concatène.

Exemple sur des tableaux à une dimension.

Code

```
array1 = np.array([1, 2, 3,4,5])
array2 = np.array([6,7,8,9])
print(array1)
print(array2)
np.concatenate([array1, array2])
```

Résultat

```
[1 2 3 4 5]
[6 7 8 9]
Out[84]:
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

À partir de deux tableaux à une dimension, on obtient un tableau unique contenant l'ensemble des éléments des deux tableaux.

Exemple sur un tableau à deux dimensions.

Code

```
array1= np.array([[1,2,3],[4,5,6],[7,8,9]], dtype="float")
array2= np.array([[2,5,6],[9,10,11],[5,6,9]], dtype="float")
print(array1)
print(array2)
np.concatenate([array1, array2])
```

Résultat

```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]

[[ 2.  5.  6.]
 [ 9. 10. 11.]
 [ 5.  6.  9.]]

array([[ 1.,  2.,  3.],
```

© Editions ENI - All rights reserved

```
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.],
       [ 2.,  5.,  6.],
       [ 9., 10., 11.],
       [ 5.,  6.,  9.]])
```

Les deux tableaux sont bien concaténés, l'un en dessous de l'autre. Bien sûr, pour pouvoir faire ça il faut, en toute logique, respecter les dimensions. Par exemple ici, les deux tableaux ont le même nombre de colonnes, donc ça marche bien.

On peut aussi concaténer plus de deux tableaux à la fois.

Code

```
array3= np.array([[4,10,13],[7,14,15]], dtype="float")
np.concatenate([array1, array2,array3])
```

Résultat

```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.],
       [ 2.,  5.,  6.],
       [ 9., 10., 11.],
       [ 5.,  6.,  9.],
       [ 4., 10., 13.],
       [ 7., 14., 15.]])
```

5.3.2 Les fonctions vstack() et hstack()

Si on souhaite combiner des tableaux de dimensions différentes, par exemple un tableau de dimension 1 avec un tableau de dimension 2, on va plutôt utiliser la fonction `vstack()` pour faire ce qu'on appelle un empilement vertical et la fonction `hstack()` pour faire un empilement horizontal.

Syntaxe d'utilisation des deux fonctions

```
import numpy as np
np.vstack([mon_array1,mon_array2])
np.hstack([mon_array1,mon_array2])
```

Pour l'exemple, on va créer un tableau de dimension 1, nommé `array1`, et un tableau de dimension 2, nommé `array2`, puis on va faire un empilement vertical et horizontal, pour voir ce que génèrent ces fonctions.

Code

```
array1 = np.array([15,16,47])
array2 = np.array([[1,2,3],[4,5,6],[7,8,9]], dtype="float")
print(array1)
print("---")
print(array2)
print("---")
print(np.vstack([array1, array2]))
print("---")
array1 = np.array([[15],[16],[47]])
print("---")
print(array1)
print("---")
print(array2)
print("---")
print(np.hstack([array1, array2]))
```

Résultat

```
[15 16 47]
---
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
---
[[15. 16. 47.]
 [ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
---
[[15]
 [16]
 [47]]
---
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
---
[[15.  1.  2.  3.]
```

© Editions ENI - All rights reserved

```
[16.  4.  5.  6.]
[47.  7.  8.  9.]]
```

Pour l'empilement vertical, il est nécessaire que `array1` ait le même nombre de colonnes que `array2`. Pour l'empilement horizontal, on redéfinit `array1` afin qu'il ait le même nombre de lignes que `array2`. Pour que les empilements fonctionnent, il est en effet nécessaire de respecter les nombres de lignes pour `hstack()` et de colonnes pour `vstack()`. On voit bien qu'avec `vstack()` on obtient un tableau avec une ligne supplémentaire et avec `hstack()` on obtient un tableau avec une colonne supplémentaire.

6. Introduction aux matrices avec NumPy

Un tableau à deux dimensions peut être considéré comme une matrice. Toutefois, NumPy propose un type particulier pour bien préciser que le tableau est une matrice, il s'agit du type `mat`.

Syntaxe pour transformer un tableau de type `ndarray` en matrice de type `mat` :

```
import numpy as np
ma_matrice=np.mat(mon_array)
```

La fonction `mat()` de NumPy permet de facilement faire cette conversion.

Il est ensuite possible de faire des opérations matricielles (multiplication par un scalaire, produit matriciel, transposition, etc.). Nous ne développerons pas les matrices dans ce livre, toutefois, voici les syntaxes des opérations matricielles élémentaires.

Les matrices, tout comme les tableaux NumPy, possèdent l'attribut `T` pour générer la transposée de la matrice. Mais les matrices possèdent aussi des attributs supplémentaires spécifiques, comme l'attribut `.H` pour générer la transposée conjuguée et l'attribut `.I` pour générer l'inverse de la matrice.

Syntaxes

```
ma_matrice.T
ma_matrice.H
ma_matrice.I
```

Pour obtenir le produit d'une matrice par elle-même, on utilisera la syntaxe :

```
ma_matrice**2
```

On obtient ainsi la matrice résultant du produit `ma_matrice*ma_matrice`.

Remarque

Si `ma_matrice` est de type `ndarray` et qu'on fait `ma_matrice**2`, on obtient un `ndarray` avec tous les éléments puissance 2 : cela montre bien la nécessité de changer le type du `ndarray` en matrice si on souhaite effectuer des calculs matriciels !

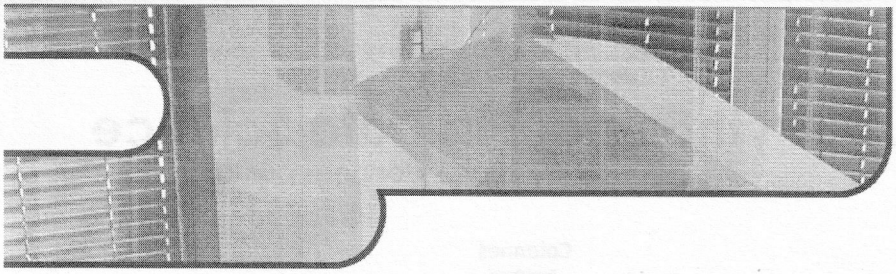
Remarque

Si on souhaite effectuer un produit matriciel sur deux `ndarrays` plutôt qu'une multiplication élément par élément, comme c'est le cas par défaut, il est possible d'utiliser la méthode `dot()`. Cette méthode permet de faire le produit matriciel entre deux `ndarrays`, sans avoir à les transformer en objet `mat`.

Exemple de code

```
# exemple de code pour obtenir un ndarray de chaque élément
de array2 puissance 2 :
print(array2**2)
# 3 exemples de code pour obtenir le produit matriciel
d'une matrice avec elle-même
print(np.mat(array2)**2)
print(np.mat(array2)*np.mat(array2))
print(array2.dot(array2))
```

© Editions ENI - All rights reserved



Chapitre 5
Maîtriser la librairie Pandas

1. Introduction

1.1 Introduction à la librairie Pandas

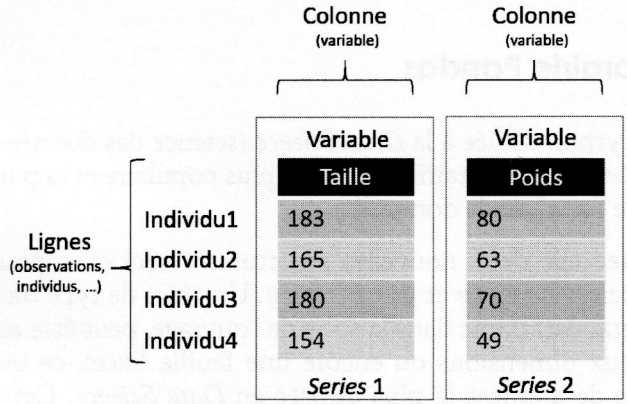
Pandas est une librairie Python dédiée à la *Data Science* (science des données, en français). Il s'agit d'ailleurs de la librairie Python la plus populaire et la plus performante pour faire de l'analyse de données.

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

Colonnes
(variables)

Lignes (observations, individus, ...)	Variable1	Variable2	Variable3	Variable4
	Taille	Poids	Age	Sexe
Individu1	183	80	20	M
Individu2	165	63	53	F
Individu3	180	70	19	M
Individu4	154	49	34	F

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



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

© Editions ENI - All rights reserved

Il faut savoir que la librairie Pandas s'appuie fortement sur la librairie NumPy, puisque ses structures de données sont basées sur les tableaux NumPy que nous avons vus précédemment (`ndarrays`). Ainsi, en plus de la structure `ndarrays`, la librairie Pandas profite aussi des performances de calcul de NumPy.

L'avantage de Pandas par rapport à NumPy est que l'objet de type `DataFrame` permet de stocker des données de types différents, donc des données hétérogènes. Cela signifie que dans un dataframe, on pourra avoir une colonne contenant des chiffres, une autre contenant du texte, une autre des booléens, etc. Cette possibilité de stocker des données hétérogènes vient du fait qu'un dataframe est constitué d'un ensemble de séries qui sont indépendantes entre elles et peuvent donc contenir des types de données différents de l'une à l'autre. En revanche, au sein d'une colonne, les données doivent être de même type.

De plus, les dataframes et les séries sont fournis avec la possibilité d'assigner des étiquettes aux données, plutôt que de travailler avec des index numériques, comme c'était le cas avec les `ndarrays` de NumPy. Avec Pandas, les lignes et les colonnes peuvent être identifiées avec des étiquettes plutôt que des nombres, vous en verrez l'utilité plus tard.

Remarque

Pour rappel, pour accéder aux valeurs des `ndarrays` ou pour effectuer du slicing, nous utilisions les index, qui sont toujours des nombres avec les objets de type `ndarrays`.

Enfin, Pandas permet de manipuler efficacement les données manquantes ainsi que les données de séries temporelles (*time series data*), deux types de données souvent rencontrées en *Data Science* et dont la manipulation serait complexe sans Pandas.

Vous l'aurez compris, les séries et les dataframes sont des structures bien mieux adaptées au travail de l'analyste de données que ne le sont les `ndarrays` de NumPy. Ainsi, pour pouvoir faire de l'analyse de données avec Python, il est impératif de maîtriser cette librairie essentielle qu'est Pandas.

1.2 Introduction au jeu de données utilisé pour les exemples

Pour s'appuyer sur des exemples tout au long de ce chapitre, nous allons utiliser le jeu de données nommé *120 years of Olympic history: athletes and results*, que vous pouvez traduire par "120 ans d'histoire des jeux olympiques : athlètes et résultats". Ce jeu de données est disponible sur la plateforme Kaggle, évoquée en introduction de ce livre, à l'adresse qui suit :
https://www.kaggle.com/heesoo37/
120-years-of-olympic-history-athletes-and-results

Remarque

Le fichier contenant les données des Jeux olympiques ainsi que le notebook associé à ce chapitre sont disponibles en téléchargement en complément de ce livre. Le fichier est nommé "120-years-of-olympic-history-athletes-and-results.csv" et le notebook est nommé "Notebook_chapitre_5_Pandas.ipynb". Vous pouvez charger directement le notebook dans votre Jupyter Notebook si vous souhaitez relancer les commandes et voir par vous-même les résultats que cela génère. Vous pouvez aussi créer votre propre notebook et écrire les commandes de ce livre par vous-même, ce qui est parfois plus formateur que de simplement exécuter les commandes.

Le fichier "120-years-of-olympic-history-athletes-and-results.csv" contient des informations sur les résultats de l'ensemble des athlètes ayant participé aux Jeux olympiques de 1896 à 2016. La personne ayant créé ce jeu de données a récupéré les informations à partir du site web : www.sports-reference.com

Il s'agit d'un tableau à deux dimensions dont nous allons nous servir pour illustrer l'utilisation de la librairie Pandas mais aussi des librairies Matplotlib et Seaborn, dans les prochains chapitres.

Ce tableau contient 271 116 lignes, une ligne correspondant à un athlète et ses résultats à l'épreuve à laquelle il a participé. Un athlète peut être présent plusieurs fois dans le tableau, puisqu'il a pu participer à plusieurs épreuves et/ou à plusieurs Jeux olympiques.

Ce tableau contient 15 colonnes, dont voici la description :

- `ID` : chaque athlète a un identifiant unique dans le tableau, rangé dans la colonne ID, ce qui permet de bien les différencier au cas où deux athlètes seraient des homonymes.

© Editions ENI - All rights reserved

- `Name` : le nom de l'athlète.
- `Sex` : le sexe de l'athlète (F pour féminin et M pour masculin).
- `Age` : l'âge de l'athlète.
- `Height` : la taille de l'athlète, en centimètres.
- `Weight` : le poids de l'athlète, en kilogrammes.
- `Team` : le nom de l'équipe à laquelle appartient l'athlète.
- `NOC` : NOC signifie Comités Nationaux Olympiques (CNO), qui est un code à trois lettres correspondant à un pays. Les CNO ont pour mission de promouvoir et protéger le mouvement olympique dans leur pays respectif.
- `Games` : l'année et la saison des Jeux olympiques. L'année est comprise entre 1896 et 2016. La saison peut prendre deux valeurs, `Summer` (pour les JO d'été) et `Winter` (pour les JO d'hiver). Il faut savoir que jusqu'en 1992, les JO d'été et d'hiver étaient organisés la même année, puis il a été décidé de les organiser tous les quatre ans avec deux années de décalage. Ainsi, en 1994, il y a eu les JO d'hiver, puis en 1996 les JO d'été, puis en 1998 les JO d'hiver, etc. Donc il faut garder en tête que jusqu'en 1992, il est possible de trouver les Jeux olympiques d'hiver et d'été la même année dans le tableau.
- `Year` : l'année où l'athlète a participé à l'épreuve sportive des JO.
- `Season` : la saison des JO (été ou hiver, `Summer` ou `Winter`).
- `City` : la ville hôte des Jeux olympiques.
- `Sport` : le sport dont fait partie l'épreuve à laquelle l'athlète a participé.
- `Event` : l'épreuve à laquelle l'athlète a participé. Par exemple, pour la natation (qui est le sport), il peut y avoir l'épreuve du 200 m brasse hommes, l'épreuve du 400 m brasse hommes, etc.
- `Medal` : la médaille obtenue par l'athlète pour cette épreuve. Les valeurs possibles sont `Gold` (or), `Silver` (argent), `Bronze` (bronze) ou `NA` (`NA` signifie *Not Available* et est couramment utilisée en informatique pour représenter des valeurs manquantes. Ici, cela correspond à "aucune médaille obtenue par l'athlète").

Remarque

Lors de la lecture du fichier, Pandas transformera ces valeurs NA, qu'il sait être des valeurs manquantes, en NaN. NaN signifie Not a Number, c'est la valeur

qu'utilise Python pour représenter des valeurs manquantes.

Ce jeu de données va non seulement nous permettre d'illustrer l'apprentissage de l'utilisation de la librairie Pandas, mais nous en profiterons aussi pour l'explorer et en apprendre plus sur les Jeux olympiques.

Par exemple, les questions auxquelles nous pourrons répondre sont les suivantes :

- Quel athlète a gagné le plus de médailles d'or aux Jeux olympiques ?
- Quel pays a gagné le plus de médailles depuis le début des Jeux olympiques ?
- Quelle est l'évolution de la présence des femmes aux Jeux olympiques au fil des années ?

Tant de questions auxquelles nous pourrons répondre au cours de ce chapitre.

Ci-dessous, voici un aperçu du tableau que nous allons explorer et manipuler, afin que vous ayez visuellement en tête la structure de ce fichier. Nous allons le voir en détail tout au long de ce chapitre.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
2	1	A Dijiang	M	24	180	80	China	CHN	1992 Sur	1992	Summer	Barcelona	Basketball	Basketb	NA
3	2	A Lamus	M	23	170	60	China	CHN	2012 Sur	2012	Summer	London	Judo	Judo Me	NA
4	3	Gunnar	M	24	NA	NA	Denmark	DEN	1920 Sur	1920	Summer	Antwerpen	Football	Football	NA
5	4	Edgar Lir	M	34	NA	NA	Denmark	DEN	1900 Sur	1900	Summer	Paris	Tug-Of-Wa	Tug-Of-\	Gold
6	5	Christine	F	21	185	82	Netherla	NED	1988 Wi	1988	Winter	Calgary	Speed Skati	Speed Sl	NA
7	5	Christine	F	21	185	82	Netherla	NED	1988 Wi	1988	Winter	Calgary	Speed Skati	Speed Sl	NA
8	5	Christine	F	25	185	82	Netherla	NED	1992 Wi	1992	Winter	Albertville	Speed Skati	Speed Sl	NA
9	5	Christine	F	25	185	82	Netherla	NED	1992 Wi	1992	Winter	Albertville	Speed Skati	Speed Sl	NA
10	5	Christine	F	27	185	82	Netherla	NED	1994 Wi	1994	Winter	Lillehamme	Speed Skati	Speed Sl	NA
11	5	Christine	F	27	185	82	Netherla	NED	1994 Wi	1994	Winter	Lillehamme	Speed Skati	Speed Sl	NA
12	6	Per Knut	M	31	188	75	United S	USA	1992 Wi	1992	Winter	Albertville	Cross Coun	Cross Cc	NA
13	6	Per Knut	M	31	188	75	United S	USA	1992 Wi	1992	Winter	Albertville	Cross Coun	Cross Cc	NA
14	6	Per Knut	M	31	188	75	United S	USA	1992 Wi	1992	Winter	Albertville	Cross Coun	Cross Cc	NA
15	6	Per Knut	M	31	188	75	United S	USA	1992 Wi	1992	Winter	Albertville	Cross Coun	Cross Cc	NA
16	6	Per Knut	M	33	188	75	United S	USA	1994 Wi	1994	Winter	Lillehamme	Cross Coun	Cross Cc	NA

© Editions ENI - All rights reserved

Remarque

Vous constatez que l'ensemble du jeu de données est en anglais. Cela sera majoritairement le cas lorsque vous aurez affaire à des données publiques, afin qu'elles soient compréhensibles par le monde entier. Si l'anglais n'est pas votre fort, ne vous inquiétez pas, chaque exemple utilisé dans ce chapitre sera traduit. Toutefois, il est conseillé d'avoir de bonnes bases d'anglais, car non seulement les jeux de données sont en anglais, mais la documentation des différentes méthodes et fonctions Python aussi, de même que les erreurs Python... L'anglais, au moins technique, est essentiel dans le domaine de l'analyse de données.

2. Lire et écrire des fichiers avec Pandas

Il est important que vous appreniez à lire des données à partir de fichiers avec Pandas, avant même de découvrir ses structures de données. En effet, pour la suite, il faut que vous puissiez importer le jeu de données concernant les Jeux olympiques dans votre notebook Jupyter.

2.1 Lecture de fichiers texte (CSV ou TXT)

Lorsqu'on souhaite utiliser Pandas pour analyser des données, la première étape est généralement d'importer les données dans son notebook. Les fichiers peuvent être de différents formats, mais le plus courant en *Data Science* est le format CSV. CSV signifie *comma-separated values*, traduisez : "valeurs séparées par une virgule". Il s'agit donc d'un format texte, dans lequel les colonnes sont séparées par des virgules. L'autre format très courant est le format TXT, qui lui aussi est un format texte, mais contenant des valeurs séparées par des tabulations, cette fois.

2.1.1 Lecture basique d'un fichier

Pour lire un fichier TXT avec Pandas, il faut utiliser la fonction `read_table()`, dont la syntaxe générale est la suivante :

```
import pandas as pd
pd.read_table("mon_fichier.txt")
```

Il est courant d'utiliser l'alias `pd` pour pandas, ce qui permet de réduire le code d'appel des fonctions appartenant à cette librairie.

La fonction `read_table()` possède un argument obligatoire, qui est le chemin complet vers le fichier ainsi que le nom du fichier, ou juste le nom du fichier si celui-ci se trouve dans votre dossier de travail, c'est-à-dire dans le même dossier que votre notebook. Dans cette syntaxe, le fichier `mon_fichier.txt` se trouve dans le même dossier que le notebook.

Pour lire un fichier CSV avec Pandas, il faut utiliser la fonction `read_csv()`, dont la syntaxe générale est la suivante :

```
import pandas as pd
pd.read_csv("mon_fichier.csv")
```

Ces deux fonctions sont les plus utilisées pour importer des données à partir d'un fichier. La seule différence entre ces deux fonctions est la valeur par défaut que prend l'option `sep`. En effet, l'option `sep`, qui signifie séparateur, permet de dire à Pandas quel est le séparateur du fichier qu'on souhaite lire. Dans la fonction `read_table()`, la valeur par défaut de `sep` est la tabulation (qu'on écrit `\t` en Python), donc cette fonction permet de lire un fichier TXT par défaut. Dans la fonction `read_csv()`, la valeur par défaut de `sep` est la virgule (`,`), donc elle permet de lire un fichier CSV par défaut.

Toutefois, cette valeur de `sep` peut être modifiée, et la fonction `read_table()`, par exemple, peut être utilisée pour lire un fichier CSV, comme suit :

```
import pandas as pd
pd.read_table("mon_fichier.csv", sep=",")
```

Ainsi, la fonction `read_table()` peut s'adapter à tout type de séparateur, que ce soit le point-virgule (`;`), l'espace (" "), etc., et il en va de même pour la fonction `read_csv()`. Il est simplement logique d'utiliser la fonction `read_table()` pour des fichiers TXT et la fonction `read_csv()` pour des fichiers CSV.

© Editions ENI - All rights reserved

À présent, importons notre jeu de données exemple dans notre notebook, avec la commande suivante :

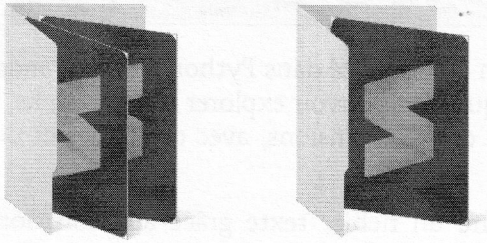
```
import pandas as pd
donnees=pd.read_csv("../datasets/120-years-of-olympic-history
-athletes-and-results.csv")
donnees.head()
```

Avec cette commande, nous importons la librairie Pandas, en lui donnant l'alias `pd` pour simplifier l'écriture. Ensuite, nous lisons le fichier de données avec la fonction `read_csv()`, et nous le stockons dans un dataframe nommé `donnees`.

Remarque

Nous verrons plus en détail dans la section suivante ce que sont les dataframes et les séries. Pour l'instant, retenez simplement qu'il s'agit d'un tableau à deux dimensions pour le premier et d'un tableau à une dimension pour le deuxième.

Comme expliqué précédemment, il faut donner le chemin vers le fichier à la fonction `read_csv()` pour qu'elle puisse le lire. Voici la structure de notre dossier de travail :



Dans le dossier `datasets` (qui signifie jeux de données, en français), il y a nos fichiers de données (dont notre fichier exemple ici) et dans le dossier `notebooks`, il y a l'ensemble de nos notebooks. Ainsi, lorsque nous travaillons sur notre notebook, nous nous trouvons dans le dossier `notebooks` par défaut. Donc, pour que Pandas puisse trouver notre fichier, on utilise les symboles `../` pour dire à Pandas de retourner dans le dossier au-dessus du dossier `notebooks`, puis d'entrer dans le dossier `datasets` et de lire le fichier 120-years-of_olympic-history-athletes-and-results.csv.

Vous pouvez créer la même structure, c'est-à-dire un dossier `datasets` où vous mettez le fichier CSV et un dossier `notebooks` où vous copiez-collez le notebook fourni en téléchargement de ce livre. Vous ouvrez ce notebook dans Jupyter Notebook et les commandes fournies dans ce chapitre fonctionneront correctement. Si vous ne souhaitez pas garder la même structure, n'oubliez pas de changer le chemin vers le fichier afin que l'importation de celui-ci fonctionne correctement.

Enfin, la méthode `head()` de Pandas, qui s'applique à une série ou à dataframe, permet d'avoir un aperçu des cinq premières lignes de notre objet afin de voir à quoi il ressemble. Vous utiliserez souvent cette méthode, car en analyse de données, nous travaillons avec de grands tableaux de données et il n'est pas conseillé ni utile de les afficher en entier.

Résultat de la commande précédente :

	ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
0	1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
1	2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
2	3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
3	4	Edgar Lindenau Aabye	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
4	5	Christine Jacoba Aaftink	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

On constate que notre fichier a bien été importé dans Python grâce à Pandas et qu'il ressemble à un joli tableau qu'on va pouvoir explorer ensemble : ici, il s'agit d'un dataframe, un tableau à deux dimensions, avec des lignes et des colonnes.

À présent, vous savez comment lire un fichier texte grâce aux fonctions `read_table()` et `read_csv()`. Toutefois, il y a quelques options intéressantes à connaître pour ces deux fonctions.

2.1.2 Gestion de l'en-tête

La première option importante est la notion de *header*, que l'on peut traduire par "en-tête" en français. Il s'agit plus simplement des noms de colonnes. Par défaut, Python prendra la première ligne du fichier comme en-tête du tableau, ce qui fonctionne bien dans le cas de notre jeu de données, où la première ligne correspond effectivement aux noms des variables.

© Editions ENI - All rights reserved

Toutefois, si votre tableau ne contient pas de *header*, et que vous ne voulez pas qu'il y en ait, vous pouvez le spécifier avec la syntaxe suivante :

```
import pandas as pd
pd.read_csv("mon_fichier.csv", header=None)
```

Ici, vos noms de colonnes correspondront à des valeurs partant de 0 et qui s'incrémentent de 1 en 1 (0, 1, 2, 3, etc.).

Si vous testez cette commande avec notre fichier, cela générera une erreur car ce fichier contient bien un header. Le fait de mettre l'option `header` à "None" fait que Pandas met la première ligne du fichier en tant que ligne du tableau, et non plus en tant qu'en-tête. Cela ne peut pas marcher, car pour les colonnes numériques comme l'âge par exemple, celle-ci se retrouvera avec une valeur de type caractère : nous le verrons plus tard, au sein d'une colonne d'un dataframe, il ne doit y avoir qu'un seul type de données.

C'est là que l'option `skiprows` devient intéressante, car elle permet d'ignorer certaines lignes lors de la lecture du fichier et elle ne les stocke pas dans le tableau. Par exemple, si on veut supprimer l'en-tête du fichier lors de la lecture, on donnera la valeur "1" à l'option `skiprows` pour ignorer la première ligne du fichier. Voici la syntaxe :

```
import pandas as pd
pd.read_csv("mon_fichier.csv", skiprows=1, header=None)
```

Si votre fichier ne possède pas d'en-tête, mais que vous voulez en créer un lors de la lecture du fichier, vous pouvez le faire avec l'option `names`. La syntaxe est la suivante :

```
import pandas as pd
pd.read_csv("mon_fichier.csv", names=['colonne_1','colonne_2'])
```

Ici, vous n'avez pas besoin de spécifier que votre fichier ne contient pas de header avec l'option `header=None`, car comme vous passez l'option `names` à la fonction `read_csv()`, Pandas comprend qu'il ne doit pas prendre le *header* dans votre fichier, mais qu'il doit le définir avec la liste que vous lui donnez dans l'option `names`.

Passons à l'exemple pratique pour illustrer ces trois nouvelles options. Imaginons que nous souhaitions supprimer l'en-tête en anglais de notre fichier et que nous voulions en mettre un nouveau, en français.

Voici le code permettant de faire cela :

```
import pandas as pd
donnees_francais=pd.read_csv("../datasets/120-years-of-olympic-history-athletes-and-results.csv",skiprows=1, names=["Id","Nom", "Sexe", "Age", "Taille", "Poids", "Equipe", "CNO", "Jeux", "Annee", "Saison", "Ville", "Sport", "Epreuve", "Medaille"])
donnees_francais.head()
```

Résultat

	Id	Nom	Sexe	Age	Taille	Poids	Equipe	CNO	Jeux	Annee	Saison	Ville	Sport	Epreuve	Medaille
0	1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
1	2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
2	3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
3	4	Edgar Lindenau Aabye	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
4	5	Christine Jacoba Aaftink	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

Ici, nous avons ignoré la première ligne, c'est-à-dire l'en-tête du fichier, avec l'option `skiprows=1`. Puis nous avons défini un nouvel en-tête grâce à l'option `names`, en lui donnant une liste Python contenant le même nombre d'éléments que le nombre de colonnes dans le tableau. Nous obtenons ainsi un tableau avec un en-tête en français.

2.1.3 Gestion des index

Nous le verrons à partir de la section suivante, Pandas propose deux structures de données principales, une correspondant à un tableau à une dimension (une colonne) et une correspondant à un tableau à deux dimensions (avec des lignes et des colonnes). Tout comme avec NumPy, ces structures de données possèdent des index, permettant de donner un nom aux colonnes (et aux lignes dans le cas d'un tableau à deux dimensions). Ces index permettent aussi d'accéder aux différentes données que les structures contiennent.

© Editions ENI - All rights reserved

Il y a différentes manières de gérer les index lors de la lecture d'un fichier. Celui-ci peut être implicite si on a une entrée en moins dans le *header* par rapport au nombre de colonnes, c'est-à-dire si le nom de la première colonne est vide. Dans ce cas, la fonction `read_csv()` (ou `read_table()`) prend par défaut la première colonne comme étant l'index.

Il est aussi possible de spécifier quelle colonne du jeu de données correspond à l'index, grâce à l'option `index_col`.

Syntaxe d'utilisation de l'option `index_col`

```
import pandas as pd
pd.read_csv("mon_fichier.csv", index_col=0)
```

Ici, Pandas définit la première colonne du fichier comme étant l'index du tableau.

Remarque

Attention, n'oubliez pas qu'en Python, on compte à partir de 0, 0 correspondant à la première valeur.

Enfin, il est possible de définir plusieurs colonnes comme étant les index, toujours grâce à l'option `index_col`.

Syntaxe

```
import pandas as pd
pd.read_csv("mon_fichier.csv", index_col=[0,1])
```

Ici, la première et la deuxième colonne deviennent les index du tableau après lecture du fichier.

Passons à l'exemple pratique pour illustrer les index. Lorsqu'on regarde notre tableau `donnees`, on constate qu'il y a une colonne `ID` qui correspond à l'identifiant de chaque athlète, on pourrait vouloir le définir comme index.

Pour cela, on peut utiliser le code suivant :

```
import pandas as pd
donnees_1index=pd.read_csv("../datasets/120-years-of-olympic-
history-athletes-and-results.csv", index_col=[0])
donnees_1index.head()
```

Résultat

ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
4	Edgar Lindenau Aabye	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
5	Christine Jacoba Aaftink	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

Ici, on constate que la colonne `ID` du fichier est devenue l'index du tableau. Avec Pandas et ses structures, il est possible d'avoir des index qui ne sont pas uniques. Ainsi, si un athlète est présent plusieurs fois dans le fichier, son ID sera lui aussi présent plusieurs fois dans l'index. Et cela permet de faire des traitements particuliers avec Pandas sur les index, ce que nous verrons plus tard. Nous verrons aussi comment utiliser les multi-index et leur utilité.

2.1.4 Création d'un tableau à une dimension à partir du fichier

Comme nous le disions, il existe deux structures de données que nous verrons juste après, correspondant à un tableau à une dimension (`Series`) ou un tableau à deux dimensions (`DataFrame`). Les fonctions de lecture de fichiers de Pandas créeront par défaut un dataframe, même à partir d'un fichier ne contenant qu'une seule colonne, et qui pourrait donc être stockée comme une série. Si on souhaite créer un tableau à une dimension plutôt que deux, il faudra spécifier l'option `squeeze`.

Syntaxe

```
import pandas as pd
pd.read_csv("fichier.csv", squeeze=True)
```

2.1.5 Filtrage des colonnes lors de la lecture du fichier

Il est possible, dès la lecture du fichier, de ne lire que certaines colonnes du tableau si toutes ne nous intéressent pas, grâce à l'option `usecols`.

Syntaxe

```
import pandas as pd
pd.read_csv("mon_fichier.csv", usecols=[3,6])
```

© Editions ENI - All rights reserved

Ici, on demande de n'importer que les colonnes 4 et 7 du fichier.

Il est aussi possible de sélectionner des colonnes en donnant une liste de noms de colonnes plutôt qu'une liste de positions de colonnes.

```
import pandas as pd
pd.read_csv("mon_fichier.csv",usecols=["macolonne_1",macolonne_2"])
```

Passons à l'exemple pratique. Si on ne souhaite sélectionner que les colonnes contenant les noms des athlètes, leur poids et leur taille, le code sera le suivant.

Code

```
import pandas as pd
donnees_usecols_position=pd.read_csv("../datasets/120-years-of-
olympic-history-athletes-and-results.csv", usecols=[1,4,5])
donnees_usecols_position.head()
```

Dans le code précédent, nous sélectionnons les colonnes grâce à leurs positions. Dans le code suivant, sélectionnons les colonnes par leurs noms plutôt que leurs positions.

Code

```
import pandas as pd
donnees_usecols =pd.read_csv("../datasets/120-years-of-olympic-
history-athletes-and-results.csv",
usecols=["Name","Height","Weight"])
donnees_usecols.head()
```

Dans les deux cas, les premières lignes de la sortie sont les suivantes.

Résultat

	Name	Height	Weight
0	A Dijiang	180.0	80.0
1	A Lamusi	170.0	60.0
2	Gunnar Nielsen Aaby	NaN	NaN
3	Edgar Lindenau Aabye	NaN	NaN
4	Christine Jacoba Aaftink	185.0	82.0

Nous avons bien sélectionné les trois colonnes qui nous intéressent directement lors de la lecture du fichier.

2.1.6 Les types des différentes colonnes

Il est possible de spécifier le type de chaque colonne afin que Pandas prenne en compte les différents traitements possibles qu'il peut faire sur chaque colonne. En effet, on ne peut pas faire les mêmes traitements sur une colonne de type numérique que sur une colonne de type caractère. Par exemple, on peut multiplier une colonne de type numérique, mais pas une colonne de type caractère.

La syntaxe générale pour spécifier le type des colonnes est la suivante :

```
import pandas as pd
pd.read_csv("mon_fichier.csv",dtype={"macolonne_1":"Int64",
"macolonne_2":"float64", "macolonne_3":"object"})
```

Ici, il suffit de donner un dictionnaire avec en clé les noms de colonnes et en valeurs le `dtype` de cette colonne.

2.1.7 Gestion des dates lors de la lecture du fichier

Enfin, certaines colonnes peuvent contenir des dates et il est important de les *parser* correctement. *Parser* pourrait être traduit par "analyser la syntaxe d'un texte". Pandas va analyser le format de la date et la formater correctement.

Lors de la lecture du fichier, on peut spécifier quelles colonnes traiter comme des dates. Par exemple, si la colonne 5 de notre fichier contient des dates, on le dit lors de la lecture.

Syntaxe

```
import pandas as pd
pd.read_csv("mon_fichier.csv",parse_dates=[4])
```

Lorsque nous visualiserons notre tableau, nous pourrons constater que le `dtype` de notre colonne est `datetime64`.

© Editions ENI - All rights reserved

Passons à l'exemple pratique. On sait que la colonne 9 de notre dataframe correspond à l'année des Jeux olympiques auxquels ont participé les athlètes. On pourrait vouloir dire à Pandas de considérer cette colonne comme une date.

Pour l'instant, si on regarde le type de la colonne 9, on a le résultat suivant.

Code

```
donnees.dtypes
```

Résultat

```
Year        int64
```

Ici, notre colonne représentant l'année des JO est considérée comme un type entier par Pandas. En précisant que la colonne 9 doit être considérée comme une date, on obtient le résultat suivant.

Code

```
import pandas as pd
donnees_dates=pd.read_csv("../datasets/120-years-of-olympic-
history-athletes-and-results.csv", parse_dates=[9])
donnees_dates.dtypes
```

Résultat

```
Year        datetime64[ns]
```

La colonne 9 est bien de type `datetime64` maintenant.

2.2 Lecture de fichiers Excel

Pour lire un fichier Excel, c'est-à-dire un fichier au format XLSX ou XLS, il suffit d'utiliser la fonction `read_excel()` de Pandas.

Syntaxe

```
import pandas as pd
fichier_excel=pd.read_excel('mon_fichier.xlsx')
```

Par défaut, la fonction `read_excel()` lit la première feuille Excel du fichier qu'on lui donne. Si on souhaite importer une autre feuille, il faut donner le nom de la feuille à l'option `sheet_name`, qui signifie 'nom de la feuille', en français.

Syntaxe

```
import pandas as pd
fichier_excel=pd.read_excel('mon_fichier.xlsx',
sheet_name="Nom_de_la_feuille")
```

À chaque fois, la structure de données créée sera un dataframe, contenant les données de la feuille Excel qui nous intéresse. Les autres options de cette fonction sont les mêmes que celles vues précédemment avec les fonctions `read_table()` et `read_csv()`, comme les options `header`, `names`, `index_col`, `usecols`, `squeeze`, ou encore `dtype`.

Passons à l'exemple pratique. En téléchargement avec ce livre, vous pouvez récupérer le fichier "120-years-of-olympic-history-athletes-and-results.xlsx", si vous souhaitez effectuer vous-mêmes ces commandes. Ce fichier contient une première feuille avec les données de JO et une deuxième feuille factice, pour vous montrer comment lire un fichier contenant plusieurs feuilles Excel.

Pour lire la première feuille du fichier Excel, le code est le suivant :

```
import pandas as pd
donnees_excel=pd.read_excel('../datasets/120-years-of-olympic-
history-athletes-and-results.xlsx')
donnees_excel.head()
```

Ces commandes permettent de lire la première feuille du fichier XLS, par défaut, de la stocker dans `donnees_excel` et d'en afficher les cinq premières lignes.

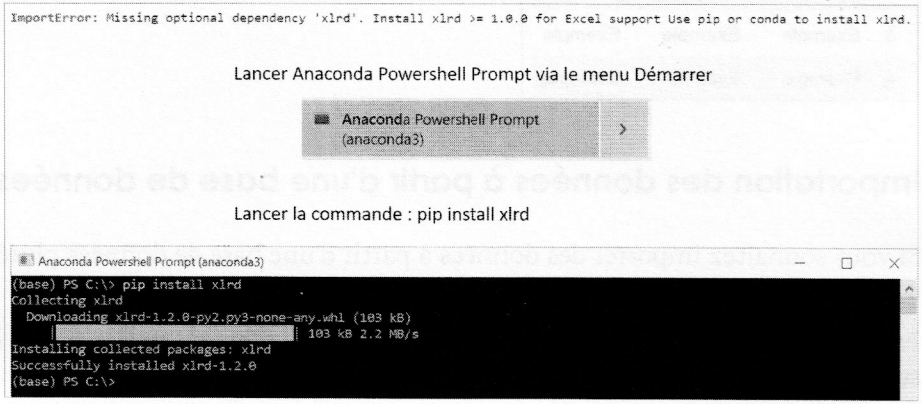
Résultat

ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN

© Editions ENI - All rights reserved

Attention, si vous rencontrez une erreur lors de cette commande, dont la dernière ligne est : `ImportError : Missing optional dependency 'xlrd'` ..., veuillez procéder de la manière suivante :

Lancez le logiciel **Anaconda Powershell Prompt** puis tapez la commande `pip install xlrd` pour installer le package xlrd, qui réglera ce problème.



Si on souhaite lire la deuxième feuille de ce fichier, on utilise l'option `sheet_name` avec le nom de la feuille qui nous intéresse.

Code

```
import pandas as pd
donnees_excel=pd.read_excel('../datasets/120-years-of-olympic-
history-athletes-and-results.xlsx', sheet_name='Feuille2')
donnees_excel.head()
```

Ce code stockera les données de la deuxième feuille Excel dans le dataframe : `donnees_excel`.

Résultat

	Exemple	Exemple.1	Exemple.2
0	Exemple	Exemple	Exemple
1	Exemple	Exemple	Exemple
2	Exemple	Exemple	Exemple
3	Exemple	Exemple	Exemple
4	Exemple	Exemple	Exemple

2.3 Importation des données à partir d'une base de données

Si vous souhaitez importer des données à partir d'une base de données plutôt que d'un fichier plat comme les formats vus précédemment, c'est possible.

Par exemple, pour une base de données SQLite, la syntaxe générale sera la suivante :

```
import pandas as pd
import sqlite3
connexion = sqlite3.connect('base_de_donnees.db')
requete = "SELECT * FROM TABLES;"
resultats = pd.read_sql(requete, con=connexion)
resultats.head()
```

Ici, on importe la librairie `Pandas` ainsi que la librairie `sqlite3`. On crée la connexion à notre fichier contenant la base de données au format `.db` grâce à la fonction `connect()` de sqlite3. Puis on écrit la requête qui nous permettra de récupérer les données qui nous intéressent dans la base de données qu'on range dans la variable `requete`. Enfin, on lance la requête sur la base de données grâce à la fonction `read_sql()` de Pandas, à laquelle on donne la requête, ainsi que la connexion créée précédemment. De cette manière, on récupère les résultats de la requête sous forme d'un dataframe.

© Editions ENI - All rights reserved

Exemple pratique

Remarque

Veuillez récupérer le fichier chinook.db disponible en téléchargement avec ce livre, si vous souhaitez effectuer vous-même ces commandes. Il s'agit d'une base de données test SQLite, disponible à l'adresse suivante : http://www.sqlitetutorial.net/sqlite-sample-database/

Nous ne rentrerons pas dans le détail de cette base de données d'exemple, car le but ici est simplement de vous expliquer comment vous connecter à une base de données et y récupérer les informations qui vous intéressent. Une fois ces données récupérées et stockées dans un dataframe, les manipulations que vous pourrez effectuer dessus seront les mêmes que celles que nous verrons par la suite avec le fichier "120-years-of-olympic-history-athletes-and-results.csv".

Pour l'exemple, sachez que cette base de données contient onze tables, dont une table contenant les informations au sujet des employés d'une entreprise. Imaginons que nous ayons envie de récupérer les informations de l'ensemble des employés de la base de données afin de les stocker dans un dataframe pour les analyser, le code serait le suivant.

Code

```
import pandas as pd
import sqlite3
connexion = sqlite3.connect('../datasets/chinook.db')
requete = "SELECT * FROM employees;"
resultats = pd.read_sql(requete, con=connexion)
resultats.head()
```

Ici, on se connecte à la base de données stockée dans le fichier `chinook.db`, lui-même stocké dans le dossier `datasets`. Puis on crée une requête pour sélectionner l'ensemble des données de la table nommée `employees`. Enfin, on lance cette requête sur la base de données et on stocke les données extraites dans un dataframe nommé `resultats`.

Résultat

	EmployeeId	LastName	FirstName	Title	ReportsTo	BirthDate	HireDate	Address	City	State	Country	PostalCode	Phone	Fax
0	1	Adams	Andrew	General Manager	NaN	1962-02-18 00:00:00	2002-08-14 00:00:00	11120 Jasper Ave NW	Edmonton	AB	Canada	T5K 2N1	+1 (780) 428-9482	+1 (780) 428-3457
1	2	Edwards	Nancy	Sales Manager	1.0	1958-12-08 00:00:00	2002-05-01 00:00:00	825 8 Ave SW	Calgary	AB	Canada	T2P 2T3	+1 (403) 262-3443	+1 (403) 262-3322
2	3	Peacock	Jane	Sales Support Agent	2.0	1973-08-29 00:00:00	2002-04-01 00:00:00	1111 6 Ave SW	Calgary	AB	Canada	T2P 5M5	+1 (403) 262-3443	+1 (403) 262-6712
3	4	Park	Margaret	Sales Support Agent	2.0	1947-09-19 00:00:00	2003-05-03 00:00:00	683 10 Street SW	Calgary	AB	Canada	T2P 5G3	+1 (403) 263-4423	+1 (403) 263-4289
4	5	Johnson	Steve	Sales Support Agent	2.0	1965-03-03 00:00:00	2003-10-17 00:00:00	7727B 41 Ave	Calgary	AB	Canada	T3B 1Y7	1 (780) 836-9987	1 (780) 836-9543

On a bien importé des données à partir d'une base de données SQLite.

2.4 Lecture de fichiers au format JSON

Le dernier format que nous verrons pour l'importation de données est le format JSON. Ce format est devenu très populaire ces dernières années et vous risquez de le rencontrer. Il est très utilisé pour échanger des données de manière structurée et légère. Ainsi, voyons à présent ensemble comment lire un fichier JSON. Pour cela, nous avons récupéré la table `employees` de la base de données précédente, que nous avons stockée au format JSON, pour l'exemple.

Remarque

Le fichier employees.json est disponible en téléchargement avec ce livre.

© Editions ENI - All rights reserved

Le format JSON est un format texte, tout comme le TXT, mais ses champs sont organisés d'une manière différente. Voici un aperçu du fichier :

Pour expliquer brièvement ce format, l'ensemble des données sont stockées entre crochets `[]`. Puis, chaque individu ou observation est stocké entre accolades `{}`, ces accolades pouvant être comparées aux lignes d'un tableau classique. Enfin, chaque accolade contient l'ensemble des informations (des colonnes) de l'individu/observation sous forme de clés-valeurs séparées par des virgules. Chaque couple clé-valeur représente une colonne de la ligne.

La librairie Pandas propose une fonction appelée `read_json()` qui permet de lire un fichier au format JSON.

Syntaxe

```
import pandas as pd
pd.read_json("mon_fichier.json")
```

Appliquons cette fonction à un exemple concret, en important le contenu du fichier `employees.json`.

Code

```
import pandas as pd
pd.read_json("../datasets/employees.json")
```

On obtient exactement le même résultat que lors de la requête sur la base de données stockée dans le fichier chinook.db.

2.5 Écriture de fichiers ou exportation de données

Lorsque vos analyses sont terminées, il est possible que vous souhaitiez exporter votre dataframe de résultats afin de le stocker dans un fichier CSV, TXT ou encore XLSX. Les méthodes à utiliser se ressemblent beaucoup.

Pour exporter un dataframe dans un fichier CSV, la syntaxe est la suivante :

```
mon_dataframe.to_csv("mes_resultats.csv")
```

Ici, on utilise la méthode `to_csv()` sur l'objet `mon_dataframe` qui est notre dataframe de résultat, et on l'écrit dans le fichier mes_resultats.csv.

Diverses options existent pour cette méthode, dont les principales sont les suivantes :

- `sep` : si on veut spécifier le séparateur ; par défaut, ce sera la virgule, étant donné qu'on utilise la méthode `to_csv()`.

Syntaxe

```
mon_dataframe.to_csv("mes_resultats.csv", sep=",")
```

© Editions ENI - All rights reserved

- `columns` : une liste avec les colonnes qu'on souhaite écrire dans le fichier, si on ne les veut pas toutes.

Syntaxe

```
mon_dataframe.to_csv("mes_resultats.csv", columns=["ma_colonne1",
"ma_colonne3"])
```

- `header` : si on souhaite écrire l'en-tête du dataframe dans le fichier ; par défaut, ce sera le cas.
- `Index` : par défaut, cette option est à `True` (vrai), ce qui spécifie à Pandas d'écrire les labels de lignes (les index de lignes) dans le fichier. À `False`, il ne le fait pas.

Pour écrire le dataframe dans un fichier au format TXT, on pourra là aussi utiliser la méthode `to_csv()`, mais en précisant le séparateur comme étant la tabulation.

Syntaxe

```
mon_dataframe.to_csv("mes_resultats.txt", sep="\t")
```

Enfin, pour écrire votre dataframe dans un fichier Excel, la syntaxe est la suivante :

```
mon_dataframe.to_excel("mes_resultats.xlsx")
```

On peut utiliser l'option `sheet_name` pour définir le nom de la feuille qui contiendra le dataframe.

Syntaxe

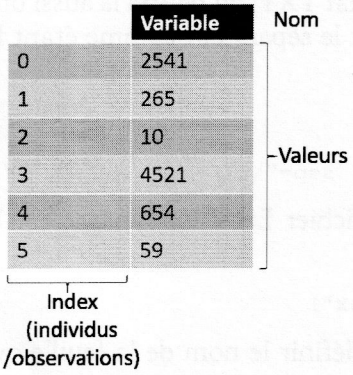
```
mon_dataframe.to_excel("mes_resultats.xlsx",
sheet_name="mes_resultats")
```

Les autres options de la méthode `to_excel()` sont assez similaires à celles de la méthode `to_csv()`. N'hésitez pas à consulter la documentation de la méthode pour prendre connaissance de toutes les options possibles.

3. Structure de données Pandas : les Series (Séries)

3.1 Introduction

On peut décrire les séries de Pandas comme étant une colonne d'un dataframe, un tableau à une dimension (vecteur) ou encore une suite de valeurs, numériques ou non. On peut aussi décrire une série comme un ensemble de valeurs représentant une variable pour un ensemble d'observations ou d'individus. Une série a un nom : par exemple dans l'image ci-dessous, le nom de la série est "Variable". Tout comme les autres structures Python que nous avons déjà vues, une série possède aussi des index.



Une série peut être créée à partir d'une liste, d'un tableau NumPy, ou encore d'un dictionnaire, grâce à la méthode `Series()`. Elle peut aussi être créée à partir d'un fichier grâce aux fonctions de lecture de fichiers que nous avons vues précédemment, telles que `read_table()` ou `read_csv()`.

La structure de données `Series` possède deux composants principaux :

- les valeurs, auxquelles on peut accéder avec la méthode `values()`,
- les étiquettes, plus couramment appelées les index, auxquelles on peut accéder avec la méthode `index()`.

© Editions ENI - All rights reserved

La série possède un seul axe, il s'agit de l'axe des index, contrairement au dataframe, que nous verrons plus tard, qui contient deux axes : les lignes et les colonnes.

Il faut savoir que pour les séries, par défaut, les index sont des nombres entiers : ils commencent à 0 et s'incrémentent, comme les `ndarrays` de NumPy. Mais il est aussi possible de remplacer ces index entiers par des étiquettes du type que l'on souhaite (nombres, caractères). Par exemple, si on préfère mettre des lettres comme index plutôt que des nombres, Pandas nous autorise à le faire. Ainsi, on pourra accéder aux valeurs de la série soit en donnant la position de la valeur entre crochets, soit en donnant l'étiquette correspondant à cette valeur, toujours entre crochets.

Enfin, au sein d'une série, il est fortement conseillé d'utiliser des données qui sont homogènes, de même type (uniquement des valeurs numériques, uniquement des valeurs de type caractères, etc.), même si le stockage de valeurs de types hétérogènes est permis.

3.2 Créer des séries

Pour pouvoir créer sa première série, il faut d'abord importer la librairie Pandas, puis utiliser la fonction `Series()`.

Voici la syntaxe générale :

```
import pandas as pd
serie=pd.Series(donnees)
```

On peut créer et remplir cette série de différentes manières.

3.2.1 À partir de valeurs aléatoires

Tout d'abord, il est possible de créer une série à partir de valeurs aléatoires, déterminées grâce à `choice()` du module `random` de NumPy : `random.choice()` (choix aléatoire).

La syntaxe d'utilisation de la fonction `random.choice()` est la suivante :

```
import numpy as np
np.random.choice(data, size)
```

On importe la librairie NumPy et on crée un alias `np` afin de simplifier l'écriture de NumPy et donc l'appel de ses fonctions. Puis on appelle la fonction `random.choice()` contenue dans la librairie NumPy. Cette fonction prend en premier argument une structure contenant des données, par exemple une liste (`data`). Puis on spécifie le nombre de valeurs qu'on veut piocher aléatoirement dans cet objet `data` avec l'argument `size` (taille). Enfin, il suffit de donner la sortie de cette fonction à la fonction `Series()` de Pandas pour créer une série contenant ces valeurs aléatoires.

Syntaxe de création de la série

```
import numpy as np
import pandas as pd
pd.Series(np.random.choice(data, size))
```

Passons à l'exemple pratique, avec le code ci-dessous :

```
import numpy as np
import pandas as pd
mes_nombres = pd.Series(np.random.choice(list(range(0,500)),10))
print(mes_nombres)
```

Ici, on importe les librairies NumPy et Pandas. Ensuite, on crée une liste contenant 500 valeurs, de 0 à 499 : il s'agit de l'objet `data` dans la syntaxe d'exemple. Enfin, on sélectionne dix valeurs aléatoirement dans cette liste grâce à la fonction `random.choice()`. En utilisant la fonction `Series()` de Pandas, on obtient un objet de classe `Series`, contenant dix valeurs aléatoires comprises entre 0 et 499, rangées dans la variable `mes_nombres`. Pour la visualiser, il suffit d'utiliser la fonction `print()` de Python.

Résultat

```
0    213
1    259
2    478
3    351
4    361
5    251
6     82
7     21
8    124
9    234
dtype: int32
```

© Editions ENI - All rights reserved

Remarque

Si vous lancez ce code dans votre propre notebook, sachez qu'il est normal que vous n'obteniez pas les mêmes valeurs dans votre série. En effet, ce sont des valeurs aléatoires.

Dans le résultat du code, nous pouvons voir qu'il y a deux "colonnes", ce sont les deux composants principaux des séries dont nous parlions précédemment : à gauche se trouvent les index et à droite les valeurs. Dans cet exemple, les index vont de 0 à 9, puisque par défaut, ceux-ci commencent à 0 puis s'incrémentent autant de fois qu'il y a de valeurs. En bas de la série, on peut apercevoir une information supplémentaire, le `dtype`, qui représente le type de données stockées dans la série ; ici, le type est `int32` (nombres entiers).

3.2.2 À partir d'une liste Python

À partir d'une liste Python, la syntaxe pour créer une série est la suivante :

```
pd.Series(liste)
```

Voici un code exemple

```
import pandas as pd
ma_serie = pd.Series([25,45,62,12,11])
print(ma_serie)
```

Avec ce code, nous venons de créer une série grâce à une liste Python, contenant une suite de valeurs numériques. Cette liste a été créée grâce aux crochets `[]`. La fonction `Series()` de Pandas prend en entrée cette liste Python, qui contient une suite de nombres, et crée un objet de type `Series` qui est stocké dans la variable `ma_serie`.

Résultat

```
0    25
1    45
2    62
3    12
4    11
dtype: int64
```

Remarque

Pour la première création de série, Python avait choisi `int32` comme type de données, puis lors de la création de la deuxième série, il a choisi `int64`. C'est un choix qu'il juge être le plus judicieux selon la manière dont nous créons les séries numériques. La différence entre `int32` et `int64` est leur capacité de stockage. Par exemple, `int32` permet de stocker des valeurs comprises entre -2 147 483 648 et +2 147 483 647 et `int64` peut stocker des valeurs comprises entre -9 223 372 036 854 775 808 et +9 223 372 036 854 775 807.

Attention toutefois lorsque vous créez des séries à partir de listes, car celles-ci acceptent des données hétérogènes, ce qui n'est pas conseillé dans le cas des séries. Voyons ensemble comment réagit Python lorsqu'on essaye de créer une série à partir d'une liste contenant des valeurs de types hétérogènes.

Code

```
ma_liste = ["hello", 1,2,3]
print(ma_liste)
```

Résultat

```
['hello', 1, 2, 3]
```

Ici, nous créons une liste contenant la chaîne de caractères `'hello'` ainsi que des chiffres. On voit que la liste contient bien une valeur de type caractère et trois valeurs entières, puisqu'elle accepte les données de types hétérogènes.

Les séries, elles, peuvent accepter des données hétérogènes, mais cela n'est pas conseillé en matière de performance. Ainsi, si on crée une série à partir de cette liste, cela donne :

Code

```
import pandas as pd
print(pd.Series(ma_liste))
```

Résultat

```
0    hello
1        1
2        2
3        3
dtype: object
```

© Editions ENI - All rights reserved

Le `dtype` ici est `object` : ce `dtype` correspond soit au type `str` de Python, pour *string* (chaîne de caractères), soit à des types mixtes (c'est-à-dire un mélange de chaînes de caractères, nombres, etc.). Ainsi, ici, on a des données hétérogènes dans la série créée, les séries les gèrent.

Pour voir les différents types que la série contient, on peut utiliser la fonction `type()` de Python.

Code

```
series_mixed=pd.Series(ma_liste)
print(type(series_mixed[0]))
print(type(series_mixed[1]))
```

Résultat

```
<class 'str'>
<class 'int'>
```

On constate que `'Hello'`, la valeur à l'index 0 de l'objet `series_mixed`, est bien considérée comme une valeur de type `str` et que la valeur `1` est bien considérée comme une valeur de type `int` par Python.

Toutefois, il n'est pas conseillé d'utiliser des données hétérogènes dans les séries et il n'y a pas d'intérêt à avoir de données mixtes dans une variable (une colonne).

Remarque

Avec Pandas, les différents types sont les suivants :

`object` = chaines de caractères, texte (`str` en Python)

`int64/int32` = valeurs entières numériques (`int` en Python)

`float64` = valeurs réelles, à virgules (`float` en Python)

`bool` = booléens, valeurs VRAIE ou FAUX (`bool` en Python)

3.2.3 À partir d'un tableau NumPy (ndarray)

Il est aussi possible de créer une série à partir d'un tableau NumPy (`ndarray`).

Syntaxe

```
import pandas as pd
series = pd.Series(numpy_array)
```

Cela ne change pas beaucoup de la création de la série à partir d'une liste Python.

3.2.4 À partir d'un fichier texte

Si vous avez lu la section de ce chapitre sur la lecture et l'écriture de fichiers avec Pandas, ces commandes ne vous seront pas inconnues. Nous allons ici lire une colonne spécifique de notre jeu de données sur les Jeux olympiques, qui nous servira d'exemple pour illustrer nos propos sur les séries.

Pour rappel, l'option `squeeze` permet de dire à Pandas que l'on souhaite créer un objet de classe `Series`, lors de la lecture du fichier, et non pas un dataframe.

Nous allons sélectionner la colonne `Weight` de notre tableau, qui correspond au poids des athlètes.

Code

```
import pandas as pd
ma_serie_de_poids_index_defaut=pd.read_csv("../datasets/120-
years-of-olympic-history-athletes-and-results.csv", usecols=[5],
squeeze=True)
print(ma_serie_de_poids_index_defaut)
```

Résultat

```
0          80.0
1          60.0
2           NaN
3           NaN
4          82.0
           ...
271111     89.0
271112     59.0
```

© Editions ENI - All rights reserved

```
271113    59.0
271114    96.0
271115    96.0
Name: Weight, Length: 271116, dtype: float64
```

Nous venons de créer une série, dont les index vont de 0 à 271115. Le nom de cette série (`Name`) est `Weight`. La taille de la série, c'est-à-dire le nombre de valeurs qu'elle contient, est de 271116. Enfin, le type des données (`dtype`) de cette série est `float64`, c'est-à-dire qu'il s'agit de nombres réels. Certaines valeurs sont égales à `NaN`, ce sont les valeurs manquantes en Python et cela signifie qu'en créant ce jeu de données sur les athlètes des JO, le poids de certains d'entre eux n'a pas pu être renseigné. Nous verrons plus en détail les valeurs manquantes au cours de ce chapitre.

3.3 Choisir l'index d'une série

Lorsque nous avons créé notre série, appelée `ma_serie_de_poids_index_defaut`, dans la section précédente, nous avons laissé les index par défaut. Ceux-ci commencent à 0 et s'incrémentent. Avec Pandas, il est possible de définir ses propres index, par exemple lors de la lecture d'un fichier.

Syntaxe pour définir soi-même les index

```
serie = pd.read_csv("mon_fichier.csv", index_col=[colonne1]
squeeze=True, usecols=[colonne1,colonne2])
```

Avec cette syntaxe, on définit quelle colonne du fichier utiliser pour créer la série. Précédemment, on sélectionnait une seule colonne, qui permettait de définir les valeurs de la série. Ici, le principe est de sélectionner deux colonnes avec l'option `usecols`, puis de définir l'une d'elles comme étant l'index de la série avec l'option `index_col`.

Illustrons cela avec un exemple. Nous allons créer une série avec la colonne `Weight` du fichier des Jeux olympiques comme valeurs et la colonne `Name` du fichier comme index.

Code

```
import pandas as pd
ma_serie_de_poids=pd.read_csv("../datasets/120-years-of-olympic-
history-athletes-and-results.csv", usecols=[1,5], squeeze=True,
index_col=[0])
ma_serie_de_poids
```

Dans ce code, on lit le fichier, on demande à Pandas de sélectionner les colonnes 1 et 5, respectivement `Name` et `Weight`, avec l'option `usecols`. On spécifie aussi à Pandas qu'on souhaite que l'objet créé soit de classe `Series`, grâce à l'option `squeeze`, et on lui définit que la première colonne de cet objet (la colonne `Name` ici) correspondra aux index, avec l'option `index_col`.

Résultat

```
Name
A Dijiang                  80.0
A Lamusi                   60.0
Gunnar Nielsen Aaby        NaN
Edgar Lindenau Aabye       NaN
Christine Jacoba Aaftink   82.0
                           ...
Andrzej ya                 89.0
Piotr ya                   59.0
Piotr ya                   59.0
Tomasz Ireneusz ya         96.0
Tomasz Ireneusz ya         96.0
Name: Weight, Length: 271116, dtype: float64
```

On constate qu'à présent, les index correspondent aux noms des athlètes. Lorsqu'on souhaite accéder à certaines valeurs d'une série, il est parfois plus pratique de le faire avec des noms d'index qui correspondent à quelque chose (par exemple des noms d'athlètes) qu'avec des chiffres incrémentés. C'est dans ce cas précis qu'on voit la puissance des séries, avec la possibilité d'ajouter l'étiquette souhaitée à nos valeurs.

© Editions ENI - All rights reserved

3.4 Accéder aux valeurs d'une série

3.4.1 Indexing via la position des valeurs

Lorsqu'on souhaite sélectionner une ou des valeurs dans une série, on va chercher à sélectionner des données à des positions particulières. Dans la documentation anglaise, vous verrez sûrement le terme *indexing*. Avec Pandas, et Python en général, *indexing* pourrait être traduit par "vouloir accéder à un sous-ensemble de données de notre structure, dans notre cas, de notre série". Nous avons déjà vu ce principe d'indexing ensemble avec les listes ou encore les `ndarrays` de NumPy.

Le premier type d'indexing est l'indexing via la position des valeurs dans la série.

Par exemple, si on souhaite récupérer la valeur en position 2 dans l'objet `ma_serie`, la syntaxe est :

```
ma_serie[1]
```

En se rappelant toujours que les positions en Python commencent à 0, donc 1 correspond à la valeur en position 2.

Pour sélectionner plusieurs valeurs, il suffit de donner une liste de positions, d'index, entre crochets, avec la syntaxe suivante :

```
ma_serie[[1,5,9]]
```

Ici, le premier couple de crochets `[1,5,9]` permet de créer la liste de valeurs et le deuxième couple de crochets `[ ]` permet d'effectuer l'indexing sur l'objet `ma_serie`.

Il est aussi possible d'utiliser des valeurs négatives. Par exemple, `-1` signifie "la dernière valeur de la série", `-2` signifie "l'avant-dernière valeur de la série", etc.

Syntaxe

```
ma_serie[-1]
```

Passons à l'exemple pratique. Pour récupérer plusieurs valeurs à plusieurs positions, il suffit de donner une liste de ces positions entre crochets. Récupérons les valeurs de poids des athlètes aux positions 0, 15, 6985 et 452.

Code

```
ma_serie_de_poids_index_defaut[[0,15,6985,452]]
```

Le premier couple de crochets `ma_serie_de_poids[]` correspond à la syntaxe pour sélectionner des données et le deuxième couple de crochets `[0,15,6985,452]` correspond à la liste des positions auxquelles on souhaite récupérer des valeurs.

Résultat

```
0       80.0
15      75.0
6985    59.0
452     56.0
Name: Weight, dtype: float64
```

Nous venons de récupérer les valeurs aux positions 0, 15, 6985 et 452.

3.4.2 Indexing via l'étiquette des valeurs

Il est aussi possible de sélectionner des valeurs selon l'étiquette de ces valeurs. Par exemple, imaginons que nous voulons sélectionner la valeur dont l'étiquette est la chaîne de caractères "`ma_valeur`". La syntaxe sera la suivante :

```
ma_serie["ma_valeur"]
```

Pour sélectionner plusieurs valeurs :

```
ma_serie[["ma_valeur1", "ma_valeur2","ma_valeur3"]]
```

Pour l'exemple de cette sous-section, nous allons repartir sur la série que nous avons créée précédemment et sur laquelle nous avons redéfini les index avec les noms des athlètes : `ma_serie_de_poids`.

Dans notre cas, il suffit de donner les noms des athlètes qui nous intéressent dans une liste. Imaginons que nous voulons récupérer le poids d'Antti Sami Aalto et de Andrzej ya.

Code

```
ma_serie_de_poids[["Antti Sami Aalto","Andrzej ya"]]
```

© Editions ENI - All rights reserved

Résultat

```
Name
Antti Sami Aalto    96.0
Andrzej ya          89.0
Name: Weight, dtype: float64
```

Ce code nous permet d'accéder à l'information de poids de ces deux athlètes qui font respectivement 96 et 89 kg.

L'indexing avec les positions des valeurs ou leurs étiquettes, entre crochets `[]`, est très répandu mais peut être à l'origine d'erreurs de mauvaise sélection de données. Pour que ces erreurs n'arrivent pas, les attributs `.loc` et `.iloc` ont été mis en place, nous allons voir cela ensemble.

3.4.3 Les indexeurs loc et iloc

Imaginons que les étiquettes d'index soient égales aux positions des valeurs, par exemple, que nous ayons une série de quatre valeurs dont les étiquettes d'index sont 1, 3, 2, 0 et dont les positions sont 0, 1, 2, 3. Cela va poser problème, car Pandas ne saura plus si nous lui demandons de récupérer la valeur selon l'étiquette de l'index, ou selon sa position dans la série.

Exemple pratique pour illustrer ce propos

Code

```
import pandas as pd
ma_serie=pd.Series([10,11,12,13], index=[1,3,2,0])
```

On crée une série nommée `ma_serie`, qui contient quatre valeurs et dont les noms d'index sont respectivement 1, 3, 2 et 0.

Résultat

```
1    10
3    11
2    12
0    13
dtype: int64
```

On constate que le nom de l'index à la position 0 est `1` et que le nom d'index à la position 4 de la série est `0`.

Imaginons que nous voulons récupérer la valeur `10`, soit la première valeur dans la série. Instinctivement, nous allons vouloir récupérer la valeur à la position 0 dans la série.

Code

```
ma_serie[0]
```

Résultat

```
13
```

Le résultat de ce code est la valeur `13` et non pas la valeur `10`, car Pandas récupère la valeur dont l'étiquette de l'index est 0. Cela peut générer des erreurs graves dans certaines analyses.

Heureusement, Pandas propose deux indexeurs pour spécifier à Python si nous souhaitons effectuer l'indexing sur les étiquettes d'index ou sur les positions des valeurs dans la série : ce sont les indexeurs `loc` et `iloc`. Ces indexeurs peuvent être considérés comme des attributs des objets de classe `Series` et `DataFrame`.

Syntaxe

```
serie.loc[etiquette]
serie.iloc[position]
```

Les attributs sont suivis des étiquettes ou positions auxquelles on souhaite récupérer des valeurs, entre crochets `[]`.

L'attribut `loc` va demander à Pandas d'utiliser les noms d'index, les étiquettes, pour sélectionner les valeurs dans la série.

Code

```
ma_serie.loc[0]
```

Résultat

```
13
```

L'attribut `loc` retourne la valeur dont le nom de l'index est `0`.

© Editions ENI - All rights reserved

L'attribut `iloc` va demander à Pandas d'utiliser la position des valeurs pour les sélectionner dans la série.

Code

```
ma_serie.iloc[0]
```

Résultat

```
10
```

L'attribut `iloc` retourne `10`, donc la valeur à la position `0` dans la série `ma_serie`.

3.4.4 Indexing via une expression booléenne

Enfin, il est possible de faire de l'indexing en utilisant une expression booléenne. Ici, cet indexing ne se base plus sur les positions ou les étiquettes des valeurs, mais réellement sur les valeurs de la série. Par exemple, si on veut récupérer l'ensemble des valeurs strictement supérieures à `10` dans la série, la syntaxe est la suivante :

```
ma_serie[ma_serie>10]
```

Ainsi, il s'agit d'utiliser les opérateurs de comparaisons (<, <=, >, >=, ==, !=) pour effectuer l'indexing.

Cet indexing se fait en deux étapes, que nous allons découper et que nous illustrerons en une ligne de code. Voici la syntaxe générale de la première étape :

```
ma_serie > 10
```

Ici, on cherche à savoir quelles valeurs sont strictement supérieures à `10` dans l'objet `ma_serie`, grâce à une expression booléenne. Cela crée une nouvelle série de même taille que l'objet `ma_serie`, contenant des valeurs de type booléen (`bool`), c'est-à-dire une suite de valeurs `True` et `False`. Si la valeur à la position 0 de l'objet `ma_serie` est égale à 20, alors la valeur à la position 0 de la nouvelle série, contenant les booléens, est `True`. Si cette valeur est égale à 9, la valeur à la position 0 de la série contenant les booléens est `False`.

Ensuite, pour récupérer l'ensemble des valeurs strictement supérieures à 10, la syntaxe est la suivante :

```
ma_serie[ma_serie > 10]
```

Cela retournera une nouvelle série, contenant uniquement les valeurs de `ma_serie` supérieures à 10, c'est-à-dire uniquement les valeurs aux positions qui sont à `True` dans la série contenant les booléens.

Illustrons cela avec un exemple. Sélectionnons uniquement les athlètes avec un poids strictement supérieur à 90 kg.

Code

```
ma_serie_de_poids>90
```

Résultat

```
Name
A Dijiang                  False
A Lamusi                   False
Gunnar Nielsen Aaby        False
Edgar Lindenau Aabye       False
Christine Jacoba Aaftink   False
                           ...
Andrzej ya                 False
Piotr ya                   False
Piotr ya                   False
Tomasz Ireneusz ya          True
Tomasz Ireneusz ya          True
Name: Weight, Length: 271116, dtype: bool
```

On constate une série de valeurs à `True` et `False`, de même taille que l'objet `ma_serie`, sachant que seuls les athlètes à `True` seront gardés dans le code ci-dessous.

Code

```
ma_serie_de_poids[ma_serie_de_poids>90]
```

© Editions ENI - All rights reserved

Résultat

```
Name
Timo Antero Aaltonen      130.0
Andreea Aanei             125.0
Dagfinn Sverre Aarskog     98.0
Hans Aasns                 93.0
Hans Aasns                 93.0
                           ...
Henk Jan Zwolle            93.0
Dominik ycki               95.0
Dominik ycki               95.0
Tomasz Ireneusz ya         96.0
Tomasz Ireneusz ya         96.0
Name: Weight, Length: 16818, dtype: float64
```

Suite à cet indexing, on sait maintenant que 16 818 valeurs sont strictement supérieures à 90 dans cette série.

Si on souhaite effectuer plusieurs expressions booléennes, il faudra utiliser les opérateurs `&` (pour et), `|` (pour ou) et `~` (pour la négation). On les appelle les opérateurs *bitwise* (comparaison bit à bit) en Python.

Remarque

Dans les chapitres précédents, et notamment dans le chapitre sur les rappels Python, nous avions utilisé les opérateurs logiques, `and`, `or` et `not`. Les opérateurs logiques fonctionnent bien sur des expressions qui renvoient une valeur `True` unique, ou une valeur `False` unique. Ici, les expressions booléennes retournent des séries de `True` et de `False`, pour chaque individu. Et dans ce cas, l'utilisation d'opérateurs logiques générera une erreur, alors que l'utilisation des opérateurs bitwise fonctionnera.

Par exemple, pour récupérer l'ensemble des valeurs strictement supérieures à 90 et strictement inférieures à 100, on peut utiliser l'opérateur `&`.

Code

```
ma_serie_de_poids[(ma_serie_de_poids>90) &
(ma_serie_de_poids<100)]
```

Résultat

```
Name
Dagfinn Sverre Aarskog     98.0
Hans Aasns                 93.0
Hans Aasns                 93.0
Hans Aasns                 93.0
Hans Aasns                 93.0
                           ...
Henk Jan Zwolle            93.0
Dominik ycki               95.0
Dominik ycki               95.0
Tomasz Ireneusz ya         96.0
Tomasz Ireneusz ya         96.0
Name: Weight, Length: 9891, dtype: float64
```

3.4.5 Slicing : découpage de valeurs successives

Tout comme pour les listes en Python, il est possible de faire du *slicing* (découpage) sur des séries, grâce aux symboles crochets `[]` et deux-points `:`.

Si vous n'êtes pas débutant en Python, vous avez déjà entendu parler des *slices*. Elles permettent de sélectionner des valeurs au sein de données, comme par exemple ici, les séries. Toutefois, contrairement aux listes, le slicing d'une série coupe/sélectionne les valeurs, mais aussi les index. La différence avec l'indexing est aussi que le slicing récupère des données consécutives, comprises dans un intervalle et n'utilise que des positions comme paramètres (on n'utilise pas les étiquettes, cela génère une erreur). Le but du slicing, tout comme l'indexing, est de récupérer un sous-ensemble de données qui nous intéressent.

Dans le cas où nous souhaitons sélectionner des valeurs au sein d'une série, la syntaxe générale est la suivante :

```
serie[start:stop:step]
```

Une autre possibilité pour le slicing qui est même conseillée est d'utiliser la méthode `iloc` plutôt que les crochets. Pour rappel, la méthode `iloc` permet de sélectionner des valeurs selon leur position. Précédemment, nous l'avons utilisée en lui précisant les positions à récupérer, mais sachez qu'il est aussi possible de lui donner des slices.

© Editions ENI - All rights reserved

Syntaxe du slicing avec `iloc`

```
serie.iloc[start:stop:step]
```

Ici, on lui donne une slice en précisant la position de départ et la position de fin des valeurs, en sachant qu'il s'agit des valeurs jusqu'à la position `stop` non incluse. Vous verrez plus souvent le slicing effectué avec l'indexeur `iloc` qu'avec les crochets seuls, pour la librairie Pandas.

Que ce soit avec les crochets seuls ou avec l'indexeur `iloc`, il y a trois informations importantes entre crochets.

- `start` : permet de donner la position de départ de la sélection. Si on souhaite commencer la sélection à partir de la deuxième valeur de la série, on mettra le chiffre 1.
- `stop` : permet de donner la position de fin de la sélection, non incluse. Attention donc, car ce qu'attend Python est la position de fin souhaitée + 1. En fait, on pourrait voir le *slicing* comme une échelle, et lorsqu'on demande à Python de sélectionner des valeurs à certaines positions, il sélectionnera les valeurs aux positions à droite de la position `start` et strictement à gauche de la position `stop`. Pour que cela soit visuel, imaginons que l'on donne comme `start` la valeur 3 et comme `stop` la valeur 5 sur notre série contenant les valeurs de poids des athlètes, créée précédemment (`ma_serie_de_poids`). Cela donnerait ceci :

0	1	2	3	4	5	...	27116

Les valeurs sélectionnées sont celles aux positions `3` et `4`. Il s'agit d'une gymnastique à retenir avec le slicing.

Exemple pratique

Pour bien comprendre le slicing, reprenons l'objet de type `Series` contenant le poids des athlètes, `ma_serie_de_poids_index_defaut`, pour pouvoir comprendre la sélection de valeurs avec le slicing.

Code

```
ma_serie_de_poids_index_defaut.iloc[3:5]
```

Résultat

```
3     NaN
4    82.0
Name: Weight, dtype: float64
```

Ici, on récupère bien les valeurs aux positions 3 et 4 de la série.

– Enfin, l'option `step` permet de spécifier le pas qu'on souhaite entre chaque valeur située entre le `start` et le `stop`. Imaginons que l'on donne comme `start` 2 et comme `stop` 11, avec un `step` de 2. La syntaxe générale serait la suivante :

```
Serie.iloc[2:11:2]
```

Les valeurs aux positions suivantes seront récupérées : 2, 4, 6, 8, 10. Alors que si nous ne donnions pas de valeur à `step`, avec la syntaxe générale suivante :

```
Serie.iloc[2:11]
```

les valeurs récupérées seraient : 2, 3, 4, 5, 6, 7, 8, 9, 10.

Exemple pratique

Appliquons ces lignes de code sur notre série exemple.

Code

```
ma_serie_de_poids_index_defaut.iloc[2:11:2]
```

Résultat

```
2     NaN
4    82.0
6    82.0
8    82.0
10   75.0
Name: Weight, dtype: float64
```

Code

```
ma_serie_de_poids_index_defaut.iloc[2:11]
```

© Editions ENI - All rights reserved

Résultat

```
2      NaN
3      NaN
4     82.0
5     82.0
6     82.0
7     82.0
8     82.0
9     82.0
10    75.0
Name: Weight, dtype: float64
```

En plus des options `start`, `stop` et `step` qui permettent de faire du *slicing*, il existe quelques autres syntaxes de slicing intéressantes à utiliser, dont nous allons voir quelques exemples tout de suite.

- Il est possible de sélectionner des données d'une position `start` jusqu'à la fin de la série, avec la syntaxe suivante :

```
Serie.iloc[start:]
```

Ici, on ne donne pas de valeur après le `:`, Pandas en déduit qu'il doit sélectionner les valeurs jusqu'à la fin de la série.

- On peut aussi sélectionner des valeurs du début de la série jusqu'à une position `stop`, avec la syntaxe :

```
Serie.iloc[:stop]
```

- Enfin, il faut savoir que les valeurs `start` et `stop` peuvent être négatives, ce qui signifie que Pandas commence à compter par la fin de la série.

Par exemple, la syntaxe suivante :

```
Serie.iloc[-1:]
```

permet d'afficher la dernière valeur de la série. En effet, `-1` signifie la dernière valeur d'une liste, série ou autre ensemble de données séquentielles. En lui demandant d'aller de `-1` jusqu'à la fin de la série, grâce à la syntaxe `[-1:]`, Pandas n'affichera donc que la dernière valeur.

Pour afficher les deux dernières valeurs de la série, la syntaxe serait la suivante :

```
Series.iloc[-2:]
```

En effet, ici on demande d'aller de la deuxième valeur avant la fin, `-2`, jusqu'à la fin, puisqu'on ne spécifie pas de valeur `stop`.

Enfin, pour afficher toutes les valeurs de la série, sans les deux dernières, la syntaxe serait la suivante :

```
Series.iloc[:-2]
```

En effet, on ne met pas de valeurs devant le `:`, ce qui signifie que le `start` part du début. Puis avec un `-2` comme option `stop`, Pandas s'arrêtera deux valeurs avant la fin.

Exemple pratique

Testons ces techniques de slicing sur notre série nommée `ma_serie_de_poids_index_defaut`.

Code

```
ma_serie_de_poids_index_defaut.iloc[1000:]
```

Résultat

```
1000      84.0
1001      84.0
1002      84.0
1003      73.0
1004      54.0
          ...
271111    89.0
271112    59.0
271113    59.0
271114    96.0
271115    96.0
Name: Weight, Length: 270116, dtype: float64
```

On constate que ce slicing génère un nouvelle série qui va de la position `1000` jusqu'à la fin de la série dont elle provient.

Pour afficher les vingt premières valeurs, le code est le suivant :

```
ma_serie_de_poids_index_defaut.iloc[:20]
```

© Editions ENI - All rights reserved

Résultat

```
0     80.0
1     60.0
2      NaN
3      NaN
4     82.0
5     82.0
6     82.0
7     82.0
8     82.0
9     82.0
10    75.0
11    75.0
12    75.0
13    75.0
14    75.0
15    75.0
16    75.0
17    75.0
18    72.0
19    72.0
Name: Weight, dtype: float64
```

On obtient bien les vingt premières valeurs de notre série, des positions 0 à 19.

Enfin, affichons la dernière valeur de notre série, avec le code suivant :

```
ma_serie_de_poids_index_defaut.iloc[-1:]
```

Résultat

```
271115    96.0
Name: Weight, dtype: float64
```

Les possibilités du slicing sont très vastes, mais vous avez à présent de bonnes bases pour découper des séries selon vos besoins.

Il faut savoir que lorsqu'on découpe un objet Python à l'aide du slicing, le sous-objet créé est de même type que l'objet dont il provient. Par exemple, si on fait du slicing sur une série de type numérique, l'objet créé après le slicing sera une série de type numérique. De même que si nous effectuons un slicing sur une liste, l'objet créé sera une liste.

3.5 Les attributs et les méthodes des objets de classe Series

Les objets de classes `Series` possèdent des attributs et des méthodes qui sont très utiles lorsqu'on souhaite les analyser. L'ensemble des méthodes et des attributs existants n'est pas couvert, mais vous pouvez tous les retrouver dans la documentation Python sur les objets de classe `Series` :
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html

3.5.1 Les attributs des objets de classe Series

Pour rappel, en Python, chaque classe a ses propres attributs, qui sont des variables appartenant à cette classe et permettant d'obtenir rapidement des informations au sujet d'un objet de cette classe. Il existe donc un ensemble d'attributs pour les objets de classe `Series`, dont voici quelques exemples :

- `dtype` : retourne le `dtype`, c'est-à-dire le type des données contenues dans la série.
- `iloc` et `loc` : permettent d'accéder aux valeurs de la série, par position ou par étiquette.
- `index` : retourne les index de la série.
- `values` : retourne les valeurs de la série.
- `name` : retourne le nom de la série.
- `size` : retourne le nombre de valeurs contenues dans la série, c'est-à-dire sa taille.

Il est primordial de retenir que, pour chaque classe d'objet, il existe des attributs associés qui permettent de simplifier la manipulation de ces objets, qui sont très utiles.

Exemple pratique

Déterminons la taille de notre série contenant le poids des athlètes.

Code

```
ma_serie_de_poids.size
```

© Editions ENI - All rights reserved

Résultat

```
271116
```

Pour rappel, les attributs sont des variables et s'appellent donc sans parenthèses, contrairement aux méthodes.

3.5.2 Les méthodes des objets de classe Series

Pour rappel, en Python, chaque classe a ses propres méthodes, qui peuvent être comparées à des fonctions capables de manipuler cette classe d'objets précisément. La différence entre les méthodes, spécifiques à une classe, et les fonctions, applicables à plusieurs classes, est la manière de les appeler. Une méthode s'appellera de la manière suivante.

```
ma_serie.methode()
```

alors que la fonction s'appellera de la manière suivante :

```
fonction(ma_serie)
```

Il existe énormément de méthodes pour les séries ; toutes sont accessibles dans la documentation liée aux séries :
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html

Il existe donc un ensemble de méthodes pour les objets de classe `Series`, dont voici quelques exemples :

- `describe()` : permet d'afficher un résumé statistique sur les valeurs de la série.
- `value_counts()` : permet de visualiser les valeurs uniques ainsi que leur nombre au sein de la série.
- `replace()` : permet de remplacer une ou plusieurs valeurs par une autre au sein d'une série.
- `set_index()` : permet de redéfinir les index de la série.

Exemple pratique

Imaginons que l'on veuille un résumé statistique de tous les poids de notre série, il faudra utiliser la méthode `describe()`.

Remarque

La méthode `describe()` est très utilisée pour avoir un premier aperçu de vos données en générant des statistiques descriptives.

Code

```
ma_serie_de_poids.describe()
```

Résultat

```
count    208241.000000
mean         70.701476
std          14.347880
min          25.000000
25%          60.000000
50%          70.000000
75%          79.000000
max         214.000000
Name: Weight, dtype: float64
```

Cette méthode fournit différentes statistiques. La première, `count`, donne le nombre de valeurs dans la série, sans compter les valeurs manquantes. Nous savons que la taille de notre série est de `271 116` valeurs, et maintenant nous savons que `208 241` valeurs ne sont pas des valeurs manquantes. Ensuite, `70.70` correspond à la valeur moyenne (`mean`) des poids dans notre série, avec une déviation standard (`std`) de `14` (moyenne de 70 kg +/- 14 kg). Le poids minimal dans ce jeu de données est de 25 kg (`min`) et le poids maximal (`max`) de 214 kg. Enfin, nous avons les quartiles, qui permettent de diviser les données en quatre parts égales. L'interprétation est la suivante : 25 % des athlètes font moins de 60 kg, 50 % des athlètes font moins de 70 kg et 75 % des athlètes font moins de 79 kg.

Cette méthode est puissante : vous pouvez constater qu'en une ligne de code, on peut déjà explorer ses données.

© Editions ENI - All rights reserved

3.6 Ajouter, supprimer et modifier les valeurs d'une série

3.6.1 Ajouter des valeurs à une série

Pour ajouter une ou des valeurs à une série, on utilise généralement la méthode `append()`. Cette méthode permet en réalité de concaténer deux ou plusieurs séries ; nous allons voir comment l'utiliser pour ajouter des valeurs à une série, à la volée.

Syntaxe

```
ma_serie.append(pd.Series([valeur], index=["label"]))
```

Dans cette syntaxe, pour ajouter la valeur à la série `ma_serie`, on doit donner un objet de type `Series` à la méthode `append()`, contenant une ou plusieurs valeurs. Ainsi, on peut découper cette syntaxe en deux parties.

Première partie :

```
pd.Series([valeur], index=["label"])
```

On utilise la fonction `Series` de Pandas, permettant de créer un objet de classe `Series`, à qui on donne une liste de valeurs, puis une liste d'index correspondants. Il est possible de ne pas spécifier l'option `index`, et dans ce cas les index sont créés par Python et correspondent à des nombres incrémentés.

Deuxième partie :

```
ma_serie.append(pd.Series([valeur], index=["label"]))
```

On applique la méthode `append()` à `ma_serie` en lui donnant notre nouvelle série comme option. Ainsi, les deux séries de valeurs sont concaténées.

Comme exemple pratique, ajoutons deux nouveaux athlètes à la série `ma_serie_de_poids`, avec leur poids respectif.

Code

```
ma_serie_de_poids=ma_serie_de_poids.append(pd.Series([90,120],
index=["Claire Muller", "Julien Villeroy"]))
ma_serie_de_poids.loc[["Claire Muller","Julien Villeroy"]]
```

Résultat

```
Claire Muller      90.0
Julien Villeroy    120.0
dtype: float64
```

Ici, la méthode `append()` retourne une nouvelle copie de la série `ma_serie_de_poids` avec les nouvelles valeurs. Étant donné que `append()` crée un nouvel objet, il ne faut pas oublier de stocker cette nouvelle série dans une variable. Ici, on la stocke dans la même variable que précédemment, nommée `ma_serie_de_poids`. Nous avons ainsi ajouté deux nouvelles valeurs à notre série.

Remarque

Contrairement à beaucoup de méthodes Pandas, `append()` génère une copie de la série en sortie et n'applique pas directement le traitement sur l'objet original. C'est une sécurité, pour ne pas modifier un objet sans que vous ne vous en rendiez compte. Pandas fait cela pour les méthodes touchant à l'intégrité des données contenues dans un objet, comme le fait la méthode `append()`.

3.6.2 Supprimer une valeur d'une série

Pour supprimer une valeur d'une série, on utilisera la méthode `drop()` qui permet de supprimer les valeurs aux étiquettes d'index spécifiées en option.

La syntaxe générale est la suivante :

```
ma_serie.drop(labels=["index1", "index2"], inplace=False)
```

L'option `labels` attend une liste de noms d'index à supprimer.

L'option `inplace` est intéressante : par défaut, elle est à `False`. Si cette option est à `False`, la méthode `drop()` retourne une copie de `ma_serie`, avec la modification effectuée. Si on met l'option à `True`, la méthode `drop()` effectue la modification directement sur l'objet `ma_serie` et ne retourne donc pas de copie. Il aurait été intéressant que cette option soit disponible pour la méthode `append()`, pour ne pas avoir à systématiquement stocker la nouvelle copie dans une variable. Toutefois, cela n'a pas encore été fait par les développeurs Pandas.

© Editions ENI - All rights reserved

Comme exemple pratique, supprimons les deux athlètes que nous avons ajoutés précédemment.

Code

```
ma_serie_de_poids.drop(labels=["Claire Muller","Julien Villeroy"],
inplace=True)
ma_serie_de_poids.loc[["Claire Muller","Julien Villeroy"]]
```

Résultat

Ce code génère une erreur ! En effet, aucun résultat ne peut être généré puisque Claire Muller et Julien Villeroy n'existent plus dans la série.

Remarque

Sachez que supprimer des données d'une série est peu courant. Il est préférable de filtrer la série et de stocker les valeurs qui nous intéressent dans une nouvelle série, plutôt que d'en modifier une. Pour filtrer la série, vous pouvez utiliser par exemple les expressions booléennes.

3.6.3 Modifier les valeurs d'une série

Pour modifier une valeur au sein d'une série, on pourra utiliser les indexeurs `loc` et `iloc` que nous avons vus précédemment lors de l'indexing (accès aux données). Ici, il s'agit de sélectionner la donnée qui nous intéresse via sa position ou son étiquette, puis d'y assigner une nouvelle valeur avec le symbole d'assignation =.

La syntaxe générale est la suivante :

```
ma_serie.loc["nom_index"]=nouvelle_valeur
```

ou

```
ma_serie.iloc[position]=nouvelle_valeur
```

Modifions les poids des athlètes Antti Sami Aalto et Andrzej ya dans notre série d'exemple.

Code

```
ma_serie_de_poids.loc[["Antti Sami Aalto"]]=56
ma_serie_de_poids.loc[["Antti Sami Aalto"]]
```

Ce code modifie le poids d'un athlète, Antti Sami Aalto, et lui attribue la valeur 56.

Résultat

```
Name
Antti Sami Aalto    56.0
Name: Weight, dtype: float64
```

Code

```
ma_serie_de_poids.loc[["Antti Sami Aalto","Andrzej ya"]]=[56,70]
ma_serie_de_poids.loc[["Antti Sami Aalto","Andrzej ya"]]
```

Ce code modifie le poids de deux athlètes. Pour cela, il faut fournir une liste contenant autant de valeurs que de poids d'athlètes à modifier, ici 2. Bien sûr, il faut que la liste de poids qu'on assigne aux athlètes soit de même taille que la liste d'athlètes que l'on donne à `loc`.

Résultat

```
Name
Antti Sami Aalto    56.0
Andrzej ya          70.0
Name: Weight, dtype: float64
```

Si un athlète est présent plusieurs fois dans une série, ce qui est le cas dans ce jeu de données puisqu'un athlète a pu participer à plusieurs JO et/ou plusieurs épreuves, son poids sera modifié à chacune de ses occurrences.

C'est le cas de l'athlète Christine Jacoba Aaftink. Regardons ses occurrences dans le jeu de données avant modification de son poids.

Résultat avant modification

```
Name
Christine Jacoba Aaftink    82.0
Christine Jacoba Aaftink    82.0
Christine Jacoba Aaftink    82.0
Christine Jacoba Aaftink    82.0
Christine Jacoba Aaftink    82.0
Christine Jacoba Aaftink    82.0
Name: Weight, dtype: float64
```

© Editions ENI - All rights reserved

Modifions son poids et assignons la valeur `60` à l'étiquette Christine Jaoba Aaftkink.

Code

```
ma_serie_de_poids.loc[["Christine Jacoba Aaftink"]]=60
ma_serie_de_poids.loc[["Christine Jacoba Aaftink"]]
```

Résultat

```
Name
Christine Jacoba Aaftink    60.0
Christine Jacoba Aaftink    60.0
Christine Jacoba Aaftink    60.0
Christine Jacoba Aaftink    60.0
Christine Jacoba Aaftink    60.0
Christine Jacoba Aaftink    60.0
Name: Weight, dtype: float64
```

Si, pour je ne sais quelle raison, nous voulions modifier une seule occurrence de cette athlète, alors il faudrait utiliser `iloc` plutôt que `loc` et préciser la position à laquelle modifier la valeur. En effet, une position étant unique, une seule valeur serait ainsi modifiée.

4. Structure de données Pandas : les objets de type DataFrame

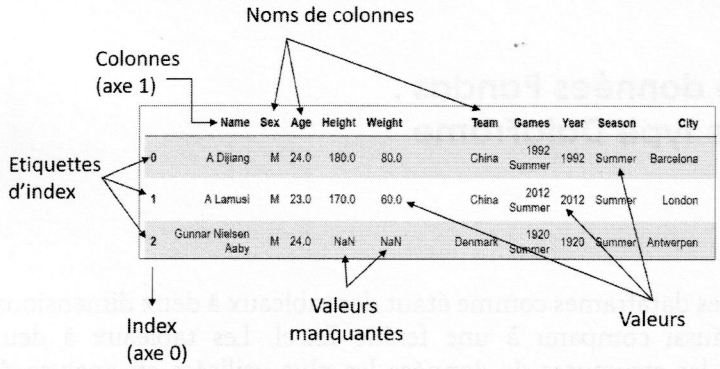
4.1 Introduction

On peut décrire les dataframes comme étant des tableaux à deux dimensions, qu'on pourrait aussi comparer à une feuille Excel. Les tableaux à deux dimensions sont les structures de données les plus utilisées en analyse de données, car cela permet de représenter en ligne les individus/observations et en colonne les variables représentant ces individus/observations. Ainsi, ce qui caractérise le dataframe, ce sont son index pour les lignes et ses noms de colonnes. Son index et ses noms de colonnes permettront d'accéder aux données.

Tout comme les séries, les dataframes sont construits sur les `ndarrays` NumPy et possèdent donc une très bonne performance. Aussi, une colonne d'un dataframe (qui est en réalité une série) contient, dans la très grande majorité des cas, des données homogènes, de même type. En revanche, le dataframe peut contenir des données hétérogènes, puisque les colonnes sont indépendantes entre elles, chacune correspondant à une série et, en termes statistiques, à une variable. Ainsi, le dataframe peut contenir des colonnes avec des données de type `int`, `float`, `bool`, etc.

Le dataframe possédant un index et des noms de colonnes, on dit aussi qu'il possède deux axes, appelés axe 0 (pour l'axe des index, ou lignes) et axe 1 (pour l'axe des colonnes). Cette notion d'axe est importante, car certaines fonctions prennent en paramètre l'axe qui est à traiter, il faut donc se rappeler que 0 signifie les lignes et 1 les colonnes. Par exemple, si on souhaite effectuer la somme de chaque colonne d'un dataframe, on utilise la fonction `sum()` sur l'axe `0`. En revanche, si on souhaite avoir une somme par ligne du dataframe, on utilise la fonction `sum()` sur l'axe `1`.

Une image étant plus claire que des mots, voyons ensemble la structure d'un dataframe sur la figure ci-dessous :



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

L'ensemble des propriétés précédemment décrites que possède le dataframe sont nécessaires dans le domaine de la *Data Science*, étant donné que l'analyste de données travaille avec des données contenant des variables de natures et de types différents (poids, sexe, dates...). Or, le dataframe a été créé pour la *Data Science* et pour les besoins de l'analyste de données. Nous allons le voir en détail tout le long de cette section.

© Editions ENI - All rights reserved

Pour les exemples concrets de cette section, nous allons à nouveau lire le fichier contenant les données de l'ensemble des athlètes ayant participé aux JO.

Code

```
import pandas as pd
donnees=pd.read_csv("../datasets/120-years-of-olympic-history-
athletes-and-results.csv", index_col=[1])
donnees.head()
```

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
A Dijiang	1	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
A Lamusi	2	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
Gunnar Nielsen Aaby	3	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN

Ici, on définit les noms d'athlètes (la colonne 1) comme étant les étiquettes d'index ; ainsi, la colonne `Name` devient l'index du dataframe et cet index correspond aux individus du jeu de données. On peut aussi voir les noms de colonnes, qui correspondent aux variables du jeu de données. Ces deux informations nous permettront de manipuler efficacement et très facilement le dataframe.

4.2 Indexing : sélectionner des valeurs d'un dataframe

Pour accéder aux valeurs d'un dataframe, la manière de faire est similaire à celle des séries, sauf qu'ici on aura deux ensembles de valeurs à donner, un ensemble pour les lignes et un pour les colonnes. Ici, l'indexing permettra d'accéder aux valeurs à des lignes et colonnes spécifiques.

Pour faire de l'indexing sur les dataframes, il faut utiliser les attributs `loc` et `iloc` comme vu précédemment pour les séries.

4.2.1 Indexing et slicing avec l'attribut loc

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

Pour sélectionner une valeur unique, la syntaxe est la suivante :

```
dataframe.loc["nom_ligne","nom de colonne"]
```

Ici, on constate qu'on accède à la valeur en donnant une étiquette d'index de ligne à gauche et un nom de colonne à droite.

Pour sélectionner plusieurs valeurs, il suffit d'utiliser des listes, avec la syntaxe suivante :

```
dataframe.loc[["nom_de_ligne_1", "nom_de_ligne_2"],
["nom_de_colonne", "nom_de_colonne"]]
```

Avec cette syntaxe, on sélectionnera deux valeurs, à deux lignes différentes pour une même colonne. En effet, la liste contenant les noms de colonnes, à droite de la virgule, contient deux mêmes noms de colonnes : "nom_de_colonne". Ainsi, la même colonne est sélectionnée dans cet exemple, pour deux lignes différentes.

Il est aussi possible d'utiliser le symbole : pour sélectionner une ligne spécifique et toutes les colonnes.

Syntaxe

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

Pour sélectionner plusieurs lignes et toutes les colonnes, la syntaxe est la suivante :

```
dataframe.loc[["nom_de_ligne1", "nom_de_ligne2"],:]
```

La différence avec la commande précédente est qu'ici on donne une liste de noms de lignes, ce qui permet de sélectionner plusieurs lignes.

© Editions ENI - All rights reserved

Enfin, pour sélectionner une colonne précise et toutes les lignes, la syntaxe sera la suivante :

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

Il est aussi possible d'utiliser la syntaxe du slicing pour récupérer des tranches de valeurs consécutives avec l'attribut `loc`. Pour rappel, l'indexing permet de sélectionner une ou plusieurs valeurs, consécutives ou non, quand le slicing permet de sélectionner des tranches de valeurs successives.

Rappel de la syntaxe du slicing :

```
dataframe[start:end:step]
```

Imaginons que l'on veuille sélectionner toutes les colonnes entre deux noms de colonnes, avec un pas de 2, c'est-à-dire qu'on garde une colonne sur deux, mais qu'on veuille aussi sélectionner une tranche de lignes. La syntaxe serait la suivante :

```
dataframe.loc['ligne1':'ligne20':1,'colonne1':'colonne10':2]
```

Ici, on sélectionne l'ensemble des lignes comprises entre `ligne1` et `ligne20`, `ligne1` et `lignes20` comprises. On sélectionne aussi l'ensemble des colonnes entre `colonne1` et `colonne10`, avec un pas de 2, `colonne_1` et `colonne10` comprises. Contrairement aux *slicing* avec les positions, ici le `end` est inclus.

Exemple pratique

Imaginons qu'on veuille regarder les informations sur le très célèbre athlète Usain Bolt. Pour cela, on peut sélectionner l'ensemble des valeurs dont l'étiquette d'index (l'étiquette de ligne) est "Usain St. Leo Bolt", de son nom complet.

Code

```
donnees.loc["Usain St. Leo Bolt"]
```

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
Usain St. Leo Bolt	13029	M	17.0	196.0	95.0	Jamaica	JAM	2004 Summer	2004	Summer	Athina	Athletics	Athletics Men's 200 metres	NaN
Usain St. Leo Bolt	13029	M	21.0	196.0	95.0	Jamaica	JAM	2008 Summer	2008	Summer	Beijing	Athletics	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	13029	M	21.0	196.0	95.0	Jamaica	JAM	2008 Summer	2008	Summer	Beijing	Athletics	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	13029	M	21.0	196.0	95.0	Jamaica	JAM	2008 Summer	2008	Summer	Beijing	Athletics	Athletics Men's 4 x 100 metres Relay	NaN
Usain St. Leo Bolt	13029	M	25.0	196.0	95.0	Jamaica	JAM	2012 Summer	2012	Summer	London	Athletics	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	13029	M	25.0	196.0	95.0	Jamaica	JAM	2012 Summer	2012	Summer	London	Athletics	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	13029	M	25.0	196.0	95.0	Jamaica	JAM	2012 Summer	2012	Summer	London	Athletics	Athletics Men's 4 x 100 metres Relay	Gold
Usain St. Leo Bolt	13029	M	29.0	196.0	95.0	Jamaica	JAM	2016 Summer	2016	Summer	Rio de Janeiro	Athletics	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	13029	M	29.0	196.0	95.0	Jamaica	JAM	2016 Summer	2016	Summer	Rio de Janeiro	Athletics	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	13029	M	29.0	196.0	95.0	Jamaica	JAM	2016 Summer	2016	Summer	Rio de Janeiro	Athletics	Athletics Men's 4 x 100 metres Relay	Gold

Usain Bolt a participé à quatre JO, remportant huit médailles d'or sur les dix épreuves auxquelles il a participé.

Imaginons que, plutôt que de tout afficher, on souhaite simplement afficher les colonnes "Games", "Event" et "Medal" ("Jeux", "Epreuve", "Médaille").

Code

```
donnees.loc["Usain St. Leo Bolt",["Games", "Event", "Medal"]]
```

Résultat

Name	Games	Event	Medal
Usain St. Leo Bolt	2004 Summer	Athletics Men's 200 metres	NaN
Usain St. Leo Bolt	2008 Summer	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	2008 Summer	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	2008 Summer	Athletics Men's 4 x 100 metres Relay	NaN
Usain St. Leo Bolt	2012 Summer	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	2012 Summer	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	2012 Summer	Athletics Men's 4 x 100 metres Relay	Gold
Usain St. Leo Bolt	2016 Summer	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	2016 Summer	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	2016 Summer	Athletics Men's 4 x 100 metres Relay	Gold

© Editions ENI - All rights reserved

Si on souhaite comparer les résultats d'Usain Bolt avec ceux de son adversaire, Justin Alexander Gatlin, il faudra utiliser le code suivant :

Code

```
donnees.loc[["Justin Alexander Gatlin","Usain St. Leo Bolt"],
["Games", "Event", "Medal"]]
```

Résultat

Name	Games	Event	Medal
Justin Alexander Gatlin	2004 Summer	Athletics Men's 100 metres	Gold
Justin Alexander Gatlin	2004 Summer	Athletics Men's 200 metres	Bronze
Justin Alexander Gatlin	2004 Summer	Athletics Men's 4 x 100 metres Relay	Silver
Justin Alexander Gatlin	2012 Summer	Athletics Men's 100 metres	Bronze
Justin Alexander Gatlin	2012 Summer	Athletics Men's 4 x 100 metres Relay	NaN
Justin Alexander Gatlin	2016 Summer	Athletics Men's 100 metres	Silver
Justin Alexander Gatlin	2016 Summer	Athletics Men's 200 metres	NaN
Justin Alexander Gatlin	2016 Summer	Athletics Men's 4 x 100 metres Relay	NaN
Usain St. Leo Bolt	2004 Summer	Athletics Men's 200 metres	NaN
Usain St. Leo Bolt	2008 Summer	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	2008 Summer	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	2008 Summer	Athletics Men's 4 x 100 metres Relay	NaN
Usain St. Leo Bolt	2012 Summer	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	2012 Summer	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	2012 Summer	Athletics Men's 4 x 100 metres Relay	Gold
Usain St. Leo Bolt	2016 Summer	Athletics Men's 100 metres	Gold
Usain St. Leo Bolt	2016 Summer	Athletics Men's 200 metres	Gold
Usain St. Leo Bolt	2016 Summer	Athletics Men's 4 x 100 metres Relay	Gold

Justin Alexander Gatlin a, quant à lui, une médaille d'or à son actif, ainsi que deux médailles d'argent et deux médailles de bronze.

Enfin, sélectionnons des tranches de lignes et de colonnes, afin d'illustrer le slicing avec `loc`.

Code

```
donnees.loc['A Dijiang':'Edgar Lindenau Aabye':1,'Age':'NOC':2]
```

Dans cet exemple, on demande de sélectionner les athlètes `A Dijiang`, `Edgar Lindenau Aabye` et l'ensemble des athlètes entre ces deux-là. Puis on demande de sélectionner les colonnes `Age`, `NOC` et les colonnes entre, avec un pas de 2.

Résultat

Name	Age	Weight	NOC
A Dijiang	24.0	80.0	CHN
A Lamusi	23.0	60.0	CHN
Gunnar Nielsen Aaby	24.0	NaN	DEN
Edgar Lindenau Aabye	34.0	NaN	DEN

4.2.2 Indexing et slicing avec l'attribut iloc

L'attribut `iloc` permet de sélectionner des valeurs selon leurs positions dans les lignes et les colonnes.

Si on souhaite sélectionner une valeur à une ligne et une colonne précise, la syntaxe est la suivante :

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

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

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

Pour sélectionner une ligne et toutes les colonnes, la syntaxe sera la suivante :

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

Pour sélectionner plusieurs lignes et toutes les colonnes, il faut donner une liste de positions de ligne, comme suit :

```
dataframe.iloc[[26,65],:]
```

© Editions ENI - All rights reserved

Enfin, pour sélectionner une colonne précise et toutes les lignes, la syntaxe sera la suivante :

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

Comme avec l'attribut `loc`, il est possible de faire du slicing pour sélectionner des tranches de lignes et de colonnes.

Syntaxe

```
dataframe.iloc[0:1,0:10]
```

Ici, on sélectionne la première ligne et les dix premières colonnes du dataframe. Contrairement au slicing avec `loc`, ici le nombre qu'on donne à l'option `end` n'est pas compris dans la sélection (`iloc` s'arrête à la position 0 pour les lignes, donc sélectionne la première ligne et s'arrête à la position 9 pour les colonnes, donc sélectionne les dix premières colonnes).

Remarque

Un exemple de slicing avec `iloc` est disponible dans le notebook dédié au chapitre sur la librairie Pandas, en téléchargement de ce livre.

4.2.3 Indexing avec une expression booléenne

Pour faire de l'indexing avec une expression booléenne, tout comme pour les séries, il faut utiliser les opérateurs de comparaison (>, <, ==, !=, >=, <=) et les opérateurs *bitwise* `&`, `|` et `~`.

Avec les dataframes, on spécifiera la colonne sur laquelle on souhaite appliquer l'expression booléenne.

Syntaxe

```
dataframe[dataframe['colonne1'] >= 10]
```

La syntaxe générale d'utilisation des opérateurs *bitwise* pour comparer plusieurs expressions booléennes est la suivante :

```
dataframe[(dataframe['colonne1'] >= 10) & (dataframe['colonne3'] == 25)]
```

Enfin, pour obtenir l'inverse de l'expression booléenne demandée, on utilisera l'opérateur *bitwise* de négation ~, avec la syntaxe suivante :

```
dataframe[~(dataframe['colonne1'] >= 10)]
```

Exemple pratique

Imaginons qu'on souhaite sélectionner l'ensemble des femmes ayant participé aux JO et ayant gagné une médaille d'or.

Code

```
donnees[(donnees["Sex"]=="F") & (donnees["Medal"]=="Gold")]
```

Résultat

Urka olnir	135219	F	30.0	173.0	64.0	Slovenia	SLO	2012 Summer	2012	Summer	London	Judo	Judo Women's Half-Middleweight	Gold
Vera Zozua (-Pyatnizin)	135290	F	24.0	157.0	69.0	Soviet Union	URS	1980 Winter	1980	Winter	Lake Placid	Luge	Luge Women's Singles	Gold
Olha Valentynivna Zubareva	135331	F	22.0	182.0	90.0	Soviet Union	URS	1980 Summer	1980	Summer	Moskva	Handball	Handball Women's Handball	Gold
Nataliya Vladimirovna Zuyeva	135488	F	19.0	176.0	62.0	Russia	RUS	2008 Summer	2008	Summer	Beijing	Rhythmic Gymnastics	Rhythmic Gymnastics Women's Group	Gold
Elina Aleksandrovna Zvereva (Kisheyeva-)	135501	F	39.0	183.0	100.0	Belarus	BLR	2000 Summer	2000	Summer	Sydney	Athletics	Athletics Women's Discus Throw	Gold
Julia Zwehl	135520	F	28.0	167.0	60.0	Germany	GER	2004 Summer	2004	Summer	Athina	Hockey	Hockey Women's Hockey	Gold
Galina Ivanovna Zybina (-Fyodorova)	135553	F	21.0	168.0	80.0	Soviet Union	URS	1952 Summer	1952	Summer	Helsinki	Athletics	Athletics Women's Shot Put	Gold

3747 rows × 14 columns

Si on descend le curseur jusqu'en bas du tableau, sur l'interface Jupyter Notebook, on peut voir sa taille, avec `3747` lignes et `14` colonnes. Il y a donc 3747 médailles d'or attribuées à des athlètes féminines depuis le début des JO, sachant que ces médailles ont pu être attribuées plusieurs fois à la même athlète.

Pour comparer le nombre de médailles d'or attribuées aux athlètes masculins, on pourrait faire la négation de notre première expression booléenne.

Code

```
donnees[~(donnees["Sex"]=="F") & (donnees["Medal"]=="Gold")]
```

`9625` lignes ont été sélectionnées, il y a donc 9625 médailles d'or attribuées aux athlètes masculins depuis le début des JO.

© Editions ENI - All rights reserved

4.3 Ajout, suppression et modification sur un dataframe

4.3.1 Ajouter une ou plusieurs colonnes à un dataframe

Nous allons voir deux méthodes pour ajouter une colonne à un dataframe.

La première, qui est la plus simple au niveau syntaxique, permet d'ajouter une colonne à partir d'une liste ou d'une série, cette colonne s'ajoutant à la fin du dataframe.

Syntaxe

```
dataframe['nouvelle colonne'] = [ma_valeur1, ma_valeur_2, ...]
dataframe['nouvelle colonne'] = ma_serie
```

Attention, il faut que le nombre de valeurs dans votre liste ou dans votre série soit de même taille que le nombre de lignes dans votre dataframe.

Si vous souhaitez ajouter une colonne contenant la même valeur à chaque ligne, pour, par exemple, modifier ces valeurs plus tard dans votre code, il est alors possible de donner une liste contenant une valeur unique et cette valeur sera alors la même dans toute la colonne.

Syntaxe

```
dataframe['nouvelle colonne']=[0]
```

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

Syntaxe

```
dataframe.insert(0, 'Nom_de_colonne', [1, 2, 3])
dataframe.insert(0, 'Nom_de_colonne', ma_serie)
```

Cette commande ajoute une colonne nommée `Nom_de_colonne`, comme première colonne du dataframe, car on donne 0 comme position de cette nouvelle colonne. Cette colonne contient les valeurs `1`, `2`, `3` ou, dans la deuxième syntaxe, les valeurs contenues dans l'objet `ma_serie`, de type `Series`. Attention, la liste de valeurs ou la série doivent avoir le même nombre de lignes que dans le dataframe.

Exemple pratique

Créons une copie de notre objet `donnees` appelée `donnees_test`, auquel nous ajouterons une colonne nommée `Test` contenant la valeur `0`, dans le dataframe des Jeux olympiques, à la fin du tableau :

```
donnees_test=donnees.copy()
donnees_test["Test"]=0
donnees_test.head()
```

Utilisons la méthode `insert()` pour ajouter notre série contenant le poids des athlètes, `ma_serie_de_poids`, comme colonne nommée `Poids_athlètes`, en troisième position du dataframe `donnees_test`.

```
donnees_test.insert(2, "Poids_athlètes", ma_serie_de_poids)
donnees_test.head()
```

Les résultats de ces codes exemples sont disponibles dans le notebook dédié à ce chapitre, disponible en téléchargement de ce livre.

4.3.2 Ajouter une ligne à un dataframe

Pour ajouter une ou plusieurs lignes, on utilise la méthode `append()` que nous avons déjà rencontrée sur les objets de classe `Series`.

Syntaxe d'ajout d'une ligne

```
import pandas as pd
dataframe_nouveau=dataframe.append(pd.Series([valeur1, valeur2, ...],
index=dataframe.columns),ignore_index=True)
```

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

© Editions ENI - All rights reserved

Par défaut, l'option `ignore_index` est à `False`, et cela précise à `append()` de garder les index de lignes du dataframe intacts et d'utiliser le nom de la série comme index de la nouvelle ligne ajoutée dans le dataframe. Ainsi, pour utiliser l'option `ignore_index=False`, il faut obligatoirement donner un nom à la série. Imaginons qu'on ait un dataframe de cinq lignes, dont les index de lignes sont `0, 1, 2, 3, 4`, et qu'on souhaite ajouter une ligne avec une série à laquelle nous n'avons pas donné de nom avec l'option `name` lors de sa création. Il faudra donc obligatoirement préciser à `append()` l'option `ignore_index=True` afin qu'elle crée des noms d'index continus, c'est-à-dire `0, 1, 2, 3, 4, 5` ; toutefois, on perdra ici les index de lignes du dataframe.

Le choix de cette option est plutôt simple : si les index de votre dataframe correspondent à des nombres incrémentés et que vous souhaitez rester là-dessus, définissez une série sans nom au sein de la méthode `append()` et utilisez l'option `ignore_index=True`. En revanche, si vous donnez un nom à la série créée dans `append()`, grâce à l'option `Name` que nous avons vue ensemble dans la section sur les séries, et que vous souhaitez que ce nom précisément apparaisse comme étiquette de la nouvelle ligne ajoutée, alors laissez l'option `ignore_index=False`.

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

Syntaxe

```
dataframe_nouveau=dataframe.append([pd.Series([valeur1, valeur2, ...],
index=dataframe.columns), pd.Series([valeur1_2, valeur2_2, ...],
index=index_colonnes), pd.Series([valeur1_3, valeur2_3, ...],
index=index_colonnes) ], ignore_index=True)
```

Ici, on donne à la méthode `append()` une liste, entre crochets `[]`, de séries, avec une série par nouvelle ligne. On ne donne pas de nom à nos séries et on utilise l'option `ignore_index=True`. Mais on pourrait donner un nom aux séries, à utiliser comme index des nouvelles lignes.

Syntaxe

```
dataframe_nouveau=dataframe.append([pd.Series([valeur1, valeur2, ...],
index=dataframe.columns, name="index_de_ligne_1"),
pd.Series([valeur1_2, valeur2_2, ...], index=index_colonnes,
name="index_de_ligne_2"), pd.Series([valeur1_3, valeur2_3, ...],
index=index_colonnes, name="index_de_ligne_3") ])
```

Remarque

Comme expliqué lors de la première utilisation de la méthode `append()`, celle-ci n'effectue pas la modification directement sur le dataframe mais crée une copie de ce dataframe, pour ne pas toucher à l'intégrité des données du dataframe original. Ainsi, il faut ranger le nouveau dataframe que retourne la méthode `append()` dans une nouvelle variable.

Exemple pratique

Avec deux exemples différents, ajoutons une ligne à notre dataframe, une fois avec l'option `ignore_index=True` et une fois avec l'option `ignore_index=False` afin d'illustrer cette option.

Code pour ajouter une ligne, avec l'option `ignore_index=True`

```
ajout_une_ligne = donnees.append(pd.Series(['135572', 'F', 29,
175, 58, 'China', 'CHN', '2012 Summer', 2012, 'Summer', 'London',
'Judo', "Judo Women's Extra-Lightweight",'NaN' ],
index=donnees.columns ),ignore_index=True)
ajout_une_ligne.tail()
```

Résultat

	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
271112	135570	M	27.0	176.0	59.0	Poland	POL	2014 Winter	2014	Winter	Sochi	Ski Jumping	Ski Jumping Men's Large Hill, Individual	NaN
271113	135570	M	27.0	176.0	59.0	Poland	POL	2014 Winter	2014	Winter	Sochi	Ski Jumping	Ski Jumping Men's Large Hill, Team	NaN
271114	135571	M	30.0	185.0	96.0	Poland	POL	1998 Winter	1998	Winter	Nagano	Bobsleigh	Bobsleigh Men's Four	NaN
271115	135571	M	34.0	185.0	96.0	Poland	POL	2002 Winter	2002	Winter	Salt Lake City	Bobsleigh	Bobsleigh Men's Four	NaN
271116	135572	F	29.0	175.0	58.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Women's Extra-Lightweight	NaN

On voit que les index de lignes, qui correspondaient aux noms des athlètes, ont disparu pour laisser place à des index de lignes incrémentés. Si on souhaite garder les noms des athlètes comme index après l'ajout de la nouvelle ligne, il faut donc donner un nom à notre série et laisser l'option `ignore_index` à `False`.

© Editions ENI - All rights reserved

Code

```
ajout_une_ligne = donnees.append(pd.Series(['135572', 'F', 29,
175, 58, 'China', 'CHN', '2012 Summer', 2012, 'Summer', 'London',
'Judo', "Judo Women's Extra-Lightweight",'NaN' ],
index=donnees.columns, name= "Claire Muller" ))
ajout_une_ligne.tail()
```

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
Piotr ya	135570	M	27.0	176.0	59.0	Poland	POL	2014 Winter	2014	Winter	Sochi	Ski Jumping	Ski Jumping Men's Large Hill, Individual	NaN
Piotr ya	135570	M	27.0	176.0	59.0	Poland	POL	2014 Winter	2014	Winter	Sochi	Ski Jumping	Ski Jumping Men's Large Hill, Team	NaN
Tomasz Ireneusz ya	135571	M	30.0	185.0	96.0	Poland	POL	1998 Winter	1998	Winter	Nagano	Bobsleigh	Bobsleigh Men's Four	NaN
Tomasz Ireneusz ya	135571	M	34.0	185.0	96.0	Poland	POL	2002 Winter	2002	Winter	Salt Lake City	Bobsleigh	Bobsleigh Men's Four	NaN
Claire Muller	135572	F	29.0	175.0	58.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Women's Extra-Lightweight	NaN

Ici, on constate bien qu'un nouvel athlète a été ajouté à notre dataframe, avec une étiquette d'index qui s'intègre bien à celles du dataframe original.

4.3.3 Supprimer des lignes ou colonnes d'un dataframe

Pour la suppression d'éléments dans un dataframe, que ce soit des lignes ou des colonnes, il faut utiliser la méthode `drop()`. Pour les colonnes, il faut utiliser le ou les noms de colonnes à supprimer en spécifier l'axe d'intérêt, ici l'axe des colonnes, donc 1.

Syntaxe de suppression de colonnes avec la méthode `drop()`

```
dataframe_nouveau=dataframe.drop(["ma_colonne_1","ma_colonne_2",
"ma_colonne_3"], axis=1)
```

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

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

Syntaxe

```
dataframe_nouveau=dataframe.drop(["ma_ligne_1","ma_ligne_2",
"ma_ligne_3"], axis=0)
```

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

```
dataframe_nouveau=dataframe.drop(dataframe.index[[2,3]], axis=0)
```

La méthode `index()` permet de récupérer les étiquettes des index en donnant leur position. Ainsi, dans cet exemple, on récupère les étiquettes des lignes aux positions 2 et 3, qu'on donne ensuite à la méthode `drop()` pour qu'elle puisse les supprimer. De cette manière, on supprime les lignes aux positions 2 et 3 du dataframe.

Nous avions vu la méthode `drop()` ainsi que son option `inplace` dans la section sur les séries. De la même manière que la méthode `append()`, la méthode `drop()` retourne une copie du dataframe avec les éléments supprimés, afin de ne pas modifier le dataframe original. Toutefois, il est possible d'utiliser l'option `inplace=True` pour effectuer la suppression directement sur le dataframe original.

Syntaxe

```
dataframe.drop(["ma_colonne_1","ma_colonne_2","ma_colonne_3"],
axis=1, inplace=True)
```

Exemple pratique

Supprimons les colonnes nommées `Test` et `Poids_athlètes` précédemment ajoutées sur le dataframe nommé `donnees_test`.

Code

```
donnees_test.drop(["Test","Poids_athlètes"], axis=1,
inplace=True)
donnees_test.head()
```

Ce code et son résultat sont disponibles dans le notebook dédié à ce chapitre, disponible en téléchargement avec ce livre.

Comme deuxième exemple, supprimons la dernière ligne ajoutée à notre jeu de données sur les JO, dont le dataframe est stocké dans la variable `ajout_une_ligne`. L'étiquette de cette ligne est "Claire Muller" et on peut l'utiliser pour supprimer la ligne.

© Editions ENI - All rights reserved

Code

```
ajout_une_ligne.drop("Claire Muller", axis=0, inplace=True)
ajout_une_ligne.tail()
```

La ligne a bien été supprimée, vous pouvez voir ce résultat dans le notebook dédié au chapitre.

4.3.4 Modifier des valeurs dans un dataframe

Comme pour les séries, on peut utiliser les attributs `loc` (en donnant les étiquettes) et `iloc` (en donnant les positions) ainsi que le symbole d'assignation = pour modifier les données.

Pour modifier une valeur avec l'attribut `loc` ou `iloc`, la syntaxe est la suivante :

```
dataframe.loc["étiquette de ligne", "nom de colonne"] = valeur
dataframe.iloc[position_ligne, position_colonne] = valeur
```

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

Syntaxe

```
dataframe.loc[dataframe["ma_colonne"] > 1000, ["ma_colonne"]] = 1000
```

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

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

Sa syntaxe est la suivante :

```
dataframe.replace(to_replace="mon_mot", value="mon_mot_2")
```

Il existe quelques autres méthodes de remplacement de données, mais les trois méthodes décrites ici vous permettent globalement de modifier votre dataframe de manière complète. Nous allons illustrer ces méthodes par des exemples concrets sur notre dataframe `donnees`.

Exemple pratique

Commençons par modifier l'âge de l'athlète A Dijiang en lui donnant vingt-cinq ans.

Code

```
donnees_modif=donnees.copy()
donnees_modif.loc["A Dijiang", "Age"] = 25
donnees_modif.head()
```

Résultat

L'âge de l'athlète a bien été modifié dans le tableau donnees_modif.

À présent, remplaçons toutes les tailles comprises entre 170 cm et 180 cm par 175 cm.

Code

```
donnees_modif=donnees.copy()
donnees_modif.loc[(donnees_modif["Height"] >= 170) &
(donnees_modif["Height"] <= 180), "Height"] = 175
donnees_modif.head()
```

Ici, on demande de sélectionner les lignes dont la valeur de la colonne `Height` (taille) est supérieure ou égale à 170 et inférieure ou égale à 180 et d'y remplacer cette valeur par 175.

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
A Dijiang	1	M	24.0	175.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
A Lamusi	2	M	23.0	175.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
Gunnar Nielsen Aaby	3	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
Edgar Lindenau Aabye	4	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
Christine Jacoba Aaftink	5	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

© Editions ENI - All rights reserved

A Dijang et A Lamusi avaient des tailles respectives de 180 et 170 cm. Leur taille a donc été remplacée par 175 cm. En revanche, Christine Jacoba Aaftink, qui mesurait 185 cm, mesure toujours 185 cm, puisque sa taille est supérieure à 180 cm.

Enfin, terminons ces exemples en utilisant la méthode `replace()`. Imaginons qu'on veuille remplacer le mot "China" par "Chine" dans l'ensemble du tableau.

Code

```
donnees_modif=donnees.copy()
donnees_modif.replace(to_replace="China", value="Chine",
inplace=True)
donnees_modif.head()
```

Résultat

L'ensemble des mots `China` du dataframe ont bien été remplacés par `Chine`. Vous pouvez visualiser ce résultat dans le notebook associé au chapitre. Vous pouvez également constater que la méthode `replace()`, comme la méthode `drop()`, possède une option `inplace`. Ainsi, en précisant `inplace = True`, la modification est effectuée directement sur le dataframe stocké dans la variable `donnees_modif`.

4.4 Nettoyage et préparation des données avec Pandas

À présent que vous connaissez les différentes structures Pandas et comment accéder aux données et les modifier, nous allons voir comme nettoyer ces données. En effet, avant de pouvoir explorer les données contenues dans un tableau, il faut le nettoyer, le préparer. Il y a deux étapes importantes, avant toute chose : prendre en compte les valeurs manquantes, et les traiter au besoin, mais aussi les valeurs dupliquées, et les supprimer si besoin.

4.4.1 Gestion des données manquantes

Lorsqu'une donnée ne peut pas être répertoriée, lorsque la valeur dans le dataframe est vide, sa valeur devient `NaN`. Il est important de prendre connaissance de la proportion des valeurs manquantes, voire de les traiter, ce que nous allons voir dans cette section.

La syntaxe générale pour compter le nombre de valeurs manquantes dans un dataframe est la suivante :

```
mon_dataframe.isna().sum()
```

La méthode `is.na()` va remplacer l'ensemble des valeurs du dataframe par `True` (si la valeur est `NaN`) ou `False` (si la valeur est différente de `NaN`). Puis la méthode `sum()` va compter le nombre de valeurs à `True` par colonne du dataframe, ce qui nous permettra de quantifier la présence des valeurs manquantes.

Voyons cela sur notre dataframe des Jeux olympiques.

Code

```
donnees.isna().sum()
```

Résultat

```
ID             0
Sex            0
Age         9474
Height     60171
Weight     62875
Team           0
NOC            0
Games          0
Year           0
Season         0
City           0
Sport          0
Event          0
Medal     231333
Name           0
dtype: int64
```

© Editions ENI - All rights reserved

Pour la variable `Medal` (médailles), sur 271 116 lignes dans le tableau, 231 333 valeurs sont à `NaN`. Cela est normal puisque les seules modalités de cette variable sont `Gold`, `Silver` et `Bronze`. Quand un athlète n'a gagné aucune de ces trois médailles, ce qui est le plus fréquent, la valeur est `NaN`. Ainsi, on peut en déduire que 39 783 médailles ont été gagnées depuis le début des JO, selon ce dataframe.

On constate que les variables `Sex` (sexe), `Year` (année), `Season` (saison), ou encore `Event` (discipline), sont des variables toujours bien renseignées dans le tableau, sans aucune valeur manquante. En revanche, les variables `Age` (âge), `Height` (taille) et `Weight` (poids) ne sont pas toujours renseignées. On peut même constater que, sur l'ensemble du jeu de données, environ 20 % des valeurs des variables taille/poids ne sont pas renseignées. Par exemple, il y a 60 171 valeurs manquantes dans la variable `Height`, ce qui correspond à 60171*100/271116 = 22 % de données manquantes, ce qui est quand même conséquent. Par ailleurs, pour 3.5 % du jeu de données, l'âge n'est pas renseigné.

Il peut arriver que la suppression des lignes contenant des `NaN` soit utile, si on veut uniquement considérer les données pour lesquelles les valeurs sont présentes.

Remarque

Dans les chapitres sur la visualisation de données avec les librairies Matplotlib et Seaborn, nous verrons ensemble dans quel cas il est intéressant de supprimer les valeurs manquantes et dans quel cas il faut impérativement les garder, sous peine de perdre de l'information.

Suppression des valeurs manquantes

Pour supprimer les lignes contenant un `NaN` dans au moins une colonne ou toutes les colonnes, il existe la méthode `dropna()`.

Syntaxe

```
dataframe.dropna(how="any")
```

L'option `how="any"` permet de spécifier à la méthode `dropna()` que si une seule valeur de la ligne est égale à `NaN`, alors il faut supprimer cette ligne.

Il est aussi possible de supprimer les lignes si l'ensemble des valeurs sont à `NaN`, c'est-à-dire si la valeur `NaN` est présente dans l'ensemble des colonnes, grâce à l'option `how="all"`.

Syntaxe

```
dataframe.dropna(how="all")
```

Enfin, pour supprimer les lignes avec une valeur manquante dans une colonne spécifique, il faudra utiliser la méthode `notna()` plutôt que la méthode `dropna()`.

Syntaxe

```
dataframe[dataframe['ma_colonne'].notna()]
```

En appliquant la méthode `notna()` sur la colonne qui nous intéresse, celle-ci retourne un objet de type `Series` contenant un ensemble de booléens, avec `True` (si la valeur n'est pas manquante) et `False` (si la valeur est manquante). En donnant cette série de booléens entre crochets à l'objet `dataframe`, cela permet de ne sélectionner que les lignes dont les valeurs sont différentes de `NaN` dans la colonne `ma_colonne`.

Exemple pratique

Dans le cas où on veut étudier uniquement la taille des athlètes ayant participé aux JO, il serait plutôt utile de supprimer les athlètes dont la taille n'est pas répertoriée dans le jeu de données. Ainsi, nous allons supprimer les lignes avec une valeur manquante dans la colonne `Height`.

Code

```
donnees[donnees["Height"].notna()]
```

© Editions ENI - All rights reserved

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal	Name
A Dijiang	1	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN	A Dijiang
A Lamusi	2	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN	A Lamusi
Christine Jacoba Aaftink	5	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN	Christine Jacoba Aaftink
Christine Jacoba Aaftink	5	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 1,000 metres	NaN	Christine Jacoba Aaftink
Christine Jacoba Aaftink	5	F	25.0	185.0	82.0	Netherlands	NED	1992 Winter	1992	Winter	Albertville	Speed Skating	Speed Skating Women's 500 metres	NaN	Christine Jacoba Aaftink
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

On constate que les lignes avec des valeurs manquantes pour la taille ont bien été supprimées. Par exemple, l'athlète Gunnar Nielsen Aaby, qui était positionné juste après A Lamusi, n'est plus présent dans le tableau, car sa taille était manquante.

Remplacement des valeurs manquantes

Il peut arriver que nous n'ayons pas envie de supprimer les lignes contenant des valeurs manquantes, sous peine de perdre de l'information, mais qu'on souhaite toutefois ne plus avoir de valeurs manquantes dans le tableau. Il est alors possible d'utiliser une méthode appelée `fillna()`, qui va remplacer les valeurs `NaN` par ce qu'on lui demande.

Par exemple, si on souhaite remplacer l'ensemble des `NaN` par 0, la syntaxe est la suivante :

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

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

```
dataframe[['colonne1', 'colonne4']] =
dataframe[['colonne1','colonne4']].fillna(dataframe[['colonne1',
'colonne4']].mean())
```

Cette syntaxe peut sembler compliquée, nous allons l'illustrer directement avec un exemple pratique, pour mieux la comprendre.

Remplaçons les valeurs manquantes des colonnes `Age`, `Height` et `Weight` par la valeur moyenne de chaque colonne, respectivement.

Code

```
donnees[['Age','Height', 'Weight']] =
donnees[['Age','Height','Weight']].fillna(donnees[['Age','Height',
'Weight']].mean())
```

Découpons ce code.

Le résultat du code :

```
donnees[['Age','Height','Weight']].mean()
```

est :

```
Age         25.556898
Height     175.338970
Weight      70.702393
dtype: float64
```

Il s'agit d'une série contenant pour chaque variable la moyenne des valeurs de cette variable.

On applique la méthode `fillna()` sur les trois colonnes qui nous intéressent, en donnant comme valeur à `fillna()` la moyenne de chaque variable. Ainsi, la méthode remplace dans chaque colonne `Age`, `Height` et `Weight` les valeurs manquantes par la moyenne des valeurs de la variable.

Et grâce au bout de code :

```
donnees[['Age','Height', 'Weight']] =
```

nous redéfinissons les trois colonnes `Age`, `Height` et `Weight` par les trois colonnes retournées par la méthode `fillna()`, et ne contenant plus de valeurs manquantes.

© Editions ENI - All rights reserved

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal	Name
A Dijiang	1	M	24.0	180.00000	80.000000	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN	A Dijiang
A Lamusi	2	M	23.0	170.00000	60.000000	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN	A Lamusi
Gunnar Nielsen Aaby	3	M	24.0	175.33897	70.702393	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN	Gunnar Nielsen Aaby
Edgar Lindenau Aabye	4	M	34.0	175.33897	70.702393	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold	Edgar Lindenau Aabye
Christine Jacoba Aaftink	5	F	21.0	185.00000	82.000000	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's	NaN	Christine Jacoba Aaftink

Les valeurs de taille et poids à la troisième et quatrième ligne du tableau étaient égales à `NaN` avant le traitement de ces valeurs manquantes. À présent, les valeurs sont `175.33897`, qui est la taille moyenne des athlètes, et `70.702393`, qui est le poids moyen des athlètes.

4.4.2 Gestion des données dupliquées

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

Des informations dupliquées peuvent être des lignes entières dupliquées, c'est-à-dire que, pour une raison inconnue, une ligne a été entrée dans le tableau deux fois, voire plus. C'est ce genre d'erreur qui peut poser problème, il faut donc faire attention et les traiter.

Détection/récupération des lignes dupliquées

La syntaxe générale pour détecter les lignes entièrement dupliquées dans un tableau est la suivante :

```
dataframe[dataframe.duplicated()]
```

Ici, on utilise la méthode `duplicated()` qui retournera une série contenant des booléens permettant de dire si la ligne est dupliquée (`True`) ou non (`False`). Ainsi, la syntaxe présentée retournera l'ensemble des lignes dupliquées, sauf la première occurrence. En effet, la méthode `duplicated()` possède deux options intéressantes, dont voici la syntaxe :

```
dataframe[dataframe.duplicated(subset=["ma_colonne1",ma_colonne2"],
keep='first')]
```

Par défaut, la méthode `duplicated()`, traite les lignes entières, c'est-à-dire l'ensemble des colonnes. Si on souhaite uniquement détecter la duplication sur un sous-ensemble de colonnes, il est possible de donner les noms de colonnes à l'option `subset`.

De même, par défaut, la méthode `duplicated()` garde la première occurrence de lignes dupliquées et supprime les autres, grâce à l'option `keep='first'`. Ainsi, quand on utilise la syntaxe présentée ci-dessus, celle-ci retourne un dataframe contenant l'ensemble des lignes dupliquées (dont la valeur est `True` après `duplicated()`), sauf la première occurrence, que `duplicated()` laisse à `False` pour ne pas perdre l'information. En effet, la suppression de données dupliquées nécessite tout de même de garder une occurrence des données dupliquées. Il est aussi possible de garder uniquement la dernière occurrence des lignes dupliquées avec l'option `keep='last'`. Dans ce cas, c'est la dernière occurrence que la méthode `duplicated()` laisserait à `False` et les autres à `True`.

Exemple pratique

Récupérons les lignes dupliquées dans le tableau `donnees`.

Code

```
donnees[donnees.duplicated()]
```

© Editions ENI - All rights reserved

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal	Name
Dsir Antoine Acket	704	M	27.0	175.33897	70.702393	Belgium	BEL	1932 Summer	1932	Summer	Los Angeles	Art Competitions	Art Competitions Mixed Painting, Unknown Event	NaN	Dsir Antoine Acket
William Truman Aldrich	2449	M	48.0	175.33897	70.702393	United States	USA	1928 Summer	1928	Summer	Amsterdam	Art Competitions	Art Competitions Mixed Painting, Drawings And ...	NaN	William Truman Aldrich
William Truman Aldrich	2449	M	48.0	175.33897	70.702393	United States	USA	1928 Summer	1928	Summer	Amsterdam	Art Competitions	Art Competitions Mixed Painting, Drawings And ...	NaN	William Truman Aldrich
Hermann Reinhard Alker	2777	M	43.0	175.33897	70.702393	Germany	GER	1928 Summer	1928	Summer	Amsterdam	Art Competitions	Art Competitions Mixed Architecture,	NaN	Hermann Reinhard Alker

Ici, l'ensemble des lignes dupliquées avec toutes leurs occurrences, sauf la première, sont retournées. Avec ce résultat, on peut conclure que la ligne avec l'ID `704` est présente deux fois dans le tableau, puisque la méthode `duplicated()` l'affiche une fois, sans la première occurrence. La ligne avec l'ID `2449` est présente trois fois dans le tableau, avec exactement les mêmes valeurs dans l'ensemble des colonnes. La présence de ces lignes dupliquées fausse clairement les résultats, car si on s'intéresse par exemple à la discipline "Art Competitions" et que l'on compte le nombre d'athlètes dans cette discipline, le résultat sera supérieur à la réalité puisqu'il y a des lignes dupliquées. C'est pour cela qu'il est important de nettoyer son jeu de données avant de commencer à l'analyser, pour ne pas faire de fausses conclusions.

Remarque

Attention, ici ce sont vraiment des lignes dupliquées entièrement, donc de l'information dupliquée. Dans ce tableau, on sait que les athlètes peuvent être présents plusieurs fois, car ils ont pu participer à plusieurs JO et/ou plusieurs épreuves, mais pour ces cas précis, l'ensemble de la ligne n'est pas dupliqué, les variables `Year` ou `Event`, au minimum, changent. Donc on ne les récupère pas ici. Le but est réellement de détecter les lignes apportant exactement la même information plusieurs fois.

Pour compter le nombre de lignes dupliquées dans le tableau, il suffit d'utiliser l'attribut `shape` sur notre code précédent. L'attribut `shape` est une variable retournant un tuple contenant la dimension d'un dataframe : (`nombre de lignes`, `nombre de colonnes`).

Code

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

Résultat

```
(1385, 14)
```

Il y a en tout 1 385 lignes dupliquées dans le tableau, sans compter leur première occurrence, et qu'il faudrait supprimer.

Suppression des lignes dupliquées

Pour nettoyer le dataframe, on va vouloir supprimer l'ensemble des lignes dupliquées, en gardant toujours une occurrence de celles-ci.

La syntaxe générale pour faire cette étape de nettoyage est la suivante :

```
dataframe.drop_duplicates(inplace=True)
```

La méthode `drop_duplicates()` va supprimer les lignes dupliquées en prenant en compte l'ensemble des colonnes et en gardant la première occurrence de chaque ligne dupliquée dans le dataframe qu'elle retourne. Cette méthode possède les deux mêmes options que la méthode `duplicated()` : `keep` pour garder la première ou la dernière occurrence et `subset` pour préciser quelles colonnes prendre en compte pour déterminer les lignes dupliquées, la méthode prenant par défaut l'ensemble des colonnes.

En ne précisant pas l'option `inplace`, la méthode `drop_duplicates()` retourne une copie du dataframe d'origine, sans lignes dupliquées. En précisant l'option `inplace=True`, la méthode `drop_duplicates()` supprime les lignes dupliquées directement sur le dataframe d'origine.

© Editions ENI - All rights reserved

Exemple pratique

Supprimons les lignes dupliquées du jeu de données des JO.

Code

```
donnees.drop_duplicates(inplace=True)
donnees.shape
```

Résultat

```
(269731, 14)
```

Il reste 269 731 lignes sur 271 116 : les 1 385 lignes dupliquées détectées précédemment ont bien été supprimées.

4.5 Exploration préliminaire d'un dataframe

Tout comme pour les séries, il existe un ensemble de méthodes et d'attributs très pratiques pour faire une première exploration des données dans son dataframe, comme par exemple l'attribut `shape`, que nous avons vu dans la section précédente. La puissance de Pandas sont les méthodes et les attributs disponibles pour ses structures.

4.5.1 Principaux attributs

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

- `index` : donne la liste des index des lignes (les étiquettes des lignes) du dataframe.

Syntaxe

```
dataframe.index
```

- `columns` : donne la liste des noms des colonnes du dataframe. Nous l'avions utilisé précédemment pour l'ajout de ligne avec la méthode `append()`.

Syntaxe

```
dataframe.columns
```

- `values` : génère un `ndarray` Numpy contenant les valeurs du dataframe.

Syntaxe

```
dataframe.values
```

Rappel : pour connaître le type d'un objet, il faut utiliser la fonction `type()`.

Code

```
type(donnees.values)
```

Résultat

```
numpy.ndarray
```

- `shape` : l'attribut `shape` donne des informations sur la dimension du dataframe. La sortie est un tuple contenant le nombre de lignes à gauche de la virgule et le nombre de colonnes à droite de la virgule.

Remarque

L'ensemble des attributs et méthodes liés aux dataframes sont disponibles dans la documentation officielle : https://pandas.pydata.org/pandas-docs/version/0.21.1/generated/pandas.DataFrame.html

- `dtypes` : retourne le type des données contenues dans chaque colonne du dataframe.

© Editions ENI - All rights reserved

Exemple pratique

Testons ces attributs sur notre dataframe stocké dans la variable `donnees`.

Code

```
donnees.shape
```

Résultat

```
(269731, 14)
```

Après suppression des lignes dupliquées, notre dataframe est constitué de 269 731 lignes et 14 colonnes.

Affichons le type des données dans chaque colonne.

Code

```
donnees.dtypes
```

Résultat

```
ID          int64
Sex        object
Age       float64
Height    float64
Weight    float64
Team       object
NOC        object
Games      object
Year        int64
Season     object
City       object
Sport      object
Event      object
Medal      object
dtype: object
```

Remarque

Pour rappel, `int64` représente des nombres entiers, `float64` des nombres réels et `object` des chaînes de caractères (et parfois, mais pas dans notre cas, des données de types mixtes).

4.5.2 Définition des termes variable, variable quantitative et variable qualitative et découverte de la méthode describe()

La méthode `describe()` génère des statistiques descriptives pour explorer, dans le cas de variables quantitatives, la distribution des données au sein des variables, la tendance centrale (moyenne, médiane, quantiles...), la dispersion des données (écart-type), etc. Elle permet aussi de générer des statistiques descriptives sur des variables qualitatives, avec la fréquence de chaque modalité au sein d'une variable.

Il faut savoir que les statistiques descriptives ne sont pas les mêmes selon qu'on travaille sur des variables **quantitatives** ou **qualitatives**.

Quand on parle de **variable**, il s'agit de chaque colonne du dataframe : `Age` est une variable, `Height` en est une autre, puis `Weight`, etc. Chaque variable décrit chaque individu ou observation du tableau (ici, les individus sont les athlètes, qui sont décrits par leur âge, poids, taille, etc.). On dit qu'une variable est une caractéristique commune à l'ensemble des individus/observations d'une étude et que la valeur de cette variable varie entre les individus/observations.

Ainsi, comme nous le disions, la méthode `describe()` ne se comporte pas de la même manière avec des variables quantitatives (des colonnes du dataframe contenant des valeurs quantifiables, numériques) qu'avec des variables qualitatives, ou catégorielles (des colonnes du dataframe contenant des modalités).

Les variables **quantitatives** permettent de mesurer des quantités, par exemple le poids d'un athlète, sa taille ou encore son âge sont les trois variables quantitatives du tableau. On peut distinguer les variables quantitatives discrètes et les variables quantitatives continues. Les variables quantitatives discrètes ne peuvent prendre qu'un nombre fini de valeurs, des valeurs numériques entières : on dit qu'il est possible de les énumérer (exemple : le nombre d'arbres, le nombre de frères et sœurs, etc.). Au contraire, une variable quantitative continue peut prendre un nombre infini de valeurs, par exemple l'âge, le poids ou la taille des athlètes sont des variables quantitatives continues.

© Editions ENI - All rights reserved

Pour les variables **qualitatives** (les autres variables du tableau des JO), on va mesurer des modalités, par exemple le genre de l'athlète. Dans notre tableau, la variable `Sex` est une variable qualitative qui contient deux modalités, `M` (homme) et `F` (femme). Il en va de même pour la variable `Team`, par exemple, dont les modalités sont les pays, ou encore la variable `Season`, qui a deux modalités, `Summer` (Eté) et `Winter` (Hiver). La variable `Year` (année), qui est un nombre, est toutefois considérée comme une variable qualitative : en effet, on préfère parler de modalités que de valeurs numériques pour les années. Par exemple, l'année 2008 des JO est une modalité de la variable `Year` du tableau.

À présent que les termes de variables, variables quantitatives et variables qualitatives sont correctement définis, revenons à la méthode `describe()`, qui permet d'afficher les statistiques descriptives des variables de notre dataframe.

Voici la syntaxe générale de la méthode `describe()` :

```
dataframe.describe()
```

Appliquons cette méthode sur notre dataframe des JO.

Code

```
donnees.describe()
```

Résultat

	ID	Age	Height	Weight	Year
count	269731.000000	269731.000000	269731.000000	269731.000000	269731.000000
mean	68264.949591	25.458303	175.338957	70.701918	1978.623073
std	39026.253843	6.056530	9.301313	12.606697	29.752055
min	1.000000	10.000000	127.000000	25.000000	1896.000000
25%	34655.500000	22.000000	170.000000	63.000000	1960.000000
50%	68233.000000	25.000000	175.338970	70.702393	1988.000000
75%	102111.000000	28.000000	180.000000	76.000000	2002.000000
max	135571.000000	97.000000	226.000000	214.000000	2016.000000

On constate que, par défaut, la méthode `describe()` est appliquée à chaque variable numérique du dataframe et on obtient ainsi un résumé statistique par variable contenant des données numériques, sur lesquelles on peut effectuer des calculs arithmétiques. La première valeur, `count`, donne le nombre de valeurs dans chaque variable. Ainsi, elle est la même pour toutes. La statistique `mean` correspond à la moyenne des valeurs pour chaque variable et la ligne `std` à l'écart-type (la dispersion des données). La ligne `min` correspond à la valeur minimale dans la colonne et la ligne `max` à la valeur maximale dans la colonne. Puis nous avons les quartiles, qui permettent de diviser les données en quatre parts égales. L'interprétation est la suivante : 25 % des valeurs sont inférieures à la valeur annoncée à la ligne 25 %, 50 % des valeurs sont inférieures à la valeur annoncée à la ligne 50 % et 75 % des valeurs sont inférieures à la valeur annoncée à la ligne 75 %.

Si on utilise la méthode `describe()` sans options particulières, les statistiques descriptives seront effectuées uniquement sur les variables qui représentent des valeurs numériques : l'âge, la taille, ou encore le poids. On constate aussi que la variable `Year` est considérée comme une variable numérique, puisque son type est `int64`.

Si on souhaite que des statistiques descriptives soient faites sur l'ensemble des variables, il faut utiliser l'option `include=all`.

Code

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

Résultat

	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event
count	269731.000000	269731	269731.000000	269731.000000	269731.000000	269731	269731	269731	269731.000000	269731	269731	269731	269731
unique	NaN	2	NaN	NaN	NaN	1184	230	51	NaN	2	42	66	765
top	NaN	M	NaN	NaN	NaN	United States	USA	2000 Summer	NaN	Summer	London	Athletics	Football Men's Football
freq	NaN	195353	NaN	NaN	NaN	17598	18604	13821	NaN	221167	22297	38624	5733
mean	68264.849591	NaN	25.556303	175.338957	70.701918	NaN	NaN	NaN	1978.623073	NaN	NaN	NaN	NaN
std	39026.253843	NaN	6.056530	9.301313	12.606697	NaN	NaN	NaN	29.752055	NaN	NaN	NaN	NaN
min	1.000000	NaN	10.000000	127.000000	25.000000	NaN	NaN	NaN	1896.000000	NaN	NaN	NaN	NaN
25%	34655.500000	NaN	22.000000	170.000000	63.000000	NaN	NaN	NaN	1960.000000	NaN	NaN	NaN	NaN
50%	68233.000000	NaN	25.000000	175.338970	70.702393	NaN	NaN	NaN	1988.000000	NaN	NaN	NaN	NaN
75%	102111.000000	NaN	28.000000	180.000000	76.000000	NaN	NaN	NaN	2002.000000	NaN	NaN	NaN	NaN
max	135571.000000	NaN	97.000000	226.000000	214.000000	NaN	NaN	NaN	2016.000000	NaN	NaN	NaN	NaN

© Editions ENI - All rights reserved

Pour les variables qualitatives, comme `Sex`, `Team`, etc., on constate que les statistiques descriptives comme la moyenne ou le minimum ne sont pas disponibles, ce qui est logique puisque ces statistiques se font sur des valeurs numériques et non sur des catégories (modalités). En revanche, la valeur `count` est donnée pour les colonnes de type numérique ou de type catégorique. Par rapport aux variables numériques, on accède à de nouvelles informations avec les variables catégorielles, qui permettent de décrire ce type de variable. La ligne `unique` donne le nombre de valeurs uniques dans la colonne, par exemple pour la colonne `Sex`, on a deux valeurs possibles, `F` et `M`. Cela nous permet d'avoir un aperçu du nombre de modalités contenues dans chaque variable qualitative. Ensuite, la ligne `top` donne la valeur la plus présente dans la colonne, par exemple pour `Sex`, il s'agit de la lettre `M`, il y a plus de lignes correspondant à des hommes qu'à des femmes dans le tableau des athlètes des JO. Enfin, la ligne `freq` représente la fréquence d'occurrences (ou nombre d'occurrences) de la valeur la plus fréquente dans la colonne. Pour la colonne `Sex`, la fréquence d'occurrence est de `195353`, ce qui signifie qu'il y a 195 353 valeurs `M` dans la colonne sur un total de 269 731 valeurs.

Cette méthode `describe()` est très utilisée lorsqu'on analyse un nouveau tableau, pour obtenir un premier aperçu descriptif des variables du dataframe.

4.5.3 Méthodes de tri d'un dataframe

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

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

```
dataframe.sort_values("Nom_de_colonne", axis = 0,
na_position="last")
```

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

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

```
dataframe.sort_index(axis=0)
dataframe.sort_index(axis=1)
```

La méthode `sort_index()` permet de trier soit sur l'axe 0, dans ce cas on trie sur les étiquettes de lignes, soit sur l'axe 1 et dans ce cas on trie sur les noms de colonnes.

Exemple pratique

Trions le tableau des Jeux olympiques sur les noms de colonnes.

Code

```
donnees.sort_index(axis=1).head()
```

Ici, on utilise la méthode `sort_index()` sur l'axe 1, puis on affiche les cinq premières lignes du dataframe retourné par la méthode `sort_index()`, grâce à la méthode `head()`.

Résultat

Name	Age	City	Event	Games	Height	ID	Medal	NOC	Season	Sex	Sport	Team	Weight	Year
A Dijiang	24.0	Barcelona	Basketball Men's Basketball	1992 Summer	180.00000	1	NaN	CHN	Summer	M	Basketball	China	80.000000	1992
A Lamusi	23.0	London	Judo Men's Extra-Lightweight	2012 Summer	170.00000	2	NaN	CHN	Summer	M	Judo	China	60.000000	2012
Gunnar Nielsen Aaby	24.0	Antwerpen	Football Men's Football	1920 Summer	175.33897	3	NaN	DEN	Summer	M	Football	Denmark	70.702393	1920
Edgar Lindenau Aabye	34.0	Paris	Tug-Of-War Men's Tug-Of-War	1900 Summer	175.33897	4	Gold	DEN	Summer	M	Tug-Of-War	Denmark/Sweden	70.702393	1900
Christine Jacoba Aaftink	21.0	Calgary	Speed Skating Women's 500 metres	1988 Winter	185.00000	5	NaN	NED	Winter	F	Speed Skating	Netherlands	82.000000	1988

On constate que les colonnes sont bien triées par ordre alphabétique à présent.

Trions maintenant sur les valeurs de la colonne `Weight` (poids des athlètes).

Code

```
donnees.sort_values("Weight", axis = 0,
na_position="last").head()
```

© Editions ENI - All rights reserved

On choisit le nom de colonne sur laquelle on veut trier, `Weight`, l'axe 0 pour trier les lignes selon cette colonne et enfin on précise de mettre les valeurs manquantes à la fin du dataframe.

Résultat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
Choi Myong-Hui	21049	F	14.0	135.0	25.0	North Korea	PRK	1980 Summer	1980	Summer	Moskva	Gymnastics	Gymnastics Women's Individual All-Around	NaN
Choi Myong-Hui	21049	F	14.0	135.0	25.0	North Korea	PRK	1980 Summer	1980	Summer	Moskva	Gymnastics	Gymnastics Women's Team All-Around	NaN
Choi Myong-Hui	21049	F	14.0	135.0	25.0	North Korea	PRK	1980 Summer	1980	Summer	Moskva	Gymnastics	Gymnastics Women's Floor Exercise	NaN
Choi Myong-Hui	21049	F	14.0	135.0	25.0	North Korea	PRK	1980 Summer	1980	Summer	Moskva	Gymnastics	Gymnastics Women's Horse Vault	NaN
Choi Myong-Hui	21049	F	14.0	135.0	25.0	North Korea	PRK	1980 Summer	1980	Summer	Moskva	Gymnastics	Gymnastics Women's Uneven Bars	NaN

L'athlète le plus léger est une athlète féminine de 14 ans, qui a un poids de 25 kg, pour une taille de 1 m 35 et qui évoluait dans la discipline Gymnastique durant les Jeux d'été de l'année 1980.

Si on souhaite voir l'athlète le plus lourd, on peut trier par ordre décroissant avec l'option `ascending`. Par défaut, cette option est à `True` (vrai) et le tri est croissant. Nous devons mettre cette option à `False`.

Code

```
donnees.sort_values("Weight", axis = 0, na_position="last",
ascending=False).head()
```

L'athlète le plus lourd des JO pesait 214 kg pour 1 m 83 et évoluait dans la discipline du Judo poids lourds lors des Jeux d'été de l'année 2008.

5. Structure de données Pandas : les panels

Il existe une dernière structure de données dans Pandas appelée `Panel`. Contrairement aux séries et aux dataframes, cette structure est peu utilisée. Toutefois, il est bon qu'elle fasse partie de votre culture générale de Python pour l'analyse de données.

Les panels peuvent être décrits comme des tableaux à trois dimensions, c'est-à-dire qu'ils contiennent plusieurs dataframes, sachant qu'un dataframe peut lui-même contenir plusieurs séries (plusieurs colonnes). En reprenant l'exemple des feuilles Excel, un dataframe correspond à une feuille Excel et un objet de type `Panel` correspond à plusieurs feuilles Excel (donc plusieurs dataframes).

Les panels ne seront pas traités dans ce livre, car ils sont peu utilisés. La structure la plus utilisée est le dataframe et c'est cette structure que nous approfondirons.

6. Manipulation avancée des données avec Pandas

6.1 Les opérations groupby

Il est possible de grouper un dataframe sur une ou plusieurs colonnes puis d'appliquer des fonctions dessus, de manière très simple, grâce à la méthode `groupby()` de Pandas, qui signifie littéralement en français "grouper par". Cette méthode est très puissante et largement utilisée en analyse de données avec Python et sa librairie Pandas.

6.1.1 groupby sur une colonne

La syntaxe globale d'utilisation de `groupby` est la suivante :

```
dataframe.groupby("nom_de_colonne")
```

Ici, un dataframe groupé sur la colonne `nom_de_colonne` est retourné par la méthode `groupby()`. Il est aussi possible d'appliquer directement une fonction de calcul après la méthode `groupby()`. Par exemple, imaginons que `nom_de_colonne` soit une variable qualitative avec des modalités. On pourrait additionner chaque colonne numérique pour chaque modalité de la colonne `nom_de_colonne` et ce très facilement avec `groupby()`.

Syntaxe

```
dataframe.groupby("nom_de_colonne").sum()
```

© Editions ENI - All rights reserved

Ici, `groupby()` retourne un dataframe avec, en index de ligne, chaque modalité de la colonne `nom_de_colonne`, ainsi que la somme de chaque colonne numérique pour chaque modalité de `nom_de_colonne`.

Afin d'illustrer ces propos, prenons un exemple avec notre dataframe des JO. Calculons la moyenne des variables numériques `Age`, `Height` et `Weight` pour chaque modalité de la variable qualitative `Sport`.

Code

```
donnees.groupby("Sport").mean()
```

Résultat

Sport	ID	Age	Height	Weight	Year
Aeronautics	107506.000000	26.000000	175.338970	70.702393	1936.000000
Alpine Skiing	66487.294371	23.226236	173.999252	71.684645	1985.203534
Alpinism	79639.320000	34.040483	175.338970	70.702393	1925.600000
Archery	71015.393316	27.853707	173.559066	70.143818	1987.533847
Art Competitions	67506.074680	41.501821	175.324897	70.762779	1933.756960

Voici un sous-ensemble du dataframe obtenu, sachant qu'il y a en réalité une ligne par sport, que vous pouvez visualiser entièrement dans le notebook associé à ce chapitre. Ici, de manière transparente, la méthode `groupby()` va créer un dataframe pour chaque modalité de la variable `Sport`, donc un dataframe contenant les lignes du sport "boxe", un autre pour le sport "basket", etc. Elle va créer autant de dataframes qu'il y a de modalités dans la variable `Sport` du dataframe de départ. Ces dataframes contiendront la totalité des lignes correspondant à leur sport respectif. Ensuite, on applique sur ces dataframes la méthode `mean()`, qui va calculer la moyenne des valeurs de chaque colonne du dataframe quand cela est possible, c'est-à-dire quand la colonne est numérique. Enfin, la méthode `groupby()` va combiner l'ensemble des dataframes pour n'en générer qu'un, contenant en noms de lignes les modalités de la variable `Sport` et en colonnes la moyenne de chaque colonne numérique du dataframe initial.

On constate que les colonnes `ID` et `Year` du résultat précédent ne sont pas forcément informatives. Si on souhaite uniquement afficher certaines colonnes, il est possible de faire du slicing directement après le calcul de la moyenne.

Code

```
donnees.groupby("Sport").mean().loc[:,
['Age','Height','Weight']]
```

En effet, étant donné que le code :

```
donnees.groupby("Sport").mean()
```

génère un dataframe, on peut directement utiliser l'indexeur `loc` après la fonction `mean()` pour ne sélectionner que les colonnes `Age`, `Height` et `Weight`. Ici, le "." permet de récupérer l'objet créé par la méthode ou la fonction précédant le point et d'y appliquer une nouvelle méthode ou fonction. Cela permet, de manière très efficace, d'enchaîner des actions en Python sur une seule ligne de code.

Résultat

Sport	Age	Height	Weight
Aeronautics	26.000000	175.338970	70.702393
Alpine Skiing	23.226236	173.999252	71.684645
Alpinism	34.040483	175.338970	70.702393
Archery	27.853707	173.559066	70.143818
Art Competitions	41.501821	175.324897	70.762779

On peut ainsi se renseigner très facilement sur la moyenne d'âge, la moyenne de taille ou la moyenne de poids dans chaque sport, grâce à la méthode `groupby()`.

© Editions ENI - All rights reserved

6.1.2 groupby sur plusieurs colonnes

Il est aussi possible de grouper un dataframe sur plusieurs colonnes avec la syntaxe globale suivante :

```
dataframe.groupby(['colonne_1','colonne_2']).sum()
```

Il suffit de donner une liste de noms de colonnes pour que le `groupby()` opère sur plusieurs colonnes.

Groupons notre dataframe des JO selon les modalités de la variable `Sport` et de la variable `Sex`.

Code

```
donnees.groupby(["Sport", "Sex"]).mean().loc[:,['Age','Height','Weight']]
```

Résultat

Sport	Sex	Age	Height	Weight
Aeronautics	M	26.000000	175.338970	70.702393
Alpine Skiing	F	22.334609	169.036674	64.521777
	M	23.784098	177.104175	76.166219
Alpinism	F	43.000000	175.338970	70.702393
	M	33.667170	175.338970	70.702393
...	...	...	...	...
Water Polo	M	25.725843	183.190182	82.024742
Weightlifting	F	24.028078	160.563751	67.724622
	M	25.701704	170.801925	79.883454
Wrestling	F	25.305921	163.865132	60.587837
	M	25.802838	173.522169	74.863361

On obtient ici la moyenne d'âge, de taille et de poids par sport et par genre des athlètes. On constate rapidement que certains sports ne possèdent pas d'équipe féminine, comme par exemple le `Water Polo`, qui ne contient qu'une modalité `M`.

Remarque

Ici, vous pouvez constater qu'en groupant sur deux colonnes, votre dataframe possède deux index. On appelle cela un index multiple : on peut avoir plusieurs index pour les lignes, comme c'est le cas ici, mais aussi plusieurs index pour les colonnes, ce que nous verrons dans la section suivante avec la méthode `aggregate`.

6.1.3 Appliquer plusieurs fonctions avec la méthode groupby et la méthode aggregate

Pour finir, il est possible d'appliquer plusieurs fonctions de calculs plutôt qu'une seule, grâce à la méthode `aggregate`, dont l'alias est `.agg()`. Voici la syntaxe générale :

```
dataframe.groupby('colonne').agg(['median','mean'])
```

Il suffit de donner la liste des fonctions qu'on souhaite appliquer au dataframe groupé par modalité de la `colonne`.

Reprenons notre code `groupby()` précédent, et plutôt que de calculer uniquement la moyenne, calculons aussi la médiane de chaque variable numérique.

Code

```
donnees.groupby(['Sport', 'Sex']).agg(['median','mean']).loc[:,
['Age','Height','Weight']]
```

La moyenne et la médiane sont calculées pour chaque colonne numérique, puis nous récupérons uniquement les colonnes qui nous intéressent : `Age`, `Height` et `Weight`.

Le résultat de ce code est disponible dans le notebook associé à ce chapitre, disponible en téléchargement avec ce livre.

Remarque

Ici, on obtient un index multiple pour les colonnes aussi, le premier index étant les variables `Age`, `Height` *et* `Weight` *et le deuxième étant* `mean` *et* `median`.

© Editions ENI - All rights reserved

Pour finir, il est aussi possible de ne pas appliquer les mêmes fonctions à toutes les colonnes. Par exemple, on peut choisir d'appliquer les fonctions `median` et `mean` à la variable `Age` et les fonctions `median` et `sum` à la variable `Height`, par exemple.

Syntaxe

```
dataframe.groupby('colonne_1').agg({'colonne_2': ['median','mean'],
'colonne_3' : ['sum','mean']})
```

Appliquons cela sur nos trois variables numériques.

Code

```
donnees.groupby(['Sport', 'Sex']).agg({'Age' : ['median','mean'],
'Height' : ['median', 'sum'], 'Weight' : ['mean','median']})
```

Ici, on groupe le dataframe sur deux variables qualitatives, `Sport` et `Sex`, grâce à `groupby()`. Puis, on utilise la méthode `agg()` en définissant qu'on souhaite appliquer les fonctions `median` et `mean` à la variable `Age`, les fonctions `median` et `sum` à la variable `Height` et les fonctions `mean` et `median` à la variable `Weight`.

Résultat

		Age		Height		Weight	
		median	mean	median	sum	mean	median
Sport	Sex						
Aeronautics	M	26.000000	26.000000	175.33897	1.753390e+02	70.702393	70.702393
Alpine Skiing	F	22.000000	22.334609	169.00000	5.743866e+05	64.521777	65.000000
	M	23.000000	23.784098	175.33897	9.618528e+05	76.166219	73.000000
Alpinism	F	43.000000	43.000000	175.33897	1.753390e+02	70.702393	70.702393
	M	30.000000	33.667170	175.33897	4.208135e+03	70.702393	70.702393
...	...	...	...	...	...	...	...

Remarque

Il n'y a pas d'intérêt exploratoire ici. Il s'agit simplement de montrer une ligne de code type pour grouper le tableau sur plusieurs variables et pour générer des statistiques différentes pour chaque variable du tableau.

6.2 Appliquer une fonction à un dataframe avec la méthode apply

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

Syntaxe générale

```
dataframe.apply(fonction, axis=0)
```

Par exemple, imaginons qu'on veuille afficher la moyenne pour les colonnes `Age`, `Height` et `Weight`, en utilisant la fonction `mean` de NumPy.

Code

```
import numpy as np
print(donnees.loc[:,["Age", "Height", "Weight"]]
.apply(np.mean, axis=0))
```

Résultat

```
Age         25.454776
Height     175.338953
Weight      70.701778
dtype: float64
```

De manière rapide, on pourra appliquer des fonctions par ligne ou par colonne.

Mais ce qui est vraiment intéressant avec la méthode `apply()` est de pouvoir créer sa propre fonction à la volée et de l'utiliser directement lors de l'appel de la méthode. Pour cela, on créera ce qu'on appelle une fonction `lambda`, une fonction anonyme, c'est-à-dire une fonction qu'on ne compte utiliser que dans cette commande. Lorsqu'on souhaite utiliser une fonction qu'on utilisera qu'une fois, on fait appel aux fonctions `lambda`, dont voici la syntaxe générale d'utilisation.

Syntaxe

```
dataframe.apply(lambda x: x * 10)
```

© Editions ENI - All rights reserved

Ici, on crée une fonction `lambda` qui va multiplier l'ensemble des valeurs qu'elle reçoit en entrée, `x`, par 10.

Dans notre jeu de données des athlètes, la taille est en centimètres. Imaginons qu'on veuille que cette taille soit en mètres : il faudrait donc diviser les valeurs par 100. Faisons-le avec la méthode `apply()` et une fonction `lambda`.

Code

```
donnees['Height'] = donnees['Height'].apply(lambda x : x / 100)
donnees.head()
```

Résulat

Name	ID	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
A Dijiang	1	M	24.0	1.80000	80.000000	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
A Lamusi	2	M	23.0	1.70000	60.000000	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
Gunnar Nielsen Aaby	3	M	24.0	1.75339	70.702393	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
Edgar Lindenau Aabye	4	M	34.0	1.75339	70.702393	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
Christine Jacoba Aaftink	5	F	21.0	1.85000	82.000000	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

On modifie directement la colonne `Height` en utilisant la méthode `apply()` et une fonction `lambda`, qui va diviser chaque valeur de la colonne `Height` par 100. On obtient ainsi la taille des athlètes en mètres plutôt qu'en centimètres.

En résumé, la méthode `apply()` est très pratique lorsqu'on souhaite appliquer des traitements selon un axe, les colonnes ou les lignes.

6.3 Remodeler/réorganiser des dataframes

6.3.1 Pivotage : la méthode pivot_table

Il existe des méthodes qui permettent d'organiser différemment son dataframe selon des variables qui nous intéressent. La méthode la plus connue est la méthode `pivot_table()`, qui permet de pivoter son dataframe et ainsi de créer des tableaux croisés dynamiques. Le but des tableaux croisés dynamiques est de regrouper, d'agréger un jeu de données selon une ou plusieurs de ses variables qualitatives afin de mieux les étudier.

Pour utiliser cette méthode, il y aura trois options à définir : la variable qu'on souhaite utiliser comme index, la variable qu'on veut voir apparaître en colonne et enfin les valeurs à croiser entre ces variables.

Voici la syntaxe générale d'utilisation de la méthode :

```
dataframe.pivot_table(index='variable3', columns='variable1',
values='variable6')
```

Imaginons que l'on souhaite, pour chaque sport de la variable `Sport` et pour chaque type de médaille (or, argent, bronze) de la variable `Medal`, savoir le genre prédominant des athlètes les ayant obtenus. La méthode `pivot_table()` permet de faire cela facilement en créant un tableau croisé dynamique. Ainsi, dans cet exemple, la colonne `Sport` sera notre option `index`, la colonne `Medal` sera notre option `columns` et enfin la variable `Sex` sera notre option `values`.

Code

```
pd.set_option('display.max_rows', 70)
donnees.pivot_table(index='Sport', columns='Medal', values="Sex",
aggfunc='max')
```

© Editions ENI - All rights reserved

Résultat

Medal Sport	Bronze	Gold	Silver
Aeronautics	NaN	M	NaN
Alpine Skiing	M	M	M
Alpinism	NaN	M	NaN
Archery	M	M	M
Art Competitions	M	M	M
Athletics	M	M	M
Badminton	M	M	M
Baseball	M	M	M
Basketball	M	M	M
Basque Pelota	NaN	M	NaN
Beach Volleyball	M	M	M

Première sous-partie de résultat

Rhythmic Gymnastics	F	F	F
Roque	M	M	M
Rowing	M	M	M
Rugby	M	M	M
Rugby Sevens	M	M	M
Sailing	M	M	M
Shooting	M	M	M
Short Track Speed Skating	M	M	M
Skeleton	M	M	M
Ski Jumping	M	M	M
Snowboarding	M	M	M
Softball	F	F	F
Speed Skating	M	M	M
Swimming	M	M	M
Synchronized Swimming	F	F	F
Table Tennis	M	M	M

Deuxième sous-partie de résultat

La commande `pd.set_option('display.max_rows', 70)` permet de dire à Pandas d'afficher les 70 premières lignes du dataframe. Étant donné qu'il y a 66 sports, cela permet d'afficher le tableau entier en sortie de la méthode `pivot_table()`.

L'option `aggfunc()` donnée dans la méthode `pivot_table` permet de dire quelle fonction d'agrégation on souhaite utiliser. Ici, on sait que ce ne sont pas des données numériques, mais des modalités, donc on demande de nous afficher la valeur qui a le maximum d'occurrences.

Remarque

Si vous mettez dans l'option `values` une colonne numérique, par exemple la variable `Age`, et que vous ne définissez pas d'option `aggfunc`, sachez que, par défaut, Python fera la moyenne des valeurs numériques s'il y en a plusieurs dans une catégorie. Ainsi, en utilisant la variable `Age` à la place de la variable `Sex`, en ne précisant pas l'option `aggfunc`, nous aurions obtenu l'âge moyen des athlètes dans chaque sport pour chaque type de médaille.

Vous pouvez constater que cette méthode permet d'afficher très facilement des résultats complexes et de croiser efficacement les données. Il est grandement recommandé de vous entraîner avec, c'est très puissant. Ici, on constate qu'il y a globalement plus d'athlètes masculins qui gagnent des médailles or, argent et bronze dans les disciplines sportives, que d'athlètes féminines. Toutefois, il y a des disciplines où elles sont majoritaires, comme la gymnastique rythmique, le softball ou encore la natation synchronisée.

6.3.2 Les méthodes stack (empiler) et unstack (désempiler)

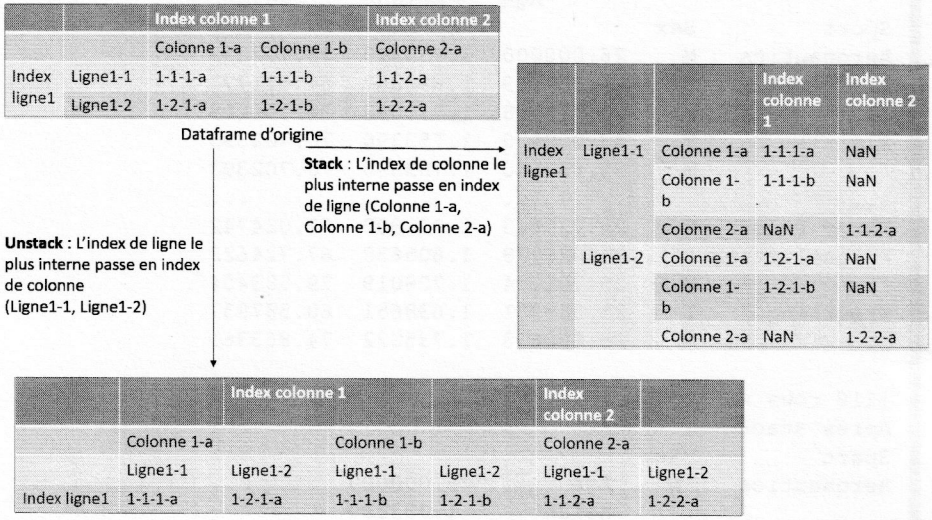
Nous finirons ce chapitre avec les méthodes `stack()` et `unstack()`, souvent utilisées dans le cas de dataframes ayant des index multiples.

La méthode `stack()` permet de transformer l'index de colonne le plus intérieur au dataframe en index de ligne.

La méthode `unstack()` permet de faire le contraire et transforme l'index de ligne le plus intérieur au dataframe en index de colonne, permettant de réorganiser le tableau.

© Editions ENI - All rights reserved

Voici un schéma illustrant l'empilement (*stacking*) et le désempilement (*unstacking*).



La syntaxe d'utilisation de ces deux méthodes est la suivante :

```
dataframe.stack()
dataframe.unstack()
```

Reprenons notre exemple précédent sur les index multiples et testons la méthode `stack()`. Pour cela, on prend le tableau contenant deux index de ligne, `Sport` et `Sex`, et un index de colonne, les noms de colonnes `Age`, `Height` et `Weight`.

Code

```
multi_index=donnees.groupby(["Sport", "Sex"]).mean()
[['Age','Height','Weight']]
print("Sans stack")
print(multi_index)
print("Après stack")
print(multi_index.stack())
```

Dans ce code, on recrée le dataframe initial, on l'affiche, puis on applique la méthode `stack()` dessus, et on affiche le nouveau dataframe créé.

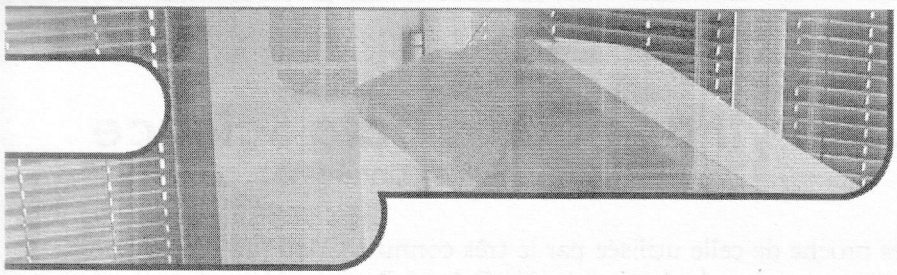
Résultat

```
Sans stack
                          Age    Height     Weight
Sport          Sex
Aeronautics    M     26.000000  1.753390  70.702393
Alpine Skiing  F     22.334609  1.690367  64.521777
               M     23.784098  1.771042  76.166219
Alpinism       F     43.000000  1.753390  70.702393
               M     33.667170  1.753390  70.702393
...                        ...       ...        ...
Water Polo     M     25.725843  1.831902  82.024742
Weightlifting  F     24.028078  1.605638  67.724622
               M     25.701704  1.708019  79.883454
Wrestling      F     25.305921  1.638651  60.587837
               M     25.802838  1.735222  74.863361

[116 rows x 3 columns]
Après stack
Sport          Sex
Aeronautics    M    Age       26.000000
                    Height     1.753390
                    Weight    70.702393
Alpine Skiing  F    Age       22.334609
                    Height     1.690367
                              ...
Wrestling      F    Height     1.638651
                    Weight    60.587837
               M    Age       25.802838
                    Height     1.735222
                    Weight    74.863361
Length: 348, dtype: float64
```

On constate que l'index de colonne le plus interne (`Age`, `Height`, `Weight`) est bien passé en index de ligne.

À ce stade, vous maîtrisez une grande partie des possibilités qu'offre la librairie Pandas. Toutefois, n'hésitez pas à chercher dans la documentation pour en apprendre davantage. Il est impossible de tout répertorier dans ce livre, l'auteur a donc introduit ce qui, selon lui, est le plus important à connaître pour avoir des bases solides de manipulations de données avec Pandas.

© Editions ENI - All rights reserved



Chapitre 6
Maîtriser la librairie Matplotlib

1. Introduction

Matplotlib est une librairie graphique très connue en Python, car elle est puissante, facile à prendre en main et chaque élément de la figure peut être configuré finement via un grand nombre de paramètres disponibles. Il est possible de paramétrer la taille de la figure, sa qualité, les couleurs, les styles, les axes, les titres, etc. De plus, les figures en sortie peuvent être de différents formats comme les formats PNG, PDF, SVG ou encore EPS.

En analyse de données, en plus d'explorer les données avec la librairie Pandas et ses différentes fonctions et méthodes que nous avons vues dans le chapitre précédent, la visualisation est une étape tout aussi importante. Elle va permettre de visualiser et de mettre en avant des informations, des tendances dans ses données. Dans ce chapitre, nous allons voir ensemble les fonctionnalités les plus importantes de Matplotlib et comment les utiliser.

Il faut savoir que la librairie Matplotlib, qui est une librairie orientée objet, est rarement utilisée en tant que telle pour générer les graphiques, car le code à écrire est très lourd. On utilisera un module de la librairie Matplotlib qui fonctionne comme une interface pour cette librairie et qui permet d'alléger grandement l'écriture du code Python.

Ainsi, dans ce chapitre, nous utiliserons le module `pyplot`, qui est une interface pour l'utilisation de Matplotlib et qui permet de créer des figures avec une

syntaxe très proche de celle utilisée par le très connu Matlab (un langage de programmation pour le calcul numérique). Grâce à Pyplot et à ses fonctions simples d'utilisation, nous pourrons créer une figure, créer une zone graphique, créer des axes, tracer des points, etc.

Il faut savoir que Matplotlib, avec son interface Pyplot, est la base de tout graphique en Python. Toutefois, les graphiques générés par défaut ne sont pas très jolis ni modernes. Pour cela, un autre package a été développé, appelé Seaborn, qui se base sur la librairie Matplotlib et qui fournit des fonctions permettant de générer des graphiques à l'allure moderne. Toutefois, Seaborn est donc basé sur Matplotlib et il est donc important de comprendre le fonctionnement de Matplotlib, notamment la création des bases du graphique, comme la figure ou encore les axes, avant de passer à l'utilisation de Seaborn. Ainsi, dans ce chapitre, nous verrons uniquement les graphiques classiques de l'analyse de données avec Matplotlib. Pour des graphiques plus avancés et travaillés, nous les verrons dans le chapitre sur l'utilisation de la librairie Seaborn.

2. Le fonctionnement de Matplotlib

2.1 Architecture de Matplotlib

Avant de commencer à utiliser le module Pyplot, voyons le fonctionnement de Matplotlib et sa définition d'un graphique. Pour commencer, il faut savoir que Matplotlib est organisé en trois couches (*layers* en anglais) empilées les unes sur les autres, dont voici un schéma :

Plus une couche est basse, plus on dit qu'elle est bas niveau. Pour un langage

© Editions ENI - All rights reserved

de programmation, par exemple, quand on dit qu'il est bas niveau cela signifie qu'il est proche de la machine et qu'on devra prendre en compte les adresses mémoires et d'autres caractéristiques de la machine. Cela n'est pas du tout le cas avec un langage haut niveau, où l'on code sans prendre en compte les caractéristiques de la machine. De ce fait, il est plus simple de programmer dans un langage haut niveau que dans un langage bas niveau, mais les possibilités d'optimisation sont restreintes pour le langage haut niveau par rapport au langage bas niveau.

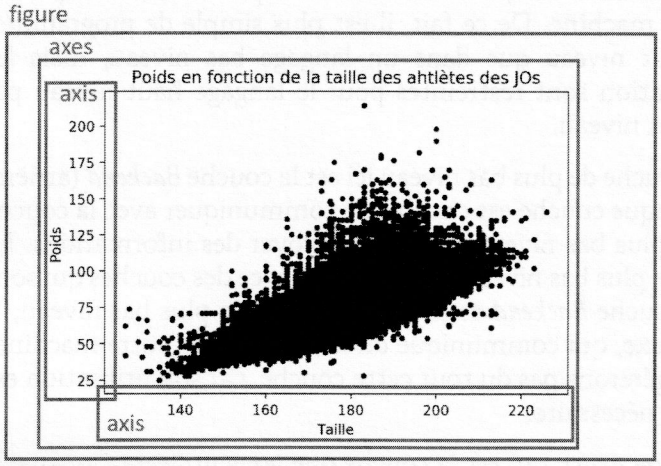
Pour Matplotlib, la couche de plus bas niveau ici est la couche *Backend* (arrière-plan en français). Chaque couche est capable de communiquer avec la couche en dessous d'elle, de plus bas niveau, et de lui envoyer des informations. En revanche, la couche de plus bas niveau n'a pas conscience des couches qui sont au-dessus d'elle. La couche *Backend* est donc la couche de plus bas niveau, la couche la plus complexe, qui communique directement avec votre machine/ordinateur. Nous ne gérerons pas du tout cette couche, car son utilisation est complexe et n'est pas nécessaire.

Ensuite, il y a la couche *Artist*, qui est la couche que vous utiliserez lorsque le module Pyplot ne vous permettra plus de personnaliser le graphique autant que vous le souhaiteriez. Cette couche permet de configurer finement chaque paramètre du graphique, et elle est notamment beaucoup utilisée lorsqu'on souhaite tracer plusieurs graphiques sur une même figure.

Enfin, la couche *Script*, celle que nous utiliserons majoritairement, est la couche de plus haut niveau ; elle permet de faire fonctionner Matplotlib avec une syntaxe de code très similaire à celles de Matlab. Elle comprend un ensemble de fonctions permettant de générer facilement des graphiques. C'est clairement cette couche qui est utilisée le plus fréquemment par les analystes de données.

2.2 Organisation des figures avec Matplotlib

Voici comment est organisée une figure selon Matplotlib :



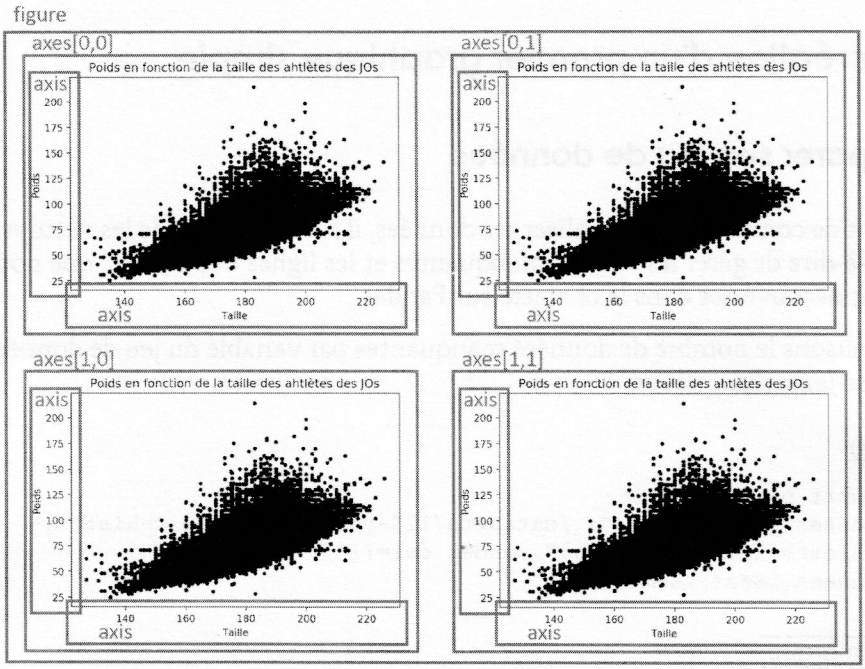
L'objet `figure` peut être représenté comme la feuille blanche sur laquelle on va tracer des graphiques. Cette figure peut contenir un ou plusieurs objets `axes`.

Un objet `axes` (qui signifie axes, au pluriel) représente un graphique, c'est-à-dire qu'il contient les deux axes x et y qui, eux, sont des objets `axis` (qui signifie axe, au singulier). Chaque objet `axes` peut avoir un titre principal et un titre lié aux `axis`. Ainsi, un objet `axes` est en fait une partie d'une figure et il peut y avoir plusieurs `axes`, plusieurs graphiques, par figure.

Les objets `axis` permettent de fixer les limites du graphique et génèrent ce qu'on appelle les *ticks* (marques en français) qui permettent de marquer les axes x et y. Sur l'exemple, les *ticks* pour l'axe x représentant la taille sont `140`, `160`, `180`, `200` et `220`. Ces *ticks* peuvent être définis soit par le dataframe ou la série qu'on donnera en entrée, soit par nous-mêmes, manuellement, lors de l'écriture du code permettant de générer la figure.

© Editions ENI - All rights reserved

Si une figure ne contient qu'un seul graphique, comme sur l'image précédente, alors elle ne contient qu'un seul objet `axes`. En revanche, si la figure contient plusieurs sous-graphiques (*subplots* en anglais), elle contiendra plusieurs `axes` (un `axes` par `subplots`).



Ici, les quatre mêmes graphiques sont représentés sur une figure, permettant de vous illustrer les objets `axes` lorsqu'il y a plusieurs `subplots`. On constate que l'on a quatre objets `axes` car on a quatre graphiques sur la figure.

Nous verrons au cours de ce chapitre comment représenter plusieurs objets `axes` sur une même figure.

Si on utilisait directement Matplotlib, il faudrait créer la figure et les axes avant de pouvoir tracer quoi que ce soit. En utilisant Pyplot, la figure et les axes sont directement créés en accord avec les données qu'on donne en entrée pour tracer le graphique, ce qui est bien pratique. Toutefois, vous risquez souvent de rencontrer les termes *figure*, *axes*, *axis*, *ticks* et *subplots* ; il était donc important de les définir.

Cette introduction peut vous sembler compliquée, avec beaucoup de nouveaux termes. Nous allons donc voir avec des exemples concrets comment créer des graphiques et vous pourrez lire à nouveau ce chapitre, lorsque vous aurez compris la dimension pratique de ces objets, il sera sans doute plus clair.

3. La création d'un premier graphique simple

3.1 Préparer son jeu de données

Avant de commencer à visualiser ses données, il est important de les nettoyer, c'est-à-dire de gérer les valeurs manquantes et les lignes dupliquées, que nous avons découvertes dans le chapitre sur Pandas.

Visualisons le nombre de données manquantes par variable du jeu de données sur les Jeux olympiques.

Code

```
import pandas as pd
donnees=pd.read_csv("../datasets/120-years-of-olympic-history-
athletes-and-results.csv", index_col=[1])
donnees.isna().sum()
```

Remarque

Rappel : pour lire le fichier, dans notre exemple, nous allons le chercher, comme précédemment, dans le dossier `datasets`, qui se trouve un cran plus haut que le dossier `notebooks` dans lequel nous travaillons. Ainsi, le `../datasets` permet de dire à Python de sortir du dossier `notebooks` puis d'aller dans le dossier `datasets` chercher le fichier CSV.

Aperçu du résultat

```
Age        9474
Height    60171
Weight    62875
Medal    231333
```

© Editions ENI - All rights reserved

Il y a quatre variables avec des données manquantes. Il faut savoir que certaines fonctions graphiques ignorent par défaut les individus ou observations contenant des valeurs manquantes, donc sans le savoir, vous travaillez sur un sous-ensemble de vos données. Il existe aussi des fonctions graphiques qui génèrent des erreurs lorsqu'il y a des valeurs manquantes dans la variable qui vous intéresse. Ainsi, il est important de traiter ces valeurs manquantes selon l'objectif de la visualisation.

Pour la variable `Medal` (médaille), les valeurs manquantes signifient que l'athlète n'a pas gagné de médaille. On peut remplacer ces valeurs manquantes par la chaîne de caractères `None`, qui signifie "aucune médaille gagnée". Ainsi, il n'y aura plus de valeurs manquantes dans cette variable et en plus, nous pourrons considérer la valeur `None` comme une modalité de notre variable qualitative `Medal` (si nous laissions les valeurs à `NaN`, ces athlètes seraient tout simplement ignorés).

Pour remplacer des valeurs manquantes par du texte, on utilisera la méthode `fillna()` que nous avons rencontrée dans le chapitre sur la librairie Pandas.

Code

```
import pandas as pd
donnees=pd.read_csv("../datasets/120-years-of-olympic-history-
athletes-and-results.csv", index_col=[1])
donnees['Medal']=donnees['Medal'].fillna("None")
```

Ici, on lit notre jeu de données à partir du fichier CSV, puis on utilise la méthode `fillna()` de Pandas sur la colonne `Medal` de notre `DataFrame` stocké dans la variable `donnees` pour remplacer les `NaN` de cette colonne par la valeur `None`.

Ensuite, trois autres variables contiennent des valeurs manquantes : `Age`, `Height` (taille) et `Weight` (poids). Nous avons deux solutions ici :

- Supprimer les athlètes qui ont une valeur manquante dans au moins une de ces trois variables.
- Calculer la moyenne de chaque variable sur l'ensemble des individus du tableau et remplacer les `NaN` de chaque variable par la valeur moyenne de la variable.

Nous allons créer les deux jeux de données, et les utiliser selon ce qu'on veut étudier.

Si on souhaite visualiser les variables `Age`, `Weight` et/ou `Height`, nous ferions mieux de supprimer les individus avec valeurs manquantes, car en utilisant la moyenne, nous risquons de fausser les résultats. En effet, si le poids moyen est de 50 kg et que les athlètes de la discipline "Lutte libre - Poids lourds" sont ceux pour lesquels le poids est majoritairement une valeur manquante, remplacer leur poids par 50 kg n'est pas le plus pertinent...

En revanche, quand nous voudrons analyser des variables différentes de ces trois-là, par exemple le nombre d'athlètes dans chaque discipline, le nombre d'athlètes ayant obtenu une médaille d'or, etc., il sera judicieux de garder les individus dont une de ces trois variables à une valeur manquante, puisqu'on ne les utilise pas pour l'analyse. Étant donné qu'environ 20 % des valeurs de taille et de poids sont manquantes, ce serait dommage d'ignorer ces athlètes alors que ces valeurs ne sont pas utiles pour l'analyse : dans ce cas, nous allons remplacer les valeurs manquantes par la moyenne de la variable concernée.

Code pour générer les deux dataframes nettoyés :

```
import pandas as pd
donnees=pd.read_csv("../datasets/120-years-of-olympic-history-
athletes-and-results.csv", index_col=[1])
donnees['Medal']=donnees['Medal'].fillna("None")
donnees_NaN_supprimes = donnees.dropna(how="any")
donnees[['Age','Height', 'Weight']] =
donnees[['Age','Height','Weight']].fillna(donnees[['Age','Height',
'Weight']].mean())
donnees_NaN_remplaces=donnees.drop_duplicates()
donnees_NaN_supprimes=donnees_NaN_supprimes.drop_duplicates()
print("Tableau avec NaN remplacés")
print(donnees_NaN_remplaces.shape)
print("Tableau avec suppression des lignes contenant des NaN")
print(donnees_NaN_supprimes.shape)
```

Ce code est un peu long, mais l'ensemble des méthodes utilisées ont été vues dans le chapitre sur la librairie Pandas.

© Editions ENI - All rights reserved

Explication du code : nous importons le tableau dans la variable `donnees`. Ensuite, nous remplaçons les valeurs manquantes de la variable `Medal` par la valeur `None`. Enfin, nous créons une variable nommée `donnees_NaN_supprimes` qui contient le jeu de données sans les athlètes ayant une valeur manquante dans les variables `Height`, `Weight` et `Age`. Ces lignes avec valeurs manquantes ont été supprimées avec la fonction `dropna()` de Pandas, qui permet de dire à Pandas "de ne garder que les lignes pour lesquelles aucune valeur manquante n'est présente". Ensuite, sur le dataframe `donnees` on remplace l'ensemble des valeurs manquantes dans les colonnes `Age`, `Height` et `Weight` par la moyenne de la variable correspondante, puis on supprime les lignes dupliquées et on range ce nouveau dataframe dans la variable `donnees_NaN_remplaces`.

Résultat

```
Dimension du tableau avec les NaN remplacés :
(269731, 14)
Dimension du tableau avec suppression des lignes contenant des NaN :
(206152, 14)
```

Entre les deux dataframes, on supprime 63 579 athlètes, ce qui est conséquent et représente une perte d'informations. Cela confirme qu'il est important de garder les informations de ces athlètes quand on souhaite étudier les variables qualitatives, c'est-à-dire les variables différentes de `Age`, `Weight` et `Height`, tout comme il est important de les ignorer lorsqu'on étudiera l'âge, la taille et le poids, car travailler avec une moyenne pour plus de 60 000 athlètes ne serait pas pertinent.

On a donc deux jeux de données nettoyés :

- `donnees_NaN_remplaces` : pour étudier les variables différentes de `Age`, `Height` et `Weight`.
- `donnees_NaN_supprimes` : pour étudier les variables `Age`, `Height` et `Weight`.

3.2 Créer un nuage de points

Dans ce chapitre, nous allons créer ensemble un premier graphique et l'améliorer au fur et à mesure, afin d'introduire l'ensemble des notions de la librairie Matplotlib et de son interface Pyplot.

Comme énoncé plus haut, au sein de la librairie Matplotlib, c'est en réalité le module Pyplot que nous allons utiliser majoritairement, qui permet de créer les graphiques et de les customiser. En réalité, Pyplot fournit un ensemble de fonctions qui permettent de tracer très facilement des graphiques.

La syntaxe générale pour tracer un graphique est la suivante :

```
import matplotlib.pyplot as plt
plt.figure()
subplot(111)
plt.plot(x,y)
plt.show()
```

Ici, nous importons le module Pyplot de la librairie Matplotlib et nous le renommons avec l'alias `plt`, ce qui est un classique.

Puis on utilise la fonction `figure()`, qui permettra de créer une nouvelle figure, encore vierge à ce moment-là. Cette commande n'est pas nécessaire, en utilisant la commande `plt.plot()`, plus bas, car la figure serait automatiquement créée par Pyplot. Mais il est plus clair ici de l'écrire, pour comprendre ce que l'on fait.

Puis vient la commande `subplot`. Dans l'exemple, le code `subplot(111)` permet de créer une grille de 1*1 puis de placer le graphique à la position 1. Par défaut, Pyplot crée un `subplot(111)`, il n'est donc pas nécessaire d'écrire cette ligne de code. Toutefois, il est ici encore plus clair, de l'écrire pour savoir ce que l'on fait. La fonction `subplot` crée l'objet `axes`, unique ici, puisqu'on a défini qu'on ne voulait qu'un seul graphique sur la figure (grille 1*1).

© Editions ENI - All rights reserved

Ensuite, la fonction `plot()` crée les objets `axis` et trace les points, en prenant les valeurs qu'on lui donne en x (abscisses) et en y (ordonnées). La fonction `plot()` de Pyplot est la fonction graphique la plus basique, qui permet de tracer les points, reliés, sur les axes x et y. Il existe d'autres fonctions permettant de créer d'autres types de graphiques, que nous verrons ensemble plus tard. Cette fonction nous servira de base pour apprendre à utiliser Matplotlib.

Enfin, la fonction `show()` permet d'afficher le graphique courant, ici le graphique généré avec la fonction `plot()`.

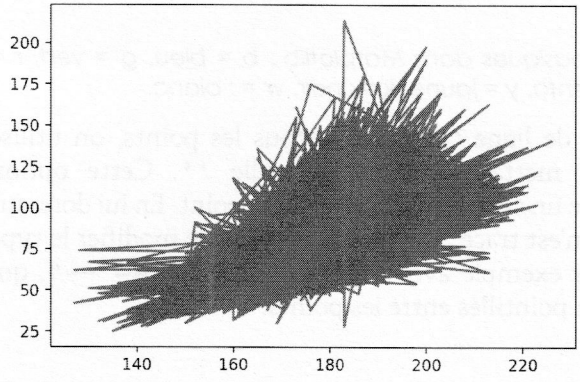
Illustrons cette syntaxe avec un exemple concret.

Nous allons tracer le poids des athlètes en fonction de leur taille, avec l'ensemble des paramètres par défaut. Pour cela, nous utiliserons donc le dataframe stocké dans la variable `donnees_NaN_supprimes`, afin de ne pas fausser les tendances qu'on pourrait observer, entre le poids et la taille.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
plt.show()
```

Résultat



Par défaut, la fonction `plot()` de Pyplot va tracer des lignes entre tous les points, ce qui n'est pas nécessaire ici. Il existe une fonction de Pyplot, nommée `scatter()`, qui permet de tracer des nuages de points sans les relier. Toutefois, avec la fonction `plot()`, il est possible d'indiquer de ne pas relier les points entre eux, grâce à l'option `linestyle`. Il est également possible de changer la forme des points grâce à l'option `marker` et la couleur des points grâce à l'option `color`.

Syntaxe générale

```
import matplotlib.pyplot as plt
plt.figure()
subplot(111)
plt.plot(x,y,marker='o', color="k", linestyle='')
plt.show()
```

Voici quelques paramètres généraux pour commencer à personnaliser son graphique avec Pyplot. Le paramètre `marker` permet de dire quel type de marqueur on veut pour représenter nos données. Par exemple, ici, la valeur `o` permet de dire à Python d'afficher des cercles pleins pour chaque point `x,y`. L'ensemble des `marker` possibles sont disponibles dans la documentation de matplotlib :
https://matplotlib.org/3.1.3/api/markers_api.html#module-matplotlib.markers

Le paramètre `color`, comme son nom l'indique, permet de définir la couleur des points : ici, `k` permet de choisir la couleur noire.

Remarque

Voici la liste des couleurs basiques dans Matplotlib : b = bleu, g = vert, r = rouge, c = cyan, m = magenta, y = jaune, k = : noir, w = : blanc.

Enfin, pour qu'il n'y ait pas de ligne tracée entre tous les points, on utilise l'option `linestyle` en lui mettant une valeur nulle `''`. Cette option `linestyle`, par défaut, trace un trait plein entre chaque point. En lui donnant une valeur nulle, aucun trait n'est tracé. Il est aussi possible de modifier le type de ligne entre les points, par exemple avec l'option `linestyle="--"`, qui permet de tracer des lignes en pointillés entre les points.

© Editions ENI - All rights reserved

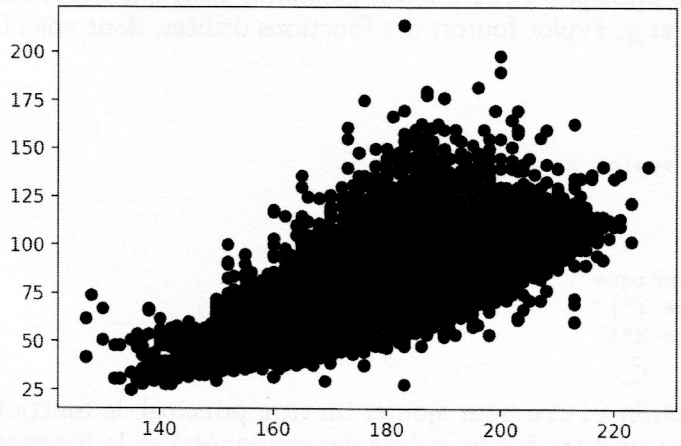
Chaque élément du graphique est finement paramétrable avec les fonctions Pyplot de Matplotlib.

Exemple sur notre jeu de données, en reprenant le graphique de la taille en fonction du poids.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes["Weight"],
marker='o', color="k", linestyle='')
plt.show()
```

Résultat



Remarque

Si vous trouvez la taille des points trop grande, vous pouvez utiliser l'option `markersize` et définir une taille de 1.5 par exemple. Cet exemple est disponible dans le notebook associé à ce chapitre, disponible en téléchargement de ce livre.

Vous pouvez constater qu'avec quelques lignes de code très simples, un graphique complet est généré grâce à Pyplot et à l'ensemble de ses fonctions. Avec ce premier graphique, on est capable de dire qu'il semble y avoir un lien, une corrélation, entre la taille et le poids des athlètes : le poids semble augmenter quand la taille augmente. Clairement, ce n'est pas une grande nouvelle, mais c'est exactement à ça que servent les graphiques et la visualisation de données : à détecter des tendances dans notre dataframe.

Toutefois, ce graphique manque de titres (ou *labels*, en anglais), nous allons donc voir comment en ajouter.

3.3 Ajouter un titre principal et des labels aux axes du nuage de points

Pour ajouter un titre principal, `title`, à son graphique ainsi que des titres, `label`, à ses axes `x` et `y`, Pyplot fournit des fonctions dédiées, dont voici la syntaxe générale.

Syntaxe

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(x,y)
plt.title("Mon graphique")
plt.ylabel("Mon axe Y")
plt.xlabel("Mon axe X")
plt.show()
```

Ici, on utilise la fonction `title` pour ajouter un titre principal, la fonction `ylabel` pour ajouter un titre à l'axe `y` (axe des ordonnées) et la fonction `xlabel` pour ajouter un titre à l'axe `x` (axe des abscisses). `title`, `ylabel` et `xlabel` étant des fonctions, il est possible de les paramétrer finement, comme par exemple changer la taille du texte, la couleur, etc.

© Editions ENI - All rights reserved

Remarque

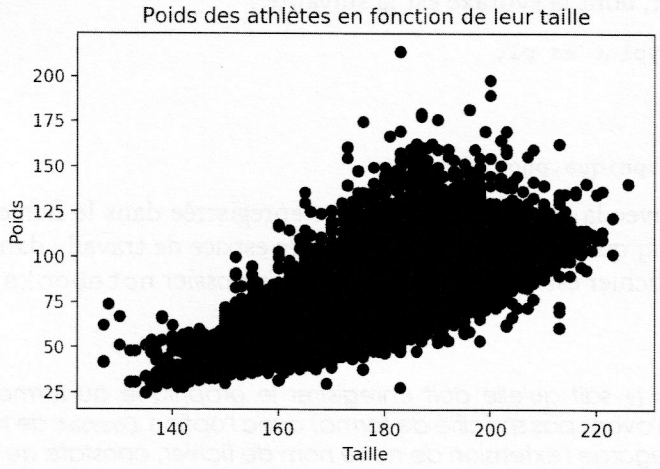
L'ensemble des fonctions (`title`, `ylabel`, `xlabel`...) effectuent leur traitement sur le graphique courant, c'est-à-dire le dernier graphique généré, ici avec la fonction `plot()`. Ainsi, ces fonctions ne fonctionnent que si un graphique courant est actif. Si vous souhaitez supprimer un graphique courant pour repartir sur une nouvelle base, utilisez la fonction `plt.clf()` puis recréez un graphique derrière.

Reprenons notre graphique précédent pour y ajouter des titres.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"],
marker='o', color="black", linestyle='')
plt.title("Poids des athlètes en fonction de leur taille")
plt.ylabel("Poids")
plt.xlabel("Taille")
plt.show()
```

Résultat



Remarque

Au sein des fonctions `title`, `xlabel` et `ylabel`, vous pouvez utiliser l'option `color` pour changer la couleur du texte ou l'option `size` pour changer la taille du texte.

Remarque

Dans cet exemple, l'option `color="black"` est utilisée plutôt que l'option `color="k"`. Ces deux options permettent de colorer les points en noir, mais correspondent à deux syntaxes de couleurs différentes. L'ensemble des couleurs "classiques" de Matplotlib, à donner en option, sont disponibles via le lien :
https://matplotlib.org/3.1.3/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py

3.4 Enregistrer son graphique

Lorsque votre graphique est terminé, il est possible de l'enregistrer dans un fichier pour ensuite l'utiliser dans un rapport ou le partager avec d'autres personnes.

Pour enregistrer un graphique dans un fichier, il faut utiliser la fonction `savefig()` de Pyplot, dont la syntaxe est la suivante :

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(x,y)
plt.savefig('mon_graphique.png')
```

Ici, la figure générée avec la fonction `plot()` est enregistrée dans le fichier `mon_graphique.png` qui est disponible dans votre espace de travail ; dans l'exemple du livre, ce fichier est donc disponible dans le dossier `notebooks`.

Remarque

La fonction `savefig()` sait qu'elle doit enregistrer le graphique au format PNG. En effet, nous n'avons pas spécifié de format avec l'option `format` de la fonction, donc elle regarde l'extension de notre nom de fichier, constate qu'il s'agit de `.png` et crée donc un fichier au format PNG.

© Editions ENI - All rights reserved

Il existe des options particulières pour enregistrer un fichier, dont voici les trois plus importantes :

- `format` : vous pouvez mettre une extension connue à votre fichier (jpg, jpeg, png, pdf, svg, eps...) et la fonction `savefig()` utilisera ce format, ou vous pouvez spécifier cela avec une option, par exemple : `format="png"`.
- `dpi` : permet de spécifier la résolution de l'image. Plus la résolution est élevée, plus l'image générée sera de qualité. Exemple : `dpi=300`.
- `transparent` : pour avoir un fond transparent, il est possible de le spécifier avec l'option `transparent=True`.

Enregistrons notre graphique précédent.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"],
marker='o', color="black", linestyle='')
plt.title("Poids en fonction de la taille des athlètes des Jos")
plt.ylabel("Poids")
plt.xlabel("Taille")
plt.savefig("athlètes.png", dpi=300, format="png")
plt.close()
```

Résultat

Si vous exécutez ce code, ce fichier sera généré dans le dossier où se trouve le notebook à partir duquel vous lancez le code. Ce fichier est de très bonne qualité, en ayant spécifié un dpi à 300, il est ainsi possible de zoomer tout en gardant une très bonne qualité d'image.

Remarque

La fonction `plt.close()` permet de fermer le graphique et de le faire disparaître de la mémoire de votre ordinateur. Cela fait partie des bonnes pratiques après avoir enregistré un graphique.

Remarque

Le DPI est une unité de mesure de la résolution. Plus sa valeur est grande, plus la qualité de l'image sera élevée. Par exemple, si vous souhaitez inclure l'image dans un PowerPoint, une valeur de 150 dpi est suffisante. Pour une impression haute définition, il est couramment choisi d'utiliser 300 dpi.

3.5 Changer la taille de la fenêtre graphique et la résolution de son graphique

Il n'est pas possible de spécifier la taille du graphique lors de l'enregistrement de celui-ci dans un fichier. Il faut le faire lors de la création de la figure, en spécifiant des options dans la fonction `figure()`. C'est cette fonction qui permet de créer une nouvelle figure, et donc elle permet de passer diverses options, dont la taille de la figure à générer et, pourquoi pas, la résolution de celle-ci, en dpi.

Syntaxe générale

```
plt.figure(figsize= [6.4, 4.8], dpi=100, facecolor='w')
```

Ici, ce sont les valeurs par défaut de cette fonction.

La première option, `figsize`, permet de définir la taille de l'image (en *inches*, pouces en français). La valeur à gauche, `6.4`, correspond à la largeur et la valeur à droite, `4.8`, correspond à la hauteur de la figure.

Ensuite, on peut définir l'option `dpi`, qui par défaut, est à `100`.

Enfin, il est possible de changer la couleur de fond du graphique, avec l'option `facecolor`, par défaut en blanc (`w = white`).

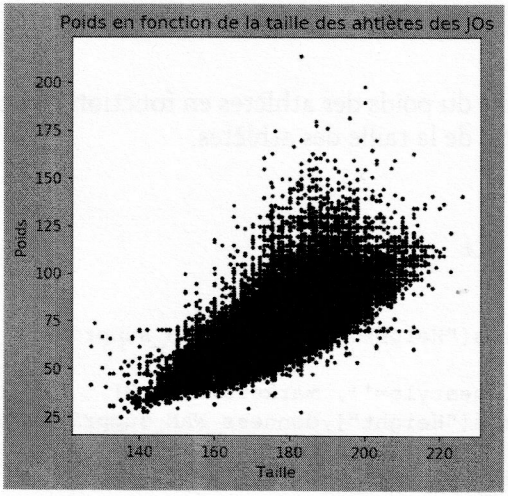
Testons cela sur notre graphique précédent.

© Editions ENI - All rights reserved

Code

```
import matplotlib.pyplot as plt
plt.figure(figsize=[5,5], dpi=150, facecolor='grey')
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"],
marker='o', color="black", linestyle='', markersize=1.8)
plt.title("Poids en fonction de la taille des athlètes des Jos")
plt.ylabel("Poids")
plt.xlabel("Taille")
plt.show()
```

Résultat



On constate bien que la figure est maintenant carrée plutôt que rectangulaire (taille 5 par 5), que le fond est gris et que la résolution est meilleure que précédemment. Dans cet exemple, l'option `markersize` est aussi utilisée pour réduire la taille des points.

3.6 Tracer plusieurs courbes sur un même graphique (sur un même objet axes)

Il est possible de tracer deux courbes représentant des variables différentes, sur un même graphique. Pour cela, il suffit d'utiliser la fonction `plot()` autant de fois que de courbes qu'on souhaite tracer.

Syntaxe

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(x1,y1)
plt.plot(x2,y2)
plt.title("Mon graphique")
plt.show()
```

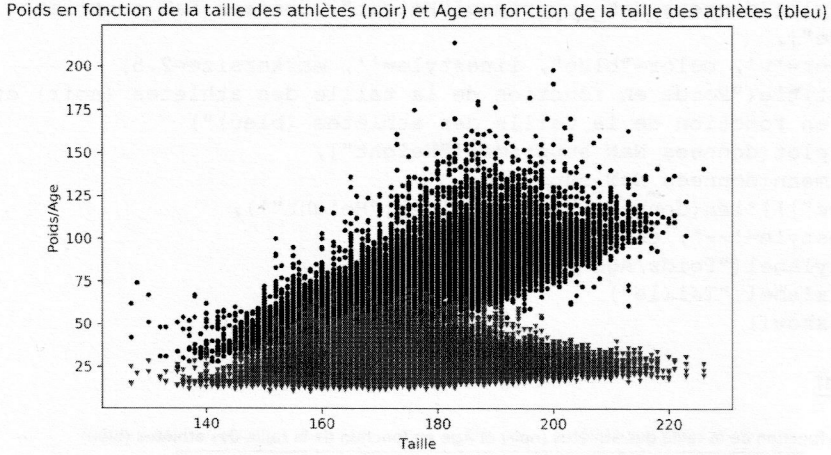
En reprenant le graphique précédent du poids des athlètes en fonction de leur taille, traçons aussi l'âge en fonction de la taille des athlètes.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"],
marker='o', color="black", linestyle='', markersize=2.5)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Age"],
marker='v', color="blue", linestyle='', markersize=2.5)
plt.title("Poids en fonction de la taille des athlètes (noir) et
Age en fonction de la taille des athlètes (bleu)")
plt.ylabel("Poids/Age")
plt.xlabel("Taille")
plt.show()
```

© Editions ENI - All rights reserved

Résultat



Tracer l'âge et le poids sur le même axe `Y`, ici, n'a pas d'intérêt, cela permet juste de vous montrer comment tracer plusieurs courbes sur un même objet `axes`. Pour bien différencier ces deux courbes, on a défini deux options `markers` différentes ainsi que deux options `color` différentes.

Remarque

On constate des athlètes avec un âge supérieur à 75 ans, ce qui est curieux. En lançant la ligne de code `donnees[donnees["Age"]>75]` on constate que ces athlètes participaient à la discipline `Art Competitions` entre 1924 et 1948 et que cela correspondait à de la peinture ou encore de la sculpture. C'est étonnant de nos jours de penser que ces disciplines aient fait partie des Jeux olympiques.

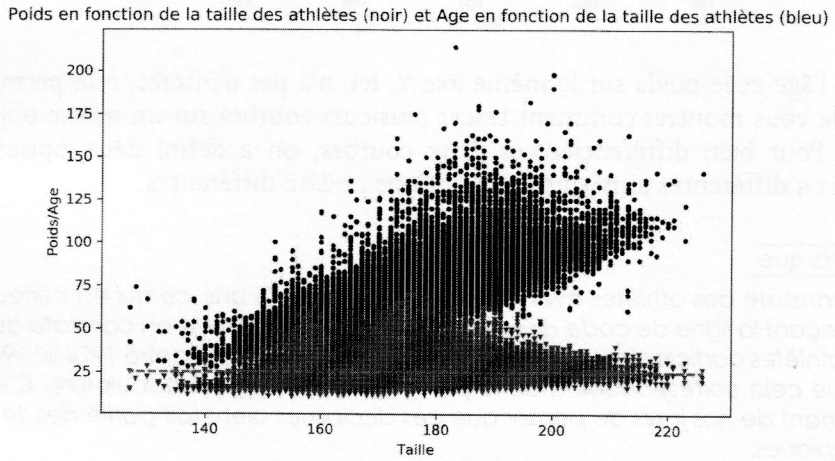
Enfin, on pourrait vouloir rajouter une troisième courbe, par exemple une droite montrant l'âge moyen des athlètes.

Code

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
```

```
["Weight"],
marker='o', color="black", linestyle='', markersize=2.5)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Age"],
marker='v', color="blue", linestyle='', markersize=2.5)
plt.title("Poids en fonction de la taille des athlètes (noir) et
Age en fonction de la taille des athlètes (bleu)")
plt.plot(donnees_NaN_supprimes["Height"],
[np.mean(donnees_NaN_supprimes
["Age"])]*len(donnees_NaN_supprimes["Height"]),
linestyle='--', color="green")
plt.ylabel("Poids/Age")
plt.xlabel("Taille")
plt.show()
```

Résultat



Pour tracer la moyenne de la variable `Age`, nous avons importé la librairie NumPy pour faire le calcul avec sa fonction `mean()`, puis nous avons utilisé la commande suivante :

```
[np.mean(donnees["Age"])]*len(donnees["Height"])
```

© Editions ENI - All rights reserved

Cette commande calcule la moyenne de la variable `Age` puis va dupliquer cette valeur autant de fois que de valeurs présentes dans la variable `Height`. En effet, quand on trace `x` et `y`, il faut que les deux objets aient le même nombre de valeurs, c'est pour cela qu'il faut dupliquer la moyenne autant de fois qu'il y a de valeurs dans la variable `Height` ; cela permet de tracer une droite à la valeur moyenne de l'âge des athlètes, 25 ans.

3.7 Ajouter une légende à son graphique

Pour ajouter une légende à un graphique, il faut spécifier un `label` (une étiquette, en français) pour chaque courbe qu'on souhaite afficher dans la légende. Puis il faut utiliser la fonction `legend()` de Pyplot.

Syntaxe générale

```
import matplotlib.pyplot as plt
plt.plot(x,y, label="mon graphique")
plt.legend(loc="upper right")
```

La fonction `legend()` affiche automatiquement la légende pour l'ensemble des graphiques possédant une option `label`. Si vous ne souhaitez pas afficher l'ensemble des labels dans la légende, il est possible de lui donner la liste des labels à afficher en utilisant les options `handles` et `labels`.

Ici, en option de la fonction, nous indiquons `loc="upper right"`, qui permet de donner la position de la légende, dans ce cas en haut (upper) à droite (right). Il n'est pas obligatoire de spécifier cette option et la fonction `legend()` cherchera l'endroit le plus adapté pour afficher la légende sur le graphique. Toutefois, cela prend plus de temps de calcul, ainsi il est conseillé, si possible, de lui donner la position que vous souhaitez. Les valeurs possibles sont :

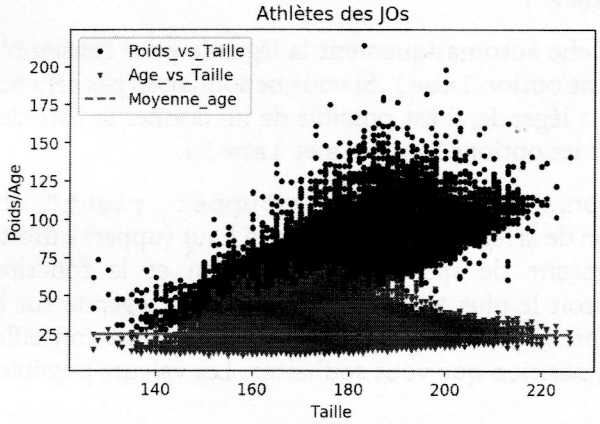
- `upper left` : en haut à gauche
- `upper right` : en haut à droite
- `lower left` : en bas à gauche
- `lower right` : en bas à droite

Reprenons le graphique précédent pour y ajouter une légende.

Code

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"], marker='o', color="black", linestyle='',
markersize=2.5, label='Poids_vs_Taille')
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Age"], marker='v', color="blue", linestyle='', markersize=2.5,
label='Age_vs_Taille')
plt.plot(donnees_NaN_supprimes["Height"],[np.mean(donnees_NaN_
supprimes ["Age"])]*len(donnees_NaN_supprimes["Height"]), linestyle='--',
color="green",label='Moyenne_age')
plt.title("Athlètes des Jos")
plt.ylabel("Poids/Age")
plt.xlabel("Taille")
plt.legend(loc='upper left')
plt.show()
```

Résultat



Dans le code, nous avons mis l'option `label` pour les trois courbes du graphique. Puis nous avons appelé la fonction `legend()` de Pyplot, en spécifiant que nous voulions que la légende soit située en haut à gauche du graphique avec `upper left`. On voit bien que pour chaque courbe avec un label, le nom du label, la forme des points ainsi que la couleur correspondante apparaissent dans la légende.

© Editions ENI - All rights reserved

3.8 Annoter son graphique avec du texte

Il est possible d'annoter son graphique avec du texte. Il est même possible de rajouter du texte à n'importe quelles coordonnées de votre graphique, pour écrire ce que vous voulez. La fonction `annotate()` de Pyplot permet d'annoter votre graphique.

Syntaxe générale de l'annotation d'un graphique

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(x,y)
plt.annotate("votre texte", xy=(1,2), xytext=(0.8,2.2),
arrowprops=dict(facecolor='black'))
plt.title("Mon graphique")
plt.show()
```

La fonction `annotate()` attend plusieurs options.

Tout d'abord, le texte que vous souhaitez afficher sur la figure, qui est le premier paramètre à donner entre parenthèses. Ensuite, les coordonnées du point sur lequel la flèche doit pointer, grâce à l'option `xy` : ici, 1 correspond à la position en `x` et 2 à la position en `y`. Puis, on donne les coordonnées où placer le texte grâce à l'option `xytext`, et enfin, les propriétés de la flèche permettant de relier votre point d'intérêt au texte : cette option, `arrowprops`, est couramment utilisée pour afficher la flèche. L'option `arrowprops` est optionnelle et contient un dictionnaire spécifiant les propriétés de la flèche à afficher. Ici, on ne donne qu'une option, on veut que la flèche soit noire, avec `facecolor`.

Remarque

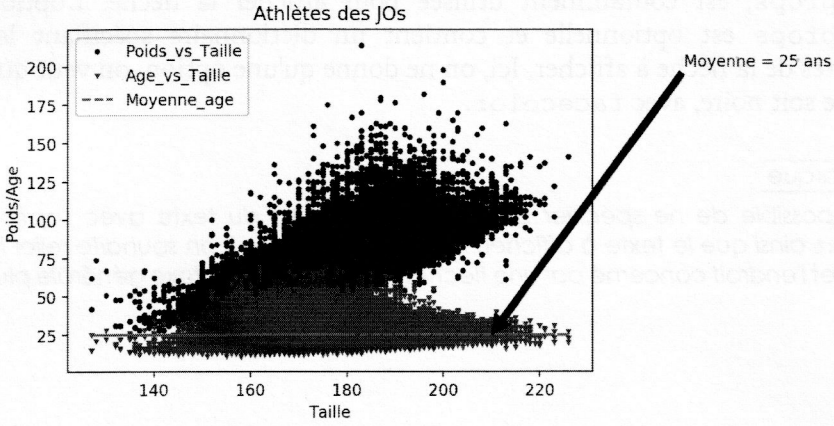
Il est possible de ne spécifier que les coordonnées du texte avec l'option `xytext` ainsi que le texte à afficher. Mais, généralement, on souhaite relier le texte et l'endroit concerné par une flèche, ce que fait la syntaxe générale plus haut.

Reprenons le graphique précédent et ajoutons un texte spécifiant la moyenne d'âge des athlètes.

Code

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"], marker='o', color="black", linestyle='',
markersize=2.5, label='Poids_vs_Taille')
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Age"], marker='v', color="blue", linestyle='', markersize=2.5,
label='Age_vs_Taille')
plt.plot(donnees_NaN_supprimes["Height"],[np.mean(donnees_
NaN_supprimes ["Age"])]*len(donnees_NaN_supprimes["Height"]),
linestyle='--', color="green",label='Moyenne_age')
plt.annotate("Moyenne = 25 ans", xy=(210,25),
xytext=(250,200),arrowprops=dict(facecolor='black'))
plt.title("Athlètes des Jos")
plt.ylabel("Poids/Age")
plt.xlabel("Taille")
plt.legend(loc='upper left')
plt.show()
```

Résultat



© Editions ENI - All rights reserved

Ici, on a pu ajouter une flèche pointant sur la droite de la moyenne d'âge des athlètes et on a pu écrire que la moyenne d'âge des athlètes est de 25 ans.

Il est aussi possible de rajouter des boîtes contenant du texte, toujours à des coordonnées précises, grâce à la fonction `text()` de Pyplot.

Voici la syntaxe générale :

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(x,y)
plt.text(x, y,"votre texte",
size=13,bbox=dict(boxstyle="round",fc="white",
ec="black"))
plt.title("Mon graphique")
plt.show()
```

Ici, on utilise la fonction `text()` en précisant la coordonnée `x,y` où placer ce texte sur la figure. Après avoir donné le texte à écrire, on définit la taille de ce texte avec l'option `size`, puis on définit une boîte à afficher autour du texte grâce à l'option `bbox`.

Comme pour l'option `arrowprops` de la fonction `annotate()`, cette option prend un dictionnaire contenant lui-même des options, comme par exemple ici le style de boîte à afficher (`round`), la couleur de l'intérieur de la boîte (`white` : blanc) et la couleur du contour de la boîte (`black` : noir).

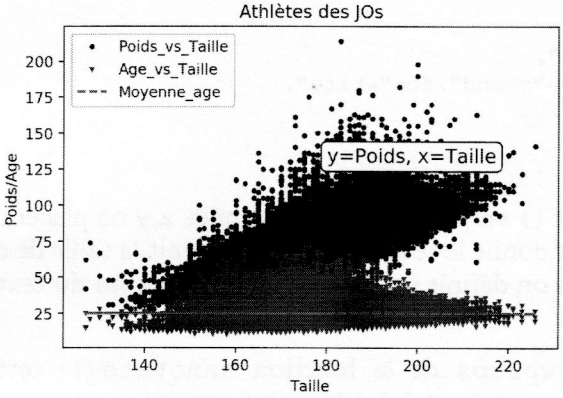
Ajoutons cette boîte à notre graphique précédent.

Code

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"], marker='o', color="black", linestyle='',
markersize=2.5, label='Poids_vs_Taille')
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Age"], marker='v', color="blue", linestyle='', markersize=2.5,
label='Age_vs_Taille')
plt.plot(donnees_NaN_supprimes["Height"],[np.mean(donnees_
NaN_supprimes["Age"])]*len(donnees_NaN_supprimes ["Height"]),
```

```
linestyle='--', color="green",label='Moyenne_age')
plt.text(180,130,"y=Poids,
x=Taille",size=13,bbox=dict(boxstyle="round", fc="white", ec="k"))
plt.title("Athlètes des Jos")
plt.ylabel("Poids/Age")
plt.xlabel("Taille")
plt.legend(loc='upper left')
plt.show()
```

Résultat



Une boîte contenant le texte `y=Poids, x=Taille` a bien été ajoutée aux coordonnées `xy` : 180,130.

Voilà pour les différentes manières d'annoter un graphique avec les fonctions `annotate()` et `text()` de Pyplot.

3.9 Combiner plusieurs graphiques grâce à subplot et subplots

3.9.1 Tracer des sous-graphiques (subplot) sur une ligne ou une colonne

Depuis le début, nous créons une figure contenant un seul `axes` avec la fonction `plt.subplot(111)`. À présent, nous allons voir comment ajouter plusieurs graphiques (plusieurs objets `axes`) à une même figure.

© Editions ENI - All rights reserved

En voici la syntaxe générale :

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(131)
plt.plot(x1,y1)
plt.subplot(132)
plt.plot(x2,y2)
plt.subplot(133)
plt.plot(x3,y3)
plt.show()
```

Sur cette syntaxe générale, on crée une figure avec la fonction `figure()` puis on va créer autant de `subplot` (sous-graphique) que l'on veut grâce à la fonction `subplot()` de Pyplot. Dans cette syntaxe, on veut tracer trois graphiques sur une ligne et trois colonnes. Pour définir cela, on donne comme première valeur à la fonction `subplot()` le nombre de lignes (1), puis le nombre de colonnes (3) et enfin la position du prochain graphique qui sera créé (1). Ensuite, on crée le graphique avec la fonction `plot()`. Puis on crée un deuxième `subplot`, cette fois-ci à la position 2, on crée le deuxième graphique avec la fonction `plot()`. Enfin, on crée un troisième `subplot`, en troisième position, et on trace le troisième graphique.

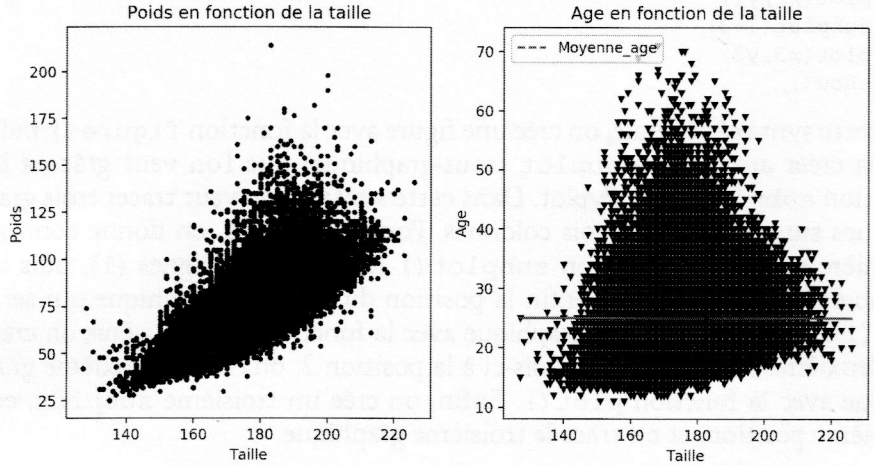
Traçons le graphique du poids en fonction de la taille puis le graphique de l'âge en fonction de la taille, l'un à côté de l'autre.

Code

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize=[10,5])
plt.subplot(121)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"], marker='o', color="black", linestyle='', markersize=2.5)
plt.title("Poids en fonction de la taille")
plt.ylabel("Poids")
plt.xlabel("Taille")
plt.subplot(122)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Age"], marker='v', color="black", linestyle='', markersize=3.5)
plt.plot(donnees_NaN_supprimes["Height"],[np.mean(donnees_
NaN_supprimes["Age"])]*len(donnees_NaN_supprimes
["Height"]), linestyle='--', color="green",label='Moyenne_age')
```

```
plt.title("Age en fonction de la taille")
plt.ylabel("Age")
plt.xlabel("Taille")
plt.legend(loc='upper left')
plt.show()
```

Résultat



Afin que le résultat soit optimal, on demande une taille de figure deux fois plus large que haute, puisqu'on souhaite placer les deux figures l'une à côté de l'autre : cela est fait grâce à l'option `figsize=[10,5]`. Puis on crée un premier `subplot` sur une grille de 1*2 (une ligne et deux colonnes) en position `1` et on trace le graphique du poids en fonction de la taille des athlètes. Enfin, on crée le deuxième `subplot` en position `2` et on trace le deuxième graphique, l'âge en fonction de la taille ainsi que l'âge moyen, sur le même `subplot`.

Remarque

Lorsqu'on crée un `subplot`, l'ensemble des graphiques qu'on crée ensuite avec la fonction `plot()` seront sur ce `subplot`, et ce jusqu'à la création d'un nouveau `subplot`. C'est pour cela que les deux graphiques "âge en fonction de la taille" et "moyenne d'âge des athlètes" sont tracés sur le même `subplot`.

© Editions ENI - All rights reserved

3.9.2 Tracer des sous-graphiques sur plusieurs lignes et plusieurs colonnes

Lorsqu'on souhaite ajouter des `axes` sur une ligne ou sur une colonne, comme précédemment, on utilisera la fonction `subplot()` de Pyplot. Mais celle-ci a ses limites et lorsqu'on souhaite créer des `axes` à la disposition plus complexe, Pyplot devient limitant. C'est à ce moment-là que nous allons passer à la couche inférieure de Matplotlib, la couche *Artist*, que nous avons vue en introduction de ce chapitre. La couche *Artist* permet de personnaliser plus finement notre graphique.

Voici la syntaxe pour créer une figure avec quatre `axes`, sur deux lignes et deux colonnes.

Syntaxe

```
import matplotlib.pyplot as plt
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
axs[0,0].plot(x1,y1)
axs[0,0].set_title("Titre de mon graphique en haut à gauche")
axs[0,0].set_ylabel("Mon Y")
axs[0,0].set_xlabel("Mon X")
axs[0,1].plot(x2,y2)
axs[0,1].set_title("Titre de mon graphique en haut à droite")
axs[0,1].set_ylabel("Mon Y")
axs[0,1].set_xlabel("Mon X")
axs[0,1].legend(loc='upper left')
axs[1,0].plot(x3,y3)
axs[1,0].set_title("Titre de mon graphique en bas à gauche")
axs[1,0].set_ylabel("Mon Y")
axs[1,0].set_xlabel("Mon X")
axs[1,1].plot(x4,y4)
axs[1,1].set_title("Titre de mon graphique en bas à droite")
axs[1,1].set_ylabel("Mon Y")
axs[1,1].set_xlabel("Mon X")
plt.show()
```

Vous pouvez constater que la syntaxe change légèrement de Pyplot, puisqu'on utilise ici les fonctions de Matplotlib directement. Par exemple, la fonction `title()` de Pyplot devient la méthode `set_title()`, la fonction `ylabel()` de Pyplot devient la méthode `set_ylabel()`, etc. De plus, vous pouvez constater que ces méthodes sont directement appliquées sur l'objet de classe `axes` défini avant le "`.`" : par exemple, `axs[0,1]` est un objet de classe `axes`. Cela permet de paramétrer finement chaque objet `axes` créé.

Explications de cette syntaxe :

Tout d'abord, on importe la librairie Pyplot, car on va tout de même utiliser deux de ses fonctions.

Ainsi, on commence par utiliser la fonction `subplots()` de Pyplot. Attention, précédemment chaque `subplot` était créé séparément grâce à la fonction `plt.subplot()` (sans "s"). Ici, on utilise la fonction `plt.subplots()`, qui permet de créer une figure et un ensemble de sous-graphiques, selon le nombre qu'on lui définit. Ici, on lui demande de créer une grille de 2*2 sous-graphiques, soit quatre graphiques sur deux lignes et deux colonnes. Cette fonction retourne deux objets : `fig`, qui correspond à la figure, et `axs`, qui est un tableau d'objets `axes`. Typiquement, avec cette syntaxe, ce tableau contient quatre `axes` sur deux lignes et deux colonnes.

Ensuite, on utilise ce tableau contenant les `axes` pour tracer un graphique sur chacun des objets `axes`. Prenons la création du premier graphique, en haut à gauche (première colonne, première ligne) avec l'objet `axs[0,0]`. Pour avoir accès à cet objet `axes`, on appelle le tableau `axs` et on lui donne l'index `0` à gauche de la virgule pour spécifier qu'on veut la première ligne et l'index `0` à droite de la virgule pour spécifier qu'on veut la première colonne.

Ensuite, on applique les méthodes graphiques liées aux objets de classe `axes` sur `axs[0,0]` en utilisant la méthode `plot()` pour tracer le graphique, puis la méthode `set_xlabel()` pour donner un titre à l'axe `X` et enfin la méthode `set_ylabel()` pour donner un titre à l'axe `Y`. Après ces commandes, le graphique à la position première ligne, première colonne sera créé. Puis on fait la même chose pour créer le graphique à la première ligne, deuxième colonne `axs[0,1]`, puis le graphique à la deuxième ligne, première colonne `axs[1,0]` et enfin le graphique à la deuxième ligne, deuxième colonne `axs[1,1]`.

© Editions ENI - All rights reserved

Remarque

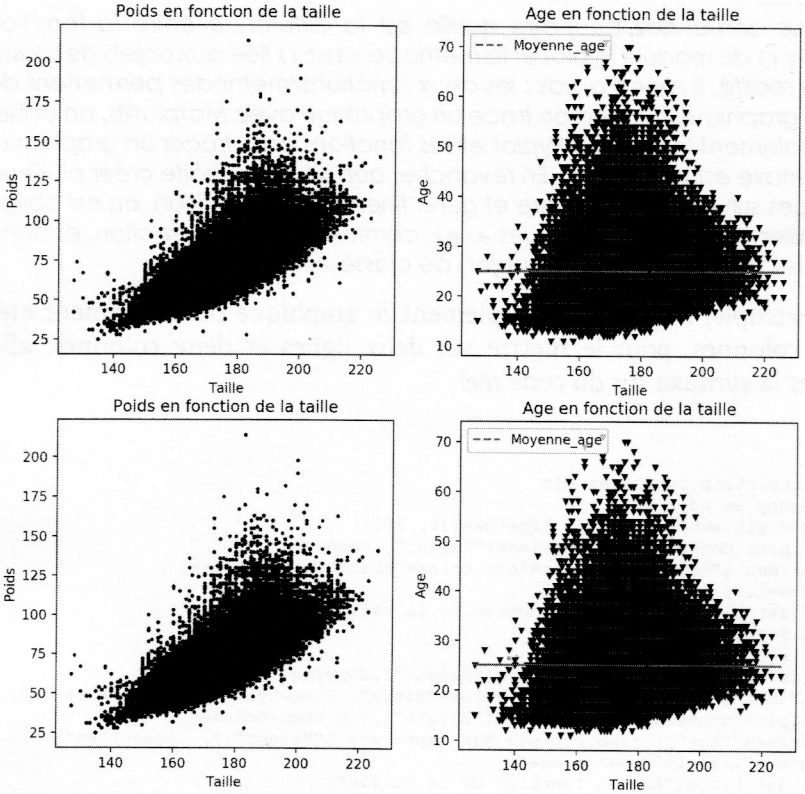
Vous vous demandez peut-être quelle est la différence entre la fonction `plt.plot()` du module Pyplot et la méthode `plot()` liée aux objets de classe `axes`. En réalité, il n'y en a pas ; les deux fonctions/méthodes permettent de tracer le graphique. Quand on trace un graphique avec Matplotlib, on utilisera généralement le module Pyplot et ses fonctions pour tracer un graphique, car la syntaxe est plus simple. En revanche, quand on souhaite créer plusieurs graphiques sur une même figure et gérer finement leur position, on est obligé de travailler directement avec les `axes`, comme dans cette section, et donc d'utiliser les méthodes liées aux objets de classe `axes`.

Comme exemple, dupliquons simplement le graphique précédemment créé sur deux colonnes, pour le mettre sur deux lignes et deux colonnes, afin d'appliquer la syntaxe sur du code réel.

Code

```
import matplotlib.pyplot as plt
import numpy as np
fig, axs = plt.subplots(2, 2, figsize=(10, 10))
axs[0,0].plot(donnees_NaN_supprimes["Height"],donnees_
NaN_supprimes ["Weight"],marker='o', color="black", linestyle='',
markersize=1.8)
axs[0,0].set_title("Poids en fonction de la taille")
axs[0,0].set_ylabel("Poids")
axs[0,0].set_xlabel("Taille")
axs[0,1].plot(donnees_NaN_supprimes["Height"],donnees_
NaN_supprimes ["Age"],marker='v', color="black", linestyle='', markersize=3.5)
axs[0,1].plot(donnees_NaN_supprimes["Height"],[np.mean(donnees_
NaN_supprimes["Age"])]*len(donnees_NaN_supprimes ["Height"]), linestyle='--',
color="green",label='Moyenne_age')
axs[0,1].set_title("Age en fonction de la taille")
axs[0,1].set_ylabel("Age")
axs[0,1].set_xlabel("Taille")
axs[0,1].legend(loc='upper left')
axs[1,0].plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes ["Weight"],marker='o', color="black", linestyle='', markersize=1.8)
axs[1,0].set_title("Poids en fonction de la taille")
axs[1,0].set_ylabel("Poids")
axs[1,0].set_xlabel("Taille")
axs[1,1].plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes ["Age"],marker='v', color="black", linestyle='', markersize=3.5)
axs[1,1].plot(donnees_NaN_supprimes["Height"],[np.mean(donnees_
NaN_supprimes["Age"])]*len(donnees_NaN_supprimes ["Height"]), linestyle='--',
color="green",label='Moyenne_age')
axs[1,1].set_title("Age en fonction de la taille")
axs[1,1].set_ylabel("Age")
axs[1,1].set_xlabel("Taille")
axs[1,1].legend(loc='upper left')
plt.show()
```

Résultat



On obtient bien nos quatre graphiques sur deux lignes et deux colonnes, de cette manière.

3.9.3 Incruster un objet axes dans un autre

Il est possible d'incruster un objet `axes` dans un autre, toujours en passant par la couche *Artist*. C'est-à-dire que sur un objet `axes`, on mettra un autre objet `axes`, de taille plus petite. Cet exemple est complexe mais permet vraiment de voir la nécessité de repasser sur une couche plus basse que la couche de *Script* pour pouvoir personnaliser très finement son graphique.

© Editions ENI - All rights reserved

Syntaxe

```
import matplotlib.pyplot as plt
fig, axe_principal = plt.subplots()
axe_principal.plot(x1,y1)
axe_principal.set_title("Titre de mon graphique en haut à gauche")
axe_principal.set_ylabel("Mon Y")
axe_principal.set_xlabel("Mon X")
axe_incruste=fig.add_axes([l,b,w,h])
axe_incruste.plot(x2,y2)
axe_incruste.set_title("Mon titre")
plt.show()
```

Ici, on crée une figure avec la fonction `subplots()` sans préciser le format de la grille, donc par défaut la figure créée contient un objet `axes`, sur une ligne et une colonne, appelé `axe_principal`. On trace le graphique sur cet axe principal, on lui donne un titre ainsi que des étiquettes à ses axes x et y. Ensuite, on crée un nouvel `axes` dans notre objet figure `fig` grâce à la méthode `add_axes()`, disponible pour les objets de classe `figure` dans la librairie Matplotlib. Ce nouvel `axes` est un rectangle, ainsi la méthode `add_axes()` attend quatre options :

- `l` = `xmin` et `b` = `ymin` : elles définissent les coordonnées du coin en bas à gauche du rectangle. Toutefois, attention, ce sont des coordonnées relatives, nous verrons cela dans l'exemple ci-dessous.
- `w` : la largeur relative du rectangle.
- `h` : la hauteur relative du rectangle.

Une fois le nouvel objet `axes` créé, on trace le graphique dessus, on lui donne un titre et on affiche la figure contenant les deux `axes` grâce à la fonction `plt.show()`.

Remarque

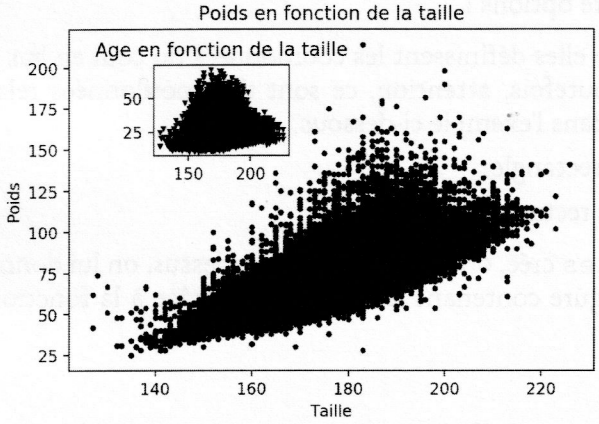
Concrètement, la fonction `plt.subplot()` et la méthode `fig.add_axes` font la même chose : elles permettent d'ajouter un axe à la figure. Seulement, la méthode `add_axes()` de l'objet de classe `figure` permet de placer l'objet `axes` où on le souhaite sur la figure : dans l'objet `axes` existant, à côté, en dessous, etc. La méthode `add_axes()` apporte une liberté supplémentaire, lorsqu'on veut faire des choses plus fines sur son graphique, tout ceci grâce à la couche Artist.

Voyons ça ensemble avec un exemple concret.

Code

```
import matplotlib.pyplot as plt
fig, axe_principal = plt.subplots()
axe_principal.plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes["Weight"],marker='o', color="black", linestyle='',
markersize=2.5)
axe_principal.set_title("Poids en fonction de la taille")
axe_principal.set_ylabel("Poids")
axe_principal.set_xlabel("Taille")
axe_incruste=fig.add_axes([0.25,0.6,.2,.2])
axe_incruste.plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes["Age"],marker='v', color="black", linestyle='',
markersize=2.5)
axe_incruste.set_title("Age en fonction de la taille")
plt.show()
```

Résultat



Ici, on crée notre objet `axes` principal, en traçant le poids des athlètes en fonction de leur taille. Puis, on ajoute un nouvel objet `axes` à la figure, en lui spécifiant ses coordonnées. Les coordonnées qu'on donne sont relatives, ainsi les quatre valeurs qu'on donne entre parenthèses doivent être comprises entre 0 et 1.

© Editions ENI - All rights reserved

Par exemple, pour les options `l` et `b`, qui déterminent la coordonnée du coin bas gauche de l'objet `axes`, si on met `0, 1`, le nouvel objet `axes` sera placé au-dessus de l'`axes` principal, en haut à gauche. Si on met `1, 1`, l'`axes` sera placé au-dessus de l'objet `axes` principal, en haut à droite. Si on met `1, 0`, le nouvel objet `axes` sera placé en dessous de l'objet `axes` principal, à droite, etc. Ici, pour que le graphique "Age en fonction de la taille" soit incrusté dans l'objet `axes` principal, nous avons donné les positions `x=0.25` et `y=0.6` du nouvel objet `axes`, ce qui place le nouvel objet `axes` dans l'objet `axes` principal. Puis on donne la taille du nouvel objet `axes`, en faisant en sorte que la largeur et la hauteur fassent globalement la taille de 20 % de la figure (`0.2`).

Remarque

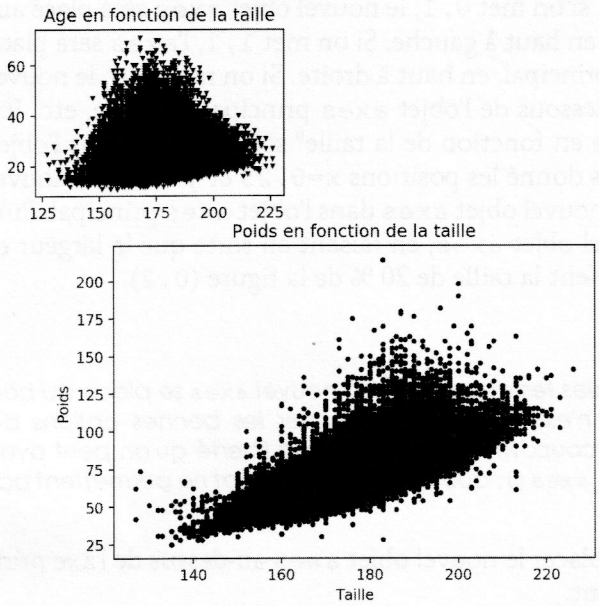
Il faut souvent faire quelques tests avant que le nouvel `axes` se place au bon endroit sur la figure. Ce n'est pas inné de trouver les bonnes options de coordonnées du premier coup. Mais cela montre la liberté qu'on peut avoir avec cette méthode `add_axes()`, que les fonctions Pyplot ne permettent pas d'avoir.

Imaginons qu'on souhaite placer le nouvel objet `axes` au-dessus de l'axe principal, le code serait le suivant.

Code

```
import matplotlib.pyplot as plt
fig, axe_principal = plt.subplots()
axe_principal.plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes["Weight"],marker='o', color="black", linestyle='',
markersize=2.5)
axe_principal.set_title("Poids en fonction de la taille")
axe_principal.set_ylabel("Poids")
axe_principal.set_xlabel("Taille")
axe_incruste=fig.add_axes([0,1,.4,.4])
axe_incruste.plot(donnees_NaN_supprimes["Height"],donnees_NaN_
supprimes["Age"],marker='v', color="black", linestyle='',
markersize=2.5)
axe_incruste.set_title("Age en fonction de la taille")
plt.show()
```

Résultat



Ici, le graphique "Age en fonction de la taille des athlètes" apparaît bien au-dessus du graphique "Poids en fonction de la taille des athlètes", grâce à la coordonnée `0,1`. De plus, on constate qu'il est plus grand, étant donné qu'on a indiqué une taille relative de `0.4`.

4. Les différents types de graphes

4.1 Types de graphiques selon les types de variables (quantitatives et qualitatives)

Nous l'avions vu précédemment avec la librairie Pandas et la méthode `describe()`, les statistiques descriptives ne sont pas les mêmes sur les variables quantitatives (valeurs quantifiables) et les variables catégorielles ou qualitatives (des colonnes de dataframe contenant des modalités).

© Editions ENI - All rights reserved

Nous avions pu constater que dans notre tableau se trouvaient trois variables quantitatives (âge, poids, taille) et le reste des variables étaient qualitatives (sexe, équipe, saison, etc.).

Il en va de même pour les types de graphiques. On ne peut pas utiliser les mêmes types de graphiques sur des variables qualitatives et quantitatives. Par exemple, les nuages de points que nous tracions précédemment se faisaient sur les variables âge, poids et taille. Cela n'est pas anodin, un nuage de points ne peut que se faire sur des variables quantitatives.

Nous allons voir dans ce chapitre quelques types de graphiques disponibles dans Matplotlib et les types de variables qu'ils peuvent représenter. Nous verrons un ensemble de graphiques plus complet dans le chapitre sur l'utilisation de la librairie Seaborn.

Ici, nous allons voir les trois types de graphiques les plus utilisés en analyse de données pour avoir une première vision de ses données : les *scatterplots* (nuages de points), les *bargraphs* (diagrammes à barres) et les *boxplots* (boîtes à moustaches).

4.2 Scatterplot

Le *scatterplot* (ou nuage de points en français) est un type de graphique très utilisé pour représenter deux variables quantitatives. En fait, précédemment, nous avons créé un nuage de points avec la fonction `plot()`, en retirant les lignes qui reliaient les points.

Le but de ce type de graphique est d'avoir deux axes x et y quantitatifs, avec des nombres, et de placer des points aux coordonnées x et y. Il existe une fonction dédiée pour créer des *scatterplots*, qui s'appelle `scatter()`. En clair, si vous souhaitez représenter deux variables quantitatives sur un graphique, notamment pour essayer de trouver une relation entre ces deux variables, le nuage de points est le graphique à utiliser.

La syntaxe générale de cette fonction est la suivante :

```
import matplotlib.pyplot as plt
plt.scatter(x, y)
```

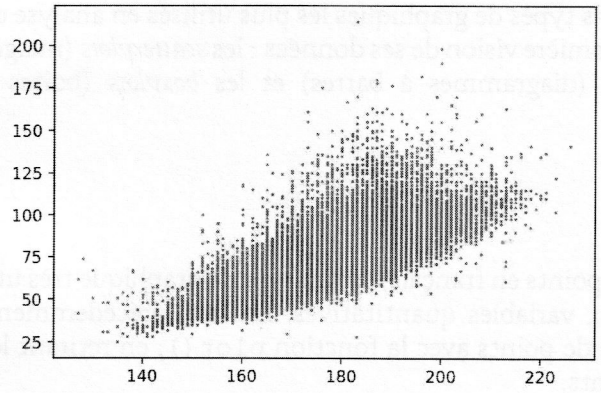
Reprenons les deux variables quantitatives précédentes, la taille et le poids.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.scatter(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"],s=0.5)
plt.show()
```

On importe le module Pyplot de Matplotlib, on crée une nouvelle figure, on trace le poids en fonction de la taille et on choisit une taille de point avec l'option `s` (précédemment, il s'agissait de l'option `markersize` pour la fonction `plot()`).

Résultat



On obtient le même graphique que précédemment, avec un visuel toutefois légèrement différent. Imaginons qu'on souhaite y ajouter des informations de couleur : par exemple, on pourrait colorer les points selon le genre de l'athlète. Avant même de tracer cette information, on s'attend à ce que les tailles les plus petites correspondent aux femmes et les tailles les plus grandes aux hommes. Vérifions cela ensemble sur notre jeu de données des JO.

© Editions ENI - All rights reserved

Voici le code pour colorer les points :

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
colors = {'F':'red', 'M':'blue'}
plt.scatter(donnees["Height"],
donnees["Weight"],c=donnees['Sex'].apply(lambda x:
colors[x]),s=0.5)
plt.show()
```

Dans ce code, après avoir généré la figure et le sous-graphique sur une grille 1*1, on définit un dictionnaire de couleurs : `colors`. Il y a une clé `F` associée à la valeur `red` et une clé `M` associée à la valeur `blue`.

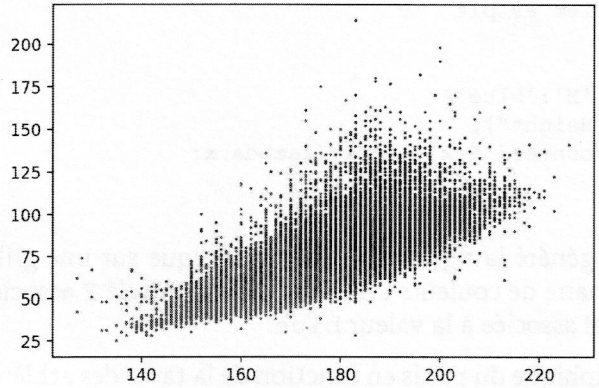
Ensuite, on trace notre graphique du poids en fonction de la taille des athlètes en ajoutant une nouvelle option : `c`. Cette option permet de définir une couleur pour chaque point tracé. Au sein de cette option, on utilise la méthode `apply()` avec une fonction lambda créée à la volée, comme nous l'avions vu dans le chapitre sur la librairie Pandas.

On applique la méthode `apply()` sur la colonne `Sex` de notre dataframe et on transforme chaque valeur de la colonne `Sex` par le nom de la couleur : grâce à la fonction lambda et au dictionnaire de couleurs, les valeurs `F` deviendront `red` et les valeurs `M` deviendront `blue`. Ainsi, l'option `c` contient une série de valeurs `red` et `blue` comme suit :

```
Name
A Dijiang                  blue
A Lamusi                   blue
Christine Jacoba Aaftink    red
Christine Jacoba Aaftink    red
Christine Jacoba Aaftink    red
                           ...
Andrzej ya                 blue
Piotr ya                   blue
Piotr ya                   blue
Tomasz Ireneusz ya         blue
Tomasz Ireneusz ya         blue
Name: Sex, Length: 206152, dtype: object
```

De cette manière, `scatter()` connaît la couleur à attribuer à chaque point du nuage de points.

Résultat



On constate bien que la couleur rouge correspondant aux femmes se situe à gauche du nuage de points, vers les tailles et les poids les plus petits, et que la couleur bleue se trouve le plus à droite, pour des tailles et poids plus grands.

4.3 Graphique à barres (bargraph)

4.3.1 Graphique à barres simple

Le *bargraph* (ou graphique à barres, en français) est le graphique le plus utilisé pour représenter des variables qualitatives. Nous allons pouvoir explorer ensemble le jeu de données et répondre à quelques questions grâce à ce type de graphique.

Le principe du *bargraph* est que chaque valeur *x* correspond aux modalités d'une catégorie : par exemple, pour la variable qualitative `Sport`, l'axe `x` comprendra les valeurs `Athletics`, `Swimming`, `Shooting`, etc., et la valeur `y` correspondant à chaque `x` sera le nombre de fois où cette modalité `x` est présente dans la variable qualitative choisie. Ce sera plus clair avec un exemple, qui arrive tout de suite après.

© Editions ENI - All rights reserved

Tout d'abord, voyons la syntaxe générale de génération d'un graphique à barres avec Pyplot :

```
import matplotlib.pyplot as plt
plt.bar(x, y)
```

Le `x` correspond aux modalités de la variable qualitative choisie et `y` correspond au nombre de fois où la modalité est présente dans le dataframe. Imaginons qu'on souhaite étudier les valeurs présentes dans la variable `Team`, pour par exemple savoir quelles sont les dix équipes des JO ayant eu le plus d'athlètes depuis le début des JO.

Il faut, pour chaque modalité de la variable `Team` (`United States`, `France`, `Italy`...), compter combien de fois cette valeur est présente dans la variable en question. Pour cela, on peut utiliser la méthode `value_counts()` de Pandas. Nous utiliserons cette fois-ci le dataframe stocké dans la variable `donnees_NaN_remplaces`, ce qui nous permettra de visualiser les informations sur l'ensemble des athlètes ayant participé aux JO, avec ou sans leurs données d'âge, taille et poids.

Code

```
import matplotlib.pyplot as plt
counts_variable=
donnees_NaN_remplaces['Team'].value_counts()[:10]
print(counts_variable)
```

Avec ce code, on sélectionne la variable qualitative souhaitée, `Team`, puis on utilise la méthode `value_counts()` sur cette série, pour compter le nombre d'occurrences de chaque modalité dans cette variable. Enfin, on sélectionne les dix premières valeurs générées en effectuant du slicing `[:10]` directement sur la série générée par la méthode `value_counts()` : on sélectionne les valeurs aux positions 0 à 9. On obtient donc une série avec les dix équipes possédant le plus d'athlètes depuis le début des jeux, ainsi que le nombre d'athlètes.

Résultat

```
United States    17598
France           11817
Great Britain    11264
Italy            10213
Germany           9230
Canada            9226
Japan             8269
Sweden            8004
Australia         7512
Hungary           6492
Name: Team, dtype: int64
```

Créons notre *bargraph* à partir de cette série. Ce graphique sera généré après création d'une figure avec Pyplot et nous pourrons y ajouter des titres et annotations grâce aux fonctions Pyplot que nous avons vues précédemment (`ylabel()`, `title()`...).

Code

```
import matplotlib.pyplot as plt
import pandas as pd
plt.figure()
plt.subplot(111)
labels_X=donnees_NaN_remplaces['Team'].value_counts()[:10].keys()
valeurs_Y=donnees_NaN_remplaces['Team'].value_counts()[:10].values
plt.bar(labels_X, valeurs_Y)
plt.xticks(labels_X,rotation=90)
plt.ylabel("Nombre d'athlètes")
plt.title("Top 10 des équipes avec le plus d'athlètes")
plt.show()
```

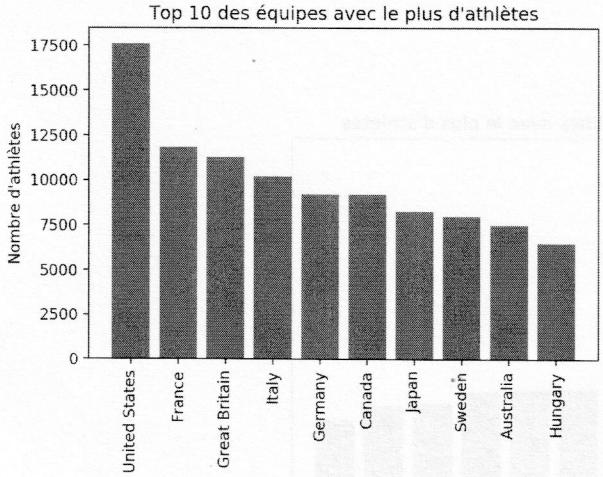
Ici, on charge le module pyplot ainsi que la librairie Pandas pour pouvoir utiliser la méthode `value_counts()`. On crée la figure et le sous-graphique.

Ensuite, on détermine les étiquettes (labels) qu'on veut en `X`, c'est-à-dire les modalités de notre variable qualitative. Pour cela, on utilise la méthode `value_counts()` pour compter le nombre d'athlètes par modalités de la variable `Team` et on garde les dix équipes ayant le plus d'athlètes. Enfin, on utilise la méthode `keys()` pour récupérer les index de la série générée, c'est-à-dire les noms des dix équipes avec le plus d'athlètes, qui seront nos étiquettes de l'axe `X`.

© Editions ENI - All rights reserved

Après avoir déterminé les étiquettes de `X`, on va déterminer les valeurs de `Y`. Pour cela, on réutilise la méthode `value_counts()`, mais cette fois-ci on récupère les valeurs de la série avec l'attribut `values`. Enfin, on trace le graphique à barres avec la fonction `bar()` de Pyplot en donnant les labels `X` et les valeurs `Y`. La fonction `xticks` de Pyplot permet d'indiquer que les labels `X` doivent être affichés verticalement plutôt qu'horizontalement, grâce à l'option `rotation=90` (sinon les textes se chevauchent et c'est illisible).

Résultat



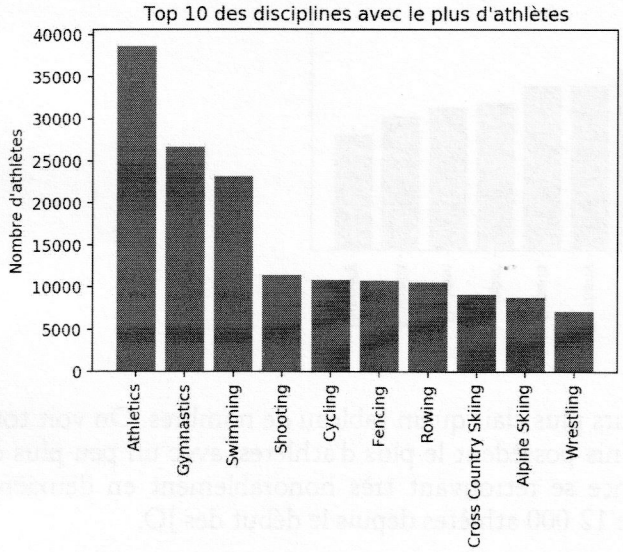
Un graphique est toujours plus clair qu'un tableau de nombres. On voit tout de suite que les États-Unis possèdent le plus d'athlètes, avec un peu plus de 17 500 athlètes, la France se retrouvant très honorablement en deuxième position avec pas loin de 12 000 athlètes depuis le début des JO.

On peut très facilement créer le même graphique pour montrer les dix disciplines avec le plus d'athlètes. Il suffit d'utiliser la variable `Sport` à la place de `Team`.

Code

```
import matplotlib.pyplot as plt
import pandas as pd
plt.figure()
plt.subplot(111)
labels_X=donnees_NaN_remplaces['Sport'].value_counts()[:10].keys()
valeurs_Y=donnees_NaN_remplaces['Sport'].value_counts()[:10].values
plt.bar(labels_X, valeurs_Y)
plt.xticks(labels_X,rotation=90)
plt.ylabel("Nombre d'athlètes")
plt.title("Top 10 des disciplines avec le plus d'athlètes")
plt.show()
```

Résultat



La discipline "Athlétisme" arrive en première position, puis la gymnastique et la natation arrivent respectivement en deuxième et troisième position.

Traduction : *Shooting* : tir, *Cycling* : cyclisme, *Fencing* : escrime, *Rowing* : aviron, *Cross country skiing* : ski de fond, *Alpine skiing* : ski alpin, *Wrestling* : Lutte.

© Editions ENI - All rights reserved

4.3.2 Graphique à barres groupées

Il peut être intéressant de représenter une variable qualitative sur un graphique à barres et de séparer les barres selon une deuxième variable qualitative, pour créer des sous-groupes. On parle aussi de barres groupées, car on va grouper les barres sur une deuxième variable qualitative, ce qui facilite la comparaison. Imaginons qu'on souhaite séparer le nombre d'athlètes dans les dix disciplines les plus populaires par leur genre. Pour cela, nous utiliserons toujours la fonction `bar()` de Pyplot, mais il faudra coder quelques lignes plus complexes pour arriver à ce résultat de barres groupées.

Pour information, la librairie Seaborn permet de créer plus facilement ces barres groupées. Mais étant donné que nous sommes dans le chapitre Matplotlib, nous allons le faire avec, cela nécessitant plus de code et d'intervention manuelle. C'est aussi comme ça que l'on comprend comment fonctionne la librairie Matplotlib, sur laquelle repose Seaborn.

La première étape est de créer un dataframe de deux colonnes, une colonne pour les femmes et une pour les hommes. Chaque ligne correspondra à l'une des dix disciplines les plus populaires et les valeurs seront le nombre d'athlètes, hommes et femmes séparés, dans chaque discipline.

Pour cela, faisons appel à nos connaissances Pandas sur la manipulation de dataframes.

Code

```
sports_10=donnees_NaN_remplaces['Sport'].value_counts()[:10].keys()
donnees_bar=donnees_NaN_remplaces.loc[donnees_NaN_remplaces
['Sport'].isin(sports_10)].groupby(["Sport", "Sex"]).size().unstack(1)
donnees_bar
```

Résultat

Sex Sport	F	M
Alpine Skiing	3398	5431
Athletics	11666	26958
Cross Country Skiing	3385	5748
Cycling	1394	9433
Fencing	2000	8735
Gymnastics	9129	17578
Rowing	2193	8402
Shooting	1724	9724
Swimming	9850	13345
Wrestling	304	6850

La première étape de ce code consiste à déterminer les dix disciplines ayant le plus d'athlètes, grâce à la méthode `value_counts()`, puis on ne garde que les index de cette `Series`, c'est-à-dire les noms des dix sports, avec la fonction `keys()`, qu'on stocke dans la variable `sports_10`.

Contenu de la variable `sports_10` :

```
Index(['Athletics', 'Gymnastics', 'Swimming', 'Shooting',
'Cycling', 'Fencing',
       'Rowing', 'Cross Country Skiing', 'Alpine Skiing',
'Wrestling'],
      dtype='object')
```

Ensuite, on crée un dataframe à partir du jeu de données JO, en ne gardant que les lignes du tableau des JO correspondant à l'un des dix sports sélectionnés précédemment. Les lignes sont sélectionnées grâce à la méthode `isin()` de Pandas, qui permet de ne garder que les lignes pour lesquelles la valeur dans la colonne `Sport` est égale à une des valeurs contenues dans la série `sports_10`.

Une fois les lignes sélectionnées, on groupe ce tableau selon deux variables, `Sport` et `Sex` grâce à la méthode `groupby()`. On obtient un dataframe à deux index, `Sport` et `Sex`.

© Editions ENI - All rights reserved

Enfin, on utilise la fonction d'aggrégation `size()` de Pandas, qui permet de compter le nombre d'éléments pour chaque groupe d'index, donc ici, le nombre d'athlètes femmes en athlétisme, le nombre d'athlètes hommes en athlétisme, le nombre d'athlètes femmes en cyclisme, etc. On obtient donc une série avec deux index, `Sport` et `Sex`, dont les valeurs correspondent au nombre d'athlètes par couple d'index.

Résultat après la fonction d'aggrégation `size()` :

```
Sport                Sex
Alpine Skiing        F       3398
                     M       5431
Athletics            F      11666
                     M      26958
Cross Country Skiing F       3385
                     M       5748
Cycling              F       1394
                     M       9433
Fencing              F       2000
                     M       8735
Gymnastics           F       9129
                     M      17578
Rowing               F       2193
                     M       8402
Shooting             F       1724
                     M       9724
Swimming             F       9850
                     M      13345
Wrestling            F        304
                     M       6850
dtype: int64
```

Enfin, on veut tracer un graphique à barres en représentant les deux variables qualitatives, `Sport` et `Sex`, dessus. Il faut donc avoir, pour chaque sport, une colonne du nombre d'athlètes masculins et une du nombre d'athlètes féminins. Pour cela, il suffit d'utiliser la méthode `unstack()`, que nous avons vue précédemment, qui permet de passer un index de colonne en index de ligne et vice versa.

Ici, on passe la variable `Sex` en nom de colonne, on précise donc de faire cela avec le deuxième index (pour Python, n'oublions pas que l'index 1 signifie : deuxième position). Au final, on obtient le dataframe présenté sur l'image précédente.

Remarque

Cet exemple est intéressant, car vous pouvez voir que les méthodes peuvent être chaînées les unes derrière les autres grâce au ".", qui donne en entrée à la méthode le résultat précédent le ".". Nous aurions pu découper ce code en plusieurs étapes en créant des sous-objets à chaque fois. Mais cette méthode de chaînage permet d'être plus synthétique dans son code, il est bon d'essayer de la maîtriser. N'hésitez pas à télécharger le notebook lié à ce chapitre et à découper ce code, à l'exécuter afin de visualiser le résultat de chaque méthode séparément.

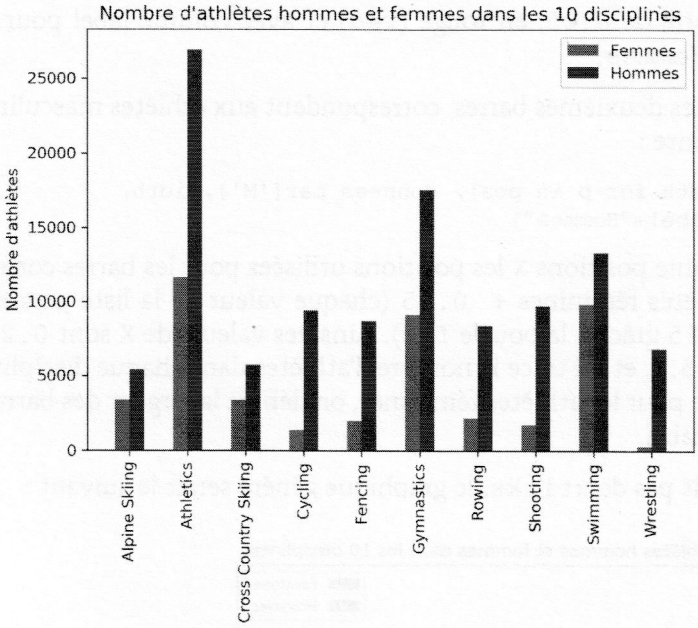
Nous allons utiliser notre dataframe `donnees_bar` pour tracer un graphique de barres groupées.

Code

```
import matplotlib.pyplot as plt
pos = list(range(len(donnees_bar['F'])))
width = 0.25
plt.figure()
plt.subplot(111)
# Créer une barre avec les données F,
# à la position pos
plt.bar(pos, donnees_bar['F'], width, color='red',label="Femmes")
# Créer une barre avec les données F,
# à la position pos + 0.25 afin que la barre M soit décalée
à droite de la première barre F
plt.bar([p + width for p in pos], donnees_bar['M'],width,
color='blue', label="Hommes")
plt.xticks([p + 0.125 for p in pos],sports_10,rotation=90)
plt.ylabel("Nombre d'athlètes")
plt.legend()
plt.title("Nombre d'athlètes hommes et femmes dans les 10 disciplines")
plt.show()
```

© Editions ENI - All rights reserved

Résultat



Ce code est lui aussi complexe, nous allons le voir ensemble.

La variable `pos` contient une liste contenant les chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Cette liste a été créée selon la taille de la colonne `F` du dataframe `donnees_bar`, c'est-à-dire qu'on a autant de chiffres dans cette liste que de disciplines que l'on veut représenter (10).

`width` correspond à la largeur qu'on souhaite donner à nos barres sur le graphique, `0.25`.

`plt.figure` et `plt.subplot` permettent de créer la figure et l'objet `axes`.

Puis on trace les premières barres, correspondant aux athlètes féminines, avec la commande suivante :

```
plt.bar(pos,donnees_bar['F'],width,color='red',label="Femmes")
```

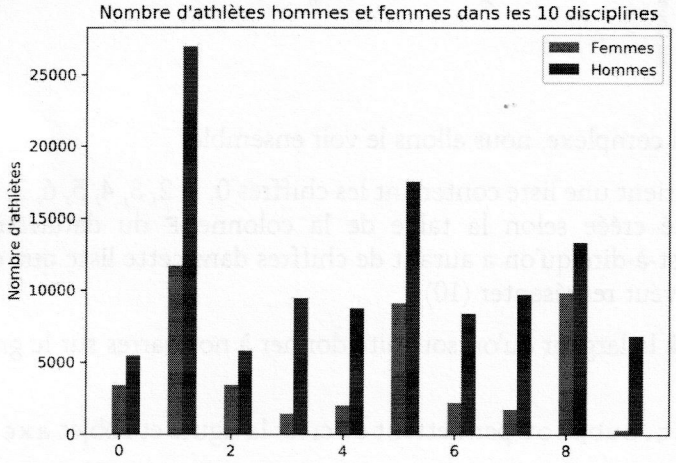
Ici, aux positions `X` égales à 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, on trace les dix valeurs de `donnees_bar['F']` en `Y` (le nombre d'athlètes féminines/discipline), avec une largeur de barre de `0.25`, en rouge (`red`) et avec comme label pour la légende le terme `Femmes`.

Ensuite, on trace les deuxièmes barres, correspondant aux athlètes masculins, avec la ligne suivante :

```
plt.bar([p + width for p in pos], donnees_bar['M'],width,
color='blue', label="Hommes")
```

Ici, on donne comme positions `X` les positions utilisées pour les barres correspondant aux athlètes féminines + `0.25` (chaque valeur de la liste `pos` est augmentée de `0.25` grâce à la boucle `for`). Ainsi les valeurs de `X` sont `0.25`, `1.25`, `2.25`, `3.25`..., et on trace le nombre d'athlètes dans chaque discipline en `Y`, puis, comme pour les athlètes féminines, on définit la largeur des barres, leur couleur et le label.

Si on ne définissait pas de `xticks`, le graphique généré serait le suivant :

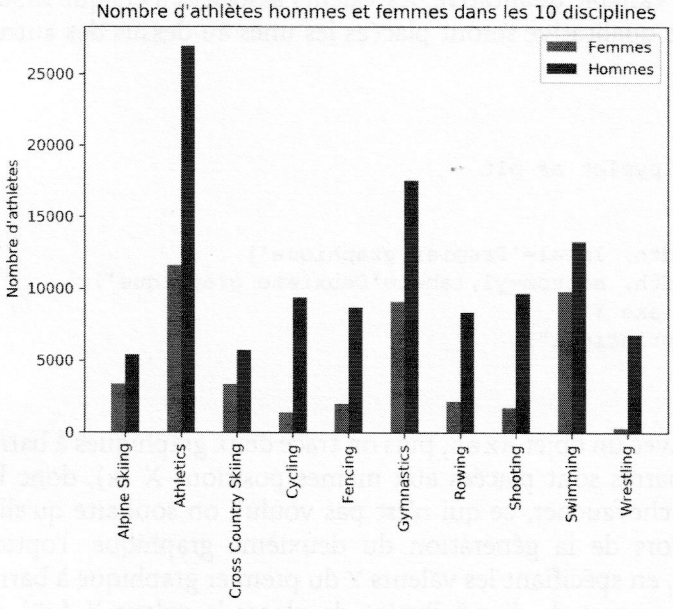


© Editions ENI - All rights reserved

On voit bien que les barres rouges sont positionnées à 0, 1, 2... et les barres bleues à 0.25, 1.25... Mais ces marqueurs X ne veulent rien dire, il faudrait que le nom de la discipline soit affiché sous chaque groupe de barres. La ligne suivante permet de faire cela :

```
plt.xticks([p + 0.125 for p in pos],donnees_bar.index,rotation=90)
```

Ici, on redéfinit les marqueurs X, grâce à la fonction `xticks()`, en les positionnant pile entre la barre des athlètes féminines et la barre des athlètes masculins, à chaque fois. Comme les barres sont positionnées à 0 et 0.25, 1 et 1.25... on va placer les marqueurs à 0.125, 1.125... au milieu des deux barres. Ainsi, on additionne p à `0.125` pour définir la position des marqueurs. On donne ensuite comme label les dix noms de disciplines correspondants (grâce à `donnees_bar.index`, l'index de la série contenant les noms des disciplines) et on affiche ces labels verticalement plutôt qu'horizontalement, grâce à l'option `rotation=90`, ce qui permet d'avoir le résultat final :



Ici, les sports sont triés par ordre alphabétique plutôt que par fréquence des athlètes dans ces sports. On constate que, globalement, il y a moins de femmes dans les disciplines sportives. Là où elles sont toutefois les plus présentes dans le top10, c'est en athlétisme, en gymnastique et en natation. En lutte, le nombre de femmes est très faible, alors que cette discipline fait partie du top10 des disciplines ayant le plus d'athlètes.

Remarque

Vous pouvez constater que pour obtenir ce graphique très personnalisé, nous avons dû écrire énormément de code... En fait, la librairie Seaborn a été développée pour simplifier encore plus la génération de graphiques avec Python et la rendre accessible à tous.

4.3.3 Graphique à barres empilées

Pour terminer, il existe les graphiques à barres empilées. Plutôt que de grouper les barres selon une variable qualitative, les valeurs associées à chaque modalité de cette variable qualitative seront placées les unes au-dessus des autres, de manière empilée.

Syntaxe

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplots(111)
plt.bar(x, y1, width, label='Premier graphique')
plt.bar(x, y2, width, bottom=y1,label='Deuxième graphique')
plt.ylabel('Titre axe Y')
plt.title("Titre principal")
plt.legend()
plt.show()
```

On crée une figure avec un objet `axes`, puis on trace deux graphiques à barres simples. Les deux barres sont placées aux mêmes positions X (`x`), donc les barres devraient se chevaucher, ce qui n'est pas voulu : on souhaite qu'elles s'empilent. Ainsi, lors de la génération du deuxième graphique, l'option `bottom` est utilisée, en spécifiant les valeurs `Y` du premier graphique à barres (`y1`). Cette option permet de dire à Pyplot de placer la valeur Y (`y2`) du deuxième graphique à barres au-dessus de la valeur Y (`y1`) du premier graphique à barres, ce qui permet d'avoir des barres empilées.

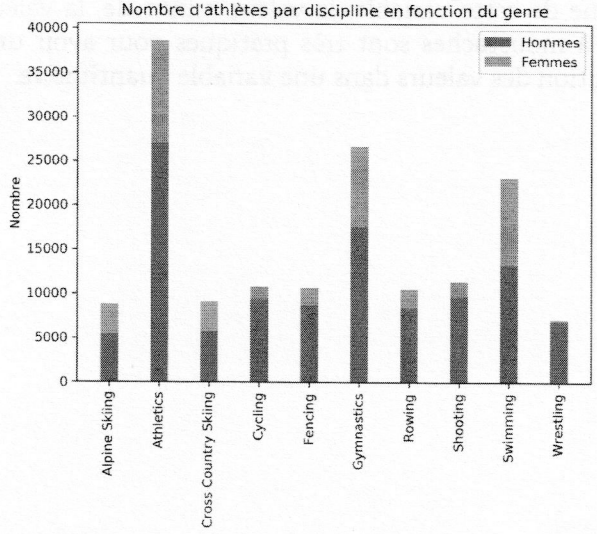
© Editions ENI - All rights reserved

Adaptons cette syntaxe pour afficher le nombre d'athlètes masculins et féminins en barres empilées.

Code

```
import matplotlib.pyplot as plt
sports_10=
donnees_NaN_remplaces['Sport'].value_counts()[:10].keys()
donnees_bar=donnees_NaN_remplaces.loc[donnees_NaN_remplaces
['Sport'].isin(sport_10)].groupby(["Sport", "Sex"]).size().unstack(1)
valeurs_hommes = donnees_bar["M"]
valeurs_femmes = donnees_bar["F"]
width = 0.35
plt.figure()
plt.subplot(111)
plt.bar(donnees_bar.index, valeurs_hommes, width, label='Hommes')
plt.bar(donnees_bar.index, valeurs_femmes, width,
bottom=valeurs_hommes,label='Femmes')
plt.ylabel('Nombre')
plt.title("Nombre d'athlètes par discipline en fonction du genre")
plt.xticks(donnees_bar.index,rotation=90)
plt.legend()
plt.show()
```

Résultat



Dans ce code, on crée les valeurs x communes aux deux graphiques, stockées dans la variable `sports_10`.

On reprend notre dataframe séparant le nombre d'athlètes masculins et féminins par discipline et on range le nombre d'athlètes masculins dans la variable `valeurs_hommes` et le nombre d'athlètes féminins dans la variable `valeurs_femmes`. On détermine la largeur souhaitée des barres avec `width`. Enfin, on trace les deux graphiques à barres en spécifiant dans le deuxième graphique l'option `bottom`, à laquelle on donne les valeurs y du premier graphique, `valeurs_hommes`. On obtient ainsi un graphique à barres empilées, ce qui est très visuel pour comparer les proportions dans différents sous-groupes.

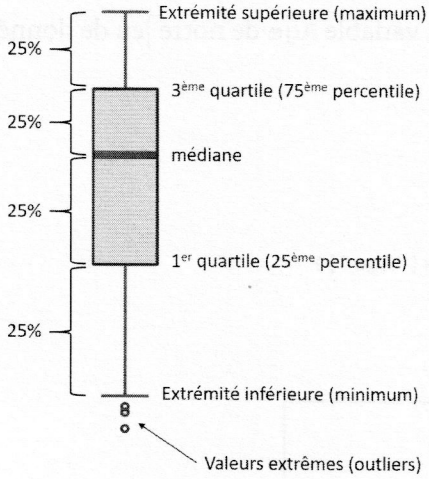
Cette section sur les graphiques à barres se termine ici. Vous aurez pu constater l'importance des graphiques à barres pour visualiser des variables qualitatives.

4.4 Boxplots

Les *boxplots* (ou boîte à moustaches, en français) permettent de visualiser la répartition des valeurs d'une variable quantitative, en permettant de connaître la valeur médiane de cette variable, la valeur minimale, la valeur maximale, etc. Les boîtes à moustaches sont très pratiques pour avoir une première idée de la distribution des valeurs dans une variable quantitative.

© Editions ENI - All rights reserved

Voici un schéma général d'une boîte à moustaches.



Les valeurs de la variable quantitative sont réparties dans cette boîte à moustaches. La médiane est la valeur pour laquelle 50 % des valeurs de la variable sont supérieures à cette valeur et 50.% des valeurs de la variable sont inférieures à cette valeur. Les bords de la boîte sont appelés les quartiles. Pour le premier quartile, cela signifie que 25 % des valeurs de la variable sont inférieures à cette valeur de premier quartile et 75 % des valeurs sont supérieures à cette valeur. C'est l'inverse pour le troisième quartile, 75 % des valeurs sont inférieures au troisième quartile et 25 % des valeurs sont supérieures.

L'extrémité inférieure de la moustaches représente la valeur minimale et l'extrémité supérieure la valeur maximale. Les ronds correspondent aux valeurs extrêmes, ce sont les valeurs qu'il faut aller regarder plus en détails pour comprendre leur position comme valeur extrême.

Voici la syntaxe générale de création d'un *boxplot* avec Pyplot :

```
import matplotlib.pyplot as plt
plt.boxplot(mon_dataframe["ma colonne"])
```

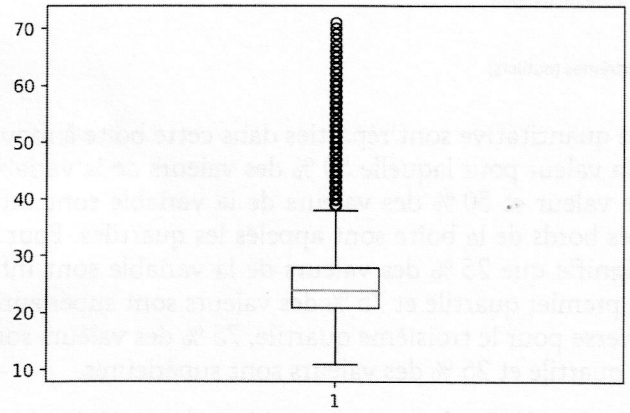
Bien entendu, ce type de graphique se fait uniquement sur des variables quantitatives, sur lesquelles on peut faire des opérations numériques, comme le calcul de la médiane par exemple.

Interprétons la boîte à moustaches de la variable `Age` de notre jeu de données des JO.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.boxplot(donnees_NaN_supprimes["Age"])
plt.show()
```

Résultat



On constate beaucoup de valeurs extrêmes pour la variable âge. En effet, nous l'avions vu précédemment, certaines disciplines artistiques qui avaient lieu aux JO entre 1924 et 1948 regroupaient des athlètes très âgés, ce qui n'est pas représentatif de la majorité des athlètes des JO : en effet, ces disciplines ont très vite disparu, laissant la place à des disciplines nécessitant d'être plus jeune pour exceller. Cela explique ces valeurs extrêmes. On peut donc afficher notre `boxplot` sans les valeurs extrêmes, car elles aplatissent notre boîte à moustaches.

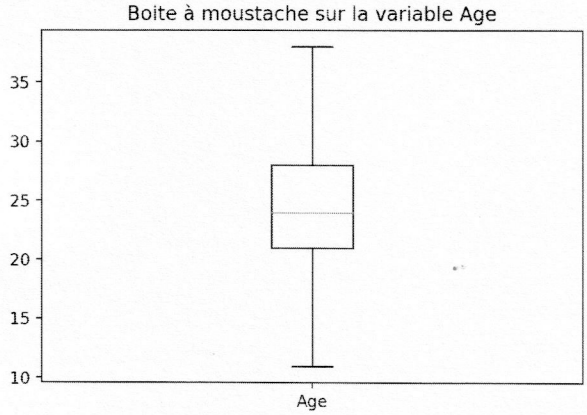
© Editions ENI - All rights reserved

Pour cela, nous mettrons l'option `showfliers` de la fonction `boxplot()` à False.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.boxplot(donnees_NaN_supprimes["Age"],showfliers=False)
plt.title("Boite à moustaches sur la variable Age")
plt.xticks([1], ['Age'])
plt.show()
```

Résultat

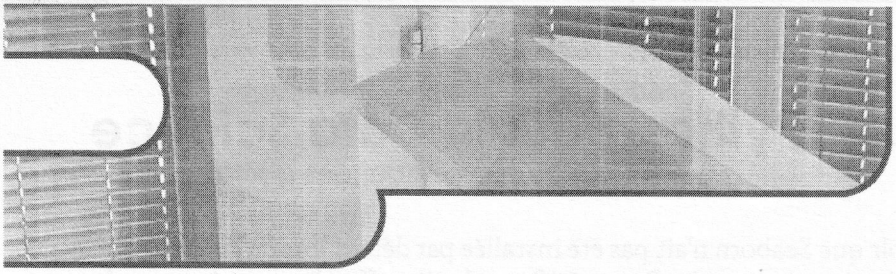


Ici, nous avons ajouté un titre au graphique et nous avons changé le label de l'axe x. Plutôt qu'il ait l'étiquette `1` (la position de la boîte sur l'axe x), nous l'avons remplacé par le mot `Age`, représentatif de la variable affichée. Comme évoqué en début de chapitre, dans la section d'introduction sur l'organisation d'une figure : "Les objets `axis` permettent de fixer les limites du graphique et génèrent ce qu'on appelle en anglais les *ticks* (marques en français), qui permettent de marquer les axes. Ces *ticks* peuvent être définis soit par le dataframe ou la série qu'on donnera en entrée, soit par nous-mêmes, manuellement." Ici, nous le définissons manuellement avec la fonction `xticks` de Pyplot. Il existe aussi la fonction `yticks` de Pyplot pour faire la même chose sur l'axe `y`.

Sur ce *boxplot*, on peut observer que la médiane est de 25 ans, donc 50 % des athlètes ont moins de 25 ans et 50 % ont plus de 25 ans. Le premier quartile est à 22 ans et le troisième quartile à 28 ans, donc 50 % des athlètes ont entre 22 et 28 ans. Enfin, l'âge maximal, sans prendre en compte les valeurs extrêmes vues sur le graphique précédent, est plutôt de 35 ans, et l'âge minimal proche des 15 ans. La boîte à moustaches est pratique pour avoir rapidement une idée de la distribution des données au sein d'une variable quantitative.

Nous avons vu les trois grands classiques de la visualisation : les nuages de points, les graphiques à barres et les boîtes à moustaches. À présent, nous allons voir, avec la librairie Seaborn, d'autres visualisations, elles aussi très utilisées.

© Editions ENI - All rights reserved



Chapitre 7
Maîtriser la librairie Seaborn

1. Introduction

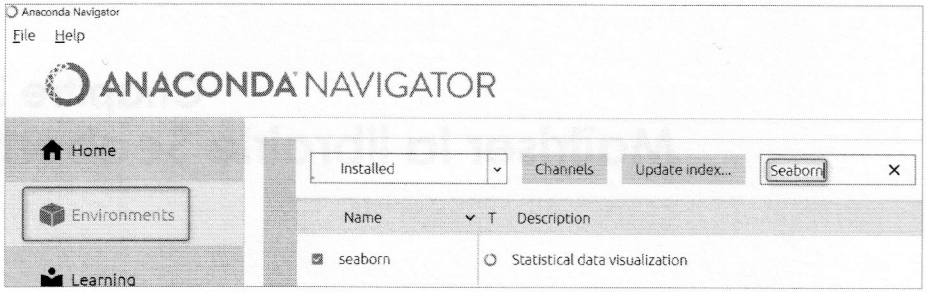
Nous avons vu dans le chapitre précédent comment visualiser des données avec la librairie Matplotlib et l'importance de la visualisation pour détecter des tendances dans nos données. Nous allons maintenant voir la deuxième librairie graphique la plus populaire, Seaborn.

En réalité, cette librairie s'appuie sur Matplotlib et apporte des fonctions supplémentaires permettant de générer des graphiques plus travaillés, plus modernes, et ce de manière très simple. Nous avons vu avec la librairie Matplotlib que lorsque nous souhaitons personnaliser nos graphiques et générer des graphiques plus complets, le code devient dense et parfois compliqué à écrire. L'objectif du développement de la librairie Seaborn était de rendre la génération de graphiques complexes plus facile, par l'appel de fonctions Seaborn.

Toutefois, cette librairie s'appuyant grandement sur la librairie Matplotlib, ses fonctions sont un complément de la librairie Matplotlib et son fonctionnement en est proche. Par exemple, nous utiliserons toujours le module Pyplot pour créer des figures, donner des étiquettes aux axes, etc. Mais nous utiliserons simplement des fonctions Seaborn pour tracer le graphique. Seaborn a été développée pour travailler avec des objets de classe `DataFrame` et nécessite beaucoup moins de formatage des données avant de pouvoir tracer les graphiques, contrairement à Matplotlib. C'est pourquoi cette librairie est devenue si populaire auprès des analystes de données.

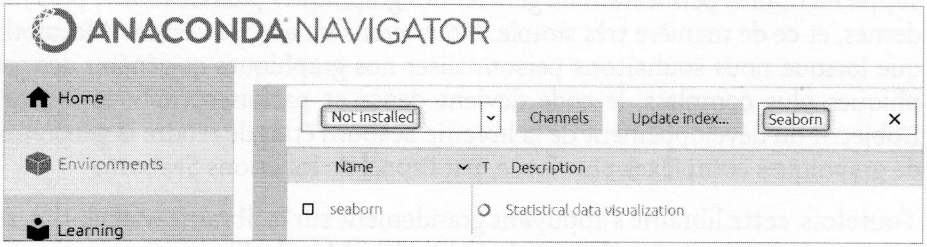
Il est possible que Seaborn n'ait pas été installée par défaut lors de l'installation de la distribution Anaconda. Pour vérifier cela, il suffit d'ouvrir **Anaconda Navigator** puis d'aller sur l'onglet **Environments**.

Si Seaborn est installée, voici ce que vous devriez voir en tapant le nom de la librairie dans la barre de recherche :



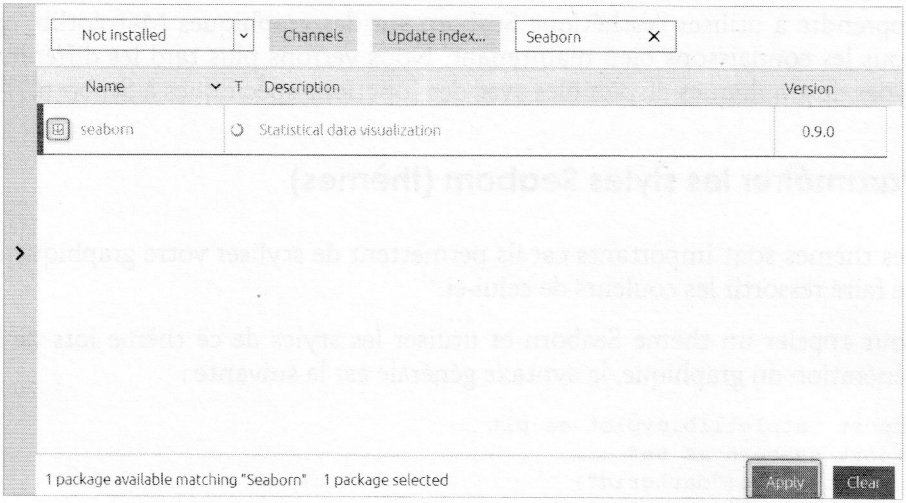
Vous voyez que l'onglet **Installed** est sélectionné et qu'en cherchant Seaborn elle apparaît avec un symbole vert à côté d'elle. La librairie est installée, vous pouvez donc continuer.

Dans le cas où elle n'est pas installée, elle n'apparaîtra pas en la cherchant ainsi. Il faut changer la valeur **Installed** par **Not installed** et vous verrez la chose suivante :



© Editions ENI - All rights reserved

Il faut alors cliquer sur le petit carré à gauche de Seaborn puis cliquer sur le bouton **Apply**, comme suit :



La librairie Seaborn va s'installer et vous pourrez continuer ce chapitre et tester le code dans votre notebook.

2. L'esthétique des figures avec Seaborn (Aesthetic)

Pour commencer, la syntaxe générale pour importer la librairie Seaborn est la suivante.

Syntaxe

```
import seaborn as sns
```

Il est courant de renommer la librairie Seaborn avec l'alias `sns`.

En plus de fournir un ensemble de fonctions permettant de tracer divers graphiques très complets, Seaborn fournit aussi un ensemble de thèmes, qui permettent de rendre les graphiques plus modernes, et ce de manière très simple. Il est possible de faire la même chose avec Matplotlib, mais il faut jouer avec beaucoup d'options liées à l'esthétique pour arriver au résultat que Seaborn fournit en choisissant juste un thème comme option.

Il est possible d'utiliser les fonctions d'esthétique de Seaborn sur les graphiques générés avec le module Pyplot (Matplotlib) ou sur les graphiques générés directement avec la librairie Seaborn. Dans cette section, nous allons apprendre à utiliser l'esthétique Seaborn sur des graphiques Matplotlib, car nous les connaissons bien maintenant. Nous verrons plus tard les différents types de graphiques disponibles avec des fonctions spécifiques à Seaborn.

2.1 Paramétrer les styles Seaborn (thèmes)

Les thèmes sont importants car ils permettent de styliser votre graphique et de faire ressortir les couleurs de celui-ci.

Pour appeler un thème Seaborn et utiliser les styles de ce thème lors de la génération du graphique, la syntaxe générale est la suivante :

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
plt.figure()
plt.subplot(111)
plt.plot(x,y)
plt.show()
```

Ici, on utilise la fonction `set_style()` de Seaborn, qui permet de définir le thème de la figure à utiliser. Par défaut, le thème est appelé `darkgrid` : ce thème permet d'avoir un fond de graphique grisé et une grille (*grid*).

Il existe d'autres thèmes, comme par exemple `whitegrid`, qui fournit une grille sur fond blanc, `dark` avec un fond grisé sans grille, `white` avec un fond blanc sans grille, ou encore le thème `ticks` qui ressemble beaucoup au thème `white`, avec un fond blanc, à la différence qu'il affiche des petits traits pour chaque valeur sur les axes x et y.

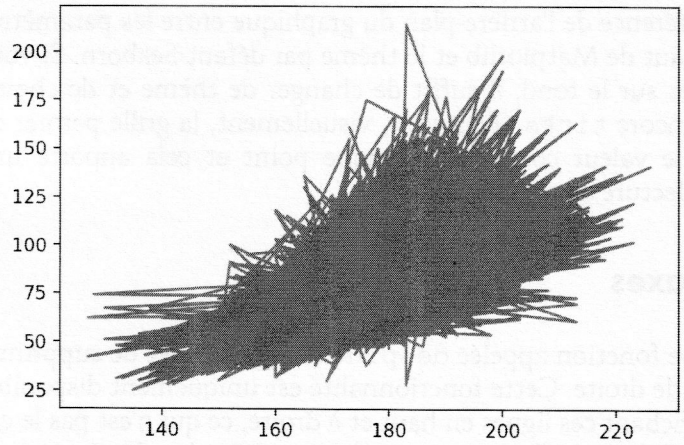
© Editions ENI - All rights reserved

Reprenons le graphique du poids en fonction de la taille des athlètes pour illustrer cet exemple et traçons un graphique avec le thème par défaut de Matplotlib et un autre avec le thème par défaut, `darkgrid`, de Seaborn.

Code

```
import matplotlib.pyplot as plt
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],donnees_NaN_supprimes
["Weight"])
plt.show()
```

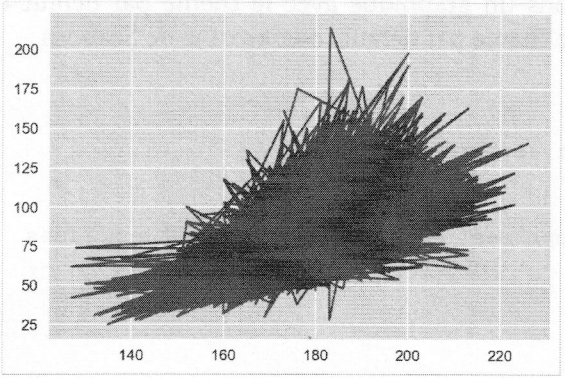
Résultat



Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
plt.show()
```

Résultat



On voit bien la différence de l'arrière-plan du graphique entre les paramètres esthétiques par défaut de Matplotlib et le thème par défaut Seaborn. Si vous n'aimez pas la grille sur le fond, il suffit de changer de thème et de choisir `dark`, `white` ou encore `ticks`. Toutefois, visuellement, la grille permet de mieux voir à quelle valeur correspond chaque point et cela apporte une certaine facilité de lecture au graphique.

2.2 Supprimer les axes

Seaborn fournit une fonction appelée `despine()`, qui permet de supprimer les axes du haut et de droite. Cette fonctionnalité est uniquement disponible pour les thèmes affichant ces lignes en haut et à droite, ce qui n'est pas le cas pour le thème par défaut de Seaborn, `darkgrid`. Mais c'est le cas pour le thème `white`, par exemple.

Syntaxe générale d'utilisation de la fonction `despine()`

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
plt.subplot(111)
plt.plot(x,y)
sns.despine(top=True, right=True, left=False, bottom=False)
plt.show()
```

© Editions ENI - All rights reserved

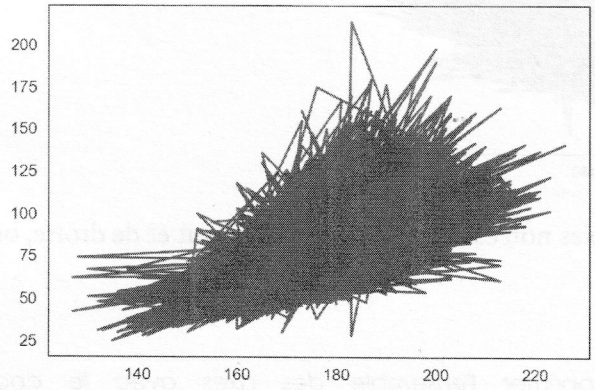
Après avoir tracé le graphique avec la fonction `plot()`, on utilise la fonction `despine()`, qui va supprimer, par défaut, les axes du haut (`top=True`) et de droite (`right=True`). Par défaut, cette option laisse l'axe de gauche, `y` (`left=False`) et l'axe du bas, `x` (`bottom=False`).

Générons le même graphique que précédemment avec le thème `white`.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
plt.show()
```

Résultat

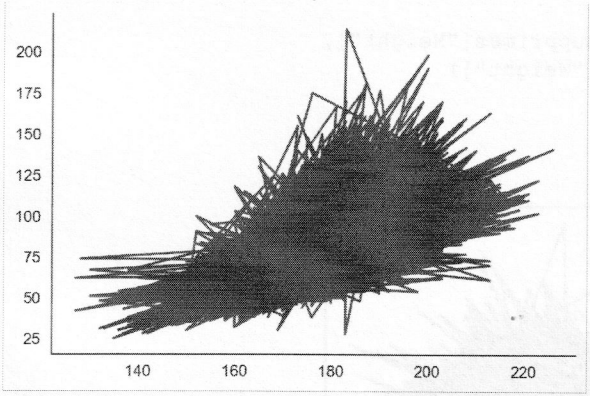


Puis générons ce même graphique en utilisant la fonction `despine()` avec ses options par défaut.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
sns.despine()
plt.show()
```

Résultat



On constate bien que les axes non essentiels ici, celui du haut et de droite, ont été supprimés.

Remarque

Il est possible de supprimer l'ensemble des axes avec le code `sns.despine(left=True, bottom=True)`*, ce qui supprime les traits de l'ensemble des axes mais laisse les étiquettes des axes (les annotations).*

© Editions ENI - All rights reserved

2.3 Paramétrer les contextes avec Seaborn

Il est possible de dire à Seaborn dans quel contexte les images seront utilisées et, en fonction de ce contexte, elle adaptera la taille des courbes et des axes.

Il existe quatre contextes disponibles avec Seaborn : `paper`, `notebook`, `talk` et `poster`. `paper` représente le contexte le plus petit, par exemple pour créer une figure pour une publication scientifique. `poster` représente le contexte le plus grand, par exemple pour créer une figure à mettre sur un poster scientifique pour présenter ses résultats. `notebook` est utilisée pour une figure à intégrer dans un notebook, et enfin, `talk` permet de dire à Seaborn que la figure sera placée dans un fichier PowerPoint pour une présentation orale de ses résultats.

Syntaxe de définition du contexte

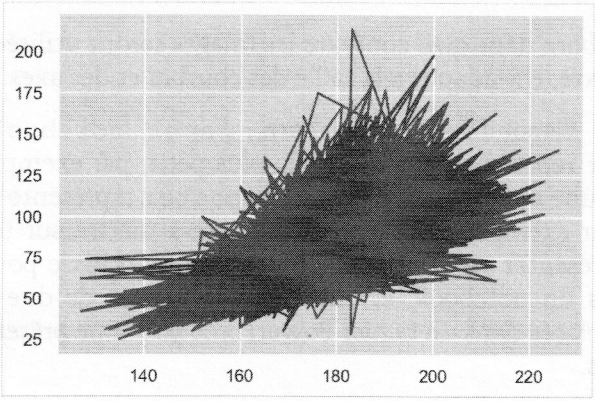
```
import seaborn as sns
sns.set_context("notebook")
```

Comparons deux contextes avec la figure précédente.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
sns.set_context("notebook")
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
plt.show()
```

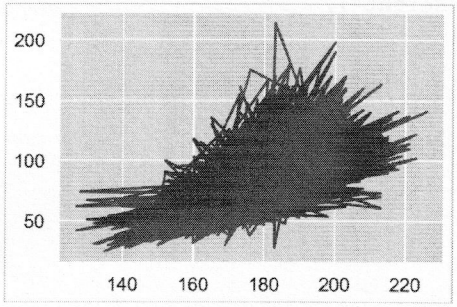
Résultat



Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
sns.set_context("talk")
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
plt.show()
```

Résultat



© Editions ENI - All rights reserved

On constate bien une différence de taille des étiquettes des axes et du graphique en lui-même entre les deux contextes. Étant donné que la figure avec le contexte `talk` est destinée à être intégrée dans un PowerPoint, il est essentiel que l'écriture soit plus grande, afin que les personnes assistant à la présentation puissent lire le graphique, même de loin.

Ces contextes modifient en fait les paramètres par défaut de Matplotlib utilisés pour générer les graphiques (taille des polices d'écriture, largeur des lignes, etc.).

Pour afficher les paramètres du contexte actuel, la syntaxe est la suivante :

```
import seaborn as sns
sns.plotting_context()
```

Affichons les paramètres des contextes utilisés précédemment, pour comprendre ce qui est modifié.

Code

```
sns.set_context("notebook")
sns.plotting_context()
```

Résultat

```
{'font.size': 12.0,
'axes.labelsize': 12.0,
'axes.titlesize': 12.0,
'xtick.labelsize': 11.0,
'ytick.labelsize': 11.0,
'legend.fontsize': 11.0,
'axes.linewidth': 1.25,
'grid.linewidth': 1.0,
'lines.linewidth': 1.5,
'lines.markersize': 6.0,
'patch.linewidth': 1.0,
'xtick.major.width': 1.25,
'ytick.major.width': 1.25,
'xtick.minor.width': 1.0,
'ytick.minor.width': 1.0,
'xtick.major.size': 6.0,
'ytick.major.size': 6.0,
'xtick.minor.size': 4.0,
'ytick.minor.size': 4.0}
```

Code

```
sns.set_context("talk")
sns.plotting_context()
```

Résultat

```
{'font.size': 18.0,
'axes.labelsize': 18.0,
'axes.titlesize': 18.0,
'xtick.labelsize': 16.5,
'ytick.labelsize': 16.5,
'legend.fontsize': 16.5,
'axes.linewidth': 1.875,
'grid.linewidth': 1.5,
'lines.linewidth': 2.25,
'lines.markersize': 9.0,
'patch.linewidth': 1.5,
'xtick.major.width': 1.875,
'ytick.major.width': 1.875,
'xtick.minor.width': 1.5,
'ytick.minor.width': 1.5,
'xtick.major.size': 9.0,
'ytick.major.size': 9.0,
'xtick.minor.size': 6.0,
'ytick.minor.size': 6.0}
```

On constate que pour le contexte `talk`, les paramètres sont tous légèrement augmentés par rapport au contexte `notebook`. Ainsi, un contexte avec Seaborn n'est autre qu'un dictionnaire de clés-valeurs avec des paramètres graphiques de taille.

Il est possible de modifier chacun de ces paramètres directement avec la fonction `set_context()` de Seaborn et l'option `rc`.

Syntaxe

```
sns.set_context("notebook", rc={"xtick.labelsize": 20})
```

Ici, la taille de la police des étiquettes de l'axe x (`xtick.labelsize`) sera de 20 et non plus la valeur par défaut du contexte `notebook`, qui est 11. En revanche, l'ensemble des autres paramètres auront bien les valeurs du contexte `notebook`.

© Editions ENI - All rights reserved

Remarque

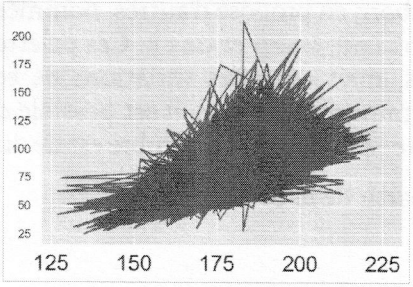
Ce livre ne décrit pas l'ensemble de ces paramètres, mais il est possible de modifier les tailles sur chaque composant du graphique, que ce soit les axes en général avec `axes.labelsize`, qui permet de définir la taille des étiquettes des axes x et y, ou `axes.titlesize`, qui permet de définir la taille de l'écriture des titres des axes. Il est possible de modifier la taille du texte de la légende avec `legend.fontsize` ou encore de changer l'épaisseur des lignes tracées (`lines.linewidth`) ou la taille des points tracés (`lines.markersize`).

Modifions la taille des étiquettes de l'axe x sur notre graphique précédent.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("darkgrid")
sns.set_context("notebook",rc={"xtick.labelsize": 20})
plt.figure()
plt.subplot(111)
plt.plot(donnees_NaN_supprimes["Height"],
donnees_NaN_supprimes["Weight"])
plt.show()
```

Résultat



On constate bien que les étiquettes de l'axe x ont une taille d'écriture plus élevée que celles de l'axe y.

À présent, vous êtes capable de customiser finement chaque partie du graphique grâce aux contextes de Seaborn.

2.4 Les palettes de couleur avec Seaborn

2.4.1 Choisir une palette de couleurs existante

Seaborn fournit une palette de couleurs qu'il utilise par défaut lors de la génération des graphiques. Mais les palettes de couleurs disponibles avec Seaborn sont nombreuses et c'est l'une des grandes raisons de sa popularité.

Pour visualiser la palette de couleurs par défaut proposée par Seaborn, il faut utiliser la syntaxe suivante :

```
sns.palplot(sns.color_palette())
```

La fonction `color_palette()` construit une palette de couleurs et attend en paramètre le nom d'une palette de couleur. Si on ne lui donne aucune option, elle utilise la palette courante, celle de Matplotlib. Seaborn utilise par défaut la palette de couleurs de Matplotlib, mais ces couleurs ont tout de même été améliorées et sont plus jolies. La fonction `palplot()` permet d'afficher la palette de couleurs.

Résultat de la syntaxe précédente :

Voici la palette de couleurs par défaut. Seaborn en propose d'autres, nommées ainsi : `pastel`, `muted`, `bright`, `colorblind`, `deep` et `dark`. Ces palettes se basent toutes sur les mêmes couleurs, mais sont des variations de ces couleurs, en matière de contraste ou de luminosité. De plus, il est possible de choisir le nombre de couleurs dans la palette ; par défaut, ce nombre est de dix.

Créons une palette de quinze couleurs à partir de la palette `dark`.

Code

```
sns.palplot(sns.color_palette("dark",15))
```

Résultat

© Editions ENI - All rights reserved

On constate que les couleurs de cette palette sont moins lumineuses que la palette par défaut et qu'il y a bien plus de couleurs disponibles.

La fonction `color_palette()` construit des palettes de couleurs, mais ne les définit pas comme palettes à utiliser. Si on souhaite définir une nouvelle palette de couleurs à utiliser par défaut sur les graphiques, il faut utiliser la fonction `set_palette()`.

La syntaxe est la suivante :

```
sns.set_palette("dark",10)
```

En dehors des palettes proposées par Seaborn, un autre groupe de palettes est très utilisé, ce sont les palettes *Color Brewer*, à la base développées pour des graphiques de cartographie. Ces palettes *Color Brewer* sont découpées en trois groupes que l'on peut importer dans Seaborn, selon ce qu'on veut faire.

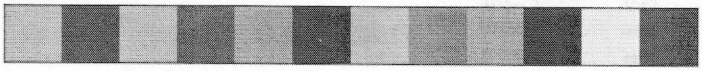
Voici les trois groupes de couleurs disponibles :

- Les palettes de couleurs pour les graphiques représentant des variables catégorielles. Généralement, on va vouloir bien séparer visuellement les différentes modalités de catégories et donc cela nécessitera des couleurs tranchantes. La palette la plus connue pour cela est nommée `Paired`.

Code

```
sns.palplot(sns.color_palette("Paired"))
```

Résultat

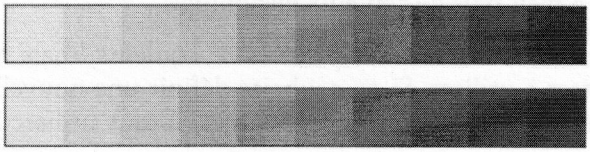


- Les palettes de couleurs séquentielles, lorsqu'on utilise des données numériques uniquement positives ou uniquement négatives et qu'on souhaite représenter une évolution de bas en haut ou de haut en bas. Voici deux palettes exemple que vous pouvez utiliser pour cela : `OrRd` ou `Blues`.

Code

```
sns.palplot(sns.color_palette("OrRd",10))
sns.palplot(sns.color_palette("Blues",10))
```

Résultat



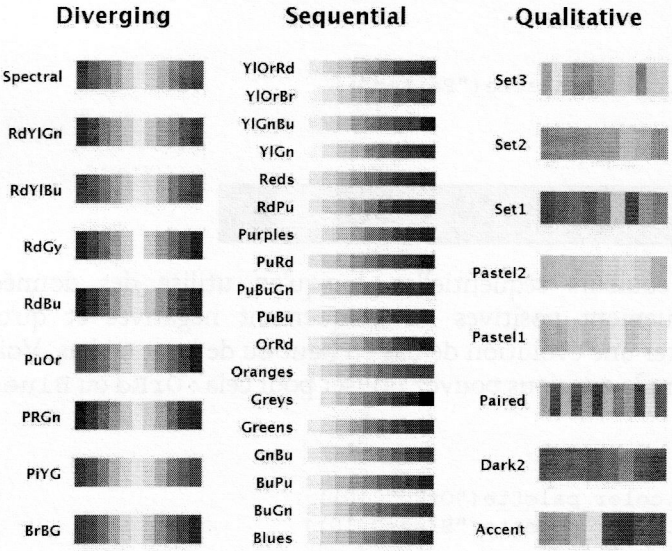
– Enfin, il y a les palettes de couleurs divergentes, qui permettent de représenter des valeurs numériques négatives et positives sur un même graphique. Cela permet de visuellement représenter des données extrêmes au sein de données numériques. Par exemple, la palette `RdBu` permet de faire cela.

Code

```
sns.palplot(sns.color_palette("RdBu", 10))
```

Résultat

Voici une image résumant les palettes *Color Brewer* disponibles et leur nom :



© Editions ENI - All rights reserved

Source :
https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/34087/versions/2/screenshot.jpg

Nous verrons l'application de ces palettes à la suite de ce chapitre, lorsque nous utiliserons les graphiques Seaborn.

2.4.2 Créer sa propre palette de couleurs

Si, parmi l'ensemble des palettes de couleurs, vous ne trouvez toujours pas votre bonheur, il est possible de créer la vôtre, grâce à la fonction `set_palette`, dont voici la syntaxe :

Syntaxe

```
ma_palette= ["#9b59b6","#3498db","#e74c3c","#34495e","#2ecc71"]
sns.set_palette(ma_palette)
```

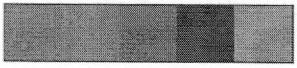
Pour créer votre palette, vous pouvez utiliser les codes hexadécimaux des couleurs. Ces codes sont facilement trouvables sur Google, ou sur ce site très bien fait : https://htmlcolorcodes.com/

Créons notre propre palette de couleurs et affichons-la.

Code

```
palette= sns.color_palette(["#9b59b6","#3498db","#e74c3c",
"#34495e", "#2ecc71"])
sns.set_palette(palette)
sns.palplot(palette)
```

Résultat



La fonction `color_palette()` ici prend une liste de cinq couleurs que l'on définit avec des codes hexadécimaux puis elle construit une palette de couleurs nommée `palette`. Ensuite, on définit cette nouvelle palette de couleurs comme palette par défaut avec la fonction `set_palette()`. Enfin, on affiche la palette avec la fonction `palplot()`.

3. Les différents types de graphiques

3.1 Préparation du jeu de données

Les mêmes jeux de données nettoyés que précédemment seront utilisés : `donnees_NaN_supprimes` et `donnees_NaN_remplaces`, selon ce que nous voulons représenter sur notre graphique. Le code pour régénérer ces objets est disponible dans le notebook associé à ce chapitre.

3.2 Nuage de points (scatterplot)

Nous l'avions vu dans le chapitre sur la librairie Matplotlib, le *scatterplot* (ou nuage de points, en français) est le graphique utilisé pour visualiser une relation statistique entre deux variables quantitatives. Le but ici est de tracer, pour chaque individu ou observation, un point, représentant sa valeur dans deux variables quantitatives.

Le grand avantage de la fonction de génération de ce type graphique avec Seaborn, par rapport à Matplotlib, est la possibilité d'augmenter les informations sur le graphique avec des modalités de variables qualitatives : pour cela, on utilisera la couleur des points, leur forme ou encore leur taille.

Syntaxe générale

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure()
plt.subplot(111)
sns.scatterplot(data=dataframe,x='Ma_colonne1',y='Ma_colonne2',
hue='Ma_colonne3',style='Ma_colonne4', size='Ma_colonne5')
plt.show()
```

© Editions ENI - All rights reserved

En regardant cette syntaxe générale, on voit l'utilité de connaître le fonctionnement de la librairie Matplotlib avant d'apprendre à utiliser Seaborn. En effet, la création de la figure, des axes ou encore l'affichage du graphique avec la fonction `show()` nécessitent de connaître Matplotlib/Pyplot. En revanche, pour créer le graphique, nous utilisons une fonction propre à Seaborn appelée `scatterplot`, qui permet de tracer un nuage de points.

L'option `data` de cette fonction attend un dataframe. Seaborn est très utilisée en *Data Science*, car elle est adaptée à la structure préférée des analystes de données, le dataframe. En donnant un dataframe à Seaborn, il est alors au courant de l'ensemble des variables de ce dataframe et de leur nom (les noms de colonnes). Ainsi, pour tracer x et y, il suffit de donner le nom de la colonne qu'on souhaite tracer.

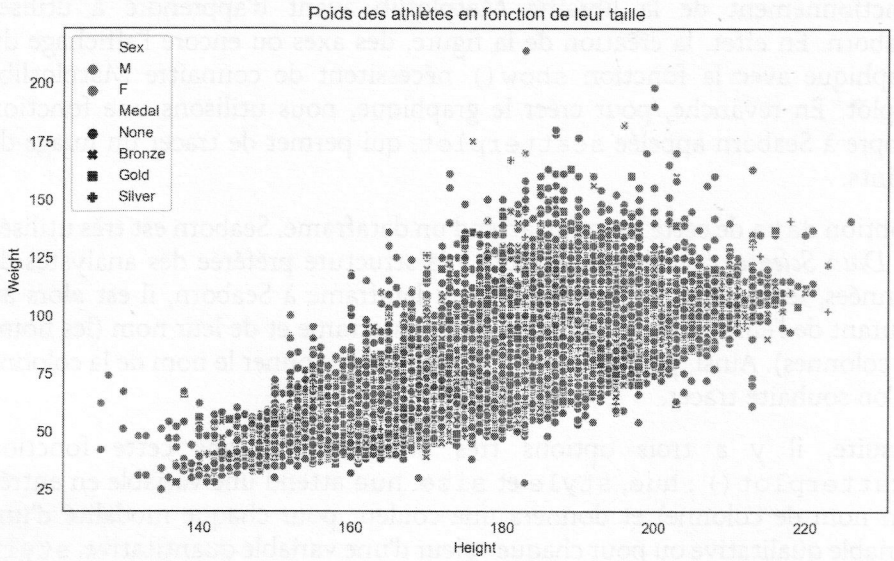
Ensuite, il y a trois options très intéressantes dans cette fonction `scatterplot()` : `hue`, `style` et `size`. `hue` attend une variable en entrée (un nom de colonne) et donnera une couleur pour chaque modalité d'une variable qualitative ou pour chaque valeur d'une variable quantitative. `style` attend aussi une variable, et donnera un style de point différent à chaque modalité/valeur de la variable. Enfin, `size` donnera une taille de point différente selon les modalités/valeurs de la variable.

Traçons un nuage de points représentant le poids des athlètes en fonction de leur taille, colorons les points selon le genre de l'athlète et mettons une forme de points différente selon la médaille obtenue par l'athlète.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
palette= sns.color_palette(["#3498db","#e74c3c"])
sns.set_palette(palette)
sns.set_style("white")
plt.figure(figsize=[10,6])
plt.subplot(111)
sns.scatterplot(data=donnees_NaN_supprimes, x='Height',
y='Weight',hue='Sex',style='Medal')
plt.title("Poids des athlètes en fonction de leur taille")
plt.legend(loc='upper left')
plt.show()
```

Résultat



En une ligne de code très simple, on a pu générer un graphique grandement personnalisé.

Explication du code :

On définit une palette, nommée `palette`, de deux couleurs, bleu et rouge, que l'on définit comme palette par défaut. On crée la figure, l'objet `axes`, et on définit la variable `Weight` comme étant `x` et la variable `Height` comme étant `y`. On indique que les couleurs seront en fonction du genre de l'athlète (variable `Sex`) et la forme des points en fonction du type de médaille obtenue (variable `Medal`). Enfin, on utilise Pyplot pour définir le titre principal du graphique et la position de la légende, en haut à gauche.

Il y a deux choses qui peuvent vous questionner ici. Nous n'avons pas défini de titres d'axes avec `plt.xlabel()` et `plt.ylabel()` et pourtant les noms des variables apparaissent en titres sur les axes x et y. C'est la fonction `scatterplot()` qui fait cela d'office. Mais vous pourriez définir vos propres labels en utilisant les fonctions de Pyplot, bien sûr.

© Editions ENI - All rights reserved

Ensuite, on ne définit pas d'option `label` dans la fonction `scatterplot()`, ce qui, précédemment, nous permettait de créer une légende avec Matplotlib. Ici, `scatterplot()` définit lui-même la légende selon ce qu'on lui donne en option `hue`, `size` ou `style`, ce qui nous facilite grandement la tâche. Et en plus de ça, cette légende est personnalisable directement avec la fonction `plt.legend()` de Pyplot, toujours !

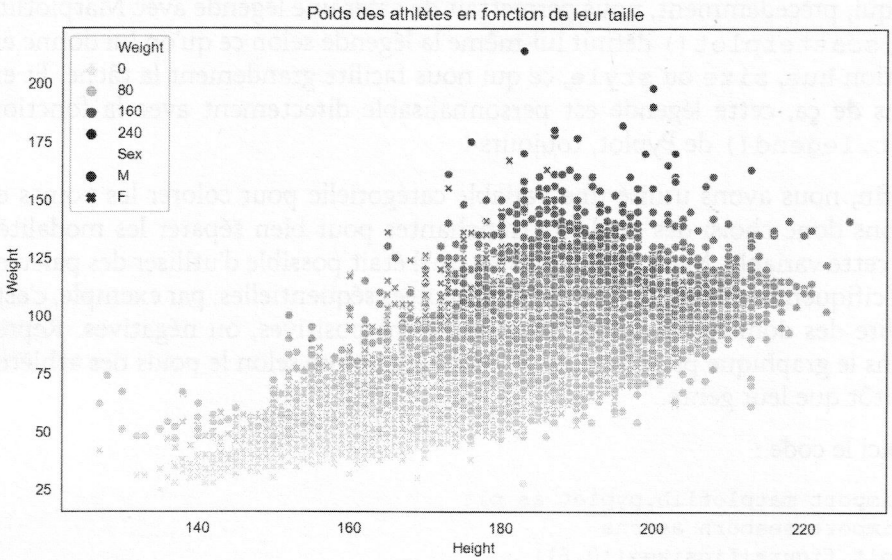
Enfin, nous avons utilisé une variable catégorielle pour colorer les points et avons donc choisi des couleurs tranchantes pour bien séparer les modalités de cette variable. Mais nous avons vu qu'il était possible d'utiliser des palettes spécifiquement pour représenter des données séquentielles, par exemple, c'est-à-dire des données numériques uniquement positives, ou négatives. Reprenons le graphique précédent, et colorons les points selon le poids des athlètes plutôt que leur genre.

Voici le code :

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=[10,6])
plt.subplot(111)
sns.scatterplot(data=donnees_NaN_supprimes, x='Height',
y='Weight',hue='Weight', style="Sex", palette="Blues")
plt.title("Poids des athlètes en fonction de leur taille")
plt.legend(loc='upper left')
plt.show()
```

Ici, nous définissons la variable `Weight` comme variable à utiliser pour la coloration des points. Plutôt que de redéfinir une palette par défaut, nous passons directement le nom de la palette à utiliser via l'option `palette` de la fonction `scatterplot()`. Nous utilisons une palette de couleurs séquentielles, `Blues`, un gradient du blanc vers le bleu que nous avons vu précédemment.

Résultat



On obtient bien un nuage de points avec un dégradé de couleur selon le poids, plus le poids augmente, plus les points sont bleus.

Ce premier exemple vous montre la puissance des fonctions Seaborn, qui permettent de faire des graphiques très personnalisés, et ce très facilement. Mais cet exemple vous montre aussi l'étroite imbrication entre Matplotlib/Pyplot et Seaborn.

Sur ce graphique, comme on l'avait déjà noté précédemment, il semble y avoir une corrélation linéaire positive entre le poids et la taille : plus la taille augmente et plus le poids augmente. Avec Seaborn, il est possible de générer des graphiques de régression afin de voir visuellement une relation linéaire entre deux variables quantitatives.

© Editions ENI - All rights reserved

3.3 Graphiques de régression

Les deux fonctions de Seaborn utilisées pour générer des graphiques de régression sont `regplot()` et `lmplot()`. Ces fonctions sont un guide visuel, elles ne remplacent pas les statistiques derrière, générées avec des librairies Python dédiées aux statistiques, comme `statsmodels`, qui n'est pas introduite dans ce livre.

Ces deux fonctions vont permettre de tracer la courbe de régression entre deux variables quantitatives et donc d'étudier leur relation linéaire. La différence entre `regplot()` et `lmplot()` est que `regplot()` est de plus bas niveau, et donc plus personnalisable. Par exemple, la fonction `regplot()` accepte différents types d'objets en `x` et `y`, tels une liste, une série, ou encore un `ndarray`. En revanche, `lmplot()` attend obligatoirement en entrée un dataframe avec l'option `data`, et `x` et `y` seront des variables de ce dataframe.

De plus, `lmplot()` est une fonction qui combine la fonction `regplot()` et un objet de type `FacetGrid` que nous verrons plus tard dans ce chapitre. Ainsi, il n'est pas nécessaire de créer la figure et les axes avec Pyplot, la fonction `lmplot()` fait tout toute seule, et c'est elle que nous allons utiliser dans cet exemple.

La fonction `lmplot()` permet de tracer un nuage de points, tout comme `scatterplot()`, mais elle va rajouter par-dessus ce nuage de points le tracé de régression.

Syntaxe générale

```
sns.lmplot(data=dataframe, x = "ma_colonne1", y = "ma_colonne2")
```

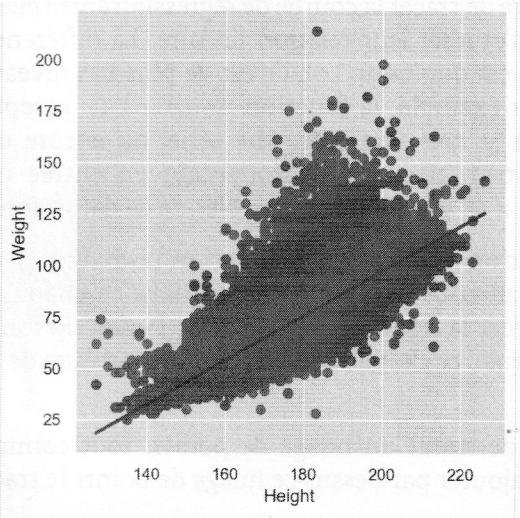
Par défaut, `lmplot()` va utiliser un modèle de régression linéaire simple, qui est une droite représentant au mieux la relation entre la variable à expliquer (`y` : exemple, le poids) et la variable explicative (`x` : exemple, la taille).

Utilisons `lmplot()` pour tracer le nuage de points du poids des athlètes en fonction de leur taille, ainsi que la droite de régression modélisant le mieux ce nuage de points.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.lmplot(x ='Height', y ='Weight', data =
donnees_NaN_supprimes,line_kws={'color': 'red'})
plt.show()
```

Résultat



Ici, on utilise la fonction `lmplot()`, à laquelle on donne le dataframe en entrée avec les variables `x` et `y` choisies. De plus, on précise la couleur de la droite de régression que l'on souhaite avec l'option `line_kws`. On obtient bien le nuage de points du poids en fonction de la taille avec le tracé de la régression.

Comme pour la fonction `scatterplot()`, il est possible de spécifier une variable qualitative, par exemple le genre des athlètes, pour colorer les points en fonction des modalités de cette variable, grâce à l'option `hue`. En plus de colorer les points, la fonction `lmplot()` tracera la droite de régression pour chaque nuage de points créé (un par modalité).

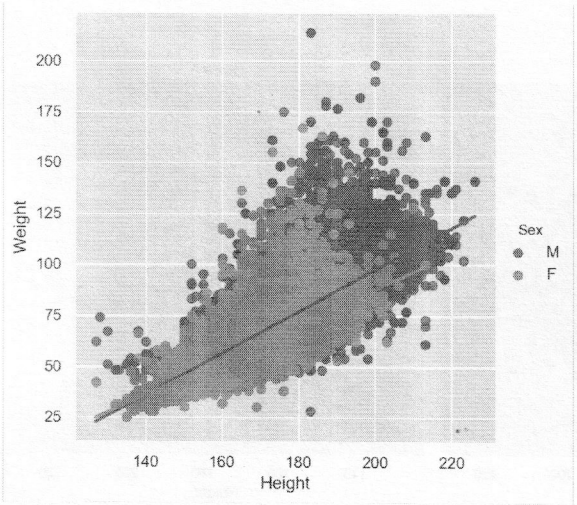
Reprenons le graphique précédent et spécifions la variable `Sex` dans l'option `hue`.

© Editions ENI - All rights reserved

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.lmplot(x ='Height', y ='Weight', data =
donnees_NaN_supprimes, hue="Sex")
plt.show()
```

Résultat



On obtient ainsi deux nuages de points séparés par leur couleur et deux droites de régression spécifiques à chaque nuage de points.

Pour finir, la spécificité de `lmplot()` est d'être basée sur les `FacetGrid`, que nous verrons en détail plus tard. Toutefois, brièvement, les `FacetGrid` permettent de positionner plusieurs graphiques sur une même figure et cela de manière optimisée par rapport à la fonction `subplots()` de Pyplot que nous utilisions précédemment.

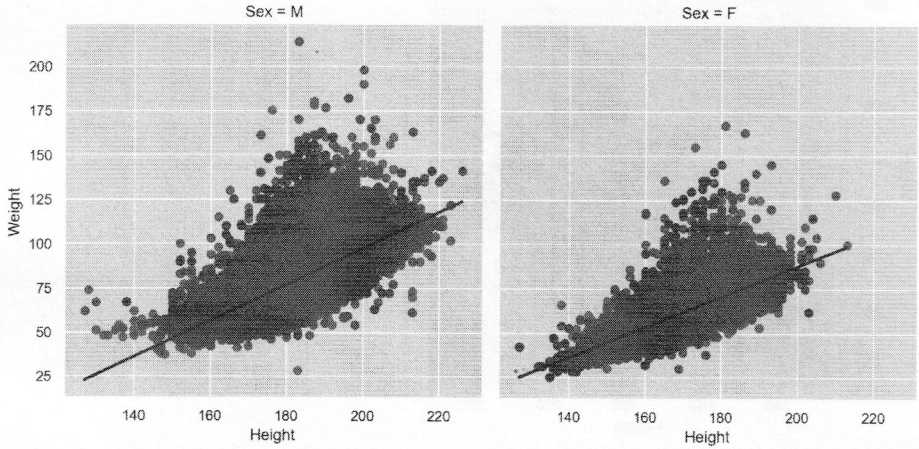
De ce fait, deux options sont disponibles dans la fonction `lmplot()` : `col` et `row`. `col` permet de placer les graphiques sur une ligne et `row` sur une colonne. En spécifiant une variable qualitative à `col` ou `row`, `lmplot()` tracera un nuage de points et une droite de régression pour chaque modalité de cette variable, et ce sur des graphiques séparés.

Testons l'option `col` avec la variable `Sex` de notre jeu de données.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.lmplot(x ='Height', y ='Weight', data =
donnees_NaN_supprimes, col="Sex",line_kws={'color': 'red'})
plt.show()
```

Résultat



On obtient deux graphiques séparés puisqu'il y a deux modalités dans la variable `Sex`, qui sont `M` et `F`.

Cette fonction `lmplot()` est très utilisée pour tracer des nuages de points et observer la relation entre deux variables quantitatives. Grâce à un grand nombre d'options et aux `FacetGrid`, elle permet de générer des graphiques très personnalisés.

© Editions ENI - All rights reserved

3.4 Pointplot

Le *pointplot*, ou graphique à points, permet de tracer, pour chaque couple de coordonnées x,y un point, puis de relier l'ensemble des points entre eux. Ce type de courbe est très utilisé pour représenter l'évolution d'une variable quantitative ou qualitative au cours du temps, en séparant les courbes via d'autres variables qualitatives. Mais il est aussi utilisé comme alternative aux graphes à barres pour représenter le nombre d'observations ou d'individus par modalité d'une variable qualitative. Nous allons voir ensemble ces différentes utilisations dans cette section.

Voici la syntaxe générale du *pointplot* :

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure()
plt.subplot(111)
sns.pointplot(x ='Variable1', y ='Variable2', data = dataframe,
hue="Variable3")
plt.show()
```

Représentons l'évolution du nombre d'athlètes dans les Jeux olympiques d'été depuis le début des JO jusqu'en 2016, année des derniers Jeux olympiques, et séparons cette évolution en fonction du genre des athlètes.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_context("notebook",rc={"xtick.labelsize":
15,"ytick.labelsize": 15})
plt.figure(figsize=[15,6])
plt.subplot(111)
donnees_summer=donnees_NaN_remplaces[donnees_NaN_remplaces
["Season"]=="Summer"]
donnees_pointplot=donnees_summer.groupby(['Year','Sex'])
.size().reset_index(name='Count')
sns.pointplot(x ='Year', y ='Count', data = donnees_pointplot,
hue="Sex")
plt.xticks(rotation=90)
plt.show()
```

On utilise le contexte `notebook` en prenant une taille de `15` pour les étiquettes des axes x et y. Puis on redéfinit la taille de la figure, pour qu'elle soit plus large que haute (largeur de 15 et hauteur de 6).

La variable `donnees_summer` contient l'ensemble des athlètes pour lesquels la colonne `Season` est égale à `Summer` (été). On récupère ainsi tous les athlètes ayant participé aux Jeux olympiques d'été. Ensuite, on groupe ce nouveau dataframe selon les deux variables qui nous intéressent, `Year` et `Sex`, puis on compte le nombre d'athlètes dans chaque catégorie avec la fonction d'agrégation `size()`. On renomme la colonne contenant les comptages avec le nom de colonne `Count` (comptage en français), grâce à la méthode `reset_index()`.

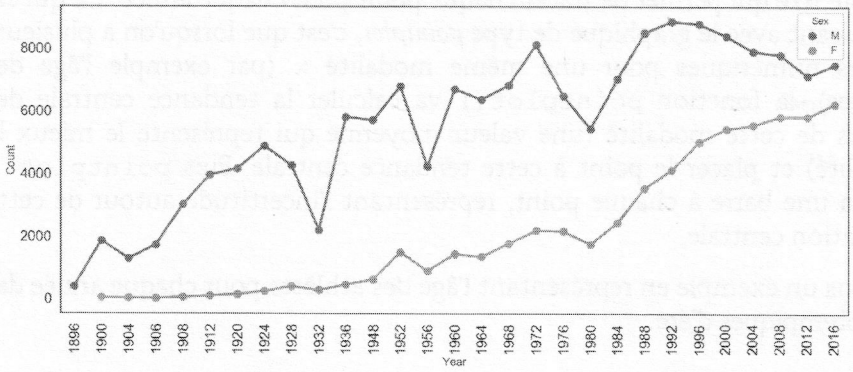
Voici un aperçu du tableau `donnees_pointplot` obtenu :

	Year	Sex	Count
0	1896	M	380
1	1900	F	33
2	1900	M	1865
3	1904	F	16
4	1904	M	1285
5	1906	F	11
6	1906	M	1722
7	1908	F	47
8	1908	M	3022
9	1912	F	87
10	1912	M	3953

À partir de ce tableau, on utilise la fonction `pointplot()` de Seaborn, à laquelle on donne en x la variable `Year`, en y la variable `Count`, et on lui passe la variable `Sex` comme option `hue`, afin qu'il y ait une courbe par genre.

© Editions ENI - All rights reserved

Résultat



On constate que le nombre d'athlètes masculins a toujours été supérieur au nombre d'athlètes féminins, mais que depuis 2012, cette différence est beaucoup moins marquée. Ensuite, on constate chez les athlètes masculins trois années avec une très forte baisse de participation : 1932, 1956 et 1980. Par exemple, en cherchant un peu sur Google, l'année 1932 correspond à la Grande Dépression et les nations européennes ont eu du mal à financer les voyages des athlètes aux États-Unis, où se déroulaient les JO, d'où la forte baisse de participation. En 1956 et 1980, suite à des contextes politiques très lourds, plusieurs pays ont boycotté les JO. De plus, si vous regardez bien les dates, vous constaterez que les JO sont bien séparés de quatre années, sauf entre 1912-1920 et 1936-1948, qui correspondent respectivement à la Première puis à la Deuxième Guerre mondiale, durant lesquelles les JO n'ont pas eu lieu.

Remarque

Ce jeu de données sur les Jeux olympiques est très intéressant car, si vous continuez de l'explorer par vous-même, vous comprendrez à quel point il est chargé d'histoire.

Ici, nous avons représenté le nombre d'athlètes par année des JO, il s'agit d'un nombre fixe qui permet de tracer chaque point pour chaque année. Ce qui est intéressant avec le graphique de type *pointplot*, c'est que lorsqu'on a plusieurs valeurs numériques pour une même modalité x (par exemple l'âge des athlètes), la fonction `pointplot()` va calculer la tendance centrale des valeurs de cette modalité (une valeur moyenne qui représente le mieux la modalité) et placer le point à cette tendance centrale. Puis `pointplot()` tracera une barre à chaque point, représentant l'incertitude autour de cette estimation centrale.

Prenons un exemple en représentant l'âge des athlètes pour chaque année des Jeux olympiques d'été.

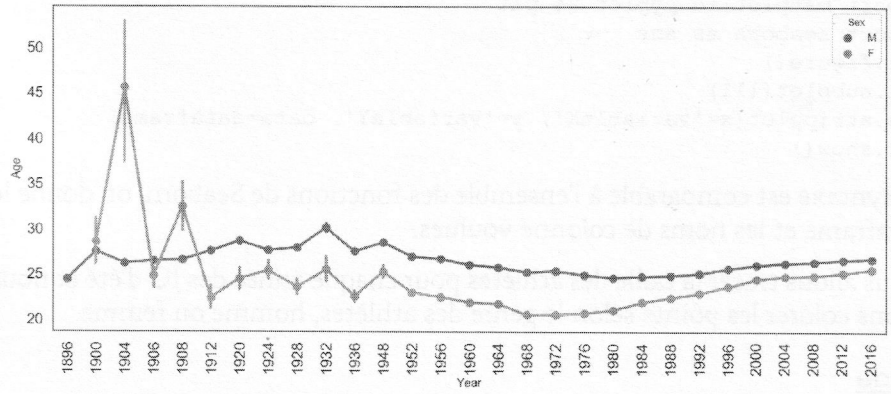
Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_context("notebook",rc={"xtick.labelsize":
15,"ytick.labelsize": 15})
plt.figure(figsize=[15,6])
plt.subplot(111)
donnees_summer=donnees_NaN_remplaces[donnees_NaN_remplaces
["Season"]=="Summer"]
sns.pointplot(x ='Year', y ='Age', data = donnees_summer, hue="Sex")
plt.xticks(rotation=90)
plt.show()
```

Ici, on récupère les athlètes ayant participé aux Jeux olympiques d'été, puis on trace l'âge des athlètes (y) en fonction des années (x), avec la fonction `poinplot()`.

© Editions ENI - All rights reserved

Résultat



Chaque point représente "l'âge moyen", la tendance centrale de l'âge des athlètes pour chaque année. On constate qu'au début des JO, l'âge des athlètes était très variable pour les femmes, peut-être aussi parce qu'elles n'étaient pas nombreuses, ce qui rend la tendance centrale plus difficile à calculer de manière représentative. Toutefois, vous pouvez voir l'utilité du *pointplot* sur des variables quantitatives comme l'âge. Grâce aux barres autour des points, cela permet de rapidement voir si la tendance centrale estimée est bien représentative de votre variable ou non.

3.5 Nuage de points avec une variable qualitative : stripplot

Nous avons vu les nuages de points entre deux variables quantitatives, permettant de visualiser leur relation. Il est possible de tracer des nuages de points entre une variable quantitative et une variable qualitative (avec ses modalités sur l'axe x ou y, au choix). Cela permet de voir la distribution des valeurs de la variable quantitative dans chaque modalité de la variable qualitative.

La fonction permettant de tracer ce graphique s'appelle `stripplot()`.

Syntaxe générale

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure()
plt.subplot(111)
sns.stripplot(x='variableX', y='variableY', data=dataframe)
plt.show()
```

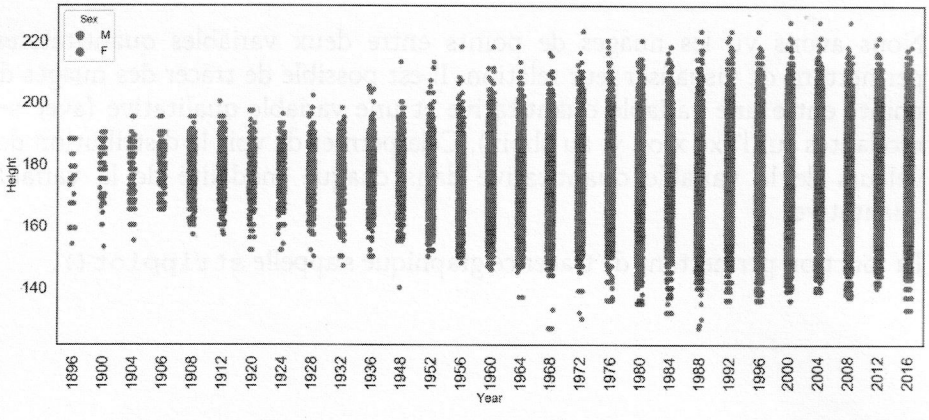
La syntaxe est comparable à l'ensemble des fonctions de Seaborn, on donne le dataframe et les noms de colonne voulues.

Nous allons tracer la taille des athlètes pour chaque année des JO d'été et nous allons colorer les points selon le genre des athlètes, homme ou femme.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
donnees_summer=donnees_NaN_supprimes[donnees_NaN_supprimes
["Season"]=="Summer"]
palette= sns.color_palette(["#3498db","#e74c3c"])
sns.set_palette(palette)
plt.figure(figsize=[15,6])
plt.subplot(111)
sns.stripplot(x='Year', y='Height', hue='Sex', data=donnees_summer)
plt.xticks(rotation=90)
plt.show()
```

Résultat



© Editions ENI - All rights reserved

Pour chaque modalité de la variable qualitative choisie, ici l'année, la taille de l'ensemble des athlètes de cette modalité est tracée sur le graphique, sous forme de points. Cela permet de voir la distribution de la taille des athlètes au fil des années.

On constate plusieurs choses sur ce graphique :

- On voit la présence d'athlètes féminines augmenter d'année en année, par la présence des points rouges.
- On constate que les athlètes féminines sont généralement de taille plus petite que les athlètes masculins.
- La taille globale des athlètes semble augmenter au fur et à mesure des années. C'est intéressant de voir ce résultat. Il pourrait y avoir diverses causes, dont deux me viennent à l'esprit :
  - Au fur et à mesure des années, les pays ont proposé des athlètes de plus en plus grands car ceux-ci montraient de meilleures performances,
  - Il a été prouvé qu'en 100 ans, la taille des Hommes a augmenté de 10 cm, cela est dû à notre changement d'alimentation (la quantité de nourriture consommée a augmenté) et au fait que nous soyons moins exposés aux infections étant enfant (ce qui limitait la croissance des enfants dans le passé). Peut-être ce gain de taille se répercute-t-il naturellement sur les athlètes des JO.

Bien sûr, ce ne sont que des suppositions.

Ce type de nuages de points est très utilisé pour visualiser la distribution de valeurs numériques dans chaque modalité d'une variable qualitative et vous avez vu ici le nombre d'informations qu'on peut en tirer.

3.6 Boxplots

Les *boxplots* ont déjà été décrits dans le chapitre sur la librairie Matplotlib, nous ne reviendrons donc pas dessus. En revanche, nous allons voir comment générer des *boxplots* beaucoup plus complets avec la librairie Seaborn qu'avec Matplotlib.

La syntaxe générale de création d'un *boxplot* avec Seaborn est la suivante :

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure()
plt.subplot(111)
sns.boxplot(x='variableX', y='VariableY', data=dataframe)
plt.show()
```

Nous l'avions vu, la boîte à moustaches permet de visualiser la distribution des valeurs au sein d'une variable numérique. L'avantage réel qu'apporte Seaborn sur les *boxplots*, c'est le fait de pouvoir séparer ces *boxplots* selon une variable qualitative, et donc de voir la distribution des valeurs d'une variable quantitative dans les différentes modalités d'une variable qualitative.

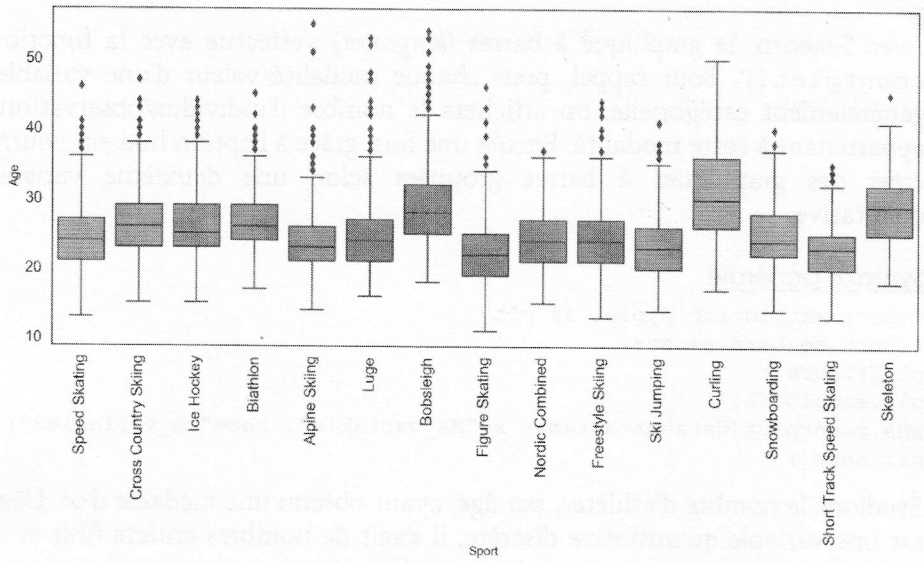
Imaginons qu'on souhaite visualiser la distribution de l'âge des athlètes dans chaque discipline des Jeux olympiques d'hiver, le code serait le suivant :

```
import matplotlib.pyplot as plt
import seaborn as sns
donnees_Winter=donnees_NaN_supprimes[donnees_NaN_supprimes
["Season"]=="Winter"]
plt.figure(figsize=[15,6])
plt.subplot(111)
sns.boxplot(x='Sport', y='Age', data=donnees_Winter)
plt.xticks(rotation=90)
plt.show()
```

On crée un sous-tableau contenant les athlètes ayant participé aux Jeux olympiques d'hiver uniquement, puis on utilise la fonction `boxplot()` en lui donnant le dataframe et les deux variables souhaitées, `Sport` en `x` et `Age` en `y`.

© Editions ENI - All rights reserved

Résultat



On constate que les athlètes avec l'âge médian le plus élevé appartiennent aux disciplines bobsleigh, curling et skeleton (un sport de glisse sur une planche ressemblant à une luge). Ce sont aussi les trois sports pour lesquels la distribution de l'âge est la plus large autour de la médiane. Le sport pour lequel l'âge médian est le plus bas est la discipline Figure Skating, qui signifie Patinage artistique en français. Il est connu que dans cette discipline, les médaillés d'or sont très jeunes, contrairement à ce qu'on peut voir en Curling par exemple, où il n'est pas rare que le médaillé d'or ait dépassé la trentaine.

Remarque

On pourrait, en plus de ça, utiliser l'option `hue` en précisant une seconde variable qualitative, ce qui générerait plusieurs boxplots/disciplines, représentant la distribution de l'âge des athlètes dans chaque sous-groupe de cette nouvelle variable qualitative, dans chaque discipline.

3.7 Graphique à barres : countplot

Avec Seaborn, le graphique à barres (*bargraph*) s'effectue avec la fonction `countplot()`. Pour rappel, pour chaque modalité/valeur d'une variable, généralement catégorielle, on affichera le nombre d'individus/observations appartenant à cette modalité. Encore une fois, grâce à l'option hue, on pourra créer des graphiques à barres groupées selon une deuxième variable qualitative.

Syntaxe générale

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure()
plt.subplot(111)
sns.countplot(data=dataframe, x="Ma_variable", hue="ma_variable2")
plt.show()
```

Étudions le nombre d'athlètes, par âge, ayant obtenu une médaille d'or. L'âge est une variable quantitative discrète, il s'agit de nombres entiers finis et la plage de valeurs est assez restreinte aux JO. Ainsi, on peut étudier chaque modalité de la variable `Age` comme si c'était une variable qualitative.

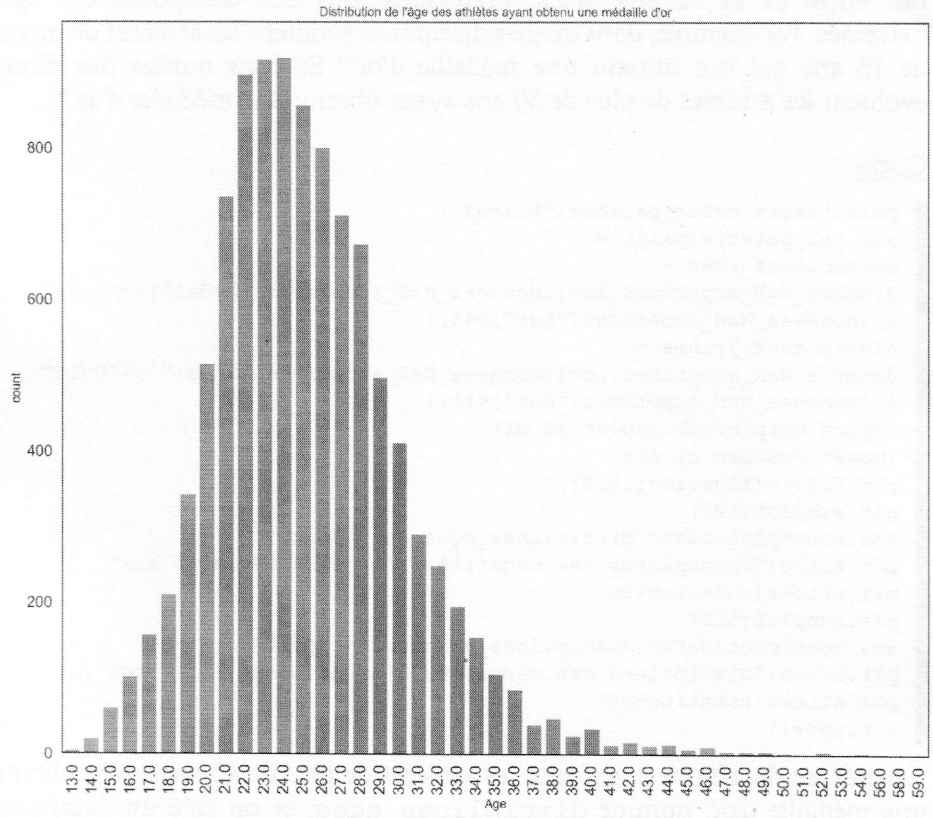
Pour cela, nous allons créer un sous-tableau avec uniquement les individus ayant la valeur `Gold` dans la colonne `Medal`, puis nous tracerons le graphe à barres de l'âge parmi ces athlètes.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=[15,13])
plt.subplot(111)
donnees_medaille_or=donnees_NaN_supprimes.loc[donnees_NaN_supprimes
["Medal"]=="Gold"]
sns.countplot(data=donnees_medaille_or, x="Age")
plt.title("Distribution de l'âge des athlètes ayant obtenu une
médaille d'or")
plt.xticks(rotation=90)
plt.show()
```

© Editions ENI - All rights reserved

Résultat



Si vous vous souvenez bien, pour obtenir ce genre de graphique avec Matplotlib, nous devions effectuer pas mal de formatage de nos données, avec les méthodes `value_counts()`, `size()` et `unstack()` de Pandas. Ici, la magie de Seaborn opère, il suffit de lui donner le dataframe ainsi que le nom de la variable qualitative que l'on veut en x, et il se charge d'afficher le nombre d'individus par modalité de cette variable. Ainsi, tracer un graphique à barres avec Seaborn est beaucoup plus simple.

On constate que, majoritairement, les athlètes ayant obtenu une médaille d'or ont entre 21 et 28 ans, mais intéressons-nous aux disciplines des âges extrêmes. Par exemple, dans quelles disciplines évoluent les athlètes de moins de 15 ans qui ont obtenu une médaille d'or ? Et dans quelles disciplines évoluent les athlètes de plus de 50 ans ayant obtenu une médaille d'or ?

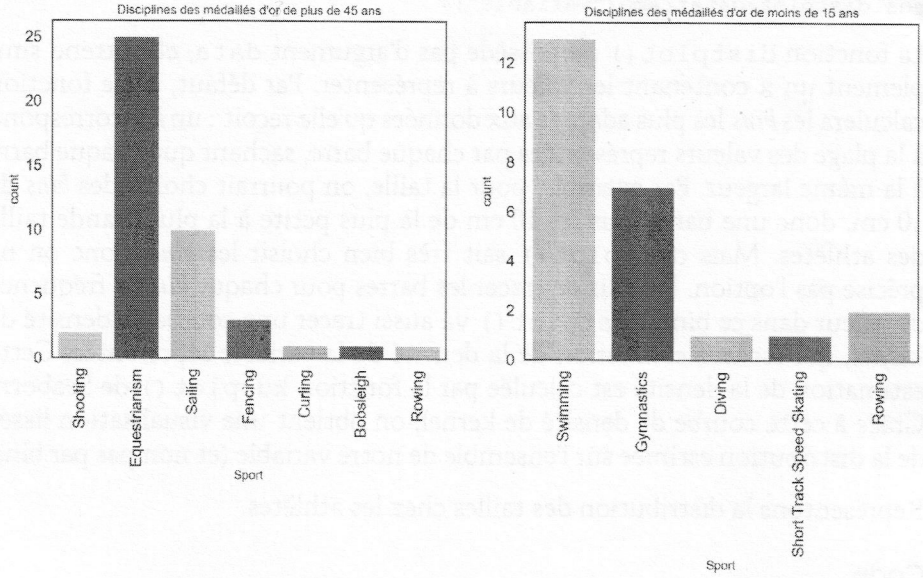
Code

```
palette=sns.color_palette("Paired")
sns.set_palette(palette)
disciplines_ages =
donnees_NaN_supprimes.loc[(donnees_NaN_supprimes["Medal"]=="Gold")
& (donnees_NaN_supprimes["Age"]>45)]
disciplines_jeunes =
donnees_NaN_supprimes.loc[(donnees_NaN_supprimes["Medal"]=="Gold")
& (donnees_NaN_supprimes["Age"]<15)]
import matplotlib.pyplot as plt
import seaborn as sns
plt.figure(figsize=[15,6])
plt.subplot(121)
sns.countplot(data= disciplines_ages, x="Sport")
plt.title("Disciplines des médaillés d'or de plus de 45 ans")
plt.xticks(rotation=90)
plt.subplot(122)
sns.countplot(data= disciplines_jeunes, x="Sport")
plt.title("Disciplines des médaillés d'or de moins de 15 ans")
plt.xticks(rotation=90)
plt.show()
```

On crée un dataframe contenant les athlètes de plus de 45 ans ayant obtenu une médaille d'or, nommé `disciplines_ages`, et on crée un dataframe contenant les athlètes de moins de 15 ans ayant obtenu une médaille d'or, nommé `disciplines_jeunes`. Puis on utilise la fonction `subplot()` pour tracer deux graphiques sur une même figure, le premier graphique représentant le nombre d'athlètes dans chaque sport du dataframe `disciplines_ages` et le deuxième graphique représentant le nombre d'athlètes dans chaque sport du dataframe `disciplines_jeunes`.

© Editions ENI - All rights reserved

Résultat



On peut voir que les médaillés d'or les plus âgés évoluaient majoritairement en équitation ou en voile (*sailing*) ou encore en escrime (*fencing*). Pour les médaillés d'or les plus jeunes, les deux principales disciplines sont la natation et la gymnastique.

3.8 Histogrammes

Lorsqu'on souhaite représenter la distribution d'une variable quantitative continue (dont les valeurs sont infinies entre un minimum et un maximum), comme le poids ou la taille, on la représentera aussi sur un graphique à barres, appelé cette fois-ci un histogramme. Pour tracer un histogramme avec Seaborn, on utilisera la fonction `distplot()`. Contrairement à la fonction `countplot()`, qui permet de visualiser le nombre d'individus par modalité, ici on va réellement étudier la distribution des valeurs au sein d'une variable quantitative.

Syntaxe générale

```
sns.distplot(dataframe["variable"])
```

La fonction `distplot()` ne possède pas d'argument `data`, elle attend simplement un `x` contenant les valeurs à représenter. Par défaut, cette fonction calculera les *bins* les plus adaptés aux données qu'elle reçoit : un *bin* correspond à la plage des valeurs représentées par chaque barre, sachant que chaque barre à la même largeur. Par exemple, pour la taille, on pourrait choisir des *bins* de 10 cm, donc une barre tous les 10 cm de la plus petite à la plus grande taille des athlètes. Mais `distplot()` sait très bien choisir les *bins*, donc on ne précise pas l'option. En plus de tracer les barres pour chaque *bin* (la fréquence de valeur dans ce bin), `distplot()` va aussi tracer une courbe de densité de kernel, qui est une estimation de la densité de la variable représentée. Cette estimation de la densité est calculée par la fonction `kdeplot()` de Seaborn. Grâce à cette courbe de densité de kernel, on obtient une visualisation lissée de la distribution estimée sur l'ensemble de notre variable (et non pas par bin).

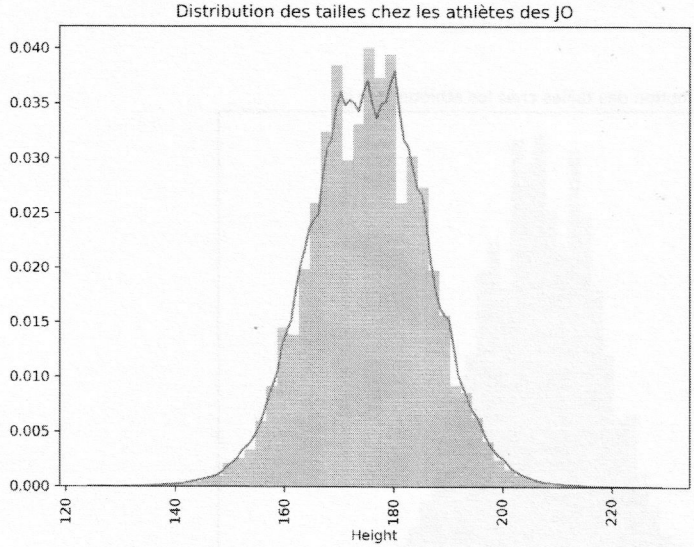
Représentons la distribution des tailles chez les athlètes.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
palette= sns.color_palette(["#3498db","#e74c3c"])
sns.set_palette(palette)
plt.figure()
plt.subplot(111)
sns.distplot(donnees_NaN_supprimes["Height"])
plt.title("Distribution des tailles chez les athlètes des Jos")
plt.xticks(rotation=90)
plt.show()
```

© Editions ENI - All rights reserved

Résultat



Par défaut, la fonction `distplot()` a choisi des *bins* de 2 cm, on obtient donc la densité de la taille des athlètes tous les 2 cm. Par défaut, la fonction `distplot()` trace la courbe d'estimation de la densité, donc l'axe y représente une densité.

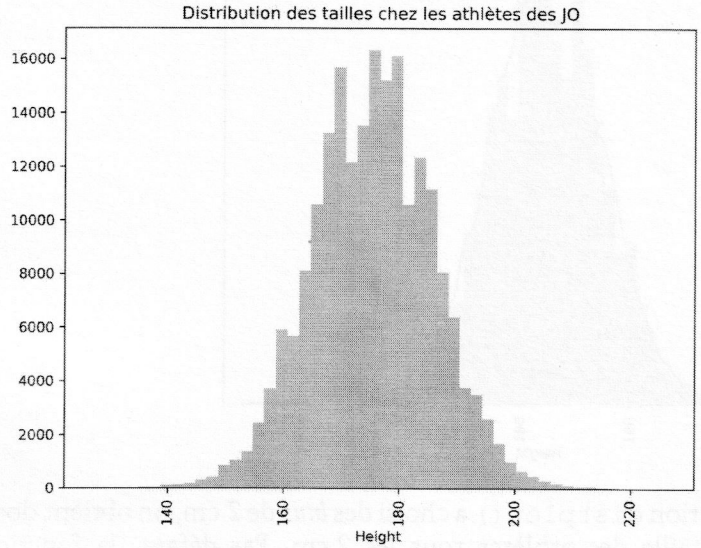
Si, en revanche, vous préférez que l'axe des y représente le nombre d'athlètes dans chaque bin plutôt qu'une densité, alors il faut mettre l'option `kde` à `False`. Ainsi, la courbe de densité ne sera pas estimée et seul l'histogramme sera tracé, avec un nombre plutôt qu'une densité.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
palette= sns.color_palette(["#3498db","#e74c3c"])
sns.set_palette(palette)
plt.figure()
plt.subplot(111)
sns.distplot(donnees_NaN_supprimes["Height"],kde=False)
plt.title("Distribution des tailles chez les athlètes des Jo")
```

```
plt.xticks(rotation=90)
plt.show()
```

Résultat



Ici, on obtient le nombre d'athlètes appartenant à chaque bin. On constate que le pic le plus haut est dans le bin représentant les athlètes de taille comprise entre 174 cm et 176 cm, très proche du second pic au bin 178-180 cm.

Étant donné qu'on sait que globalement les hommes sont plus grands que les femmes, on pourrait tracer un histogramme par genre. Malheureusement, la fonction `distplot()` ne prenant pas d'option `data`, elle ne prend pas non plus d'option `hue`. Ainsi, il faudra tracer, "à la main", un histogramme pour les athlètes masculins et un pour les athlètes féminins.

Code

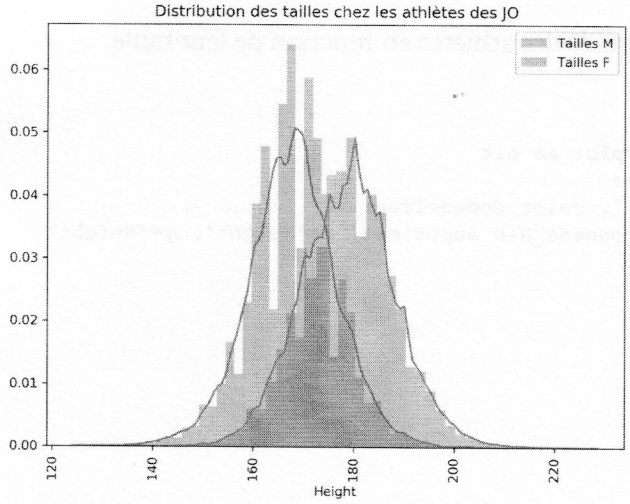
```
import matplotlib.pyplot as plt
import seaborn as sns
palette= sns.color_palette(["#3498db","#e74c3c"])
sns.set_palette(palette)
valeurs_athletes_feminines=donnees_NaN_supprimes.loc
[(donnees_NaN_supprimes["Sex"]=="F")]
```

© Editions ENI - All rights reserved

```
valeurs_athletes_masculins=donnees_NaN_supprimes.loc
[(donnees_NaN_supprimes["Sex"]=="M")]
plt.figure()
plt.subplot(111)
sns.distplot(valeurs_athletes_masculins["Height"], label="Tailles M")
sns.distplot(valeurs_athletes_feminines["Height"], label="Tailles F")
plt.title("Distribution des tailles chez les athlètes des Jo")
plt.xticks(rotation=90)
plt.legend()
plt.show()
```

Pour tracer ces deux histogrammes, il faut d'abord créer deux sous-tableaux, un contenant les lignes correspondant aux athlètes féminines et un contenant les lignes des athlètes masculins. On crée une figure avec un unique objet `axes`, sur lequel on trace deux courbes avec `distplot()`, en donnant respectivement les tailles des athlètes masculins puis féminins. Grâce à l'option `label` dans la fonction `distplot()`, il est ensuite possible de tracer la légende avec matplotlib/Pyplot et sa fonction `legend()`.

Résultat



On constate que le pic de taille pour les athlètes féminines est d'environ 168 cm et pour les athlètes masculins environ 180 cm.

3.9 Jointplot

Le *jointplot* permet de comparer deux variables quantitatives, comme par exemple le poids et la taille des athlètes, avec la distribution de chaque variable quantitative. Ainsi, sur un même graphique on représentera un *scatterplot*, graphique bivarié représentant la relation entre deux variables quantitatives, ainsi que deux histogrammes, graphiques univariés, représentant la distribution de chaque variable quantitative.

Syntaxe générale

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.jointplot(data=dataframe, x='Variable1', y='Variable2')
```

Ici, il n'est pas utile de créer la figure et les axes avec les fonctions `figure()` et `subplot()` de Pyplot. En effet, la fonction les crée elle-même : elle crée un objet de type `JointGrid`, similaire au `FacetGrid` dont on a parlé plus haut et qui sera introduit en fin de chapitre. Ce type d'objet contient déjà une figure et des *subplots*.

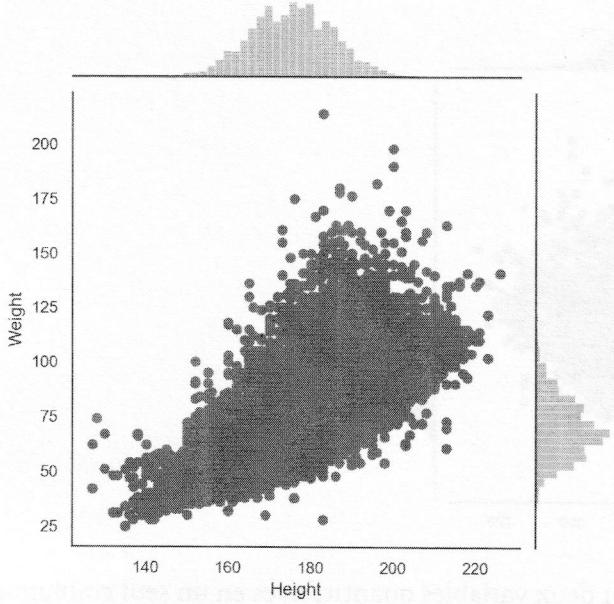
Traçons un *jointplot* du poids des athlètes en fonction de leur taille.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.jointplot(data=donnees_NaN_supprimes, x='Height', y='Weight')
plt.show()
```

© Editions ENI - All rights reserved

Résultat



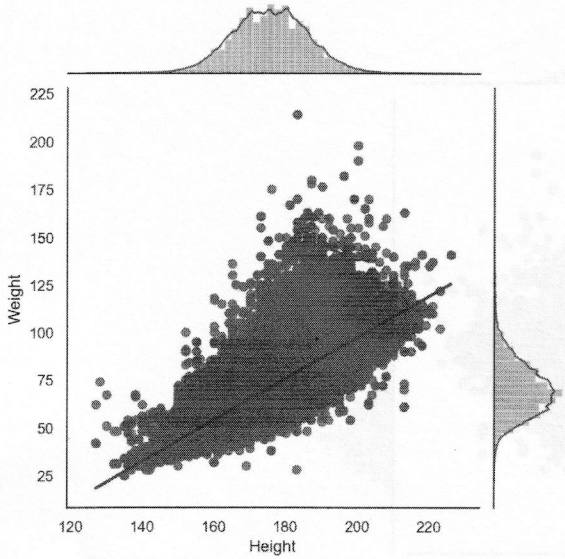
On obtient bien le nuage de points et les histogrammes spécifiques à chaque variable. L'histogramme du haut correspond à la distribution de la variable x, ici la taille, et l'histogramme de droite correspond à la distribution de la variable y, ici le poids des athlètes. En une figure, on peut voir la relation entre deux variables quantitatives ainsi que leur distribution respective.

Il est aussi possible de tracer directement la droite de régression sur le nuage de points et la courbe de densité de kernel sur les histogrammes grâce à l'option `kind="reg"`.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.jointplot(data=donnees_NaN_supprimes, x='Height', y='Weight',
kind="reg",line_kws={'color': 'red'})
plt.show()
```

Résultat



Lorsqu'on souhaite étudier deux variables quantitatives en un seul graphique, le *jointplot* est vraiment le graphique de choix à utiliser.

3.10 Pairplot

Avec la fonction `jointplot()`, nous avons vu comment visualiser la relation bivariée (deux variables quantitatives représentées sur un nuage de points) ainsi que la distribution univariée de chaque variable. Seulement ce graphique se fait avec deux variables choisies.

La fonction `pairplot()` génère une figure un cran plus complète que `jointplot()` : elle crée un *scatterplot* et les distributions associées pour l'ensemble des variables quantitatives d'un dataframe. Ce graphique est très pratique pour avoir une première vision globale des relations entre chaque paire de variables quantitatives.

© Editions ENI - All rights reserved

Syntaxe générale

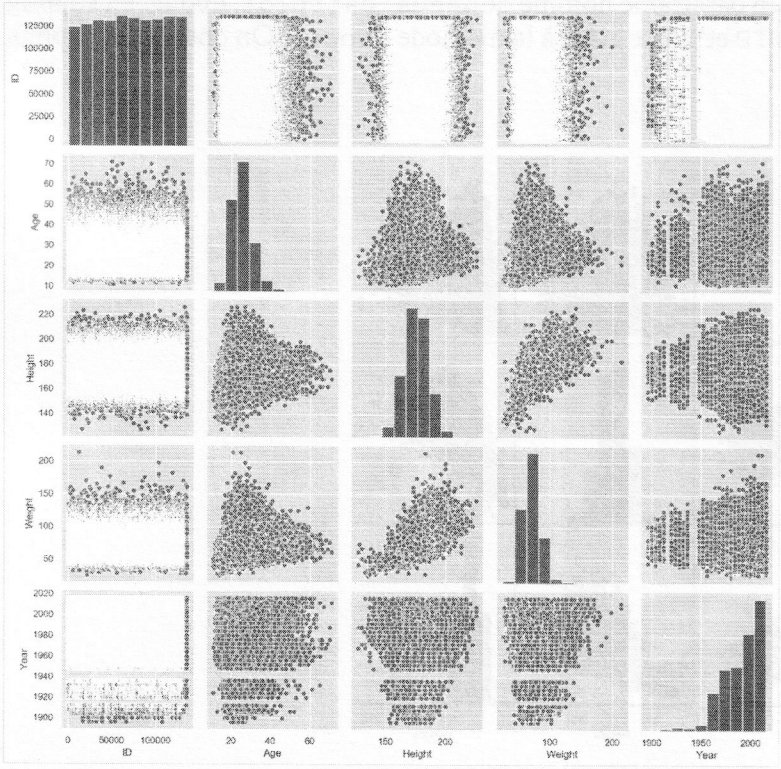
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.pairplot(dataframe)
```

Traçons le pairplot de notre tableau de données des JO.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.pairplot(data=donnees_NaN_supprimes)
plt.show()
```

Résultat



Pour tracer le *pairplot* de l'ensemble des variables quantitatives, on donne le dataframe entier. Toutefois, on constate que certaines variables quantitatives ne sont pas utiles, par exemple la variable `ID` des athlètes n'est pas une variable informative. De même, on pourrait enlever la variable `Year`.

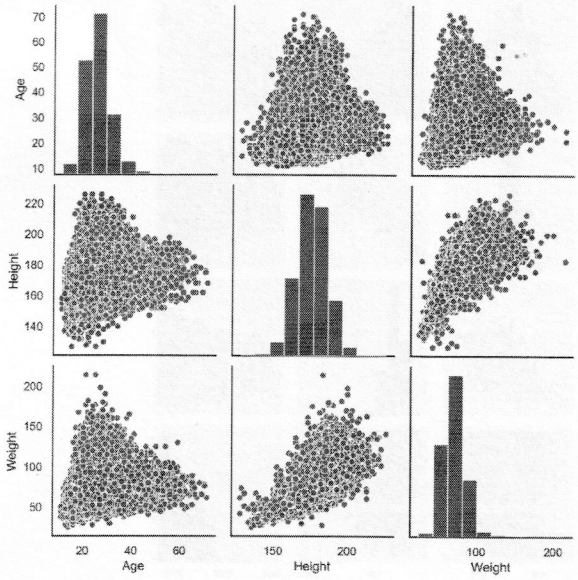
Pour supprimer ces variables du graphique, il faut créer un nouveau dataframe avec ces colonnes en moins.

Code

```
donnees_pairplot = donnees_NaN_supprimes.drop(columns = ['ID', 'Year'])
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="white", color_codes=True)
sns.pairplot(data=donnees_pairplot)
plt.show()
```

Ici, on crée un dataframe nommé `donnees_pairplot`, duquel on supprime les colonnes `ID` et `Year` grâce à la méthode `drop()`. On obtient le graphique ci-dessous.

Résultat



© Editions ENI - All rights reserved

En une seule vue, on constate les relations entre l'ensemble des variables quantitatives intéressantes, ainsi que leur distribution. Par exemple, en haut à gauche, on a l'histogramme de la distribution des valeurs dans la variable âge, puis à sa droite le nuage de points de la variable `Age` en `y` avec la variable `Height` en `x`, et tout en haut à droite, le nuage de points de la variable `Age` en `y` et la variable `Weight` en `x`, etc.

Remarque

On peut tracer les droites de régression sur les nuages de points avec l'option `kind="reg"`.

3.11 Heatmap

Une *heatmap* (ou carte thermique) est un graphique permettant de représenter les valeurs d'une matrice par des couleurs. Chaque case de la *heatmap* représente une case de la matrice, et celle-ci est colorée selon une échelle de couleur définie.

Cela permet de visuellement voir si des groupes se forment au sein de nos données, ou encore si des tendances sortent dans ces données. Il est toujours plus simple, pour un humain, de voir des couleurs et graphiques qu'une succession de nombres dans un tableau.

La matrice qu'on représente doit contenir des valeurs numériques. Ces valeurs peuvent être des comptages, mais la *heatmap* est aussi beaucoup utilisée pour représenter des corrélations linéaires entre variables quantitatives avec le coefficient de corrélation de Pearson, par exemple : pour chaque paire de variables, on aura un coefficient de corrélation compris entre -1 (corrélation linéaire négative forte) et 1 (corrélation linéaire positive forte), 0 signifiant qu'il n'y a pas de corrélation linéaire entre les deux variables.

Voici la syntaxe générale pour tracer une *heatmap* avec Seaborn :

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.heatmap(matrice)
```

Calculons une matrice de corrélation sur quelques variables intéressantes dans notre jeu de données. Cette matrice nous permettra de voir quelles variables sont corrélées.

Nous allons créer un sous-ensemble de notre dataframe contenant uniquement les données sur les athlètes médaillés, puis nous garderons les variables `Sex`, `Age`, `Weight`, `Height`, `Medal`, `Year` et `Team`. Comme écrit plus haut, une corrélation peut être calculée sur des variables quantitatives, ce qui n'est pas le cas de la variable `Medal` par exemple.

Ainsi, avant de pouvoir calculer une corrélation sur cette variable, il faut transformer ses valeurs en nombres. Nous allons voir ensemble deux manières de transformer les modalités d'une variable qualitative en nombres.

Code

```
donnees_heatmap=donnees_NaN_supprimes.loc[(donnees_NaN_supprimes
["Medal"]=="Gold") | (donnees_NaN_supprimes["Medal"]=="Silver") |
(donnees_NaN_supprimes["Medal"]=="Bronze")]
donnees_heatmap=donnees_heatmap[["Sex","Age","Weight","Height",
"Medal","Year","Team"]]
dictionnaire = {"Bronze" : 1, "Silver" : 2, "Gold" : 3}
donnees_heatmap["Medal"] =
donnees_heatmap["Medal"].map(dictionnaire)
donnees_heatmap["Team"]=donnees_heatmap["Team"].astype("category")
.cat.codes
correlation = donnees_heatmap.corr()
sns.heatmap(correlation, annot=True, linewidths=1.0)
```

Ici, on crée un dataframe ne contenant que les athlètes médaillés, dont la colonne Medal est égale à `Bronze`, `Silver` ou `Gold`. Puis on sélectionne les variables qu'on veut représenter sur la *heatmap*. Pour les variables qualitatives, il faut les transformer en valeurs numériques.

Il y a deux techniques possibles pour le faire :

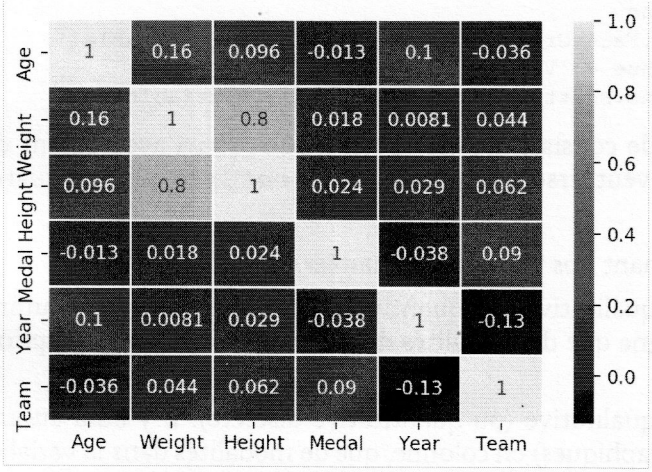
- On crée un dictionnaire avec en clé les modalités de la variable et en valeur, un nombre associé. Par exemple, on sait que la variable médaille possède trois modalités, Bronze, Silver et Gold, donc on assigne respectivement les chiffres 1, 2 et 3 à ces modalités. Puis on utilise la fonction `map()`, qui permet de remplacer les modalités par les chiffres du dictionnaire.

© Editions ENI - All rights reserved

– Quand une variable catégorielle a trop de modalités, comme la variable `Team`, on utilisera la fonction `astype()` puis `cat.codes()` pour déterminer un code numérique unique par modalité, et ce de manière automatique. `astype("category")` permet de convertir la colonne en type `categorie`, puis l'attribut `cat.codes()` permet de convertir ces modalités en valeurs numériques.

Enfin, la fonction `corr()` permet de calculer le coefficient de corrélation entre chaque paire de variables quantitatives du dataframe qu'on lui donne.

Résultat



Ici, on constate qu'il n'y a qu'une corrélation, positive, entre les variables poids et taille. Nous l'avions déjà remarqué. On peut aussi conclure, sans grande surprise, qu'il n'y a pas de corrélation entre le fait de gagner une médaille et la taille, le poids, l'âge ou encore le pays de l'athlète, puisque la variable `Medal` ne présente aucune corrélation sur cette *heatmap*.

4. Les graphiques multi-grilles

4.1 FacetGrid

Lorsqu'on souhaite tracer plusieurs graphiques sur une même figure avec Seaborn et donc créer plusieurs `axes` sur la figure, selon différentes modalités, il faut utiliser la fonction `FacetGrid()`.

Syntaxe générale

```
import matplotlib.pyplot as plt
import seaborn as sns
mon_facetgrid = sns.FacetGrid(mon_dataframe, row = 'Variable1',
col = 'Variable2', hue = 'Variable3')
mon_facetgrid.map(sns.scatterplot, 'Variable4', 'Variable5')
```

La première commande consiste à créer l'objet `FacetGrid` avec autant de grilles que ce qu'on veut tracer. La fonction `FacetGrid` attend quatre options :

- Le dataframe contenant nos variables souhaitées.
- `row` : une variable qualitative (ou quantitative discrète). Il y aura autant d'objets `axes` en ligne que de modalités dans la variable donnée à l'option `row`.
- `col` : une variable qualitative (ou quantitative discrète). Il y aura autant d'objets `axes` (de graphiques) en colonne, que de modalités dans la variable donnée.
- `hue` : la variable selon laquelle on veut colorer/séparer les valeurs sur l'ensemble des graphiques. `row` et `col` créent les différents `axes` des différents graphiques, `hue` permettra, pour un nuage de points, de colorer les points selon la variable ou, pour un *countplot*, de séparer les barres selon cette même variable, etc.

© Editions ENI - All rights reserved

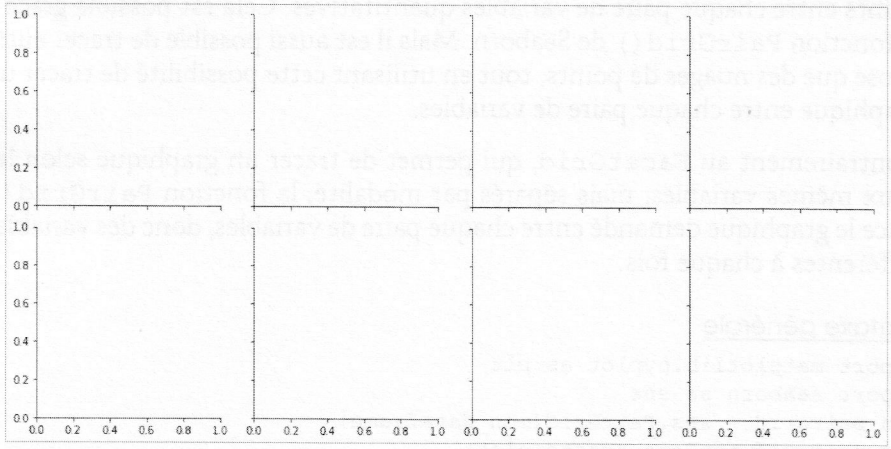
Une fois l'objet `mon_facetgrid` créé, les `axes` sont vides, il faut y tracer des courbes. Pour cela, on utilise la méthode `map()` sur l'objet `FacetGrid` créé. En première option, on détermine le type de graphique à tracer, ici un *scatterplot*, et les variables à tracer en x et en y sur ce *scatterplot*. On obtiendra un *scatterplot* pour chaque paire de modalités des variables `row` et `col`, avec les points colorés selon la variable `hue`.

Créons un nuage de points du poids en fonction de la taille, mais selon différentes modalités. En ligne, nous mettrons la saison des JO, pour séparer les Jeux olympiques d'été et d'hiver. En colonne, nous mettrons les médailles, pour séparer les nuages de points selon la médaille gagnée par l'athlète. Enfin, nous colorerons les points selon le genre de l'athlète.

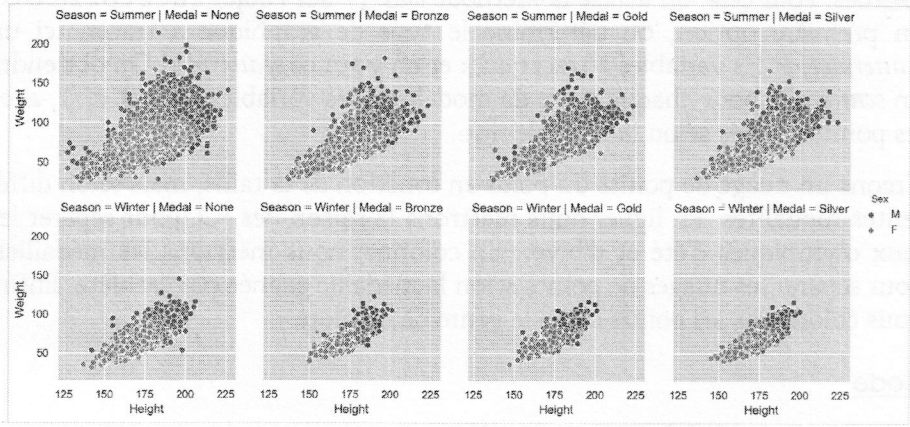
Code

```
import matplotlib.pyplot as plt
import seaborn as sns
mon_facetgrid = sns.FacetGrid(donnees_NaN_supprimes,
row = 'Season', col ='Medal', hue = 'Sex')
mon_facetgrid.map(sns.scatterplot, 'Height', 'Weight')
mon_facetgrid.add_legend()
plt.show()
```

Voici la grille créée par `FacetGrid`, avant de tracer les graphiques dessus :



Résultat après la méthode `map()` :



Vous pouvez constater la puissance de la fonction `FacetGrid()` et la facilité de séparer des graphiques selon différentes modalités.

4.2 PairGrid

Nous avons vu la fonction `pairplot()` de Seaborn pour tracer un nuage de points entre chaque paire de variables quantitatives. Cela est possible grâce à la fonction `PairGrid()` de Seaborn. Mais il est aussi possible de tracer autre chose que des nuages de points, tout en utilisant cette possibilité de tracer un graphique entre chaque paire de variables.

Contrairement au `FacetGrid`, qui permet de tracer un graphique selon les deux mêmes variables, mais séparés par modalité, la fonction `PairGrid()` trace le graphique demandé entre chaque paire de variables, donc des variables différentes à chaque fois.

Syntaxe générale

```
import matplotlib.pyplot as plt
import seaborn as sns
mon_pairgrid = sns.PairGrid(mon_dataframe)
mon_pairgrid.map(sns.scatterplot)
```

© Editions ENI - All rights reserved

Comme précédemment, on initialise la grille, avec la fonction `PairGrid`, cette fois-ci, puis on trace l'ensemble des graphiques avec la méthode `map()`.

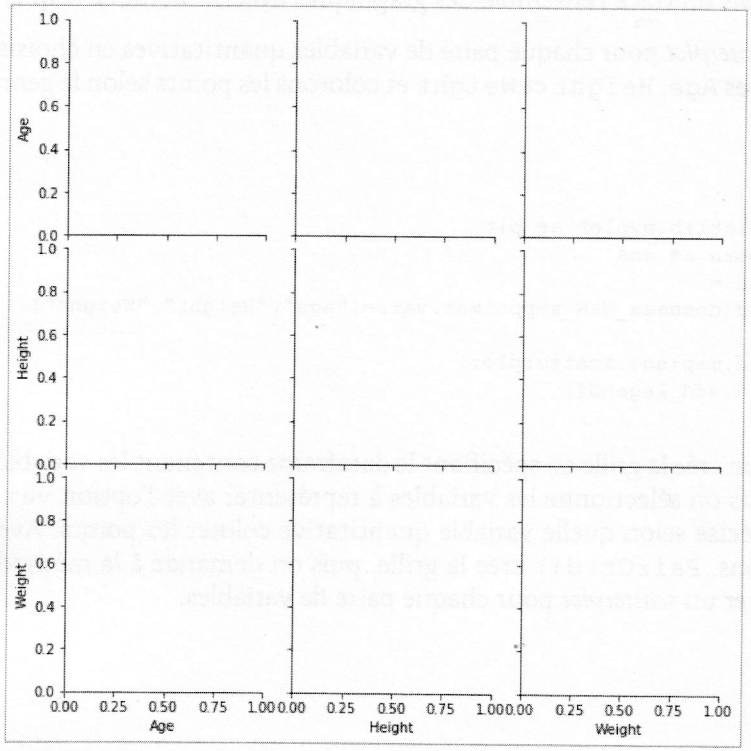
Traçons un *scatterplot* pour chaque paire de variables quantitatives en choisissant les variables `Age`, `Height` et `Weight` et colorons les points selon le genre des athlètes.

Code

```
import matplotlib.pyplot as plt
import seaborn as sns
mon_pairgrid =
sns.PairGrid(donnees_NaN_supprimes,vars=["Age","Height","Weight"],
hue="Sex")
mon_pairgrid.map(sns.scatterplot)
mon_pairgrid.add_legend()
plt.show()
```

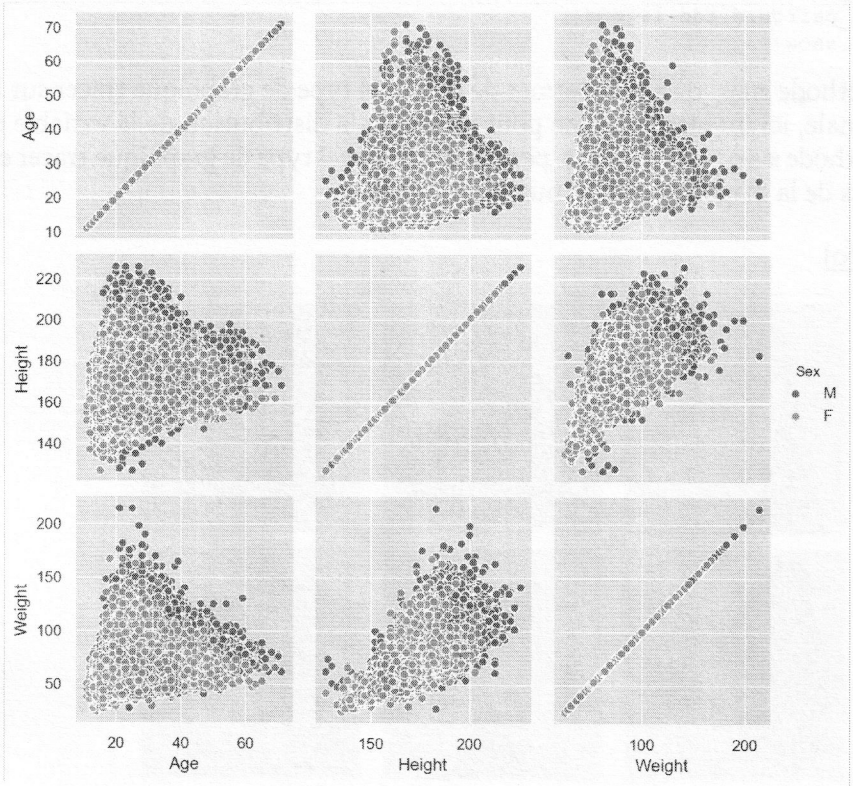
Avec ce code, on crée la grille en spécifiant le dataframe contenant les variables souhaitées, puis on sélectionne les variables à représenter avec l'option `vars` et enfin on précise selon quelle variable quantitative colorer les points. Avec ces informations, `PairGrid()` crée la grille, puis on demande à la méthode `map()` de tracer un *scatterplot* pour chaque paire de variables.

Grille avant de tracer les graphiques :



© Editions ENI - All rights reserved

Résultat



Le graphique sur la diagonale n'a pas d'intérêt, il s'agit des valeurs d'une variable tracée en fonction de ces mêmes valeurs. Il est possible de tracer un type de graphique différent sur la diagonale, grâce aux méthodes `map_diag()` (pour tracer sur la diagonale) et `map_offdiag()` (pour tracer en dehors de la diagonale).

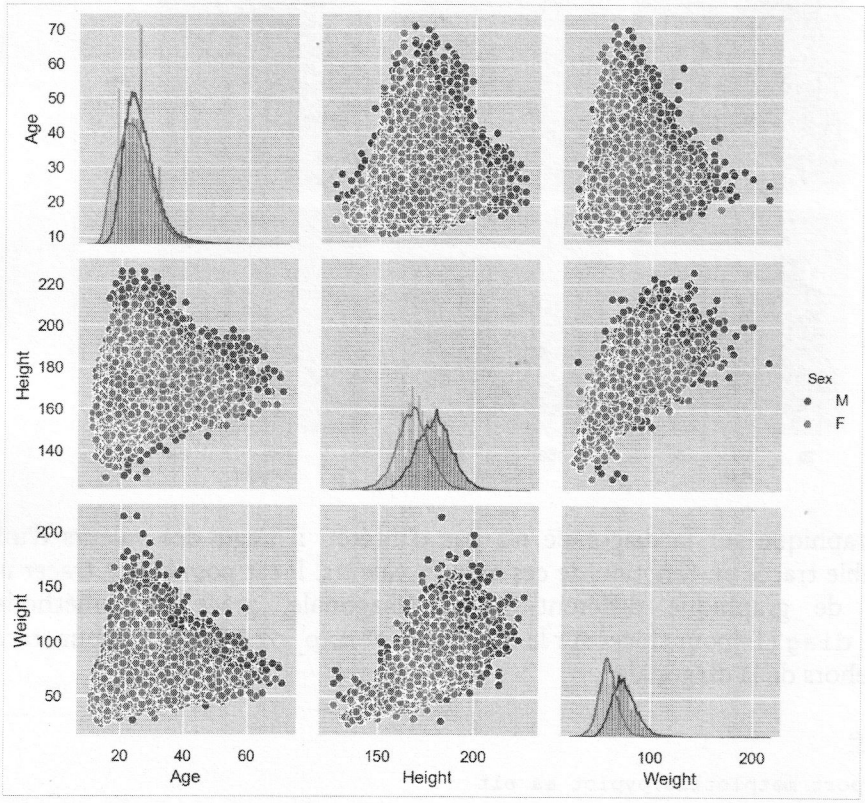
Code

```
import matplotlib.pyplot as plt
import seaborn as sns
mon_pairgrid =
sns.PairGrid(donnees_NaN_supprimes,vars=["Age","Height","Weight"],
hue="Sex")
```

```
mon_pairgrid.map_diag(sns.distplot)
mon_pairgrid.map_offdiag(sns.scatterplot)
mon_pairgrid.add_legend()
plt.show()
```

La méthode `map_diag()` permet de dire quel type de graphique tracer sur la diagonale, ici un histogramme pour visualiser la distribution de la variable et la méthode `map_offdiag()` permet de dire quel type de graphique tracer en dehors de la diagonale, ici un nuage de points.

Résultat



© Editions ENI - All rights reserved

4.3 JointGrid

Précédemment, nous avons vu la fonction `jointplot()`, qui permettait de tracer un nuage de points avec la distribution de chaque variable quantitative tracée. La fonction `jointplot()` crée un objet de type `JointGrid`.

Le but de cet objet est de tracer un graphique bivarié (exemple du nuage de points), ainsi que des graphiques univariés pour chaque variable (par exemple leur distribution). Nous ne nous étendrons pas sur cette fonction, nous vous la présentons pour que vous compreniez comment fonctionne le `jointplot()` et parce que `JointGrid` fait partie des fonctions Seaborn permettant de représenter différents graphiques sur une même figure.

Syntaxe

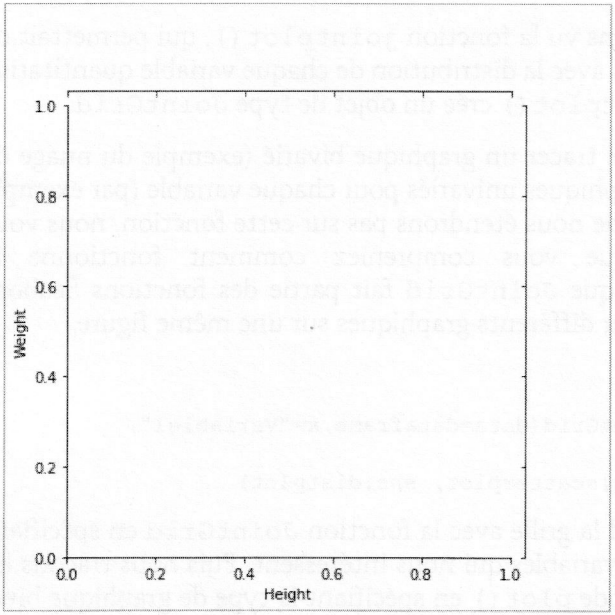
```
mon_JointGrid=sns.JointGrid(data=dataframe,x="Variable1",
y="Variable2")
mon_JointGrid.plot(sns.scatterplot, sns.distplot)
```

Nous créons tout d'abord la grille avec la fonction `JointGrid` en spécifiant le dataframe et les deux variables qui nous intéressent. Puis nous traçons les graphiques avec la méthode `plot()` en spécifiant le type de graphique bivarié, puis le type de graphique univarié à tracer.

Code

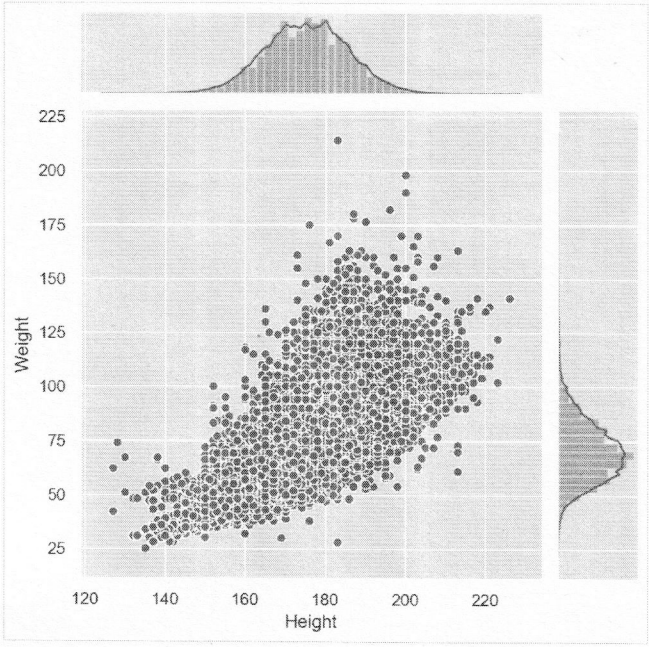
```
import matplotlib.pyplot as plt
import seaborn as sns
mon_JointGrid=sns.JointGrid(data=donnees_NaN_supprimes,x="Height",
y="Weight")
mon_JointGrid.plot(sns.scatterplot, sns.distplot)
plt.show()
```

Grille créée avant de tracer les graphiques :



© Editions ENI - All rights reserved

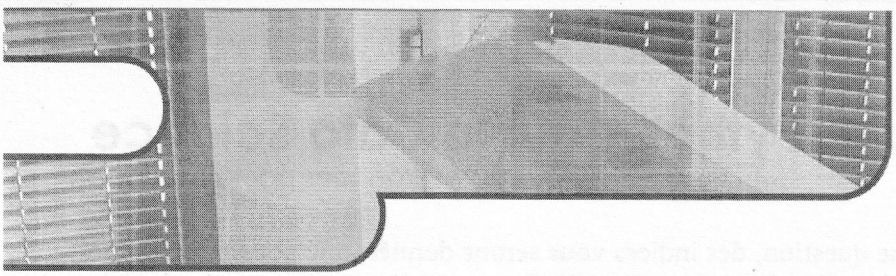
Résultat



À présent, vous avez des connaissances très avancées sur l'utilisation de la librairie Seaborn. Il faut donc maintenant pratiquer, sur votre propre jeu de données ou sur un jeu de données à récupérer sur le site kaggle.com par exemple.

5. Conclusion

Vous voici arrivé à la fin de la lecture de ce livre sur l'analyse de données avec Python et ses librairies dédiées. À présent, vous êtes capable de manipuler, explorer et visualiser des jeux de données structurés avec ce langage très populaire. Afin de tester les connaissances acquises dans ce livre, le chapitre suivant laisse place à un exercice complet, sur un nouveau jeu de données.



Chapitre 8
Exercice complet sur jeu de données réel

1. Introduction

Le but de ce chapitre est d'analyser un nouveau jeu de données afin de s'entraîner sur les notions acquises dans ce livre. Bien sûr, vous pouvez aller plus loin que les questions qui vous seront posées : elles ne couvrent pas forcément toutes les tendances et informations intéressantes à découvrir dans ce nouveau jeu de données.

Dans un premier temps, nous vous présenterons le jeu de données, puis l'énoncé de l'exercice. Pour chaque question qui vous sera posée, ou action qui vous sera demandée, le but sera d'écrire le code permettant de résoudre le problème.

Comme pour les autres chapitres, un notebook associé à ce chapitre est disponible en téléchargement et contient la correction de cet exercice. Bien sûr, si vous trouvez la réponse à la question sans écrire le même code que celui disponible dans le notebook, il n'y a aucun problème. Il existe toujours plusieurs façons et plusieurs codes pour arriver à une même solution. Tout le monde n'a pas la même manière de penser et donc de procéder. Si votre code vous permet d'arriver à la même solution que celle affichée dans le notebook, alors considérez votre code comme juste. Il se peut même que votre code soit plus efficient que celui du corrigé. Prenez le temps de comparer les deux pour vous en faire une idée.

Pour chaque question, des indices vous seront donnés, que vous pourrez lire ou non. Ces indices vous permettront de structurer ce que votre code doit faire, voire vous donneront les fonctions, méthodes ou attributs principaux à utiliser. Le but de cet exercice est vraiment de vous faire réfléchir, donc n'allez pas directement voir la solution proposée. N'hésitez pas à chercher la réponse à votre problématique dans un moteur de recherche ou encore à aller voir la documentation des fonctions, méthodes ou attributs proposés en indices. Il est très courant, pour un informaticien, un développeur ou un analyste de données de chercher des indices via un moteur de recherche ou la documentation de la librairie utilisée. Cela permet d'avoir une piste sur la manière d'adapter la solution d'une problématique proche de la vôtre à votre propre problématique. Toutefois, certaines questions sont complexes et il est normal de ne pas réussir à y répondre : dans ce cas, allez voir la solution pour comprendre la procédure de résolution du problème.

Enfin, dans ce chapitre, nous introduirons une nouvelle notion : celles des méthodes Pandas spécifiques au traitement de données texte. En effet, il est impossible de couvrir l'ensemble de la librairie Pandas dans un livre et nous n'avons pas jugé utile de les intégrer dans celui-ci. Toutefois, cette nouvelle notion est essentielle pour répondre à certaines questions de cet exercice, ce sera donc l'occasion de vous l'expliquer. Cependant, la librairie Pandas est extrêmement vaste, n'hésitez pas à aller vous perdre dans le guide utilisateur de Pandas, disponible ici :
https://pandas.pydata.org/docs/user_guide/index.html#user-guide

Celui-ci est en anglais et contient la description de l'ensemble des concepts, fonctions, méthodes et attributs disponibles dans la librairie Pandas.

© Editions ENI - All rights reserved

2. Présentation du jeu de données

Le jeu de données proposé regroupe l'ensemble des films et séries disponibles sur Netflix jusqu'en 2019. Il est disponible sur le site kaggle.com présenté en introduction de ce livre : https://www.kaggle.com/shivamb/netflix-shows

Ce jeu de données est sous licence CC0 : Public Domain. Ainsi, il peut être copié, modifié, distribué et analysé, même à des fins commerciales.

Il contient douze variables que l'on pourra étudier :

- **show_id** : correspond à l'ID unique du film ou de la série
- **type** : permet de dire s'il s'agit d'un film ("Movie") ou d'une série télévisée ("TV Show")
- **title** : le titre du film/de la série
- **director** : le directeur de production
- **cast** : les acteurs présents dans le film/la série
- **country** : pays dans lequel le film/la série a été produit
- **date_added** : date à laquelle le film/la série a été ajouté sur Netflix
- **release_year** : l'année de sortie réelle du film/de la série
- **rating** : classement du contenu (TV-MA, TV-14, TV-PG, R, PG-13, NR, TV-Y7, TV-G, TV-Y, TV-Y7-FV, G, UR, NC-17). Pour voir la signification des différents classements de contenu : https://fr.wikipedia.org/wiki/TV_Parental_Guidelines
- **duration** : pour les films - durée du film en minutes. Pour les séries - nombre de saisons
- **listed_in** : la ou les catégories du film/de la série (drame, comédie, documentaire, etc.)
- **description** : la description du film/de la série

Ce jeu de données est intéressant car il possède de nombreuses variables avec des informations diverses qui nous permettront une belle exploration. Ainsi, à partir de ce jeu de données, nous vous proposons de répondre aux questions ou d'effectuer les actions énoncées dans la section suivante.

3. Énoncé de l'exercice

Vous pouvez télécharger le fichier `netflix_titles.csv` disponible avec ce livre et de vous créer un nouveau notebook via le tableau de bord Jupyter (pour créer un nouveau notebook, voir la section Premiers pas avec les notebooks du chapitre Mise en place de l'environnement de travail).

3.1 Lire le fichier

Ici, il s'agit de récupérer le fichier disponible en téléchargement de ce livre : netflix_titles.csv. Vous pouvez aussi le récupérer directement via le site : https://www.kaggle.com

Action : écrire le code qui permet de lire le fichier netflix_titles.csv, de définir la première colonne du fichier comme l'index du dataframe et de stocker ce dataframe dans une variable nommée `donnees`. Enfin, afficher un aperçu de ce dataframe pour prendre connaissance visuellement des différentes variables du tableau.

Indices :

- N'oubliez pas d'importer la librairie Pandas pour pouvoir utiliser ses fonctions.
- La fonction `read_csv()` de Pandas permet de lire un fichier.
- L'option `index_col` permet de définir le numéro de colonne à utiliser comme index.
- La méthode `head()` permet d'afficher un aperçu.

© Editions ENI - All rights reserved

3.2 Afficher les dimensions du dataframe

Action :écrire le code pour afficher les dimensions du dataframe, c'est-à-dire le nombre de lignes et le nombre de colonnes. Cela vous permettra de connaître le nombre de films/séries stockés dans ce tableau (nombre de lignes), ainsi que le nombre de variables (nombre de colonnes).

Indice :

L'attribut `shape` permet d'afficher la dimension d'un dataframe.

3.3 Compter les films et les séries

Action : écrire le code qui permet d'afficher le nombre de films et le nombre de séries stockés dans ce jeu de données.

Indices :

- La variable utile est la variable `type`, qui contient les deux modalités `Movie` (film) ou `TV Show` (série). Il faut la sélectionner avec l'attribut `loc`.
- La méthode `value_counts()` de Pandas permet de compter le nombre d'occurrences de chaque valeur unique (chaque modalité) d'une variable.

3.4 Générer le résumé statistique du dataframe

Action : générer le code pour afficher le résumé statistique du dataframe pour l'ensemble des variables.

Indices :

- La méthode `describe()` de Pandas permet de générer un résumé statistique.
- L'option `include=all` de la méthode `describe()` permet d'afficher le résumé statistique sur l'ensemble des variables. Par défaut, ce résumé se fait uniquement sur les variables numériques.

3.5 Compter les valeurs manquantes

Action : écrire le code pour afficher le nombre de valeurs manquantes par colonne du dataframe.

Indices :

- La méthode `isna()` permet de générer un dataframe de booléens avec la valeur `False` si la valeur n'est pas `NaN` et `True` si la valeur est `NaN`.
- La méthode `sum()` permet de compter le nombre de valeurs `True` dans un dataframe et donc le nombre de valeurs manquantes.

3.6 Explorer les valeurs manquantes

3.6.1 Sur la colonne des directeurs de production

Action : écrire le code pour sélectionner les lignes pour lesquelles la valeur est manquante dans la colonne `director`, puis compter le nombre de films et de séries sans directeur de production.

Indices :

- Créer un sous-tableau contenant uniquement les lignes avec une valeur manquante dans la colonne `director` grâce à l'attribut `loc` et à la méthode `isna()`.
- Sélectionner la colonne `type` sur ce sous-tableau grâce à l'attribut `loc` et compter les occurrences de chaque modalité avec la méthode `value_counts()`.

3.6.2 Sur la colonne des acteurs

Action : écrire le code pour sélectionner les films et séries sans acteurs, c'est-à-dire ceux pour lesquels la valeur dans la colonne `cast` est manquante. Puis afficher les dix catégories (colonne `listed_in`) qui ont le plus de valeurs manquantes dans la colonne `cast`.

© Editions ENI - All rights reserved

Indices :

- Créer un sous-tableau contenant uniquement les lignes avec une valeur manquante dans la colonne `cast` grâce à l'attribut `loc` et la méthode `isna()`.
- Sélectionner la colonne `listed_in` sur ce sous-tableau grâce à l'attribut `loc` et compter les occurrences de chaque modalité avec la méthode `value_counts()`.
- La méthode `head()` permet d'afficher un aperçu d'un dataframe, l'option `n` permet de déterminer le nombre de lignes à afficher.

3.7 Supprimer les lignes dupliquées

Action : écrire le code pour compter le nombre de lignes dupliquées et pour les supprimer du dataframe.

Indices :

- La méthode `duplicated()` permet de générer une série de booléens et donc de l'utiliser en *indexing* pour afficher les lignes dupliquées.
- La méthode `drop_duplicates()` permet de supprimer les lignes dupliquées.

3.8 Compter les films/séries produits par les États-Unis et par la France

Action : écrire le code pour compter le nombre de films/séries produits par les États-Unis et par la France.

Indices :

- L'indexation booléenne permet de sélectionner les lignes pour lesquelles une colonne est égale à une certaine valeur. Ici, la variable utile est `country`.
- La syntaxe suivante permet de faire de l'indexation booléenne : `dataframe[(dataframe["variable"]=="valeur")]`.
- L'attribut `shape` permet d'afficher la dimension d'un dataframe et donc le

nombre de lignes.

3.9 Afficher le contenu le plus vieux disponible sur Netflix

Action : écrire le code permettant d'afficher la ligne du contenu pour lequel l'année de sortie réelle (`release_year`) est la plus ancienne.

Indices :

- La méthode `min()` permet de récupérer la valeur minimale d'un ensemble de valeurs et donc la valeur minimale d'une colonne de dataframe.
- Une fois l'année minimale connue, l'indexation booléenne permet de récupérer la ligne pour laquelle la valeur dans la colonne `release_year` est égale à cette date.

3.10 Afficher le film avec la durée la plus longue sur Netflix

3.10.1 Nouvelle notion : les méthodes str

Il existe des méthodes Pandas spécifiques pour traiter des données de type chaînes de caractères (`str` en Python). Généralement, ces méthodes sont précédées du terme `str` (qui est un attribut) et sont dédiées, comme leur nom l'indique, au traitement des données de type `str` spécifiquement.

Ces méthodes permettent, entre autres, de remplacer des chaînes de caractères par d'autres (`str.replace()`) ou encore de sélectionner des lignes contenant une sous-chaîne de caractères (`str.contains()`). Ce sont les deux méthodes que nous verrons dans ce chapitre. Retenez simplement que si vous avez des chaînes de caractères à traiter, il faut aller regarder spécifiquement la partie de la documentation Pandas dédiée à ce thème :
https://pandas.pydata.org/docs/user_guide/text.html

Syntaxe générale de `str.replace()` :

```
ma_serie.str.replace("chaine1", "chaine2")
```

Ici, la chaîne de caractères `chaine1` sera remplacée par `chaine2` dans l'ensemble de la série.

© Editions ENI - All rights reserved

Syntaxe générale de `str.contains()` :

```
mon_dataframe[mon_dataframe["ma_colonne"].str.contains("chaine")]
```

Dans le cas d'un dataframe, on pourra utiliser la méthode `str.contains()` pour faire de l'indexation et récupérer les lignes où une colonne spécifique contient le mot `chaine`.

3.10.2 Énoncé

Action : le problème de la colonne `duration` est que les données qu'elle contient sont des chaînes de caractères, du fait que le mot `" min"` est écrit à la suite du nombre de minutes. Il n'est donc pas possible de trouver la valeur numérique la plus grande tant que cette colonne est une chaîne de caractères. Ainsi, il faut écrire le code pour convertir les données de la colonne `duration` en entier puis trier numériquement ces valeurs pour afficher les cinq films avec la durée la plus longue. Ce code n'est pas simple, n'hésitez pas à vous aider des indices ou à aller étudier la solution proposée dans le notebook.

Indices :

- Créer un sous-tableau contenant uniquement les contenus de type `Movie` grâce à l'indexation booléenne.
- À partir du sous-tableau, créer une série correspondant à la colonne `duration`, et la ranger dans une variable nommée, par exemple, `duree`.
- Utiliser la fonction `str.replace()` sur la série `duree` pour remplacer la chaîne de caractères `" min"` par une chaîne vide `""`, ce qui permet d'enlever le texte des valeurs contenues dans la série.
- Utiliser la méthode `astype()` pour transformer le type des valeurs contenues dans la variable `duree` en un autre type, ici on souhaite des entiers (`int`).
- Trier les données de cette série avec la méthode `sort_values`, dans l'ordre décroissant.
- Utiliser les index de la série contenant les valeurs de durée triées, grâce à l'attribut `index`, pour récupérer les lignes du dataframe originel, afin que celles-ci soient triées selon la durée. Enfin, afficher les cinq premières lignes.

3.11 Étudier les catégories avec le plus de contenu

Ici, nous allons étudier ensemble les catégories avec le plus de contenu, c'est-à-dire que nous allons nous intéresser à la colonne `listed_in`. La difficulté des colonnes `listed_in`, `director` et `cast` est qu'elles peuvent contenir des listes de valeurs.

Nous allons voir ensemble comment analyser des listes de valeurs en prenant comme exemple la colonne `listed_in`. Ensuite, vous devrez adapter ce code pour les colonnes `director` et `cast`.

Voici un aperçu de la colonne `listed_in`.

Code

```
import pandas as pd
donnees=pd.read_csv("../datasets/netflix_titles.csv",
index_col=[0])
donnees["listed_in"].head()
```

Ici, on lit le fichier et on affiche un aperçu de la colonne `listed_in`.

Résultat

```
show_id
81145628    Children & Family Movies, Comedies
80117401                       Stand-Up Comedy
70234439                               Kids' TV
80058654                               Kids' TV
80125979                               Comedies
Name: listed_in, dtype: object
```

© Editions ENI - All rights reserved

On constate que chaque film/série peut appartenir à une ou plusieurs catégories. Dans le cas où il appartient à plusieurs catégories, celles-ci sont séparées par une virgule et un espace `", "`. Nous devons *splitter*, couper, ces catégories pour pouvoir les étudier, sinon Python considérera `"Children & Family Movies, Comedies"` comme une unique catégorie, alors qu'en réalité il y en a deux ici : `"Children & Family Movies"` et `"Comedies"`.

Pour faire ça, nous allons voir une méthode Pandas non introduite dans le cours, mais que vous connaissez peut-être si vous travaillez déjà avec Python, car cette méthode existe plus généralement en Python. Il s'agit de la méthode `join()`, qui permet de joindre tous les éléments d'un tuple, d'une liste ou même d'une série, séparés par un caractère que nous lui donnons en début de code.

Pour illustrer ce propos, utilisons la méthode `join()` sur la colonne `listed_in` de notre dataframe stocké dans la variable `donnees` et joignons l'ensemble des éléments de la colonne avec le séparateur `", "` (une virgule et un espace). On ignore les valeurs manquantes grâce à la méthode `dropna()`.

Code

```
", ".join(donnees["listed_in"].dropna())
```

Aperçu du résultat

```
"Children & Family Movies, Comedies, Stand-Up Comedy, Kids' TV,
Kids' TV, Comedies, Crime TV Shows, International TV Shows,
Spanish-Language TV Shows, International Movies, Sci-Fi & Fantasy,
Thrillers, Stand-Up Comedy, Docuseries, Science & Nature TV,
Action & Adventure, Thrillers, Stand-Up Comedy, Action & Adventure,
Dramas, International Movies, Children & Family Movies,
Children & Family Movies, Children & Family Movies,..."
```

Chaque valeur de la colonne `listed_in`, contenait une ou plusieurs valeurs séparées par `", "`. En joignant l'ensemble des valeurs par le séparateur `", "`, on obtient une chaîne de caractères unique contenant toutes les catégories au sein de la série, films et séries confondus.

Le but, à présent, est d'utiliser la méthode `split()` pour couper cette chaîne de caractères selon le motif `", "`, ce qui nous permettra d'obtenir une série avec une valeur pour chaque catégorie. Grâce au séparateur `", "`, nous allons pouvoir séparer chaque catégorie et enfin, compter le nombre d'occurrences de chaque catégorie.

Code

```
", ".join(donnees["listed_in"].dropna()).split(", ")
```

Résultat

```
['Children & Family Movies',
'Comedies',
'Stand-Up Comedy',
"Kids' TV",
"Kids' TV",
'Comedies',
...
]
```

Ici, on obtient une liste contenant toutes les catégories du jeu de données Netflix, films et séries confondus. On constate que `"Children & Family Movies"` et `"Comedies"` sont maintenant deux catégories bien séparées. Ainsi, nous allons transformer cette liste en série, pour pouvoir ensuite utiliser la méthode `value_counts()` que nous connaissons bien, afin d'avoir le nombre d'occurrences de chaque catégorie dans le dataframe. En effet, la méthode `value_counts()` est une méthode Pandas, qui fonctionne donc sur une série et non sur une liste.

Code

```
categories=pd.Series
(", ".join(donnees["listed_in"].dropna()).split(", "))
categories.value_counts().head()
```

© Editions ENI - All rights reserved

Résultat

```
International Movies      1926
Dramas                    1622
Comedies                  1113
International TV Shows    1001
Documentaries              668
dtype: int64
```

À présent, on peut conclure que la catégorie contenant le plus de contenu est la catégorie "International Movies", suivi par "Dramas" et "Comedies". Si nous n'avions pas fait tout ce traitement, les résultats auraient été grandement biaisés.

3.12 Afficher les directeurs qui ont produit le plus de films/séries disponibles sur Netflix

Action : adapter le code précédent pour la colonne `director` plutôt que la colonne `listed_in`, afin d'afficher les dix directeurs de production les plus présents dans le jeu de données Netflix.

Indices :

- Remplacer le nom de colonne `listed_in` par `director` dans le code précédent.

3.13 Voir si Jan Suter travaille souvent avec les mêmes acteurs

Action : afficher les cinq acteurs les plus présents dans les contenus produits par Jan Suter et en déduire si ce directeur travaille préférentiellement avec certains acteurs ou non.

Indices :

- Garder uniquement les lignes du dataframe où les valeurs de la colonne `director` ne sont pas manquantes, grâce à la méthode `notna()`.
- Utiliser la méthode `str.contains()` pour sélectionner uniquement les lignes dont la colonne `director` contient la chaîne de caractères "Jan

`Suter"` et ranger ce nouveau dataframe dans une variable nommée, par exemple, `donnees_jan_suter`.

- Adapter le code précédent, utilisé pour les colonnes `listed_in` et `director`, afin d'afficher les cinq acteurs les plus présents dans le dataframe `donnees_jan_suter`.

3.14 Représenter les dix pays qui ont produit le plus de contenus disponibles sur Netflix, avec le nombre de contenus par pays

Action : écrire le code pour sélectionner les dix pays avec le plus de contenus, créer un sous-dataframe contenant uniquement les données de ces pays, puis tracer un *countplot* sur ce dataframe et la variable `country`.

Indices :

- Sélectionner les dix pays ayant produit le plus de contenus grâce à la méthode `value_counts()` et la méthode `head()` et stocker cette série dans une variable.
- Créer un dataframe contenant uniquement les lignes pour ces dix pays, c'est-à-dire les lignes pour lesquelles la valeur dans la colonne `country` est comprise dans les index de la série créée précédemment. Pour cela, utiliser l'indexation booléenne, la méthode `.isin()` et l'attribut `index`.
- Tracer un *countplot* grâce à la fonction `countplot()` de Seaborn en lui donnant le nouveau dataframe et la variable souhaitée : `country`.

© Editions ENI - All rights reserved

3.15 Tracer un graphe à barres du nombre de films/séries par classement de contenu (rating)

Action : tracer le graphe à barres en utilisant le dataframe d'origine et la colonne d'intérêt représentant le classement de contenu.

Indice :

- La fonction pour tracer un graphe à barres est `countplot()`.
- La colonne utile est la colonne `rating`.

3.16 Afficher l'évolution du nombre de films/séries disponibles sur Netflix au cours du temps

3.16.1 Notions supplémentaires sur les dates

Nous avions vu comment formater une colonne `date` directement à la lecture du fichier. Il est possible de le faire après lecture grâce à la fonction `to_datetime()` de Pandas, qui permet de formater une colonne de dataframe dans le type `datetime64`.

Il existe des attributs pour les données de type `datetime64` qui permettent d'accéder à des valeurs spécifiques, comme le jour `dt.day`, le mois `dt.month` ou encore l'année `dt.year`.

Syntaxe pour l'utilisation de ces attributs :

```
dataframe["colonne"].dt.day
dataframe["colonne"].dt.month
dataframe["colonne"].dt.year
```

3.16.2 Énoncé

Action : écrire le code pour formater correctement la colonne `date_added` et créer une nouvelle colonne dans le dataframe contenant uniquement l'année d'ajout du contenu sur Netflix. Grouper le dataframe sur les variables `year_added` et `type`, afin de tracer l'évolution du nombre de contenus ajoutés par an grâce à la fonction `pointplot()` de Seaborn. Séparer les courbes selon le type du contenu, `Movie` ou `TV Show`.

Indices :

- La méthode `to_datetime()` permet de formater la colonne `date_added`, date d'ajout du contenu sur Netflix.
- À partir de la colonne `date_added` correctement formatée, récupérer l'année d'ajout du contenu sur Netflix avec l'attribut `dt.year` et créer une nouvelle colonne dans le dataframe, nommée par exemple `year_added`.
- Grouper le dataframe sur les variables souhaitées grâce à la méthode `groupby()` de Pandas, déterminer le nombre d'occurrences par groupe grâce à la méthode `size()` puis renommer la colonne de comptage en `Count`. Un exemple est disponible dans la section sur la fonction `pointplot()` du chapitre Seaborn.
- Utiliser la fonction `pointplot()` de Seaborn avec en `x` la variable `year_added`, en `y` la variable `Count` et comme option `hue` la variable `type`, afin d'avoir une courbe par type de contenu.

3.17 Afficher la distribution de la durée des films disponibles sur Netflix

Action : écrire le code pour sélectionner uniquement les contenus correspondant au type `Movie` et transformer la durée des films en nombre entier, comme dans la section Afficher le film avec la durée la plus longue sur Netflix de ce chapitre. Enfin, tracer un histogramme sur la série contenant les données de durée des films.

Indices :

- Créer une série contenant les durées de films en nombres entiers.
- Utiliser la fonction `distplot()` de Seaborn pour tracer la distribution de ces durées.

© Editions ENI - All rights reserved

3.18 Tracer un graphique représentant le nombre de séries par modalité de nombre de saisons

Action : créer un dataframe des contenus de type `TV Show` uniquement puis tracer un `countplot()` pour représenter le nombre de séries pour chaque modalité du nombre de saisons.

Indices :

- Utiliser la sélection booléenne pour créer le dataframe des séries.
- La variable utile en x est la variable `duration`, contenant l'information du nombre de saisons pour chaque série.

!= (opérateur relationnel, inégalité), 105
(), parenthèses, 70
[], crochets, 62, 103
{}, accolades, 73
* (opérateur arithmétique, multiplication), 90
& (opérateur bitwise, et), 177
% (opérateur arithmétique, modulo), 79
+ (opérateur arithmétique, addition), 61
- (opérateur arithmétique, soustraction), 78
< (opérateur relationnel, strictement inférieur), 105
<= (opérateur relationnel, inférieur ou égal), 105
= (symbole d'assignation), 56
== (opérateur relationnel, égalité), 105
> (opérateur relationnel, strictement supérieur), 105
>= (opérateur relationnel, supérieur ou égal), 105
| (opérateur bitwise, ou), 177
~ (opérateur bitwise, négation), 177
/ (opérateur arithmétique, division), 79
// (opérateur arithmétique, résultat entier de la division), 79

Aesthetic, 303
Ajout d'éléments
    dataframe, 201
    dictionnaire, 74
    liste, 66
    ndarray, 124
    série, 187

Anaconda, 23
Anaconda Navigator, 28
and (opérateur logique, et), 81
append(), 66
Architecture de Matplotlib, 242
    axes, 244
    axis, 244
    figure, 244
    subplot, 245
Attribut
    dtype, 96
    iloc, 175
    loc, 174
    shape, 122
    size, 85

C

Cellule, 41
Contexte (seaborn), 309
Créer
    dictionnaire, 73
    liste, 62
    ndarray, 94
    série, 163

D

Données
    dupliquées, 215
    manquantes, 210
    non structurées, 13
    ouvertes, 12
    structurées, 13

© Editions ENI - All rights reserved

E

Écriture de fichiers
savefig(), 256
savetxt(), 103
to_csv(), 160
to_excel(), 161

F

Format
CSV, 143
DB, 156
JSON, 158
TXT, 143
XLS, 153
XLSX, 153

G

Graphiques divers
FacetGrid, 352
heatmap, 349
JointGrid, 359
PairGrid, 354
pointplot, 327
Graphiques pour variables qualitatives
bargraph (graphe à barres), 282
countplot (graphe à barres), 336
stripplot (nuage de points), 331

Graphiques pour variables quantitatives
boxplot (boîte à moustaches), 296, 334
histogramme, 339
jointplot, 344
pairplot, 346
régression, 323
scatterplot (nuage de points), 279, 318

I

Index (indice), 63
Indexation
booléenne, 105
simple, 103
IPython, 20

J

Jeu de données (JO), 140

K

Kernel, 35
redémarrer, 39

© Editions ENI - All rights reserved

L

Lecture de fichier
- connect(), 156
- genfromtxt(), 101
- read_csv(), 144
- read_excel(), 153
- read_json(), 160
- read_sql(), 156
- read_table(), 144

M

Markdown, 45

Méthodes principales des dataframes et séries
- agg(), 232
- apply(), 234
- describe(), 185, 222
- duplicated(), 216
- fillna(), 213
- groupby(), 228
- head(), 146
- is.na(), 210
- pivot_table(), 236
- replace(), 207
- sort_index(), 226
- sort_values(), 225
- stack(), 238
- sum(), 210
- unstack(), 238
- value_counts(), 185

Modification des éléments
dataframe, 207
dictionnaire, 75
liste, 68
ndarray, 104
série, 189

N

not (opérateur logique, négation), 82
Notebook, 34

O

Opérateur
arithmétique, 78
bitwise, 177
logique, 81
relationnel, 80
or (opérateur logique, ou), 81

P

Package, 87
Matplotlib, 53
NumPy, 52
Pandas, 53
Seaborn, 53
Palettes de couleur, 314

© Editions ENI - All rights reserved

Python, 14
attribut, 85
fonction, 82
librairie, 87
méthode, 84
module, 85

R

R, 14

S

Sélectionner des éléments
indexing, 103, 194
slicing, 64, 194
Structure de données
dataframe, 191
dictionnaire, 72
liste, 62
matrice, 135
ndarray, 93
panel, 227
série, 162
tuple, 70
Suppression des éléments
dataframe, 205
dictionnaire, 76
liste, 69
ndarray, 127
série, 188

T

Thème (seaborn), 304
Type de données, 58
    bool, 60
    float, 58
    int, 58
    object, 167
    str, 60

V

Variable, 56, 222
    qualitative, 223
    quantitative, 222

© Editions ENI - All rights reserved

livre & vidéo

vBook

public

Data Scientist

Python pour la Data Science

livre

Un **livre de référence** sur la Data Science avec le langage Python, alliant théorie et pratique, pour apprendre à analyser des données.

Amandine VELT

Diplômée du master Bioinformatique Modélisation et Statistique de l'Université de Rouen Normandie, **Amandine VELT** est ingénieure en Bioinformatique à l'INRAE (Institut National de Recherche pour l'Agriculture, l'Alimentation et l'Environnement). Passionnée par la programmation, le traitement de données de grandes volumétries et l'enseignement, elle est aussi formatrice indépendante en Data Science avec les langages de programmation Python et R. Ses deux formations en ligne dans le domaine regroupent environ 3000 participants. Ce livre, qu'elle a voulu pratique et concret, est dédié aux lecteurs désireux d'apprendre à analyser des données ave le langage Python.

vidéo

Un **approfondissement sous forme de vidéo*** (1 H 15) qui présente NumPy et Pandas, deux bibliothèques dédiées à la Data Science et à la visualisation de données.

Madjid KHICHANE

Après un diplôme d'ingénieur en Informatique obtenu à l'université Mouloud Mammeri à Tizi-Ouzou en Algérie puis un Master en Intelligence Artificielle - Systèmes multi-agents obtenu à l'université Paris 5 (René Descartes), **Madjid KHICHANE** a soutenu son PhD en Intelligence Artificielle en collaboration entre l'Université Claude Bernard Lyon 1 et IBM. Cette thèse doctorale a donné naissance à des innovations algorithmiques dans le domaine de l'apprentissage par renforcement qui sont aujourd'hui publiées dans des conférences internationales de premier niveau.

* Accessible via Internet quel que soit votre support et votre navigateur (PC, tablette...) grâce à un numéro d'activation personnel.

Flashez-moi pour en savoir plus et découvrir un extrait gratuit

ISBN : 978-2-409-03930-0

49 €

eni Editions

www.editions-eni.fr