

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.

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.

a. 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.

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`.



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 :



datasets



notebooks

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.

b. 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.

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 souhaitons 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 Djiang | 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 Jacobsa 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.

c. 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.

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.



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 Lindenu 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 Jacobsa Aaltink | 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.

d. 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)
```

e. 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])
```

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.

f. 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.

g. 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 pourrions constater que le `dtype` de notre colonne est `datetime64`.

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. 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.

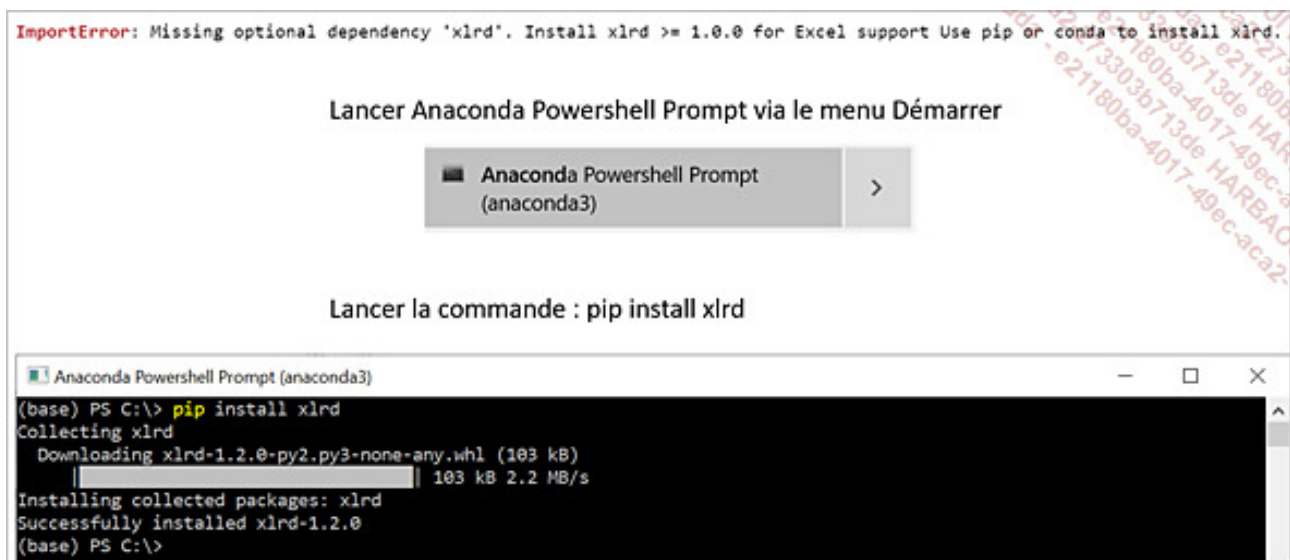
Résultat

| ID | Name | Sex | Age | Height | Weight | Team | NOC | Games | Year | Season | City | Sport | Event | Medal |
|----|---------------------|-----|------|--------|--------|---------|-----|-------------|------|--------|-----------|------------|------------------------------|-------|
| 1 | A Dīāng | 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 |

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`.

Structure de données Pandas : les Series (Séries)

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.

| | Variable | Nom |
|---------------------------------------|----------|---------|
| 0 | 2541 | Valeurs |
| 1 | 265 | |
| 2 | 10 | |
| 3 | 4521 | |
| 4 | 654 | |
| 5 | 59 | |
| Index (individus /observations) | | |

Une série peut être créée à partir d'une liste, d'un tableau NumPy, ou encore d'un dictionnaire, grâce à la méthode `Series()`. Elle peut aussi être créée à partir d'un fichier grâce aux fonctions de lecture de fichiers que nous avons vues précédemment, telles que `read_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()`.

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.

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.

a. 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

```



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).

b. 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
```



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
```

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).



Avec Pandas, les différents types sont les suivants :

`object` = chaînes 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)

c. 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.

d. 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_default=pd.read_csv("../datasets/120-
years-of-olympic-history-athletes-and-results.csv", usecols=[5],
squeeze=True)
print(ma_serie_de_poids_index_default)
```

Résultat

```
0    80.0
1    60.0
2     NaN
3     NaN
4    82.0
...
```

```

271111  89.0
271112  59.0
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 271 115. 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 271 116. 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. 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 Aaby    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.

4. Accéder aux valeurs d'une série

a. 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_default[[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.

b. 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"]]
```

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.

c. 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`.

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`.

d. 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]
```

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.



Dans les chapitres précédents, et notamment dans le chapitre sur les rappels Python, nous avons 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
```

e. 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.

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_default.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_default.iloc[2:11]
```

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_default`.

Code

```
ma_serie_de_poids_index_default.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 une 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_default.iloc[:20]
```

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_default.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.

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>

a. 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
```

Résultat

```
271116
```

Pour rappel, les attributs sont des variables et s'appellent donc sans parenthèses, contrairement aux méthodes.

b. 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()`.



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.

6. Ajouter, supprimer et modifier les valeurs d'une série

a. 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.



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()`.

b. 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.

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.



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.

c. 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
```

Modifions son poids et assignons la valeur 60 à l'étiquette Christine Jaoba Aaftink.

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.