Manipulation de données avec `pandas`

- *Author* : **Ibrahima SY**
- *Email* : syibrahima31@gmail.com
- *Github* : Cliquez
- *Linkdin* : Cliquez
- *School* : Institut Supérieur Informatique (**ISI**)
- *Spécialité* : Master 2 **IAGE**

`pandas` est une librairie open-source basée sur `NumPy` fournissant des structures de données facile à manipuler, et des outils d'analyse de données. Le lecteur familier avec les fonctions de base du langage `R` retrouvera de nombreuses fonctionnalités similaires avec `pandas`.

Pour avoir accès aux fonctionnalités de `pandas`, il est coutume de charger la librairie en lui accordant l'alias `pd` :

```
>>> import pandas as pd
```

Nous allons également utiliser des fonctions de `numpy` (vu dans le chapitre précédent). Assurons-vous de charger cette librairie, si ce n'est pas déjà fait :

```
>>> import numpy as np
```

1. Structures

Nous allons nous pencher sur deux types de structures, les séries (`serie`) et les dataframes (`DataFrame`).

1.1 Séries

Les séries sont des tableaux à une dimension de données indexées.

1.1.1 Création de séries à partir d'une liste

Pour en créer,on peut définir une liste, puis appliquer la fonction `Series` de `pandas` :

```
>>> s = pd.Series([1, 4, -1, np.nan, .5, 1])
>>> print(s)
## 0    1.0
## 1    4.0
## 2   -1.0
## 3    NaN
## 4    0.5
## 5    1.0
## dtype: float64
```

L'affichage précédent montre que la série `s` créée contient à la fois les données et un `index` associé. L'attribut `values` permet d'afficher les valeurs qui sont stockées dans un tableau `numpy` :

```
>>> print("valeur de s : ", s.values)
## valeur de s :  [ 1.   4.  -1.   nan  0.5  1. ]
>>> print("type des valeurs de s : ", type(s.values))
## type des valeurs de s :  <class 'numpy.ndarray'>
```

L'indice est quand à lui stocké dans une structure spécifique de `pandas` :

```
>>> print("index de s : ", s.index)
## index de s :  RangeIndex(start=0, stop=6, step=1)
>>> print("type de l'index de s : ", type(s.index))
## type de l'index de s :  <class 'pandas.core.indexes.range.RangeIndex'>
```

Il est possible d'attribuer un nom à la série ainsi qu'à l'index :

```
>>> s.name = "ma_serie"
>>> s.index.name = "nom_index"
>>> print("nom de la série : {} , nom de l'index : {}".format(s.name,
s.index.name))
## nom de la série : ma_serie , nom de l'index : non_index
>>> print("série s : \n", s)
## série s :
## nom_index
##  0    1.0
## 1    4.0
## 2   -1.0
## 3    NaN
## 4    0.5
## 5    1.0
## Name: ma_serie , dtype: float64
```

1.1.2 Définition de l'index

L'index peut être défini par l'utilisateur, au moment de la création de la série :

```
>>> s = pd.Series([1, 4, -1, np.nan],
                  index = ["o", "d", "i", "l"])
>>> print(s)
## o    1.0
## d    4.0
## i   -1.0
## l    NaN
## dtype: float64
```

On peut définir l'indice avec des valeurs numériques également, sans être forcé de respecter un ordre précis :

```
>>> s = pd.Series([1, 4, -1, np.nan],
                  index = [4, 40, 2, 3])
>>> print(s)
## 4     1.0
## 40    4.0
## 2    -1.0
## 3     NaN
## dtype: float64
```

L'index peut être modifié par la suite, en venant écraser l'attribut `index` :

```
>>> s.index = ["o", "d", "i", "l"]
>>> print("Série s : \n", s)
## Série s :
##  o    1.0
## d    4.0
## i   -1.0
## l    NaN
## dtype: float64
```

1.1.3 Création de séries particulières

Il existe une petite astuce pour créer des séries avec une valeur répétée, qui consiste à fournir un scalaire à la fonction `Series` de `NumPy` et un index dont la longueur correspondra au nombre de fois où le scalaire sera répété :

```
>>> s = pd.Series(5, index = [np.arange(4)])
>>> print(s)
## 0    5
## 1    5
## 2    5
## 3    5
## dtype: int64
```

On peut créer une série à partir d'un dictionnaire :

```
>>> dictionnaire = {"Roi": "Arthur",
                    "Chevalier_pays_galles": "Perceval",
                    "Druide": "Merlin"}
>>> s = pd.Series(dictionnaire)
>>> print(s)
## Roi                        Arthur
## Chevalier_pays_galles    Perceval
## Druide                     Merlin
## dtype: object
```

Comme on le note dans la sortie précédente, les clés du dictionnaire ont été utilisées pour l'index. Lors de la création de la série, on peut préciser au paramètre clé des valeurs spécifiques : cela aura pour conséquence de ne récupérer que les observations correspondant à ces clés :

```
dictionnaire = {"Roi": "Arthur",
                "Chevalier_pays_galles": "Perceval",
                "Druide": "Merlin"}
s = pd.Series(dictionnaire, index = ["Roi", "Druide"])
print(s)
## Roi       Arthur
## Druide    Merlin
## dtype: object
```

1.2 Dataframes

Les Dataframes correspondent au format de données que l'on rencontre classiquement dans tous les domaines , des tableaux à deux dimensions, avec des variables en colonnes et des observations en ligne. Les colonnes et lignes des dataframes sont indexées.

1.2.1 Création de dataframes à partir d'un dictionnaire

Pour créer un dataframe, on peut fournir à la fonction `DataFrame()` de `pandas` un dictionnaire pouvant être transformé en `serie`. C'est le cas d'un dictionnaire dont les valeurs associées aux clés ont toutes la même longueur :

```
>>> dico = {"height" :
                [58, 59, 60, 61, 62,
                 63, 64, 65, 66, 67,
                 68, 69, 70, 71, 72],
         "weight":
                [115, 117, 120, 123, 126,
                 129, 132, 135, 139, 142,
                 146, 150, 154, 159, 164]
        }
>>> df = pd.DataFrame(dico)
>>> print(df)
##     height  weight
## 0       58     115
## 1       59     117
## 2       60     120
## 3       61     123
## 4       62     126
## 5       63     129
## 6       64     132
## 7       65     135
```

```
## 8       66     139
## 9       67     142
## 10      68     146
## 11      69     1`
## 12      70     154
## 13      71     159
## 14      72     164
```

La position des éléments dans le dataframe sert d'index. Comme pour les séries, les valeur sont accessibles dans l'attribut `values` et l'index dans l'attribut `index`. Les colonnes sont également indexées :

```
>>> print(df.columns)
## Index(['height', 'weight'], dtype='object')
```

La méthode `head()` permet d'afficher les premières lignes (les 5 premières, par défaut). On peut modifier son paramètre `n` pour indiquer le nombre de lignes à retourner :

```
>>> df.head(2)
```

Lors de la création d'un dataframe à partir d'un dictionnaire, si on précise le nom des colonnes à importer par une liste de chaînes de caractères fournie au paramètree `columns` de la fonction `DataFrame`, on peut non seulement définir les colonnes à remplir mais également leur ordre d'apparition.

Par exemple, pour n'importer que la colonne `weight` :

```
>>> df = pd.DataFrame(dico, columns = ["weight"])
>>> print(df.head(2))
##    weight
## 0     115
## 1     117
```

Et pour définir l'ordre dans lequel les colonnes apparaîtront :

```
>>> df = pd.DataFrame(dico, columns = ["weight", "height"])
>>> print(df.head(2))
##    weight  height
## 0     115      58
## 1     117      59
```

Si on indique un nom de colonne absent parmi les clés du dictionnaires, le dataframe résultant contiendra une colonne portant ce nom mais remplie de valeurs `NaN` :

```
>>> df = pd.DataFrame(dico, columns = ["weight", "height", "age"])
>>> print(df.head(2))
##    weight  height  age
## 0     115      58  NaN
## 1     117      59  NaN
```

1.2.2 Création de dataframes à partir d'une série

Un dataframe peut être créé à partir d'une série :

```
>>> s = pd.Series([1, 4, -1, np.nan], index = ["o", "d", "i", "l"])
>>> s.name = "nom_variable"
>>> df = pd.DataFrame(s, columns = ["nom_variable"])
>>> print(df)
##    nom_variable
## o           1.0
## d           4.0
## i          -1.0
## l           NaN
```

Si on n'attribue pas de nom à la série, il suffit de ne pas renseigner le paramètre `columns` de la fonction `DataFrame`. Mais dans ce cas, la colonne n'aura pas de non, juste un index numérique.

```
>>> s = pd.Series([1, 4, -1, np.nan], index = ["o", "d", "i", "l"])
>>> df = pd.DataFrame(s)
>>> print(df)
##      0
## o  1.0
## d  4.0
## i -1.0
## l  NaN
print(df.columns.name)
## None
```

10.1.2.3 Création de dataframes à partir d'une liste de dictionnaire

Un dataframe peut être créé à partir d'une liste de dictionnaires :

```
>>> dico_1 = {
    "Nom": "Pendragon",
    "Prenom": "Arthur",
    "Role": "Roi de Bretagne"
}
>>> dico_2 = {
    "Nom": "de Galles",
    "Prenom": "Perceval",
    "Role": "Chevalier du Pays de Galles"
}

>>> df = pd.DataFrame([dico_1, dico_2])
>>> print(df)
##          Nom    Prenom                         Role
## 0  Pendragon    Arthur              Roi de Bretagne
## 1  de Galles  Perceval  Chevalier du Pays de Galles
```

Si certaines clés sont absentes dans un ou plusieurs des dictionnaires de la liste, les valeurs correspondantes dans le dataframe seront `NaN` :

```
>>> dico_3 = {
    "Prenom": "Guenièvre",
    "Role": "Reine de Bretagne"
}
>>> df = pd.DataFrame([dico_1, dico_2, dico_3])
>>> print(df)
##          Nom             ...                                    Role
## 0  Pendragon             ...                          Roi de Bretagne
## 1  de Galles             ...              Chevalier du Pays de Galles
## 2        NaN             ...                         Reine de Bretagne
##
## [3 rows x 3 columns]
```

1.2.4 Création de dataframes à partir d'un dictionnaire de séries

On peut aussi créer un dataframe à partir d'un dictionnaire de séries. Pour illustrer la méthode, créons deux dictionnaires :

```
# PIB annuel 2017
# En millions de dollars courants
>>> dico_gdp_current = {
    "France": 2582501.31,
    "USA": 19390604.00,
    "UK": 2622433.96
}
# Indice annuel des prix à la consommation
>>> dico_cpi = {
    "France": 0.2,
    "UK": 0.6,
    "USA": 1.3,
    "Germany": 0.5
}
```

À partir de ces deux dictionnaires, créons deux séries correspondantes :

```
>>> s_gdp_current = pd.Series(dico_gdp_current)
>>> s_cpi = pd.Series(dico_cpi)

>>> print("s_gdp_current : \n", s_gdp_current)
## s_gdp_current :
##  France     2582501.31
## USA       19390604.00
## UK         2622433.96
## dtype: float64
>>> print("\ns_cpi : \n", s_cpi)
##
## s_cpi :
##  France     0.2
## UK         0.6
## USA        1.3
## Germany    0.5
## dtype: float64
```

Puis, créons un dictionnaire de séries :

```
>>> dico_de_series = {
    "gdp": s_gdp_current,
    "cpi": s_cpi
}
>>> print(dico_de_series)
## {'gdp': France     2582501.31
## USA       19390604.00
## UK         2622433.96
## dtype: float64, 'cpi': France     0.2
## UK         0.6
## USA        1.3
## Germany    0.5
## dtype: float64}
```

Enfin, créons notre dataframe :

```
>>> s = pd.DataFrame(dico_de_series)
print(s)
##                   gdp  cpi
## France    2582501.31  0.2
## Germany          NaN  0.5
## UK        2622433.96  0.6
## USA      19390604.00  1.3
```

Le dictionnaire `dico_gdp_current` ne contient pas de clé `Germany`, contrairement au dictionnaire `dico_cpi`. Lors de la création du dataframe, la valeur du PIB pour l'Allemagne a dont été assignée comme `NaN`.

10.1.2.5 Création de dataframes à partir d'un tableau `NumPy` à deux dimensions

On peut aussi créer un dataframe à partir d'un tableau `Numpy`. Lors de la création, avec la fonction `DataFrame()` de `NumPy`, il est possible de préciser le nom des colonnes (à défaut, l'indiçage des colonnes sera numérique) :

```
>>> liste = [
    [1, 2, 3],
    [11, 22, 33],
    [111, 222, 333],
    [1111, 2222, 3333]
]
>>> tableau_np = np.array(liste)
>>> print(df = pd.DataFrame(tableau_np,
                    columns = ["a", "b", "c"]))

```

1.2.6 Dimensions

On accède aux dimensions d'un dataframe avec l'attribut `shape`.

```
>>> print("shape : ", df.shape)
## shape :  (3, 3)
```

On peut aussi afficher le nombre de lignes comme suit :

```
>>> print("shape : ", len(df))
## shape :  3
```

Et le nombre de colonnes :

```
print("shape : ", len(df.columns))
## shape :  3
```

1.2.7 Modification de l'index

Comme pour les séries, on peut modifier l'index une fois que le dataframe a été créé, en venant écraser les valeurs des attributs `index` et `columns`, pour l'index des lignes et colonnes, respectivement :

```
>>> dico = {"nom":["sy", "Faye"], "age":[39,39]}
>>> df = pd.DataFrame(dico)
>>> dico.index = ["persone1", "persone2"]
>>> dico.columns = ["Nom", "Age"]
```

2 Sélection

Dans cette section, nous regardons différentes manières de sélectionner des données dans des séries et dataframes. On note deux manières bien distinctes :

- une première basée sur l'utiliation de crochets directement sur l'objet pour lequel on souhaite sélectionner certaines parties ;
- seconde s'appuyant sur des indexeurs, accessibles en tant qu'attributs d'objets `NumPy` (`loc`, `at`, `iat`, etc.)

La seconde méthode permet d'éviter certaines confusions qui peuvent apparaître dans le cas d'index numériques.

2.1 Pour les séries

Dans un premier temps, regardons les manières d'extraire des valeurs contenues dans des séries.

2.1.1 Avec les crochets

On peut utiliser l'index pour extraire les données :

```
>>> s = pd.Series([1, 4, -1, np.nan, .5, 1])
>>> s[0] # 1er élément de s
>>> s[1:3] # du 2e élément (inclus) au 4e (non inclus)
>>> s[[0,4]] # 1er et 5e éléments
```

On note que contrairement aux tableaux `numpy` ou aux listes, on ne peut pas utiliser des valeurs négatives pour l'index afin de récupérer les données en comptant leur position par rapport à la fin :

```
s[-2]
## KeyError: -2
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

```
##   File "/anaconda3/lib/python3.6/site-packages/pandas/core/series.py",
line 766, in __getitem__
##     result = self.index.get_value(self, key)
##   File "/anaconda3/lib/python3.6/site-
packages/pandas/core/indexes/base.py", line 3103, in get_value
##     tz=getattr(series.dtype, 'tz', None))
##   File "pandas/_libs/index.pyx", line 106, in
pandas._libs.index.IndexEngine.get_value
##   File "pandas/_libs/index.pyx", line 114, in
pandas._libs.index.IndexEngine.get_value
##   File "pandas/_libs/index.pyx", line 162, in
pandas._libs.index.IndexEngine.get_loc
##   File "pandas/_libs/hashtable_class_helper.pxi", line 958, in
pandas._libs.hashtable.Int64HashTable.get_item
##   File "pandas/_libs/hashtable_class_helper.pxi", line 964, in
pandas._libs.hashtable.Int64HashTable.get_item
```

Dans le cas d'un indice composé de chaînes de caractères, il est alors possible, pour extraire les données de la série, de faire référence soit au contenu de l'indice (pour faire simple, son nom), soit à sa position :

```
s = pd.Series([1, 4, -1, np.nan],
              index = ["o", "d", "i", "l"])
print("La série s : \n", s)
## La série s :
##  o    1.0
## d    4.0
## i   -1.0
## l    NaN
## dtype: float64
print('s["d"] : \n', s["d"])
## s["d"] :
##  4.0
print('s[1] : \n', s[1])
## s[1] :
##  4.0
print("éléments o et i : \n", s[["o", "i"]])
## éléments o et i :
##  o    1.0
## i   -1.0
## dtype: float64
```

Par contre, dans le cas où l'indice est défini avec des valeurs numériques, pour extraire les valeurs à l'aide des crochets, ce sera par la valeur de l'indice et pas en s'appuyant sur la position :

```
>>> s = pd.Series([1, 4, -1, np.nan],
                  index = [4, 40, 2, 3])
>>> print(s[40])
## 4.0
```

2.1.2 Avec les indexeurs

Pandas propose deux types d'indiçage multi-axes : `loc`, `iloc`. Le premier est principalement basé sur l'utilisation des labels des axes, tandis que le second s'appuie principalement sur les positions à l'aide d'entiers.

Pour les besoins de cette partie, créons deux séries ; une première avec un index textuel, une deuxième avec un index numérique :

```
>>> s_num = pd.Series([1, 4, -1, np.nan],
              index = [5, 0, 4, 1])
>>> s_texte = pd.Series([1, 4, -1, np.nan],
              index = ["c", "a", "b", "d"])
```

2.1.2.1 Extraction d'un seul élément

Pour extraire un objet avec `loc`, on utilise le nom de l'indice :

```
>>> print(s_num.loc[5], s_texte.loc["c"])
## 1.0 1.0
```

Pour extraire un élément unique avec `iloc`, il suffit d'indiquer sa position :

```
>>> (s_num.iloc[1], s_texte.iloc[1])
```

2.1.2.2 Extraction de plusieurs éléments

Pour extraire plusieurs éléments avec `loc`, on utilise les noms (labels) des indices, que l'on fournit dans une liste :

```
>>> print("éléments aux labels 5 et 4 :\n", s_num.loc[[5,4]])
## éléments aux labels 5 et 4 :
##  5    1.0
## 4   -1.0
## dtype: float64
>>> print("éléments aux labels c et b : \n", s_texte.loc[["c", "b"]])
## éléments aux labels c et b :
##  c    1.0
## b   -1.0
## dtype: float64
```

Pour extraire plusieurs éléments avec `iloc` :

```
>>> print("éléments aux positions 0 et 2 :\n", s_num.iloc[[0,2]])
## éléments aux positions 0 et 2 :
##  5    1.0
## 4   -1.0
## dtype: float64
>>> print("éléments aux positions 0 et 2 : \n", s_texte.iloc[[0,2]])
## éléments aux positions 0 et 2 :
##  c    1.0
## b   -1.0
## dtype: float64
```

2.1.2.3 Découpage

On peut effectuer des découpages de séries, pour récupérer des éléments consécutifs :

```
print("éléments des labels 5 jusqu'à 4 :\n", s_num.loc[5:4])
## éléments des labels 5 jusqu'à 4 :
##  5    1.0
## 0    4.0
## 4   -1.0
## dtype: float64
print("éléments des labels c à b : \n", s_texte.loc["c":"b"])
## éléments des labels c à b :
##  c    1.0
## a    4.0
## b   -1.0
## dtype: float64
```

Pour extraire plusieurs éléments avec `iloc` :

```
>>> print("éléments aux positions de 0 à 2 :\n", s_num.iloc[0:2])
## éléments aux positions de 0 à 2 :
##  5    1.0
## 0    4.0
## dtype: float64
>>> print("éléments aux positions de 0 à 2 : \n", s_texte.iloc[0:2])
## éléments aux positions de 0 à 2 :
##  c    1.0
## a    4.0
## dtype: float64
```

Comme ce que l'on a vu jusqu'à présent, la valeur supérieur de la limite n'est pas incluse dans le découpage.

2.1.2.4 Masque

On peut aussi utiliser un masque pour extraire des éléments, indifféremment en utilisant `loc` ou `iloc` :

```
>>> print("\n",s_num.loc[[True, False, False, True]])
##
##  5    1.0
## 1    NaN
## dtype: float64
>>> print("\n", s_texte.loc[[True, False, False, True]])
##
##  c    1.0
## d    NaN
## dtype: float64
>>> print("\n", s_num.iloc[[True, False, False, True]])
##
##  5    1.0
## 1    NaN
## dtype: float64
>>> print("\n", s_texte.iloc[[True, False, False, True]])
##
##  c    1.0
```

```
## d    NaN
## dtype: float64
```

2.1.2.5 Quel est l'intérêt ?

Pourquoi introduire de telles manières d'extraire les données et ne pas se contenter de l'extraction à l'aide des crochers sur les objets ? Regardons un exemple simple. Admettons que nous disposons de la série `s_num`, avec un indice composé d'entiers n'étant pas une séquence allant de 0 au nombre d'éléments. Dans ce cas, si nous souhaitons récupérer récupérer le 2e élément, du fait de l'indice composé de valeurs numériques, nous ne pouvons pas l'obtenir en demandant `s[1]`. Pour extraire le 2e de la série, on doit savoir que son indice vaut `0` et ainsi demander :

```
>>> print("L'élément dont l'index vaut 0 : ", s_num[0])
## L'élément dont l'index vaut 0 :  4.0
```

Pour pouvoir effectuer l'extraction en fonction de la position, il est bien pratique d'avoir cet attribut `iloc` :

```
>>> print("L'élément en 2e position :", s_num.iloc[1])
## L'élément en 2e position : 4.0
```

2.2 Pour les dataframes

À présent, regardons différentes manières d'extraire des données depuis un dataframe. Créons deux dataframes en exemple, l'une avec un index numérique ; une autre avec un index textuel :

```
>>> dico = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "taille": [162, 156, 172, 160, 158],
       }
>>> df_num = pd.DataFrame(dico)
>>> df_texte = pd.DataFrame(dico, index=["a", "e", "c", "b", "d"])
>>> print("df_num : \n", df_num)
## df_num :
##     height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
## 3      61     123   31     160
## 4      62     126   29     158
>>> print("df_texte : \n", df_texte)
## df_texte :
##     height  weight  age  taille
## a      58     115   28     162
## e      59     117   33     156
## c      60     120   31     172
## b      61     123   31     160
## d      62     126   29     158
```

Pour faire simple, lorsqu'on veut effectuer une extraction avec les attributs `iloc`, la syntaxe est la suivante :

```
>>> df.iloc[selection_lignes, selection_colonnes]
```

avec `selection_lignes` :

- une valeur unique : `1` (seconde ligne) ;
- une liste de valeurs : `[2, 1, 3]` (3e ligne, 2e ligne et 4e ligne) ;
- un découpage : `[2:4]` (de la 3e ligne à la 4e ligne (non incluse)).

pour `selection_colonnes` :

- une valeur unique : `1` (seconde colonne) ;
- une liste de valeurs : `[2, 1, 3]` (3e colonne, 2e colonne et 4e colonne) ;
- un découpage : `[2:4]` (de la 3e colonne à la 4e colonne (non incluse)).

Avec `loc`, la syntaxe est la suivante :

```
>>> df.loc[selection_lignes, selection_colonnes]
```

avec `selection_lignes` :

- une valeur unique : `"a"` (ligne nommée `a`) ;
- une liste de noms : `["a", "c", "b"]` (lignes nommées “a”, “c” et “b”) ;
- un masque : `df.['a']<10` (lignes pour lesquelles les valeurs du masque valent `True`).

pour `selection_colonnes` :

- une valeur unique : `"a"` (colonne nommée `a`) ;
- une liste de valeurs : `["a", "c", "b"]` (colonnes nommées “a”, “c” et “b”) ;
- un découpage : `["a":"c"]` (de la colonne nommée “a” à la colonne nommée “c”).

2.2.1 Extraction d’une ligne

Pour extraire une ligne d’un dataframe, on peut utiliser le nom de la ligne avec `loc` :

```
>>> print("Ligne nommée 'e':\n", df_texte.loc["e"])
## Ligne nommée 'e':
##  height     59
## weight    117
## age        33
## taille    156
## Name: e, dtype: int64
>>> print("\nLigne nommée 'e':\n", df_num.loc[1])
##
## Ligne nommée 'e':
##  height     59
## weight    117
## age        33
## taille    156
## Name: 1, dtype: int64
```

Ou bien, sa position avec `iloc` :

```
>>> print("Ligne en position 0:\n", df_texte.iloc[0])
## Ligne en position 0:
##  height     58
## weight    115
```

```
## age        28
## taille    162
## Name: a, dtype: int64
>>> print("\nLigne en position 0:\n", df_num.iloc[0])
##
## Ligne en position 0:
##  height     58
## weight    115
## age        28
## taille    162
## Name: 0, dtype: int64
```

2.2.2 Extraction de plusieurs lignes

Pour extraire plusieurs lignes d'un dataframe, on peut utiliser leur noms avec `loc` (dans un tableau) :

```
>>> print("Lignes nommées a et c :\n", df_texte.loc[["a", "c"]])
## Lignes nommées a et c :
##     height  weight  age  taille
## a      58     115   28     162
## c      60     120   31     172
>>> print("\nLignes nommées 0 et 2:\n", df_num.loc[[0, 2]])
##
## Lignes nommées 0 et 2:
##     height  weight  age  taille
## 0      58     115   28     162
## 2      60     120   31     172
```

Ou bien, leur position avec `iloc` :

```
>>> print("Lignes aux positions 0 et 3:\n", df_texte.iloc[[0, 3]])
## Lignes aux positions 0 et 3:
##     height  weight  age  taille
## a      58     115   28     162
## b      61     123   31     160
>>> print("\nLignes aux positions 0 et 3:\n", df_num.iloc[[0, 3]])
##
## Lignes aux positions 0 et 3:
##     height  weight  age  taille
## 0      58     115   28     162
## 3      61     123   31     160
```

2.2.3 Découpage de plusieurs lignes

On peut récupérer une suite de ligne en délimitant la première et la dernière à extraire en fonction de leur nom et en utilisant `loc` :

```
>>> print("Lignes du label a à c:\n", df_texte.loc["a":"c"])
## Lignes du label a à c:
##     height  weight  age  taille
## a      58     115   28     162
## e      59     117   33     156
## c      60     120   31     172
>>> print("\Lignes du label 0 à 2:\n", df_num.loc[0:2])
## \Lignes du label 0 à 2:
##     height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
```

Avec l'attribut `iloc`, c'est également possible (encore une fois, la borne supérieure n'est pas incluse) :

```
>>> print("Lignes des positions 0 à 3 (non incluse):\n", df_texte.iloc[0:3])
## Lignes des positions 0 à 3 (non incluse):
##     height  weight  age  taille
## a      58     115   28     162
## e      59     117   33     156
## c      60     120   31     172
>>> print("\nLignes des positions 0 à 3 (non incluse):\n", df_num.iloc[0:3])
##
## Lignes des positions 0 à 3 (non incluse):
##     height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
```

2.2.4 Masque

On peut aussi utiliser un masque pour sélectionner certaines lignes. Par exemple, si on souhaite récupérer les lignes pour lesquelles la variable `height` a une valeur supérieure à 60, on utilise le masque suivante :

```
>>> masque = df_texte["height"]> 60
>>> print(masque)
## a    False
## e    False
## c    False
## b     True
## d     True
## Name: height, dtype: bool
```

Pour filtrer :

```
>>> print(df_texte.loc[masque])
##    height  weight  age  taille
## b      61     123   31     160
## d      62     126   29     158
```

2.2.5 Extraction d'une seule colonne

Pour extraire une colonne d'un dataframe, on peut utiliser des crochets et faire référence au nom de la colonne (qui est indexée par les noms) :

```
>>> print(df_texte['weight'].head(2))
## a    115
## e    117
## Name: weight, dtype: int64
```

En ayant sélectionné une seule colonne, on obtient une série (l'index du dataframe est conservé pour la série) :

```
>>> print(type(df_texte['weight']))
## <class 'pandas.core.series.Series'>
```

On peut également extraire une colonne en faisant référence à l'attribut du dataframe portant le nom de cette colonne :

```
>>> print(df_texte.weight.head(2))
## a    115
## e    117
## Name: weight, dtype: int64
```

Comme pour les séries, on peut s'appuyer sur les attributs `loc` et `iloc` :

```
>>> print("Colone 2 (loc):\n", df_texte.loc[:,"weight"])
## Colone 2 (loc):
##  a    115
## e    117
## c    120
## b    123
## d    126
## Name: weight, dtype: int64
>>> print("Colonne 2 (iloc):\n", df_texte.iloc[:,1])
## Colonne 2 (iloc):
##  a    115
## e    117
## c    120
## b    123
## d    126
## Name: weight, dtype: int64
```

10.2.2.6 Extraction de plusieurs colonnes

Pour extraire plusieurs colonnes, il suffit de placer les noms des colonnes dans un tableau :

```
>>> print(df_texte[["weight", "height"]])
##    weight  height
## a     115      58
## e     117      59
## c     120      60
## b     123      61
## d     126      62
```

L'ordre dans lequel on appelle ces colonnes correspond à l'ordre dans lequel elles seront retournées.

À nouveau, on peut utuliser l'attribut `loc` (on utilise les deux points ici pour dire que l'on veut toutes les lignes) :

```
>>> print("Colonnes de weight à height:\n", df_texte.loc[:,["weight",
"height"]])
## Colonnes de weight à height:
##     weight  height
## a     115      58
## e     117      59
## c     120      60
## b     123      61
## d     126      62
```

Et l'attribut `iloc` :

```
>>> print("Colonnes 2 et 1 :\n", df_num.iloc[:,[1,0]])
## Colonnes 2 et 1 :
##     weight  height
## 0     115      58
## 1     117      59
## 2     120      60
## 3     123      61
## 4     126      62
```

2.2.7 Découpage de plusieurs colonnes

Pour effectuer un découpage, on peut utiliser les attributs `loc` et `iloc`. Attention, on ne place pas le nom des colonnes servant pour le découpage dans un tableau ici :

Avec `loc` :

```
>>> print("Colones 2 et 2:\n", df_texte.loc[:, "height":"age"])
## Colones 2 et 2:
##     height  weight  age
## a      58     115   28
## e      59     117   33
## c      60     120   31
## b      61     123   31
## d      62     126   29
```

Et avec l'attribut `iloc` :

```
>>> print("Colonnes de la position 0 à 2 (non incluse) :\n", df_texte.iloc[:,
0:2])
## Colonnes de la position 0 à 2 (non incluse) :
##     height  weight
## a      58     115
## e      59     117
## c      60     120
## b      61     123
## d      62     126
```

2.2.8 Extraction de lignes et colonnes

À présent que nous avons passé en revue de nombreuses manières de sélectionner une ou plusieurs lignes ou colonnes, nous pouvons également mentionner qu'il est possible de faire des sélections de colonnes et de lignes dans une même instruction.

Par exemple, avec `iloc`, sélectionnons les lignes de la position 0 à la position 2 (non incluse) et les colonnes de la position 1 à 3 (non incluse) :

```
>>> print(df_texte.iloc[0:2, 1:3])
##    weight  age
## a     115   28
## e     117   33
```

Avec `loc`, sélectionnons les lignes nommées `a` et `c` et les colonnes de celle nommée `weight` jusqu'à `age`.

```
df_texte.loc[["a", "c"], "weight":"age"]
```

3. Renommage des colonnes dans un dataframe

Pour renommer une colonne dans un dataframe, `pandas` propose la méthode `rename()`. Prenons un exemple avec notre dataframe `df` :

```
>>> dico = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "taille": [162, 156, 172, 160, 158],
       }
>>> df = pd.DataFrame(dico)
>>> print(df)
##    height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
## 3      61     123   31     160
## 4      62     126   29     158
```

Renommons la colonne `height` en `taille`, à l'aide d'un dicionnaire précisé au paramètre `columns`, avec comme clé le nom actuel de la colonne, et en valeur le nouveau nom :

```
>>> df.rename(index=str, columns={"height": "taille"}, inplace=True)
>>> print(df)
##    taille  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
## 3      61     123   31     160
## 4      62     126   29     158
```

Pour que le changement soit effectif, on indique `inplace=True`, sinon, la modification n'est pas apportée au dataframe.

Pour renommer plusieurs colonnes en même temps, il suffit de fournir plusieurs couples de clés valeurs dans le dictionnaire :

```
>>> df.rename(index=str,
              columns={"weight": "masse", "age" : "annees"},
              inplace=True)
>>> print(df)
##    taille  masse  annees  taille
## 0      58    115      28     162
## 1      59    117      33     156
## 2      60    120      31     172
## 3      61    123      31     160
## 4      62    126      29     158
```

4. Filtrage

Pour effectuer une filtration des données dans un tableau, en fonction des valeurs rencontrées pour certaines variables, on utilise des masques.

Redennons quelques exemples ici, en redéfinissant notre dataframe :

```
>>> dico = {"height" : [58, 59, 60, 61, 62],
            "weight": [115, 117, 120, 123, 126],
            "age": [28, 33, 31, 31, 29],
            "taille": [162, 156, 172, 160, 158],
           }
>>> df = pd.DataFrame(dico)
>>> print(df)
##    height  weight  age  taille
## 0      58     115   28     162
## 1      59     117   33     156
## 2      60     120   31     172
## 3      61     123   31     160
## 4      62     126   29     158
```

L'idée consiste à créer un masque retournant une série contenant des valeurs booléennes, ligne par ligne. Lorsque la valeur de la ligne du masque vaut `True`, la ligne du dataframe sur lequel sera appliqué le masque sera retenue, tandis qu'elle ne le sera pas quand la valeur de la ligne du masque vaut `False`.

Regardons un exemple simple, dans lequel nous souhaitons conserver les observations uniquement pour lesquelles la valeur de la variable `age` est inférieure à 30 :

```
>>> masque = df["age"] < 30
>>> print(masque)
## 0     True
## 1    False
## 2    False
## 3    False
## 4     True
## Name: age, dtype: bool
```

Il reste alors à appliquer ce masque, avec `loc`. On souhaite l'ensemble des colonnes, mais seulement quelques lignes :

```
>>> print(df.loc[masque])
##    height  weight  age  taille
## 0      58     115   28     162
## 4      62     126   29     158
```

Note : cela fonctionne aussi sans `loc` :

```
>>> print(df[masque])
##    height  weight  age  taille
## 0      58     115   28     162
## 4      62     126   29     158
```

Plus simplement, on peut utiliser la méthode `query()` de `pandas`. On fournit une expression booléenne à évaluer à cette méthode pour filtrer les données :

```
>>> print(df.query("age<30"))
##    height  weight  age  taille
## 0      58     115   28     162
## 4      62     126   29     158
```

La requête peut être un peu plus complexe, en combinant opérateurs de comparaison et opérateurs logiques . Par exemple, admettons que nous voulons filtrer les valeurs du dataframe pour ne retenir que les observations pour lesquelles la taille est inférieure ou égale à 62 et la masse strictement supérieure à 120. La requête serait alors :

```
>>> print(df.query("weight > 120 and height < 62"))
##    height  weight  age  taille
## 3      61     123   31     160
```

On peut noter que l'instruction suivante donne le même résultat :

```
>>> print(df.query("weight > 120").query("height < 62"))
##    height  weight  age  taille
## 3      61     123   31     160
```

4.1 Test d'appartenance

Pour créer un masque indiquant si les valeurs d'une série ou d'un dataframe appartiennent à un ensemble, on peut utiliser la méthode `isin()`. Par exemple, retournons un masque indiquant si les valeurs de la colonne `height` de `df` sont dans l'intervalle [59,60][59,60] :

```
>>> df.height.isin(np.arange(59,61))
```

5. Valeurs manquantes

Il est assez fréquent de récupérer des données incomplètes. La manière dont les données manquantes sont gérées par `pandas` est le recours aux deux valeurs spéciales : `None` et `NaN`.

La valeur `None` peut être utilisée dans les tableaux `NumPy` uniquement quand le type de ces derniers est `object`.

```
>>> tableau_none = np.array([1, 4, -1, None])
>>> print(tableau_none)
## [1 4 -1 None]
>>> print(type(tableau_none))
## <class 'numpy.ndarray'>
```

Avec un tableau de type `object`, les opérations effectuées sur les données seront moins efficaces qu'avec un tableau d'un type numérique.

La valeur `NaN` est une valeur de nombre à virgule flottante . `NumPy` la gère différemment de `None`, et n'assigne passe type `object` d'emblée en présence de `NaN` :

```
>>> tableau_nan = np.array([1, 4, -1, np.nan])
>>> print(tableau_nan)
## [ 1.  4. -1. nan]
>>> print(type(tableau_nan))
## <class 'numpy.ndarray'>
```

Avec `pandas`, ces deux valeurs, `None` et `NaN` peuvent être présentes :

```
>>> s = pd.Series([1, None, -1, np.nan])
>>> print(s)
## 0    1.0
## 1    NaN
## 2   -1.0
## 3    NaN
## dtype: float64
>>> print(type(s))
## <class 'pandas.core.series.Series'>
```

Cela tient aussi pour les tableaux :

```
>>> dico = {"height" : [58, 59, 60, 61, np.nan],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, np.nan, 29],
        "taille": [162, 156, 172, 160, 158],
       }
>>> df = pd.DataFrame(dico)
>>> print(df)
##    height  weight   age  taille
## 0    58.0     115  28.0     162
## 1    59.0     117  33.0     156
## 2    60.0     120  31.0     172
## 3    61.0     123   NaN     160
## 4     NaN     126  29.0     158
```

On note toutefois que seule le type des variables pour lesquelles existent des valeurs manquantes sont passées en `float64` :

```
>>> print(df.dtypes)
## height    float64
## weight      int64
## age       float64
## taille      int64
## dtype: object
```

On note que les données sont enregistrées sur un type `float64`. Lorsqu'on travaille sur un tableau ne comportant pas de valeurs manquantes, dont le type est `int` ou `bool`, si on introduit une valeur manquante, `pandas` changera le type des données en `float64` et `object`, respectivement.

`pandas` propose différentes pour manipuler les valeurs manquantes.

5.1 Repérer les valeurs manquantes

Avec la méthode `isnull()`, un masque de booléens est retournée, indiquant `True` pour les observations dont la valeur est `NaN` ou `None` :

```
>>> print(s.isnull())
## 0    False
## 1     True
## 2    False
## 3     True
## dtype: bool
```

Pour savoir si une valeur n'est pas nulle, on dispose de la méthode `notnull()` :

```
>>> print(s.notnull())
## 0     True
## 1    False
## 2     True
## 3    False
## dtype: bool
```

5.2 Retirer les observations avec valeurs manquantes

La méthode `dropna()` permet quant à elle de retirer les observations disposant de valeurs nulles :

```
>>> print(df.dropna())
##    height  weight   age  taille
## 0    58.0     115  28.0     162
## 1    59.0     117  33.0     156
## 2    60.0     120  31.0     172
```

5.3 Retirer les valeurs manquantes par d'autres valeurs

Pour remplacer les valeurs manquantes par d'autres valeurs, `pandas` propose d'utiliser la méthode `fillna()` :

```
>>> print(df.fillna(-9999))
##    height  weight     age  taille
## 0    58.0     115    28.0     162
## 1    59.0     117    33.0     156
## 2    60.0     120    31.0     172
## 3    61.0     123 -9999.0     160
## 4 -9999.0     126    29.0     158
```

6. Suppressions

Pour supprimer une valeur sur un des axes d'une série ou d'un dataframe, `NumPy` propose la méthode `drop()`.

6.1 Suppression d'éléments dans une série

Pour illustrer le fonctionnement de la méthode `drop()`, créons une série avec un index numérique, une autre avec un index textuel :

```
>>> s_num = pd.Series([1, 4, -1, np.nan],
              index = [5, 0, 4, 1])
>>> s_texte = pd.Series([1, 4, -1, np.nan],
              index = ["c", "a", "b", "d"])
```

On peut supprimer un élément d'une série en utilisant son nom :

```
>>> print("pour s_num : \n", s_num.drop(5))
## pour s_num :
##  0    4.0
## 4   -1.0
## 1    NaN
## dtype: float64
>>> rint("\npour s_texte : \n", s_texte.drop("c"))
##
## pour s_texte :
##  a    4.0
## b   -1.0
## d    NaN
## dtype: float64
```

On peut aussi aller récupérer le nom en fonction de la position, en passant par un détour en utilisant la méthode `index()` :

```
>>> print(s.drop(s_num.index[0]))

>>> print("s_num.index[0] : ", s_num.index[0])
## s_num.index[0] :  5
print("s_texte.index[0] : ", s_texte.index[0])
## s_texte.index[0] :  c
>>> print("pour s_num : \n", s_num.drop(s_num.index[0]))
## pour s_num :
##  0    4.0
## 4   -1.0
## 1    NaN
## dtype: float64
```

```
>>> print("\npour s_texte : \n", s_texte.drop(s_texte.index[0]))
##
## pour s_texte :
##  a    4.0
## b   -1.0
## d    NaN
## dtype: float64
```

Pour supprimer plusieurs éléments, il suffit de fournir plusieurs noms d'indice dans une liste à la méthode `drop()` :

```
>>> print("pour s_num : \n", s_num.drop([5, 4]))
## pour s_num :
##  0    4.0
## 1    NaN
## dtype: float64
print("\npour s_texte : \n", s_texte.drop(["c", "b"]))
##
## pour s_texte :
##  a    4.0
## d    NaN
## dtype: float64
```

À nouveau, on peut aller récupérer le nom en fonction de la position, en passant par un détour en utilisant la méthode `index()` :

```
>>> print(s.drop(s_num.index[0]))

>>> print("s_num.index[[0,2]] : ", s_num.index[[0,2]])
## s_num.index[[0,2]] :  Int64Index([5, 4], dtype='int64')
>>> print("s_texte.index[[0,2]] : ", s_texte.index[[0,2]])
## s_texte.index[[0,2]] :  Index(['c', 'b'], dtype='object')
>>> print("pour s_num : \n", s_num.drop(s_num.index[[0,2]]))
## pour s_num :
##  0    4.0
## 1    NaN
## dtype: float64
>>> print("\npour s_texte : \n", s_texte.drop(s_texte.index[[0,2]]))
##
## pour s_texte :
##  a    4.0
## d    NaN
## dtype: float64
```

Il est possible d'utiliser un découpage également pour obtenir la série sans le ou les éléments .

6.2 Suppression d'éléments dans un dataframe

Pour illustrer le fonctionnement de la méthode `drop()` sur un dataframe, créons-en un :

```
>>> s_num = pd.Series([1, 4, -1, np.nan],
                      index = [5, 0, 4, 1])
s_texte = pd.Series([1, 4, -1, np.nan],
                      index = ["c", "a", "b", "d"])
>>> dico = {"height" : [58, 59, 60, 61, np.nan],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, np.nan, 29],
        "taille": [162, 156, 172, 160, 158],
       }
>>> df = pd.DataFrame(dico)
```

6.2.1 Suppressions de lignes

Pour supprimer une ligne d'un dataframe, on peut faire référence à son nom (ici, les noms sont des numéros, mais ce sont bien des labels) :

```
>>> print("Supprimer la première ligne :  \n", df.drop(0))
## Supprimer la première ligne :
##     height  weight   age  taille
## 1    59.0     117  33.0     156
## 2    60.0     120  31.0     172
## 3    61.0     123   NaN     160
## 4     NaN     126  29.0     158
```

Si les lignes ont des labels textuels, on peut au préalable aller les récupérer à l'aide de la méthode `index()` :

```
>>> label_pos_0 = df.index[0]
>>> print("Supprimer la première ligne :  \n", df.drop(label_pos_0))
## Supprimer la première ligne :
##     height  weight   age  taille
## 1    59.0     117  33.0     156
## 2    60.0     120  31.0     172
## 3    61.0     123   NaN     160
## 4     NaN     126  29.0     158
```

Pour supprimer plusieurs lignes, on donne le nom de ces lignes dans une liste à la méthode `drop()` :

```
>>> print("Supprimer les 1ère et 4e lignes :  \n", df.drop([0,3]))
## Supprimer les 1ère et 4e lignes :
##     height  weight   age  taille
## 1    59.0     117  33.0     156
## 2    60.0     120  31.0     172
## 4     NaN     126  29.0     158
```

Ou encore, en indiquant les positions des lignes :

```
>>> label_pos = df.index[[0, 3]]
>>> print("Supprimer les 1ère et 4e lignes :  \n", df.drop(label_pos))
## Supprimer les 1ère et 4e lignes :
##     height  weight   age  taille
## 1    59.0     117  33.0     156
## 2    60.0     120  31.0     172
## 4     NaN     126  29.0     158
```

Il est possible d'utiliser un découpage également pour obtenir la série sans le ou les éléments .

6.2.2 Suppressions de colonnes

Pour supprimer une colonne d'un dataframe, on procède de la même manière que pour les lignes, mais en ajoutant le paramètre `axis=1` à la méthode `drop()` pour préciser que l'on s'intéresse aux colonnes :

```
>>>  print("Supprimer la première colonne :  \n", df.drop("height", axis=1))
## Supprimer la première colonne :
##     weight   age  taille
## 0     115  28.0     162
## 1     117  33.0     156
## 2     120  31.0     172
## 3     123   NaN     160
## 4     126  29.0     158
```

On peut au préalable aller récupérer les labels des colonnes en fonction de leur position à l'aide de la méthode `columns()` :

```
>>> label_pos = df.columns[0]
>>> print("label_pos : ", label_pos)
## label_pos :  height
print("Supprimer la première colonne :  \n", df.drop(label_pos, axis=1))
## Supprimer la première colonne :
##     weight   age  taille
## 0     115  28.0     162
## 1     117  33.0     156
## 2     120  31.0     172
## 3     123   NaN     160
## 4     126  29.0     158
```

Pour supprimer plusieurs colonnes, on donne le nom de ces colonnes dans une liste à la méthode `drop()` :

```
>>> print("Supprimer les 1ère et 4e colonnes :  \n", df.drop(["height",
"taille"], axis = 1))
## Supprimer les 1ère et 4e colonnes :
##     weight   age
## 0     115  28.0
## 1     117  33.0
## 2     120  31.0
## 3     123   NaN
## 4     126  29.0
```

Ou encore, en indiquant les positions des colonnes :

```
>>> label_pos = df.columns[[0, 3]]
>>> print("Supprimer les 1ère et 4e colonnes :  \n", df.drop(label_pos,
axis=1))
## Supprimer les 1ère et 4e colonnes :
##     weight   age
## 0     115  28.0
## 1     117  33.0
## 2     120  31.0
## 3     123   NaN
## 4     126  29.0
```

Il est possible d'utiliser un découpage également pour obtenir la série sans le ou les éléments

7. Remplacement de valeurs

Nous allons à présent regarder comment modifier une ou plusieurs valeurs, dans le cas d'une série puis d'un dataframe.

7.1 Pour une série

Pour modifier une valeur particulière dans une série ou dans un dataframe, on peut utiliser le symbole égale (`=`) en ayant au préalable ciblé l'emplacement de la valeur à modifier, à l'aide des techniques d'extraction expliquées.

Par exemple, considérons la série suivante :

```
>>> s_num = pd.Series([1, 4, -1, np.nan],
              index = [5, 0, 4, 1])
>>> print("s_num : ", s_num)
## s_num :  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## dtype: float64
```

Modifions le deuxième élément élément de `s_num`, pour lui donner la valeur -3 :

```
>>> s_num.iloc[1] = -3
print("s_num : ", s_num)
## s_num :  5    1.0
## 0   -3.0
## 4   -1.0
## 1    NaN
## dtype: float64
```

Il est évidemment possible de modifier plusieurs valeurs à la fois.

Il suffit à nouveau de cibler les positions (on peu utiliser de nombreuses manières de le faire) et de fournir un objet de dimensions équivalentes pour venir remplacer les valeurs ciblées. Par exemple, dans notre série `s_num`, allons remplacer les valeurs en position 1 et 3 (2e et 4e valeurs) par -10 et -9 :

```
>>> s_num.iloc[[1,3]] = [-10, -9]
print(s_num)
## 5     1.0
## 0   -10.0
## 4    -1.0
## 1    -9.0
## dtype: float64
```

7.2 Pour un dataframe

Considérons le dataframe suivant :

```
>>> dico = {"ville" : ["Marseille", "Aix",
                       "Marseille", "Aix", "Paris", "Paris"],
         "annee": [2019, 2019, 2018, 2018,2019, 2019],
         "x": [1, 2, 2, 2, 0, 0],
         "y": [3, 3, 2, 1, 4, 4],
        }
>>> df = pd.DataFrame(dico)
print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
## 5      Paris   2019  0  4
```

7.2.1 Modifications d'une valeur particulière

Modifions la valeur de la première ligne de `df` pour la colonne `annee`, pour que celle-ci vaille 2020. Dans un premier temps, récupérons la position de la colonne `annee` dans le dataframe, à l'aide de la méthode `get_loc()` appliquée à l'attribut `colnames` du dataframe :

```
>>> pos_annee = df.columns.get_loc("annee")
>>> print("pos_annee : ", pos_annee)
## pos_annee :  1
```

Ensuite, effectuons la modification :

```
>>> df.iloc[0,pos_annee] = 2020
>>> print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2020  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
## 5      Paris   2019  0  4
```

7.2.2 Modifications sur une ou plusieurs colonnes

Pour modifier toutes les valeurs d'une colonne pour y placer une valeur particulière, par exemple un 2 dans la colonne `x` de `df` :

```
>>> df.x = 2
>>> print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2020  2  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  2  4
## 5      Paris   2019  2  4
```

On peut également modifier les valeurs de la colonne en fournissant une liste de valeurs :

```
>>> df.x = [2, 3, 4, 2, 1, 0]
>>> print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2020  2  3
## 1        Aix   2019  3  3
## 2  Marseille   2018  4  2
## 3        Aix   2018  2  1
## 4      Paris   2019  1  4
## 5      Paris   2019  0  4
```

On peut donc imaginer modifier les valeurs d'une colonne en fonction des valeurs que l'on lit dans une autre colonne. Par exemple, admettons le code suivant : si la valeur de `y` vaut 2, alors celle de x vaut "a", si la valeur de `y` vaut 1, lors celle de `x` vaut "b", sinon, elle vaut `NaN`. Dans un premier temps, construisons une liste contenant les valeurs à insérer (que nous nommerons `nv_val`), à l'aide d'une boucle. Nous allons parcourir tous les éléments de la colonne `y`, et à chaque itération ajouter à `nv_val` la valeur obtenue en effectuant nos comparaisons :

```
>>> nv_val = []
>>> for i in np.arange(len(df.index)):
        if df.y[i] == 2:
            nv_val.append("a")
        elif df.y[i] == 1:
            nv_val.append("b")
        else:
            nv_val.append(np.nan)
>>> print("nv_val : ", nv_val)
## nv_val :  [nan, nan, 'a', 'b', nan, nan]
```

Nous sommes prêts à modifier le contenu de la colonne `x` de `df` pour le remplacer par `nv_val` :

```
>>> df.x = nv_val
>>> print("df : \n", df)
## df :
##         ville  annee    x  y
## 0  Marseille   2020  NaN  3
## 1        Aix   2019  NaN  3
## 2  Marseille   2018    a  2
## 3        Aix   2018    b  1
## 4      Paris   2019  NaN  4
## 5      Paris   2019  NaN  4
```

Pour remplacer plusieurs colonnes en même temps :

```
>>> df[["x", "y"]] = [[2, 3, 4, 2, 1, 0], 1]
print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2020  2  1
## 1        Aix   2019  3  1
## 2  Marseille   2018  4  1
## 3        Aix   2018  2  1
## 4      Paris   2019  1  1
## 5      Paris   2019  0  1
```

Dans l'instruction précédente, nous avons remplacé le contenu des colonnes `x` et `y` par une vecteur de valeurs écrites à la main pour `x` et par la valeur 1 pour toutes les observations pour `y`.

7.2.3 Modifications sur une ou plusieurs lignes

Pour remplacer une ligne par une valeur constante (peu d'intérêt ici) :

```
>>> df.iloc[1,:] = 1
>>> print("df : \n", df)

```

Il peut être plus intéressant de remplacer une observation comme suit :

```
>>> df.iloc[1,:] = ["Aix", 2018, 1, 2]

print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2020  2  1
## 1  Aix          2018 1  2
## 2  Marseille   2018  4  1
## 3        Aix   2018  2  1
## 4      Paris   2019  1  1
## 5      Paris   2019  0  1


```

Pour remplacer plusieurs lignes, la méthode est identique :

```

```

```
>>> df.iloc[[1,3],:] = [
    ["Aix", 2018, 1, 2],
    ["Aix", 2018, -1, -1]
]

>>> print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2020  2  1
## 1       Aix    2018  1  2
## 2  Marseille   2018  4  1
## 3        Aix   2018 -1  -1
## 4      Paris   2019  1  1
## 5      Paris   2019  0  1
```

8 Ajout de valeurs

Regardons à présent comment ajouter des valeurs, dans une série d'abord, puis dans un dataframe.

10.8.1 Pour une série

Considérons la série suivante :

```
>>> s_num = pd.Series([1, 4, -1, np.nan],
             index = [5, 0, 4, 1])
>>> print("s_num : ", s_num)
## s_num :  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## dtype: float64
```

8.1.1 Ajout d'une seule valeur dans une série

Pour ajouter une valeur, on utlise la méthode `append()`. Ici, avec `s_num`, comme l'index est manuel, nous sommes obligé de fournir une série avec une valeur pour l'index également :

```
>>> s_num_2 = pd.Series([1], index = [2])
>>> print("s_num_2 : \n", s_num_2)
## s_num_2 :
##  2    1
## dtype: int64
>>> s_num = s_num.append(s_num_2)
>>> print("s_num : \n", s_num)
## s_num :
##  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## 2    1.0
## dtype: float64
```

On note que la méthode `append()` retourne une vue, et que pour répercuter l'ajout, il est nécessaire d'effectuer une nouvelle assignation.

En ayant une série avec un index numérique généré automatiquement, on peut préciser la valeur `True` pour le paramètre `ignore_index` de la méthode `append()` pour indiquer de ne pas tenir compte de la valeur de l'index de l'objet que l'on ajoute :

```
>>> s = pd.Series([10, 2, 4])
>>> s = s.append(pd.Series([2]), ignore_index=True)
print("s : \n", s)
## s :
##  0    10
## 1     2
## 2     4
## 3     2
## dtype: int64
```

8.1.2 Ajout de plusieurs valeurs dans une série

Pour ajouter plusieurs valeurs, on utlise la méthode `append()`. Ici, avec `s_num`, comme l'index est manuel, nous sommes obligé de fournir une série avec une valeur pour l'index également :

```
>>> s_num_2 = pd.Series([1], index = [2])
>>> s_num.append(s_num_2)
>>> print("s_num : ", s_num)
## s_num :  5    1.0
## 0    4.0
## 4   -1.0
## 1    NaN
## 2    1.0
## dtype: float64
```

En ayant une série avec un index numérique généré automatiquement :

```
>>> s = pd.Series([10, 2, 4])
>>> s.append(pd.Series([2]), ignore_index=True)
```

8.2 Pour un dataframe

Reprenons notre dataframe :

```
>>> dico = {"ville" : ["Marseille", "Aix",
                       "Marseille", "Aix", "Paris", "Paris"],
        "annee": [2019, 2019, 2018, 2018,2019, 2019],
        "x": [1, 2, 2, 2, 0, 0],
        "y": [3, 3, 2, 1, 4, 4],
       }
>>> df = pd.DataFrame(dico)
>>> print("df : \n", df)
## df :
##         ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
## 5      Paris   2019  0  4
```

8.2.1 Ajout d'une ligne dans un dataframe

Comme pour une série, pour ajouter une ligne, on utlise la méthode `append()`. Dans un premier temps, créons un nouveau dataframe avec la ligne à ajouter :

```
>>> nv_ligne = pd.DataFrame([["Marseille", "2021", 2, 4]],
                        columns = df.columns)
>>> print("nv_ligne : \n", nv_ligne)
## nv_ligne :
##         ville annee  x  y
## 0  Marseille  2021  2  4
```

On s'est assuré d'avoir le même nom de colonnes ici, en indiquant au paramètre `columns` de la méthode `pd.DataFrame` le nom des colonnes de `df`, c'est-à-dire `df.columns`.

Ajoutons la nouvelle ligne à `df` :

```
>>> df = df.append(nv_ligne, ignore_index=True)
```

À nouveau,la méthode `append()` appliquée à un dataframe, retourne une vue et n'affecte pas l'objet.

On peut noter que lors de l'ajout d'une ligne, si le nom des colonnes n'est pas indiqué dans le même ordre que dans le dataframe dans lequel est effectué l'ajout, il faut rajouter une indication au paramètre `sort` de la méthode `append()` :

- si `sort=True`, l'ordre des colonnes de la ligne ajoutée sera appliqué au dataframe de destination ;
- si `sort=False`, l'odre des colonnes du dataframe de destination ne sera pas modifié.

```
>>> nv_ligne = pd.DataFrame([["2021", "Marseille", 2, 4]],
                        columns = ["annee", "ville", "x", "y"])
>>> print("nv_ligne : \n", nv_ligne)
## nv_ligne :
##    annee      ville  x  y
## 0  2021  Marseille  2  4
>>> print("avec sort=True : \n", df.append(nv_ligne, ignore_index=True, sort
= True))
## avec sort=True :
##    annee      ville  x  y
## 0  2019  Marseille  1  3
## 1  2019        Aix  2  3
## 2  2018  Marseille  2  2
## 3  2018        Aix  2  1
## 4  2019      Paris  0  4
## 5  2019      Paris  0  4
## 6  2021  Marseille  2  4
## 7  2021  Marseille  2  4
```

8.2.2 Ajout de plusieurs lignes dans un dataframe

Pour ajouter plusieurs lignes, c'est exactement le même principe qu'avec une seule, il suffit juste d'ajouter un dataframe de plusieurs lignes, avec encore une fois les mêmes noms.

Les lignes à insérer :

```
>>> nv_lignes = pd.DataFrame([
    ["Marseille", "2022", 2, 4],
    ["Aix", "2022", 3, 3]],
    columns = df.columns)
>>> print("nv_ligne : \n", nv_lignes)
## nv_ligne :
##         ville annee  x  y
## 0  Marseille  2022  2  4
## 1        Aix  2022  3  3
```

Puis l'insertion :

```
>>> df = df.append(nv_lignes, ignore_index=True)
```

8.2.3 Ajout d'une colonne dans un dataframe

Pour ajouter une colonne dans un dataframe :

```
from numpy import random
df["z"] = random.rand(len(df.index))
print("df : \n", df)
```

9. Retrait des valeurs dupliquées

Pour retirer les valeurs dupliquées dans un dataframe, `NumPy` propose la méthode `drop_duplicates()`, qui prend plusieurs paramètres optionnels :

- `subset` : en indiquant un ou plusieurs noms de colonnes, la recherche de doublons se fait uniquement sur ces colonnes ;
- `keep` : permet d'indiquer quelle observation garder en cas de doublons identifies :
- si `keep='first'`, tous les doublons sont retirés sauf la première occurrence,
- si `keep='last'`, tous les doublons sont retirés sauf la dernière occurrence, -si `keep='False'`, tous les doublons sont retirés ;
- `inplace` : booléen (défaut : `False`) pour indiquer si le retrait des doublons doit s'effectuer sur le dataframe ou bien si une copie doit être retournée (par défaut).

Donnons quelques exemples à l'aide de ce dataframe qui compose deux doublons quand on considère sa totalité. Si on se concentre uniquement sur les années ou les villes, ou les deux, d'autres doublons peuvent être identifiés.

```
dico = {"ville" : ["Marseille", "Aix",
                   "Marseille", "Aix", "Paris", "Paris"],
        "annee": [2019, 2019, 2018, 2018,2019, 2019],
        "x": [1, 2, 2, 2, 0, 0],
        "y": [3, 3, 2, 1, 4, 4],
       }
df = pd.DataFrame(dico)
print(df)
##        ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
```

```
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
## 5      Paris   2019  0  4
```

Pour retirer les doublons :

```
print(df.drop_duplicates())
##        ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
```

Retirer les doublons en gardant la dernière valeur des doublons identifiés :

```
>>> df.drop_duplicates(keep='last')
```

Pour retirer les doublons identifiés quand on se concentre sur le nom des villes, et en conservant uniquement la première valeur :

```
>>> print(df.drop_duplicates(subset = ["ville"], keep = 'first'))
##        ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 4      Paris   2019  0  4
```

Idem mais en se concentrant sur les couples (ville, annee)

```
>>> print(df.drop_duplicates(subset = ["ville", "annee"], keep = 'first'))
##        ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
```

On note que le dataframe original n'a pas été impacté, puisque nous n'avons pas touché au paramètre `inplace`. Si à présent, nous demandons à ce que les changement soient opérés sur le dataframe plutôt que de récupérer une copie :

```
>>> df.drop_duplicates(subset = ["ville", "annee"], keep = 'first', inplace =
True)
>>> print(df)
##        ville  annee  x  y
## 0  Marseille   2019  1  3
## 1        Aix   2019  2  3
## 2  Marseille   2018  2  2
## 3        Aix   2018  2  1
## 4      Paris   2019  0  4
```

Pour savoir si une valeur est dupliquée dans un dataframe, `NumPy` propose la méthode `duplicated()`, qui retourne un masque indiquant pour chaque observation, si elle est dupliquée ou non. Son fonctionnement est similaire à `df.drop_duplicates()`, hormis pour le paramètre `inplace` qui n'est pas présent.

```
print(df.duplicated(subset = ["ville"], keep = 'first'))
## 0    False
## 1    False
## 2     True
## 3     True
## 4    False
## dtype: bool
```

On peut utiliser la méthode `any()` par la suite pour savoir s'il existe des doublons :

```
>>> print(df.duplicated(subset = ["ville"], keep = 'first').any())
## True
```

10. Opérations

Il est souvent nécessaire de devoir effectuer des opérations sur les colonnes d'un dataframe, notamment lorsqu'il s'agit de créer une nouvelle variable.

En reprenant les principes de modification de colonnes, on imagine assez facilement qu'il est possible d'appliquer les fonctions et méthodes de `NumPy` sur les valeurs des colonnes.

Par exemple, considérons le dataframe suivant :

```
>>> dico = {"height" :
                [58, 59, 60, 61, 62,
                 63, 64, 65, 66, 67,
                 68, 69, 70, 71, 72],
         "weight":
                [115, 117, 120, 123, 126,
                 129, 132, 135, 139, 142,
                 146, 150, 154, 159, 164]
        }
>>> df = pd.DataFrame(dico)
print(df)
##     height  weight
## 0       58     115
## 1       59     117
## 2       60     120
## 3       61     123
## 4       62     126
## 5       63     129
## 6       64     132
## 7       65     135
## 8       66     139
## 9       67     142
## 10      68     146
## 11      69     150
## 12      70     154
## 13      71     159
## 14      72     164
```

Ajoutons la colonne `height_2`, élevant les valeurs de la colonne `height` au carré :

```
>>> df["height_2"] = df.height**2
print(df.head(3))
##    height  weight  height_2
## 0      58     115      3364
## 1      59     117      3481
## 2      60     120      3600
```

À présent, ajoutons la colonne `imc`, fournissant les valeurs de l'indicateur de masse corporelle pour les individus du dataframe (IMC=weightheight2IMC=weightheight2) :

```
>>> df["imc"] = df.weight / df.height_2
>>> print(df.head(3))
##    height  weight  height_2       imc
## 0      58     115      3364  0.034185
## 1      59     117      3481  0.033611
## 2      60     120      3600  0.033333
```

10.1 Statistiques

`pandas` propose quelques méthodes pour effectuer des statistiques descriptives pour chaque colonne ou par ligne. Pour cela, la syntaxe est la suivante (tous les paramètres ont une valeur par défaut, la liste est simplifiée ici) :

```
dataframe.fonction_stat(axis, skipna)
```

- `axis` : 0 pour les lignes, 1 pour les colonnes ;
- `skipna` : si `True`, exclue les valeurs manquantes pour effectuer les calculs.

Parmi les méthodes disponibles : - `mean()` : moyenne ; - `mode()` : mode ; - `median()` : médiane ; - `std()` : écart-type ; - `min()` : minimum ; - `max()` : maximum - `mad()` : écart absolu à la moyenne ; - `sum()` : somme des valeurs ; - `prod()` : produit de tous les éléments ; - `count()` : comptage du nombre d'éléments.

Par exemple, pour calculer la moyenne des valeurs pour chaque colonne :

```
dico = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, 117, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "taille": [162, 156, 172, 160, 158],
        "married": [True, True, False, False, True],
        "city": ["A", "B", "B", "B", "A"]
       }
df = pd.DataFrame(dico)
print(df.mean())
## height      60.0
## weight     120.2
## age         30.4
## taille     161.6
## married      0.6
## dtype: float64
```

Si on le souhaite, on peut faire la moyenne des valeurs en colonne (sans aucun sens ici) :

```
print(df.mean(axis=1))
## 0    72.8
## 1    73.2
## 2    76.6
## 3    75.0
## 4    75.2
## dtype: float64
```

Ces fonctions peuvent s'appliquer sur une seule colonne. Par exemple, pour afficher la valeur minimum :

```
print("min : ", df.height.min())
## min :  58
```

Il est aussi utile de pouvoir obtenir la position des valeurs min et max ; ce qu'on peut obtenir avec les méthodes `idxmin()` et `idxmax()`, respectivement.

```
print("pos min : ", df.height.idxmin())
## pos min :  0
print("pos min : ", df.height.idxmax())
## pos min :  4
```

Une méthode très pratique est `describe()`, elle permet de retourner des statistiques descriptives sur l'ensemble des colonnes numériques :

```
print(df.describe())
##           height      weight        age      taille
## count   5.000000    5.000000   5.000000    5.000000
## mean   60.000000  120.200000  30.400000  161.600000
## std     1.581139    4.438468   1.949359    6.228965
## min    58.000000  115.000000  28.000000  156.000000
## 25%    59.000000  117.000000  29.000000  158.000000
## 50%    60.000000  120.000000  31.000000  160.000000
## 75%    61.000000  123.000000  31.000000  162.000000
## max    62.000000  126.000000  33.000000  172.000000
```

11 Tri

Il est aisé de trier un dataframe par ordre croissant ou décroissant d'une ou plusieurs de ses colonnes. Pour ce faire, on utilise la méthode `sort_values()`. La syntaxe est la suivante :

```
DataFrame.sort_values(by, axis=0, ascending=True,
                      inplace=False, kind="quicksort",
                      na_position="last")

```

- `by` : nom ou liste de nom de la ou les colonnes utilisées pour effectuer le tri ;
- `axis` : `0` pour l'index (par défaut), `1` pour les colonnes
- `ascending` : booléen ou liste de booléens, quand `True` le tri est fait par valeurs croissantes (par défaut), quand `False` il est effectué par valeurs décroissantes
- `inplace` : si `True`, le tri affecte le dataframe, sinon il retourne une vue ;
- `kind` : choix de l'algorithme de tri (`quicksort` (par défaut), `mergesort`, `heapsort`) ;

- `na_position` : si `first`, les valeurs manquantes sont placées au début ; si `last` (par défaut), à la fin.

Donnons quelques exemples :

```
>>> dico = {"height" : [58, 59, 60, 61, 62],
        "weight": [115, np.nan, 120, 123, 126],
        "age": [28, 33, 31, 31, 29],
        "taille": [162, 156, 172, 160, 158],
        "married": [True, True, np.nan, False, True],
        "city": ["A", "B", "B", "B", "A"]
       }
>>> df = pd.DataFrame(dico)
```

Si on trie les valeurs par ordre décroissant des valeurs de la colonne `height` :

```
>>> df.sort_values(by="height", ascending=False)
```

Pour effectuer un tri par ordre croissant des valeurs de `married` (rappel, `True` est interprété comme 1 et `False` comme 0), puis décoissant de `weight`, en plaçant les valeurs `NaN` en premier :

```
>>> df.sort_values(by=["married", "weight"],
                ascending=[True, False],
                na_position="first")
```

On note que les valeurs `NaN` sont remontées en avant pour les sous-groupes composés en fonction des valeurs de `married`.

12 Concaténation

Il est fréquent d'avoir des données en provenance de plusieurs sources lorsque l'on réalise une analyse. Il est alors nécessaire de pouvoir combiner les différentes sources dans une seule. Dans cette section, nous allons nous contenter de concaténer différents dataframes entre-eux, dans des cas simples dans lesquels on sait *a priori* qu'il suffit de coller deux dataframes côte-à-côte ou l'un en-dessous de l'aure. Le cas des jointures un peu plus élaborées avec appariement en fonction d'une ou plusieurs colonnes est abordé dans la Section 13.

Dans un premier temps, créons deux dataframes avec le même nombre de lignes :

```
>>> x_1 = pd.DataFrame(np.random.randn(5, 4),
                       columns=["a", "b", "c", "d"])
>>> x_2 = pd.DataFrame(np.random.randn(5, 2),
                       columns = ["e", "f"])
>>> print("x_1 : \n", x_1)
## x_1 :
##           a         b         c         d
## 0  0.058606 -0.559564 -2.227457 -0.674820
## 1  1.014849 -0.557025 -0.424606  0.137496
## 2 -0.070513  0.325394 -0.473522 -1.609218
## 3  0.532768  1.413138  0.231711 -0.474710
## 4 -0.309147 -2.032396 -0.174468 -0.642475
>>> print("\nx_2 : \n", x_2)
##
```

```
## x_2 :
##           e         f
## 0 -0.625023  1.325887
## 1  0.531255  1.275284
## 2 -0.682826 -0.948186
## 3  0.777362  0.325113
## 4 -1.203486  1.209543
```

Pour “coller” le dataframe `x_2` à droite de `x_1`, on peut utiliser la méthode `concat()` de `pandas`. Pour indiquer que la concaténation s’effectue sur les colonnes, on précise la valeur `1` pour le paramètre `axis` comme suit :

```
>>> print(pd.concat([x_1, x_2], axis = 1))
##          a         b         c         d         e         f
## 0  0.058606 -0.559564 -2.227457 -0.674820 -0.625023  1.325887
## 1  1.014849 -0.557025 -0.424606  0.137496  0.531255  1.275284
## 2 -0.070513  0.325394 -0.473522 -1.609218 -0.682826 -0.948186
## 3  0.532768  1.413138  0.231711 -0.474710  0.777362  0.325113
## 4 -0.309147 -2.032396 -0.174468 -0.642475 -1.203486  1.209543
```

Pour coller les dataframes les uns en-dessous des autres, on peut utiliser la méthode `append()`, comme indiqué dans la Section 8.2.1, ou on peut aussi utiliser la méthode `concat()`.

```
>>> x_3 = pd.DataFrame(np.random.randn(5, 2),
                       columns = ["e", "f"])
>>> print("x_3 : \n", x_3)
## x_3 :
##           e         f
## 0  0.157622 -0.293555
## 1  0.111560  0.597679
## 2 -1.270093  0.120949
## 3 -0.193898  1.804172
## 4 -0.234694  0.939908
```

Rajoutons les observations de `x_3` en-dessous de celles de `x_2` :

```
>>> print(pd.concat([x_2, x_3], axis = 0))
##           e         f
## 0 -0.625023  1.325887
## 1  0.531255  1.275284
## 2 -0.682826 -0.948186
## 3  0.777362  0.325113
## 4 -1.203486  1.209543
## 0  0.157622 -0.293555
## 1  0.111560  0.597679
## 2 -1.270093  0.120949
## 3 -0.193898  1.804172
## 4 -0.234694  0.939908
```

Comme on peut le voir, l’indice des lignes de `x_2` n’a pas été modifié. Si on souhaite qu’il le soit, on peut le préciser via le paramètre `ignore_index` :

```
>>> print(pd.concat([x_2, x_3], axis = 0, ignore_index=True))
##           e         f
## 0 -0.625023  1.325887
## 1  0.531255  1.275284
## 2 -0.682826 -0.948186
## 3  0.777362  0.325113
## 4 -1.203486  1.209543
## 5  0.157622 -0.293555
## 6  0.111560  0.597679
## 7 -1.270093  0.120949
## 8 -0.193898  1.804172
## 9 -0.234694  0.939908
```

Si le nom des colonnes n'est pas ientique, des valeurs `NaN` seront introduites :

```
>>> x_4 = pd.DataFrame(np.random.randn(5, 2),
                       columns = ["e", "g"])
>>> print("x_4 : \n", x_4)
## x_4 :
##            e         g
## 0 -0.171520 -0.153055
## 1 -0.363095 -0.067318
## 2  1.444721  0.325771
## 3 -0.855732 -0.697595
## 4 -0.276134 -1.258759
>>> pd.concat([x_2, x_4], axis = 0, sort=False, ignore_index=True)
```

13 Jointures

Il est plus fréquent d'avoir recours à des jointures un peu plus élaborées pour rassembler les différentes sources de données en une seule. `pandas` offre un moyen performant pour rassembler les données, la fonction `merge()`.

Pour illustrer les différentes jointures de cette section, créons quelques dataframes :

```
>>> exportations_fr = pd.DataFrame(
    {"country" : "France",
     "year" : np.arange(2014, 2017),
     "exportations" : [816.8192172, 851.6632573, 867.4014253]
    })

>>> importations_fr = pd.DataFrame(
    {"country" : "France",
     "year" : np.arange(2015, 2018),
     "importations" : [898.5242962, 936.3691166, 973.8762149]
    })

>>> exportations_us = pd.DataFrame(
    {"country" : "USA",
     "year" : np.arange(2014, 2017),
     "exportations" : [2208.678084, 2217.733347, 2210.442218]
    })

>>> importations_us = pd.DataFrame(
    {"country" : "USA",
```

```
     "year" : np.arange(2015, 2018),
     "importations" : [2827.336251, 2863.264745, np.nan]
    })

>>> importations_maroc = pd.DataFrame(
    {"pays" : "Maroc",
     "annee" : np.arange(2015, 2018),
     "importations" : [46.39884177, 53.52375588, 56.68165748]
    })
>>> exportations_maroc = pd.DataFrame(
    {"country" : "Maroc",
     "year" : np.arange(2014, 2017),
     "exportations" : [35.50207915, 37.45996653, 39.38228396]
    })

>>> exportations = pd.concat([exportations_fr, exportations_us],
ignore_index=True)
>>> importations = pd.concat([importations_fr, importations_us],
ignore_index=True)

>>> print("exportations : \n", exportations)
## exportations :
##    country  year  exportations
## 0  France  2014    816.819217
## 1  France  2015    851.663257
## 2  France  2016    867.401425
## 3     USA  2014   2208.678084
## 4     USA  2015   2217.733347
## 5     USA  2016   2210.442218
>>> print("\nimportations : \n", importations)
##
## importations :
##    country  year  importations
## 0  France  2015    898.524296
## 1  France  2016    936.369117
## 2  France  2017    973.876215
## 3     USA  2015   2827.336251
## 4     USA  2016   2863.264745
## 5     USA  2017           NaN
```

La fonction `merge()` de `pandas` nécessite de préciser la table de gauche (que l'on appellera ici `x`) via le paramètre `left` sur qui viendra s'effectuer la jointure de la table de droite (que l'on appellera ici `y`) via le paramètre `right`.

Par défaut, la fonction `merge()` réalise une jointure de type `inner`, c'est-à-dire que toutes les toutes les lignes de `x` qui trouvent une correspondance dans `y`, et toutes les colonnes de `x` et `y` seront dans le résultat de la jointure :

```
>>> print(pd.merge(left = importations, right = exportations))
##   country  year  importations  exportations
## 0  France  2015    898.524296    851.663257
## 1  France  2016    936.369117    867.401425
## 2     USA  2015   2827.336251   2217.733347
## 3     USA  2016   2863.264745   2210.442218
```

Si on désire changer le type de jointure, on peut modifier la valeur du paramètre `how` de la fonction `merge()`, pour lui donner une des valeurs suivantes :

- `left` : toutes les lignes de `x`, et toutes les colonnes de `x` et `y`. Les lignes dans `x` pour lesquelles il n'y a pas de correspondance dans `y` auront des valeurs `NaN` dans les nouvelles colonnes. S'il y a plusieurs correspondances dans les noms entre `x` et `y`, toutes les combinaisons sont retournées ;
- `inner` : toutes les lignes de `x` pour lesquelles il y a des valeurs correspondantes dans `y`, et toutes les colonnes de `x` et `y`. S'il y a plusieurs correspondances dans les noms entre `x` et `y`, toutes les combinaisons possibles sont retournées ;
- `right` : toutes les lignes de `y`, et toutes les colonnes de `y` et `x`. Les lignes dans `y` pour lesquelles il n'y a pas de correspondance dans `x` auront des valeurs `NaN` dans les nouvelles colonnes. S'il y a plusieurs correspondances dans les noms entre `y` et `x`, toutes les combinaisons sont retournées ;
- `outer` : toutes les lignes de `x` et de `y`, et toutes les colonnes de `x` et `y`. Les lignes de `x` pour lesquelles il n'y a pas de correspondance dabs `y` et celles de `y` pour lesquelles il n'y a pas de correspondance dans `x` auront des valeurs `NaN`.

```
>>> print("left : \n", pd.merge(left = importations, right = exportations, how="left"))
## left :
##    country  year  importations  exportations
## 0  France  2015    898.524296    851.663257
## 1  France  2016    936.369117    867.401425
## 2  France  2017    973.876215           NaN
## 3     USA  2015   2827.336251   2217.733347
## 4     USA  2016   2863.264745   2210.442218
## 5     USA  2017           NaN           NaN
>>>  print("\nright : \n", pd.merge(left = importations, right = exportations, how="right"))
##
## right :
##    country  year  importations  exportations
## 0  France  2015    898.524296    851.663257
## 1  France  2016    936.369117    867.401425
## 2     USA  2015   2827.336251   2217.733347
## 3     USA  2016   2863.264745   2210.442218
## 4  France  2014           NaN    816.819217
## 5     USA  2014           NaN   2208.678084
>>> print("\nouter : \n", pd.merge(left = importations, right = exportations, how="outer"))
##
## outer :
##    country  year  importations  exportations
## 0  France  2015    898.524296    851.663257
## 1  France  2016    936.369117    867.401425
## 2  France  2017    973.876215           NaN
## 3     USA  2015   2827.336251   2217.733347
## 4     USA  2016   2863.264745   2210.442218
## 5     USA  2017           NaN           NaN
## 6  France  2014           NaN    816.819217
## 7     USA  2014           NaN   2208.678084
```

Le paramètre `on`, qui attend un nom de colonne ou une liste de noms sert à désigner les colonnes permettant de faire la jointure. Les noms de colonnes doivent être identiques dans les deux dataframes.

```
>>> print(pd.merge(left = importations, right = exportations, on =
"country"))
##    country  year_x  importations  year_y  exportations
## 0   France    2015    898.524296    2014    816.819217
## 1   France    2015    898.524296    2015    851.663257
## 2   France    2015    898.524296    2016    867.401425
## 3   France    2016    936.369117    2014    816.819217
## 4   France    2016    936.369117    2015    851.663257
## 5   France    2016    936.369117    2016    867.401425
## 6   France    2017    973.876215    2014    816.819217
## 7   France    2017    973.876215    2015    851.663257
## 8   France    2017    973.876215    2016    867.401425
## 9      USA    2015   2827.336251    2014   2208.678084
## 10     USA    2015   2827.336251    2015   2217.733347
## 11     USA    2015   2827.336251    2016   2210.442218
## 12     USA    2016   2863.264745    2014   2208.678084
## 13     USA    2016   2863.264745    2015   2217.733347
## 14     USA    2016   2863.264745    2016   2210.442218
## 15     USA    2017           NaN    2014   2208.678084
## 16     USA    2017           NaN    2015   2217.733347
## 17     USA    2017           NaN    2016   2210.442218
```

Si le nom des colonnes devant servir à réaliser la jointure sont différents entre le dataframe de gauche et celui de droite, on indique au paramètre `left_on` le ou les noms de colonnes du dataframe de gauche à utiliser pour la jointure ; et au paramètre `right_on`, le ou les noms correspondants dans le dataframe de doite :

```
>>> pd.merge(left = importations_maroc, right = exportations_maroc,
         left_on= ["pays", "annee"], right_on = ["country", "year"] )
```

Avec le paramètre `suffixes`, on peut définir des suffixes à ajouter aux noms des colonnes lorsqu'il existe des colonnes dans `x` et dans `y` portant le même nom mais ne servant pas à la jointure. Par défaut, les suffixes (`_x` et `_y`) sont rajoutés.

```
>>> print(pd.merge(left = importations, right = exportations,
                on = "country",
                suffixes=("_gauche", "_droite")).head(3))
##   country  year_gauche  importations  year_droite  exportations
## 0  France         2015    898.524296         2014    816.819217
## 1  France         2015    898.524296         2015    851.663257
## 2  France         2015    898.524296         2016    867.401425
```

14. Agrégation

Il arrive de vouloir agréger les valeurs d'une variable, pour passer par exemple d'une dimension trimestrielle à annuelle. Avec des observations spatiales, cela peut aussi être le cas, comme par exemple lorsque l'on dispose de données à l'échelle des départements et que l'on souhaite connaître les valeurs agrégées à l'échelle des régions.

Pour illustrer les différentes opérations d'agrégation, créons un dataframe avec des des données de chômage dans différentes régions, départements et années :

```
>>> chomage = pd.DataFrame(
    {"region" : (["Bretagne"]*4 + ["Corse"]*2)*2,
     "departement" : ["Cotes-d'Armor", "Finistere",
                      "Ille-et-Vilaine", "Morbihan",
                      "Corse-du-Sud", "Haute-Corse"]*2,
     "annee" : np.repeat([2011, 2010], 6),
     "ouvriers" : [8738, 12701, 11390, 10228, 975, 1297,
                   8113, 12258, 10897, 9617, 936, 1220],
     "ingenieurs" : [1420, 2530, 3986, 2025, 259, 254,
                     1334, 2401, 3776, 1979, 253, 241]
    })
>>> print(chomage)
##       region      departement  annee  ouvriers  ingenieurs
## 0   Bretagne    Cotes-d'Armor   2011      8738        1420
## 1   Bretagne        Finistere   2011     12701        2530
## 2   Bretagne  Ille-et-Vilaine   2011     11390        3986
## 3   Bretagne         Morbihan   2011     10228        2025
## 4      Corse     Corse-du-Sud   2011       975         259
## 5      Corse      Haute-Corse   2011      1297         254
## 6   Bretagne    Cotes-d'Armor   2010      8113        1334
## 7   Bretagne        Finistere   2010     12258        2401
## 8   Bretagne  Ille-et-Vilaine   2010     10897        3776
## 9   Bretagne         Morbihan   2010      9617        1979
## 10     Corse     Corse-du-Sud   2010       936         253
## 11     Corse      Haute-Corse   2010      1220         241
```

Comme nous l'avons vu précédemment (c.f. Section 10.10.1), on peut utiliser des méthodes permettant de calculer des statistiques simples sur l'ensemble des données. Par exemple, pour afficher la moyenne de chacune des colonnes numériques :

```
>>> print(chomage.mean())
## annee         2010.500000
## ouvriers      7364.166667
## ingenieurs    1704.833333
## dtype: float64
```

Ce qui nous intéresse dans cette section, est d'effectuer des calculs sur des sous-groupes de données. Le principe est simple : dans un premier temps, on sépare les données en fonction de groupes identifiés (*split*), puis on applique une opération sur chacun des groupes (*apply*), et enfin on rassemble les résultats (*combine*). Pour effectuer le regroupement, en fonction de facteurs avant d'effectuer les calculs d'agrégation, `pandas` propose la méthode `groupby()`. On lui fournit en paramètre le ou les noms de colonnes servant à effectuer les groupes.

14.1 Agrégation selon les valeurs d'une seule colonne

Par exemple, admettons que nous souhaitons obtenir le nombre total de chomeurs ouvriers par année. Dans un premier temps, on utilise la méthode `groupby()` sur notre dataframe en indiquant que les groupes doivent être créés selon les valeurs de la colonne `annee`

```
>>> print(chomage.groupby("annee"))
## <pandas.core.groupby.groupby.DataFrameGroupBy object at 0x128de5cc0>
```

Ensuite, on récupère la variable `ouvriers` :

```
>>> print(chomage.groupby("annee").annee)
# Ou bien
## <pandas.core.groupby.groupby.SeriesGroupBy object at 0x12890a0f0>
>>> print(chomage.groupby("annee")["annee"])
## <pandas.core.groupby.groupby.SeriesGroupBy object at 0x128df30b8>
```

Et enfin, on peut effectuer le calcul sur chaque sous-groupe et afficher le résultat :

```
print(chomage.groupby("annee")["ouvriers"].sum())
## annee
## 2010    43041
## 2011    45329
## Name: ouvriers, dtype: int64
```

Si on veut effectuer ce calcul pour plusieurs colonnes, par exemple `ouvriers` et `ingenieurs`, il suffit de sélectionner *a priori* la variale de regroupement et les variables pour lesquelles on désire effectuer le calcul :

```
chomage.loc[:,["annee", "ouvriers", "ingenieurs"]].groupby("annee").sum()
```

14.2 Agrégation selon les valeurs de plusieurs colonnes

À présent, admettons que l'on souhaite effectuer une agrégation en fonction de l'année et de la région. Il s'agit simplement de donner une liste contenant les noms des colonnes utilisées pour créer les différents groupes :

```
chomage.loc[:,["annee", "region",
               "ouvriers", "ingenieurs"]].groupby(["annee",
                                                   "region"]).sum()
```

À compléter

15. Exportation et importation de données

`pandas` offre de nombreuses fonctions pour importer et exporter des données dans différents formats.

15.1 Exportation des données

15.1.1 Exportation de données tabulaires

15.1.1.1 Vers un fichier CSV {pandas-export_csv}

Pour exporter des données tabulaires, comme celles contenues dans un dataframe, `NumPy` propose la méthode `to_csv()`, qui accepte de nombreuses spécifications. Regardons quelques-unes d'entre-elles qui me semblent les plus courantes :

Paramètre	Description
`path_or_buf`	chemin vers le fichier
`sep`	caractère de séparation des champs
`decimal`	Caractère à utiliser pour le séparateur de décimales
`na_rep`	représentation à utiliser pour les valeurs manquantes
`header`	indique si le nom des colonnes doit être exporté (`True` par défaut)
`index`	indique si le nom des lignes doit être exporté (`True` par défaut)
`mode`	mode d'écriture python (c.f. Tableau 5.1, par défaut `w`)
`encoding`	encodage des caractères (`utf-8` par défaut)
`compression`	compression à utiliser pour le fichier de destination (`gzip`, `bz2`, `zip`, `xz`)
`line_terminator`	caractère de fin de ligne
`quotechar`	Caractère utilisé pour mettre les champs entre *quotes*
`chunksize`	(entier) nombre de lignes à écrire à la fois
`date_format`	format de dates pour les objets `datetime`

Admettons que nous souhaitons exporter le contenu du dataframe `chomage` vers un fichier CSV dont les champs sont séparés par des points-virgules, et en n'exportant pas l'index :

```
>>> chomage = pd.DataFrame(
    {"region" : (["Bretagne"]*4 + ["Corse"]*2)*2,
     "departement" : ["Cotes-d'Armor", "Finistere",
                      "Ille-et-Vilaine", "Morbihan",
                      "Corse-du-Sud", "Haute-Corse"]*2,
     "annee" : np.repeat([2011, 2010], 6),
     "ouvriers" : [8738, 12701, 11390, 10228, 975, 1297,
                   8113, 12258, 10897, 9617, 936, 1220],
     "ingenieurs" : [1420, 2530, 3986, 2025, 259, 254,
                     1334, 2401, 3776, 1979, 253, 241]
    })
>>> print(chomage)
##       region      departement  annee  ouvriers  ingenieurs
## 0   Bretagne    Cotes-d'Armor   2011      8738        1420
## 1   Bretagne        Finistere   2011     12701        2530
## 2   Bretagne  Ille-et-Vilaine   2011     11390        3986
## 3   Bretagne         Morbihan   2011     10228        2025
## 4      Corse     Corse-du-Sud   2011       975         259
## 5      Corse      Haute-Corse   2011      1297         254
## 6   Bretagne    Cotes-d'Armor   2010      8113        1334
## 7   Bretagne        Finistere   2010     12258        2401
## 8   Bretagne  Ille-et-Vilaine   2010     10897        3776
## 9   Bretagne         Morbihan   2010      9617        1979
## 10     Corse     Corse-du-Sud   2010       936         253
## 11     Corse      Haute-Corse   2010      1220         241
```

Pour l'exportation :

```
chemin = "./fichiers_exemples/chomage.csv"
chomage.to_csv(chemin, decimal=";", index=False)
```

Si on désire que le fichier CSV soit compressé dans un fichier `gzip`, on le nomme avec l'extention `.csv.gz` et on ajoute la valeur `gzip` au paramètre `compression` :

```
>>> chemin = "./Python_pour_economistes/fichiers_exemples/chomage.csv.gz"
>>> chomage.to_csv(chemin, decimal=";", index=False, compression="gzip")
## FileNotFoundError: [Errno 2] No such file or directory:
'./Python_pour_economistes/fichiers_exemples/chomage.csv.gz'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
##   File "/anaconda3/lib/python3.6/site-packages/pandas/core/frame.py",
line 1745, in to_csv
##     formatter.save()
##   File "/anaconda3/lib/python3.6/site-
packages/pandas/io/formats/csvs.py", line 136, in save
##     compression=None)
##   File "/anaconda3/lib/python3.6/site-packages/pandas/io/common.py", line
400, in _get_handle
##     f = open(path_or_buf, mode, encoding=encoding)
```

17. Importation des données

`pandas` propose de nombreuses fonctions pour importer des données. Dans cette version des notes de cours, nous allons en aborder 3 : `read_csv()`, pour lire des fichiers CSV ; `read_excel()`, pour lire des fichiers Excel ;

Dans la prochaine version, des ajouts sur `read_html()`, `read_fwf()`, `read_stata()`, `read_json()`.

17.1 Fichiers CSV

Pour importer des données depuis un fichier CSV, `pandas` propose la fonction `read_csv()` :

```
>>> chemin = "./fichiers_exemples/chomage.csv"
>>> chomage = pd.read_csv(chemin, decimal=";", index=False)
## TypeError: parser_f() got an unexpected keyword argument 'index'
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
```

Il est possible de fournir une URL pointant vers un fichier CSV comme chemin, la fonction `read_csv()`.

Parmi les paramètres que l'on utilise fréquemment :

- `sep`, `delimiter` : séparateur de champs ;
- `decimal` : séparateur de décimales ;
- `header` : numéro(s) de ligne(s) à utiliser comme en-tête des données ;

- `skiprows` : numéro(s) de ligne(s) à sauter au début ;
- `skipfooter` : numéro(s) de ligne(s) à sauter à la fin ;
- `nrows` : nombre de ligne à lire ;
- `na_values` : chaînes de caractères supplémentaires à considérer comme valeurs manquantes (en plus de `#N/A`, `#N/A N/A`, `#NA`, `-1.#IND`, `-1.#QNAN`, `-NaN`, `-nan`, `1.#IND`, `1.#QNAN`, `N/A`, `NA`, `NULL`, `NaN`, `n/a`, `nan`, `null`) ;
- `quotechar` : caractère de *quote* ;
- `encoding` : encodage des caractères (défaut `utf-8`).