Introduction

- Pandas est une **librairie Python spécialisée** dans **l'analyse des données**.
- Dans ce support, nous nous intéresserons surtout aux fonctionnalités de manipulations de données qu'elle propose.
  - Un objet de type "data frame", bien connu sous R, permet de réaliser de nombreuses opérations de filtrage, prétraitements, etc., préalables à la modélisation statistique.
- **Fonctionnalités de pandas**
  - La richesse des fonctionnalités de la librairie pandas est une des raisons, si ce n'est la principale, d'utiliser Python pour extraire, préparer, éventuellement analyser, des données.
  - **Objets**: les classes Series et DataFrame ou table de données.
  - **Lire, écrire** création et exportation de tables de données à partir de fichiers textes (séparateurs,.csv, format fixe, compressés), binaires (HDF5 avec Pytable), HTML, XML, JSON, MongoDB, SQL...
  - **Gestion d'une table** : sélection des lignes, colonnes, transformations, réorganisation par niveau d'un facteur, discrétisation de variables quantitatives, exclusion ou imputation élémentaire de données manquantes, permutation et échantillonnage aléatoire, variables indicatrices, chaînes de caractères...
  - **Statistiques élémentaires** uni et bivariées, tri à plat (nombre de modalités, de valeurs nulles, de valeurs manquantes...), graphiques associés, statistiques par groupe, détection élémentaire de valeurs atypiques...
  - **Manipulationde tables** : concaténations, fusions, jointures, tri, gestion destypes et formats..

Les structures de données pandas

- De même que la librairie Numpy introduit le type **array indispensable à la manipulation de matrices en calcul scientifique**,
- le module **pandas introduit les classes Series** (séries chronologiques) et **DataFrame** ou table de données **indispensables en data Science**.
- Le cœur de pandas est composé des **deux structures de données primaires** sur lesquelles toutes les **manipulations, qui sont généralement effectuées** lors de l'analyse des données, sont centralisées :
  - **Series**
  - **Dataframes**
- La **Series** constitue la structure de données conçue pour accueillir une **séquence** de données **unidimensionnelles**,
- tandis que le **dataframe**, une structure de données **plus complexe**, est conçue pour contenir **des observations à plusieurs dimensions**.

- Bien que ces structures de données ne soient pas la solution universelle à tous les problèmes, elles fournissent un outil valide et robuste pour la plupart des applications.
  - En fait, ils restent très simples à comprendre et à utiliser.
  - De plus, de nombreux cas de structures de données plus complexes peuvent encore être attribués à ces deux cas simples.
- Cependant, leurs **particularités** reposent sur une intégration de fonctionnalités particulières dans leur structure d'objets **d'index et d'étiquettes**.
  - cette **fonctionnalité** permet de **manipuler facilement ces structures** de données.

Les Series

- La **Serie** est l'objet de la bibliothèque pandas conçue pour représenter des **structures de données unidimensionnelles**, similaires **à un tableau** mais avec quelques **fonctionnalités supplémentaires**.
- Sa structure interne est **simple** et se compose de **deux tableaux associés** l'un à l'autre.
  - Le tableau principal contient les **données** (données de tout type NumPy) auxquelles chaque élément est associé
  - à une **étiquette**, contenue dans l'autre tableau, appelé **index**.

Series	
index	value
0	12
1	-4
2	7
3	9

Création d'une Series

- Pour créer une série spécifiée, il vous suffit d'appeler le **constructeur Series()** et de passer **en argument un tableau contenant les valeurs** à y inclure.
  - **s = pd.Series([12, -4, 7, 9])**
  - Si aucun index n'est spécifié lors de la définition de la série, par défaut, pandas attribuera des valeurs numériques partant de 0 en tant qu'étiquettes.

- Cependant, il est souvent **préférable** de créer une **série en utilisant des étiquettes significatives** afin **de distinguer et d'identifier chaque élément** quel que soit l'ordre dans lequel il a été inséré dans la série.
  - Dans ce cas, il sera nécessaire, lors de l'appel du constructeur, d'inclure l'option **index** et d'affecter un tableau de chaînes contenant les étiquettes.
    - s = **pd.Series(**[12,-4,7,9], **index=['a','b','c','d'])**

- Si vous souhaitez voir individuellement les deux tableaux qui composent cette structure de données, vous pouvez appeler les deux attributs de la série comme suit: **index** et **values**.
  - **s.values**
  - **s.index**

Accès aux éléments

- Vous pouvez sélectionner des éléments individuels **sous forme de tableaux numpy ordinaires**, en spécifiant la clé.
  - **s[2]**
- Ou vous pouvez **spécifier l'étiquette** correspondant à la position de l'index
  - **s['b']**
- Ou sélectionner **plusieurs éléments** dans un tableau numpy
  - **s[0:2]**
- Ou vous pouvez utiliser les libellés correspondants, mais spécifiez la liste des libellés dans un tableau.
  - **s[['b','c']]**
- **Modification de valeurs d'éléments**
  - Maintenant que vous savez comment sélectionner des éléments individuels, vous savez également comment leur attribuer de nouvelles valeurs. En fait, vous pouvez sélectionner la valeur par index ou par étiquette.
  - **s[1] = 0**
  - **s['b'] = 1**

Filtrage des valeurs et Opérations et fonctions mathématiques

- Grâce à NumPy comme base de pandas et, par conséquent, pour ses structures de données, de nombreuses **opérations applicables** aux tableaux NumPy sont étendues à la série.
  - L'une d'elles **consiste à filtrer les valeurs contenues dans la structure** de données par le **biais de conditions**.
- Par exemple, si vous avez besoin de savoir quels éléments de la série sont supérieurs à 8, écrivez ce qui suit:

  s[s > 8]

- D'autres opérations telles que les opérateurs (+, -, * et /) et les fonctions mathématiques applicables au tableau NumPy peuvent être étendues à des séries.
- Vous pouvez simplement écrire l'expression arithmétique des opérateurs.

  s/2

Évaluation des valeurs

- Il y a souvent des valeurs en double dans une série.

```
serd = pd.Series([1,0,2,1,2,3], index=['white','white','blue','green','green','yellow'])
```

- Pour connaître **toutes les valeurs contenues dans la série, à l'exclusion des doublons**, vous pouvez utiliser la fonction **unique()**.
  - La valeur de retour est un tableau contenant les valeurs uniques de la série, mais pas nécessairement dans l'ordre.

```
serd.unique()
```

- Une fonction similaire à **unique()** est **value_counts()**, qui non seulement renvoie des valeurs uniques, mais calcule également les occurrences dans une série.

```
serd.value_counts()
```

- Enfin, **isin()** évalue l'appartenance, c'est-à-dire la liste de valeurs donnée.
  - Cette fonction vous indique si les valeurs sont contenues dans la structure de données.
  - Les valeurs booléennes renvoyées peuvent être très utiles lors du filtrage de données dans une série ou dans une colonne d'un dataframe.

```
serd.isin([0,3])
serd[serd.isin([0,3])]
```

Valeurs NaN

- La valeur spécifique **NaN** (Not a Number) est utilisée dans les structures de données pandas pour indiquer la présence d'un **champ vide** ou quelque chose qui n'est pas **numériquement définie**
  - En général, ces valeurs NaN posent problème et doivent être gérées d'une manière ou d'une autre, en **particulier lors de l'analyse des données**.
  - Ces données sont souvent générées lors de **l'extraction de données à partir** d'une source douteuse ou lorsque la source manque de données.
- pandas vous permet de **définir explicitement les NaN** et de les ajouter à une structure de données, telle qu'une série.
- Dans le tableau contenant les valeurs, vous entrez **np.NaN** partout où vous souhaitez définir une valeur manquante.
  - **s2 = pd.Series([5, -3, np.NaN, 14])**
- Les méthodes **isnull()** et **notnull()** sont très utiles pour identifier les index sans valeur. s2.isnull()
  s2.notnull()
- En fait, ces fonctions renvoient deux séries avec des valeurs booléennes qui contiennent les valeurs True et False, selon que l'élément est une valeur NaN ou non.
  - La fonction **isnull()** renvoie True aux valeurs NaN de la série; inversement, la fonction **notnull()** renvoie True s'ils ne sont pas NaN.
- Ces fonctions sont souvent placées à l'intérieur de filtres pour créer une condition.
  - **s2[s2.notnull()]**
  - **s2[s2.isnull()]**

Series à partir de dictionnaire

- Une autre façon de penser une série est de la **considérer comme un objet dict (dictionnaire)**.
- Cette similitude est également exploitée lors de la définition d'une série d'objets.
  - En fait, vous pouvez créer une série à partir d'un dict préalablement défini.

    mydict = {'red': 2000, 'blue': 1000, 'yellow': 500, 'orange': 1000}

    myseries = pd.Series(mydict)
  - Le **tableau de l'index** est rempli avec **les clés** tandis que **les données** sont remplies avec les **valeurs** correspondantes.
- Vous pouvez également **définir les index** de tableau **séparément**.
  - Dans ce cas, le **contrôle de la correspondance** entre les clés du tableau d'index dict et labels s'exécutera.
  - En cas d'incohérence, les **pandas ajouteront la valeur NaN**.

```
colors = ['red','yellow','orange','blue','green']
myseries = pd.Series(mydict, index=colors)
```

Operations entre Series

- Nous avons vu comment effectuer des opérations arithmétiques entre séries et valeurs scalaires.
  - La même chose est possible en **effectuant des opérations entre deux séries**, mais **dans ce cas même les étiquettes entrent en jeu**.
- En fait, l'un des grands potentiels de ce type de structures de données est que les séries peuvent aligner des données adressées différemment entre elles en **identifiant leurs étiquettes correspondantes**.
- Dans l'exemple suivant, vous additionnez deux séries n'ayant que certains éléments en commun avec l'étiquette.

```
mydict2 = {'red':400,'yellow':1000,'black':700}
myseries2 = pd.Series(mydict2)
myseries + myseries2
```

- Vous obtenez une nouvelle série d'objets dans laquelle seuls les éléments **avec la même étiquette sont additionnés**.
  - **Toutes les autres étiquettes présentes dans l'une des deux séries** sont toujours ajoutées au résultat mais **ont une valeur NaN**.

Exercice

- Création de 2 series :
  - S1 de chaines sans index
  - S2 de valeurs numériques avec index
- Accès aux éléments
  - S1
    - Afficher l’élément à un indice donné par l’utilisateur
    - Slicing
  - S2
    - Accès par les indexes

Le DataFrame

- Le dataframe est une **structure de données tabulaire** très **similaire à une feuille de calcul**.
- Cette structure de données est conçue pour **étendre les séries à plusieurs dimensions**.
- En fait, le dataframe consiste en une **collection ordonnée de colonnes**, chacune pouvant contenir une valeur d'un type différent (numérique, chaîne, booléen, etc.).

DataFrame			
	columns		
index	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Dataframe

- Contrairement aux séries, qui ont un tableau d'index contenant des étiquettes associées à chaque élément, **le dataframe a deux tableaux d'index**.
  - Le **premier** tableau d'index, **associé aux lignes**, a des fonctions très similaires au tableau d'index en série.
    - En fait, chaque étiquette est **associée à toutes les valeurs de la ligne**.
  - Le **deuxième** tableau contient une **série d'étiquettes**, chacune **associée à une colonne particulière**.

- Un dataframe peut également être compris comme un **dictionnaire de série**, où les **clés sont les noms de colonnes** et les **valeurs sont les séries** qui formeront les colonnes du dataframe.
  - De plus, **tous les éléments de chaque série sont mappés** selon **un tableau d'étiquettes, appelé index**.

Création de Dataframe

- Le moyen le plus courant de créer un nouveau dataframe est précisément de passer un objet dict au constructeur **DataFrame()**.
- Cet **objet dict** contient une clé pour chaque colonne que vous souhaitez définir, avec un tableau de valeurs pour chacune d'elles.

```
data = { 'color' : ['blue','green','yellow','red','white'],
         'object' : ['ball','pen','pencil','paper','mug'],
         'price' : [1.2,1.0,0.6,0.9,1.7]}
frame = pd.DataFrame(data)
```

- Si l'objet dict à partir duquel vous souhaitez créer un dataframe contient plus de données que vous ne le souhaitez, vous **pouvez effectuer une sélection**.
  - **Dans le constructeur du dataframe**, vous pouvez **spécifier une séquence de colonnes** à l'aide de l'option **columns**.
  - Les **colonnes seront créées dans l'ordre de la séquence**, quelle que soit la manière dont elles sont contenues dans l'objet dict.

```
frame2 = pd.DataFrame(data, columns=['object','price'])
```

Création de Dataframe

- Même pour les objets dataframe, si les étiquettes ne sont pas explicitement spécifiées dans le tableau Index, pandas attribue automatiquement une **séquence numérique à partir de 0**.
- Au lieu de cela, si vous souhaitez attribuer des étiquettes aux index d'un dataframe, vous devez utiliser **l'option index** et affectez-lui un **tableau contenant les étiquettes**.
  - frame2 = **pd.DataFrame(**data, **index**=['one','two','three','four','five'] **)**

- Au lieu d'utiliser un objet dict, vous pouvez **définir trois arguments** dans le **constructeur**, dans l'ordre suivant: une **matrice de données**, un **tableau contenant les étiquettes** affectées à l'option **index** et un **tableau contenant les noms des colonnes** affectées à l'option de **colums** .
  - Pour créer rapidement et facilement une matrice de valeurs, vous pouvez utiliser
    np.arange (16).reshape((4,4))
  - qui génère une matrice 4x4 de nombres augmentant de 0 à 15.

```
frame3 = pd.DataFrame( np.arange(16).reshape((4,4)) ,
                       index= ['red','blue','yellow','white'] ,
                       columns=['ball','pen','pencil','paper'] )
```

Sélection d'éléments

- Pour connaître **les noms de toutes les colonnes d'un dataframe**, vous pouvez spécifier **l'attribut columns** sur l'instance de l'objet dataframe.

  frame.columns

- De même, pour obtenir la liste des index, vous devez spécifier **l'attribut index**.

  frame.index

- Vous pouvez également obtenir l'ensemble complet des données contenues dans la structure de données à l'aide de **l'attribut values**.

  frame.values

- Ou, si vous souhaitez sélectionner **uniquement le contenu d'une colonne**, vous pouvez **écrire le nom de la colonne**.

  frame['price']

  - Comme vous pouvez le voir, la valeur de **retour est un objet série**.

- **Une autre façon** de procéder consiste à utiliser le nom de la **colonne** comme attribut de l'instance du dataframe.

  frame.price

- Pour les **lignes d'un dataframe**, il est possible d'utiliser **l'attribut loc** avec la **valeur d'index de la ligne** que vous souhaitez extraire.

  frame.loc[2]

  - **L'objet renvoyé** est à **nouveau une série** dans laquelle les **noms des colonnes sont devenus l'étiquette de l'index** du tableau et les valeurs sont devenues les données de la série.

Sélection d'éléments

- Pour sélectionner **plusieurs lignes**, vous **spécifiez un tableau avec la séquence** de lignes à insérer:

```
frame.loc[[2,4]]
```

- Si vous devez **extraire une partie d'un DataFrame**, en sélectionnant les lignes que vous souhaitez extraire, vous pouvez **utiliser les numéros de référence des index**.
  - En fait, vous pouvez **considérer une ligne comme une partie d'un dataframe** qui a l'index de la ligne comme valeur source et la ligne au-dessus de celle que nous voulons comme deuxième valeur.

```
frame[0:1]
   color object  price
0  blue   ball    1.2
```

  - Comme vous pouvez le voir, la valeur de retour est un dataframe contenant une **seule ligne**.
- Si vous voulez **plus d'une ligne**, vous devez **étendre la plage de sélection**.

```
frame[1:3]
    color  object  price
1   green    pen    1.0
2  yellow  pencil   0.6
```

- Enfin, si vous **souhaitez obtenir une valeur unique** dans un dataframe, vous utilisez d'abord le **nom de la colonne**, puis **l'index ou le libellé** de la ligne.

```
frame['object'][3]
```

Affectation de valeurs

- Une fois que vous avez compris comment **accéder aux différents éléments** qui composent un dataframe, **vous suivez la même logique pour affecter ou modifier** les valeurs qu'il contient.

- Par exemple, vous avez déjà vu que dans la structure dataframe, un tableau d'index est spécifié par l'attribut **index**, et la ligne contenant le nom des colonnes est spécifiée avec l'attribut **colums**.
  - Vous pouvez également attribuer une étiquette, en utilisant l'attribut **name**, à ces deux sous-structures pour les identifier.

```
>>> frame.index.name = 'id'
>>> frame.columns.name = 'item'
>>> frame
```

item	color	object	price
id			
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Affectation de valeurs

- L'une des meilleures caractéristiques des structures de données des pandas est leur grande flexibilité.
  - En fait, vous pouvez toujours intervenir à n'importe quel niveau pour modifier la structure interne des données.
  - Par exemple, une opération très courante consiste à **ajouter une nouvelle colonne**.
    - Vous pouvez le faire en attribuant simplement une valeur à l'instance de la trame de données et en spécifiant un nouveau nom de colonne.

frame['new'] = 12

```
>>> frame
   colors  object price  new
0    blue    ball   1.2   12
1   green     pen   1.0   12
2  yellow  pencil   0.6   12
3     red   paper   0.9   12
4   white     mug   1.7   12
```

- Si, cependant, vous souhaitez **mettre à jour** le contenu d'une colonne, **vous devez utiliser un tableau**.

frame['new'] = [3.0,1.3,2.2,0.8,1.1]

Affectation de valeurs

- Vous pouvez suivre une approche similaire si vous souhaitez mettre à jour une colonne entière,
  - par exemple en utilisant la fonction **np.arange()** pour mettre à jour les valeurs d'une colonne avec une séquence prédéterminée.

```
ser = pd.Series(np.arange(5))
frame['new'] = ser
```

- Enfin, pour **changer une seule valeur**, il vous suffit de sélectionner l'élément et de lui donner la nouvelle valeur.

```
frame['price'][2] = 3.3
```

Appartenance de valeurs

- Vous avez déjà vu la fonction **isin()** appliquée à la série pour déterminer l'appartenance à un ensemble de valeurs.
  - Cette fonctionnalité est également applicable aux objets dataframe.

    frame.isin([1.0,'pen'])
  - Vous obtenez un dataframe contenant des valeurs booléennes, où True indique des valeurs qui correspondent à l'appartenance.
- Si vous transmettez la valeur renvoyée en tant que condition, vous obtiendrez un nouveau dataframe contenant uniquement les valeurs qui satisfont à la condition.
  - **frame[frame.isin([1.0,'pen'])]**

Suppression de colonnes et filtrage

- **Suppression de colonnes**
  - Si vous souhaitez **supprimer une colonne entière et tout son contenu,** utilisez la commande **del**.

```
del frame['new']
```

Filtrage

- Même lorsqu'il s'agit d'un dataframe, vous pouvez appliquer le filtrage via l'application de certaines conditions.
  - Par exemple, disons que vous souhaitez obtenir toutes les valeurs inférieures à un certain nombre, par exemple 1.2.
  - `frame[frame.price < 1.2]`

	couleur	objet	prix
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9

  - Vous obtiendrez un dataframe contenant des valeurs inférieures à 1.2

DataFrame à partir d'un dictionnaire imbriqué

- Une structure de données très courante utilisée dans Python est un **dict imbriqué**, comme suit:

```
nestdict = { 'red': { 2012: 22, 2013: 33 },
             'white': { 2011: 13, 2012: 22, 2013: 16},
             'blue': {2011: 17, 2012: 27, 2013: 18} }
```

- Cette structure de données, **lorsqu'elle est transmise directement en tant qu'argument au constructeur DataFrame(),** sera interprétée par les pandas pour
  - traiter **les clés externes** comme des **noms de colonnes**
  - et les **clés internes** comme **des étiquettes pour les index**.

- Lors de l'interprétation de la structure imbriquée, il est possible que tous les champs ne trouvent pas une correspondance réussie.
  - **pandas compense** cette incohérence en **ajoutant la valeur NaN** aux valeurs manquantes.

```
nestdict = { 'red': {2012: 22, 2013: 33}, 'white': {2011: 13, 2012: 22, 2013: 16}, 'blue': {2011: 17, 2012: 27, 2013: 18} }
frame2 = pd.DataFrame(nestdict)
```

Transposition de Dataframe

- Une opération dont vous pourriez avoir besoin lorsque vous traitez des structures de **données tabulaires est la transposition**
  - c'est-à-dire que **les colonnes deviennent des lignes** et les **lignes deviennent des colonnes**
- pandas vous permet de le faire d'une manière très simple.
  - Vous pouvez obtenir la transposition du dataframe en ajoutant **l'attribut** T à son application.

  frame2.T

Exercice

- Soient dictionnaire contenant les :
  - Noms de produits
  - Prix
  - Quantité en stock
- Créer un dataframe produit à partir de ce dictionnaire
  - Afficher la colonne prix
  - Afficher deux colonnes
  - Afficher la 3ième ligne du DF
  - Slicing et afficher plusieurs lignes
- Afficher les produit dont le prix est inférieur de 2000
- Ajout d'une colonne catégorie

Index avec des étiquettes en double

- Jusqu'à présent, vous avez rencontré tous les cas dans lesquels les **index d'une même structure de données ont une étiquette unique**.
  - Bien que de nombreuses fonctions nécessitent cette condition pour s'exécuter, **cette condition n'est pas obligatoire** sur les structures de données des pandas.
  - Exemple, une série avec quelques étiquettes en double.

```
serd = pd.Series(range(6), index=['white','white','blue','green', 'green','yellow'])
```

  - Concernant la sélection des éléments dans une structure de données, s'il y a plus de valeurs en correspondance de la même étiquette, vous obtiendrez une série à la place d'un seul élément.

```
serd['white']
```

- La même logique s'applique aux dataframes, avec des index en double qui renverront un dataframe.
- Avec de petites structures de données, il est facile d'identifier les index en double, mais si la structure devient progressivement plus grande, cela commence à devenir difficile.
- À cet égard, pandas vous fournit l'attribut **is_unique** appartenant aux objets Index.
  - Cet attribut vous dira **s'il y a des index avec des étiquettes en double** dans les données de structure (à la fois série et dataframe).

```
serd.index.is_unique
frame.index.is_unique
```

Suppression

- La suppression d'une ligne ou d'une colonne devient simple, du fait des libellés utilisés pour indiquer les index et les noms de colonnes.
- pandas fournit une fonction spécifique pour cette opération, appelée **drop()**.
  - Cette méthode renverra un nouvel objet sans les éléments que vous souhaitez supprimer.

  Exemple

  - ser = **pd.Series(**np.arange(4.), index=['red','blue','yellow','white']**)**
  - **ser.drop('yellow')**
- Pour supprimer **plus d'éléments**, passez simplement un tableau avec les étiquettes correspondantes.

  ser.drop(['blue','white'])

Suppression

- Concernant le dataframe à la place, les valeurs peuvent être supprimées en se référant aux étiquettes des deux axes.

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)), index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
```

  - Pour **supprimer des lignes**, il vous suffit de **passer les index des lignes**.

```
frame.drop(['blue','yellow'])
```

  - Pour **supprimer des colonnes**, vous devez spécifier **les index des colonnes**, et **spécifier l'axe à partir duquel** supprimer les éléments,
    - et cela peut être fait à l'aide de l'option **axis**.
    - Donc, pour faire référence aux **noms de colonnes**, vous devez spécifier **axis = 1**.
  - **frame.drop(['pen','pencil'], axis=1)**

Arithmétique et alignement de données

- Les pandas se révèlent très puissants pour aligner les index lors des opérations atithmétiques.

```
s1 = pd.Series([3,2,5,1], ['white','yellow','green','blue'] )
s2 = pd.Series( [1,4,7,2,1], ['white','yellow','black','blue','brown'] )
```

- Considérons la **somme simple**.
  - Certaines étiquettes sont présentes dans les deux, tandis que d'autres ne sont présentes que dans l'une des deux.
  - Lorsque les étiquettes **sont présentes dans les deux opérandes**, leurs valeurs seront **additionnées**,
  - tandis que dans le **cas contraire**, elles seront également affichées dans le résultat (nouvelle série), mais avec la valeur **NaN**.

```
s1 + s2
```

Arithmétique et alignement de données

- Dans le cas de la dataframe, bien que cela puisse paraître plus complexe, l'alignement suit le même principe, mais **est réalisé à la fois pour les lignes et pour les colonnes**.

```
frame1 = pd.DataFrame(np.arange(16).reshape((4,4)), index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
frame2 = pd.DataFrame(np.arange(12).reshape((4,3)), index=['blue','green','white','yellow'],
columns=['mug','pen','ball'])
frame1 + frame2
```

Méthodes arithmétiques flexibles

- Les mêmes **opérations** arihtmétiques peuvent également **être effectuées à l'aide de méthodes** appropriées, appelées **méthodes arithmétiques flexibles**.
  - **add()**
  - **sub()**
  - **div()**
  - **mul()**
- Pour appeler ces fonctions, vous devez utiliser une spécification différente de celle que vous avez l'habitude de traiter avec les opérateurs mathématiques.
  - Par exemple, au lieu d'écrire une somme entre deux dataframes, telles que **frame1 + frame2**, vous devez utiliser le format suivant:

```
frame1.add(frame2)
```

  - Comme vous pouvez le voir, les résultats sont les mêmes que ceux que vous obtiendriez en utilisant l'opérateur d'addition +.

Operations entre DataFrame et Series

- Pour en revenir aux opérateurs arithmétiques, pandas vous permet d'effectuer des manipulation entre différentes structures : par exemple, entre un dataframe et une série.
- Par exemple, vous pouvez définir ces deux structures de la manière suivante.

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)), index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])

ser = pd.Series(np.arange(4), index=['ball','pen','pencil','paper'])
```

- Les deux structures de données nouvellement définies ont été créées spécifiquement pour **que les index des séries correspondent** aux **noms des colonnes du dataframe**.

- De cette façon, vous pouvez appliquer une opération directe.

```
frame – ser
```

- Comme vous pouvez le voir, les éléments de la série sont soustraits des valeurs du dataframe **correspondant au même index sur la colonne**.
- La valeur est soustraite pour toutes les valeurs de la colonne, quel que soit leur index.
- Si un index n'est pas présent dans l'une des deux structures de données, le résultat sera une nouvelle colonne avec cet index uniquement que tous ses éléments seront NaN.

```
ser['mug'] = 9
frame - ser
```

Fonctions par élément

- La bibliothèque pandas est construite sur les fondations de **NumPy** et étend ensuite nombre de ses fonctionnalités en les adaptant à de nouvelles structures de données sous forme de séries et de dataframe.
  - Parmi celles-ci figurent les **fonctions universelles**, appelées **ufunc**.
  - Cette **classe de fonctions opère par élément** dans la structure de données.

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)), index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
```

- Par exemple, vous pouvez calculer la racine carrée de chaque valeur dans le dataframe à l'aide de NumPy **np.sqrt()**.

```
np.sqrt(frame)
```

Fonctions statistiques

- La plupart des fonctions statistiques des tableaux sont toujours valides pour dataframe
- Par exemple, des fonctions telles que **sum()** et **mean()** peuvent calculer la somme et la moyenne, respectivement, des éléments contenus dans un dataframe.

  frame.sum()

  frame.mean()

- Il existe également une fonction appelée **describe()** qui vous permet **d'obtenir des statistiques récapitulatives** à la fois.

  frame.describe()

Tri

- Le tri des données est souvent une nécessité et il est très important de pouvoir le faire facilement. pandas fournit la fonction **sort_index()**, qui **renvoie un nouvel objet identique** au début, mais **dans lequel les éléments sont ordonnés**.
- L'opération est assez triviale puisque la **liste des index à ordonner** est une seule.

```
ser = pd.Series( [5,0,3,8,4], index=['red','blue','yellow','white','green'])
ser.sort_index()
```

  - Comme vous pouvez le voir, les éléments ont été triés par ordre alphabétique croissant en fonction de leurs étiquettes (de A à Z).
  - Il s'agit du comportement par défaut, mais vous pouvez définir l'ordre inverse en définissant l'option ***ascending*** sur False.

```
ser.sort_index(ascending=False)
```

- Avec le dataframe, le tri peut être effectué **indépendamment sur chacun de ses deux axes.**
  - Si vous préférez classer par colonnes, vous devez définir les options de l'axe sur 1.

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)),   index=['red','blue','yellow','white'],
 columns=['ball','pen','pencil','paper'] )
frame.sort_index()
frame.sort_index(axis=1)
```

- Si vous souhaitez classer la série, vous devez utiliser la fonction **sort_values().**
  - **ser.sort_values()**
- Si vous avez besoin d'ordonner les valeurs dans un dataframe, utilisez la fonction sort_values() avec l'option **by**, ensuite, vous devez spécifier le **nom de la colonne sur laquelle trier**.

```
frame.sort_values(by='pen')
```

- Si les critères de tri seront basés sur **deux colonnes ou plus**, vous pouvez attribuer un **tableau contenant les noms des colonnes** à l'option by.

```
frame.sort_values(by= ['pen','pencil'] )
```

Corrélation et Covariance

- Deux calculs statistiques importants sont la corrélation et la covariance, exprimées en pandas par les fonctions **corr()** et **cov().**
- Ces types de calculs **impliquent normalement deux séries**.

```
seq2 = pd.Series([3,4,3,4,5,4,3,2],['2006','2007','2008',  '2009','2010','2011','2012','2013'])
seq = pd.Series([1,2,3,4,4,3,2,1],['2006','2007','2008',   '2009','2010','2011','2012','2013'])
seq.corr(seq2)
0.7745966692414835
seq.cov(seq2)
0.8571428571428571
```

- La covariance et la corrélation peuvent également être appliquées à un seul dataframe.
  - Dans ce cas, ils renvoient leurs matrices correspondantes sous la forme de **deux nouveaux objets dataframe**.

```
frame2 = pd.DataFrame([[1,4,3,6],[4,5,6,1],[3,3,1,5],[4,1,6,4]], index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
frame2.corr()
frame2.cov()
```

Corrélation et Covariance

- En utilisant la méthode **corrwith(),** vous pouvez calculer les **corrélations par paires entre les colonnes ou les lignes d'un dataframe avec une série ou un autre DataFrame**.

```
ser = pd.Series([0,1,2,3,9],  index=['red','blue','yellow','white','green'])
frame2.corrwith(ser)
frame2.corrwith(frame)
```

Les données “Not a Number”

- Les **données manquantes** sont reconnaissables dans les structures de données par la valeur NaN (Not a Number).
  - Ainsi, avoir des valeurs qui ne sont pas définies dans une structure de données est assez courant dans l'analyse de données.
- Dans la bibliothèque pandas, **le calcul des statistiques descriptives exclut implicitement les valeurs NaN**.

Affection de valeur NaN

- Pour affecter spécifiquement une valeur NaN à un élément, vous pouvez utiliser la valeur **np.NaN** (ou **np.nan**) de la bibliothèque NumPy.

```
ser = pd.Series([0,1,2,np.NaN,9], index=['red','blue','yellow','white','green'])
ser['white'] = None
```

Filtrer des valeurs NaN

Il existe différentes manières **d'éliminer les valeurs NaN** lors de l'analyse des données.

- Les éliminer manuellement, élément par élément, peut être très fastidieux et risqué, et vous n'êtes jamais sûr d'avoir éliminé toutes les valeurs NaN.

- C'est là que la fonction **dropna()** vous vient en aide.

  ser.dropna()

- Vous pouvez également exécuter directement la fonction de filtrage en plaçant **notnull()** dans la condition de sélection.

  ser[ser.notnull()]

Si vous avez affaire à un dataframe, cela devient un peu plus complexe.

- Si vous utilisez la fonction **dropna()** sur ce type d'objet et qu'il n'y a qu'une seule valeur NaN sur une colonne ou une ligne, elle l'éliminera.

```
frame3 = pd.DataFrame([[6,np.nan,6],[np.nan,np.nan,np.nan],[2,np.nan,5]],  index =
['blue','green','red'] ,  columns = ['ball','mug','pen'] )
frame3.dropna()
```

- Par conséquent, pour **éviter que des lignes et des colonnes entières disparaissent** complètement, vous devez spécifier l'option **how**, en lui attribuant une valeur **all**.
- Cela indique à la fonction **dropna()** de ne supprimer que les lignes ou colonnes dans lesquelles tous les éléments sont NaN.

  frame3.dropna(how='all')

Filtrer sur des occurrences NaN

- Plutôt que de filtrer les valeurs NaN dans les structures de données, **avec le risque de les ignorer avec les valeurs** qui pourraient être pertinentes dans le contexte de l'analyse des données, vous **pouvez les remplacer par d'autres nombres**.
- Dans la plupart des cas, la fonction **fillna()** est un excellent choix.
  - Cette méthode **prend un argument**, la **valeur avec laquelle remplacer tout NaN**.
  - Cela peut être le même dans tous les cas.

```
frame3.fillna(0)
```

  - Ou vous pouvez **remplacer NaN par des valeurs différentes en fonction de la colonne**, en spécifiant **un par un les index et les valeurs** associées.

```
frame3.fillna({'ball':1,'mug':0,'pen':99})
```

Exercice

- Création de 2 dataframes dont tous les éléments sont des numériques avec random
- Utilisation des méthodes sum, mean,…
- Statistiques
  - Globale
  - Par ligne
  - Par colonne

- Mise en oeuvre de la méthode describe
- Ordonner les produits selon le prix décroissant

- Corrélation entre deux series