tp-intro-Python

September 28, 2021

1 UE M2 Master MAS-MSS Projet Données Massives

2 Introduction à Python pour l'analyse statistique de données &

Résumé: Ce calepin propose une rapide présentation de Pyhton, en insistant sur l'exécution de commandes interactives ou de scripts avec un IDE (*integrated Development Environment*) et l'utilisation d'un calepin. On présente brièvement les types et structures élémentaires de données, les structures de contrôle, les fonctions, classes et modules. Une rapide description des librairies scientifiques: `Numpy, Matplotlib, Scipy` est également proposée. La présentation et la rédaction de calepin sont très largement inspirées de ceux disponibles sur le site de .

2.1 1 Introduction

2.1.1 1.1 Prérequis

Ce calepin introduit le langage libre Python et décrit les premières commandes nécessaires au pré-traitement des données avant l'utilisation de méthodes statistiques avec ce langage. Pour une présentation plus détaillée, de très nombreuses ressources pédagogiques sont disponbiles dont le tutoriel officiel de Python 3.4., les sites `pythontutor.com`, `courspython.com`, le livre de Sheppard (2014) qui présentent une introduction à Python pour l'Econométrie et la Statistique et celui de Mac Kinney (2013), principal auteur de la bibliothèque `pandas`.

2.1.2 1.2 Installation

Python et ses librairies peuvent être installés dans quasiment tout environnement matériel et système d'exploitation à partir du site officiel. Voici les principales librairies scientifiques définissant des structures de données et fonctions de calcul indispensables. - `ipython`: pour une utilisation interactive de Python, - `numpy`: pour utiliser vecteurs et tableaux, - `scipy`: intègre les principaux algorithmes numériques, - `matplotlib`: pour les graphes, - `pandas`: structure de données et feuilles de calcul, - `patsy`: formules statistiques, - `statsmodels`: modélisation statistique, - `seaborn`: visualisation de données, - `scikit-learn`: algorithmes d'apprentissage statistique.

Néanmoins, compte tenu de la complexité de l'opération, il est plus simple de faire appel à une procédure d'installation intégrant les principales librairies. Dans cette UE, on propose de privilégier l'utilisation d'Anaconda développé par l'entreprise commerciale *Continuum Analytics* (mais libre de droits pour une utilisation académique) avec le choix de la version 3.4 de Python. `Conda` est l'utilitaire (commande en ligne) qui permet les mises à jour et installations des librairies complémentaires.

2.2 2 Utilisation de Python

Dans cette UE, on choisit d'utiliser le langage Python pour exécuter des programmes ou scripts à l'aide d'un interprète de commande (`IDLE`) de manière interactive. En situation pédagogique, c'est l'utilisation et la réalisation d'un *notebook Jupyter* (calepin) qui est privilégiée à partir d'un simple navigateur .

2.2.1 2.1 Calepin *Jupyter*

Les commandes sont regroupées dans des cellules suivies de leur résultat après exécution. Ces résultats et commentaires sont stockés dans un fichier spécifique `.ipynb` et sauvegardés. Les commandes LaTeX sont acceptées pour intégrer des formules, la mise en page est assurée par des balises HTML ou *Markdown*.

La commande de sauvegarde permet également d'extraire les seules commandes Python dans un fichier d'extension `.py`. C'est une façon simple et efficace de conserver tout l'historique d'une analyse pour en faire une présentation ou créer un tutoriel. Le calepin peut être en effet chargé sous un autre format: page `html`, fichier `.pdf` ou diaporama.

Le projet Jupyter propose cet environnement de calepin pour beaucoup de langages (Pyhton, Julia, Scala…) dont R. Il devient un outil indispensable pour assurer simplement la *reproductibilité* des analyses.

L'ouverture d'un navigateur sur un calepin Jupyter est obtenu, selon l'installation, à partir des menus ou en exécutant: `jupyter notebook` dans une fenêtre de commande. Une fois le calepin ouvert, - Entrer des commandes Python dans une cellule, - Cliquer sur le bouton d'exécution de la cellule. - Ajouter une ou des cellules de commentaires et balises HTML ou Markdown. Itérer l'ajout de cellules. Une fois l'exécution terminée: - Sauver le calepin `.ipynb` - Charger éventuellement une version `.html` pour une page web ou une verion `.pdf` - Charger le fichier `.py` regroupant les commandes python pour une version opérationnelle.

2.2.2 2.2 IDE `Spyder`

Pour la réalisation d'applications et programmes plus complexes, l'usage d'un IDE libre comme Spyder est recommandé. Ce dernier est intégré à la distribution `Anaconda` et sa présentation proche de celles de Matlab ou RStudio.

Comme pour RStudio, `Spider` ouvre plusieurs fenêtres: - un éditeur de commandes dont les boutons du menu exécutent tout le fichier ou interactivement la cellule courante, sauvent le fichier, contrôlent le débogage. Une cellule débute par la balise: `#%%`. - Un explorateur d'objets avec aide en ligne, des variables en cours, du répertoire courant. Les boutons de l'explorateur de variables permettent de supprimer, sauver les objets créés ou encore d'importer des données. - La console IPython avec les résultats et son historique.

2.2.3 2.3 Exemple

Entrer les commandes ci-dessous dans le calepin et les exécuter cellule par cellule en cliquant sur le bouton d'exécution de la cellule courante. Sauvergarder le calepin sous les différents formats `.ipynb`, `.html`, `.pdf` et `.py`

```
In [1]: import sys
        print(sys.version)
```

```
3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)]
```

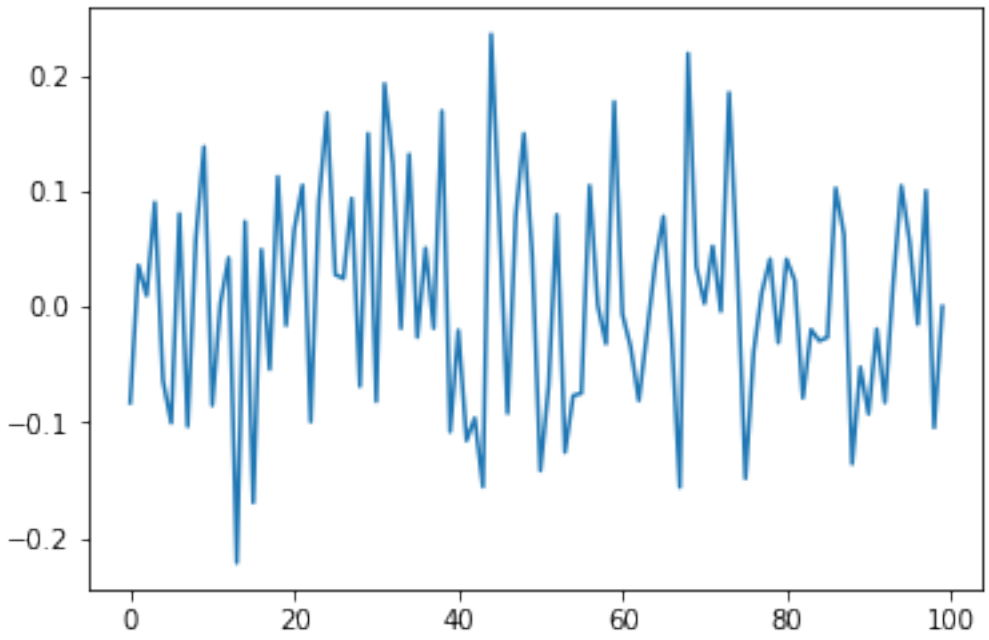
In [2]:
```
# Début d'une session Python gérée à l'aide d'un calepin.
# Le script est divisé en cellules avec généralement l'affichage d'au plus un résultat
## Importer les librairies nécessaires
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pylab import *
import os

## Commande pour obtenir les graphiques dans le calepin
%matplotlib inline
```

In [3]:
```
#%% Créer un data frame avec pandas
data = pd.DataFrame({
    'Gender': ['f', 'f', 'm', 'f', 'm', 'm', 'f', 'm', 'f', 'm'],
    'TV': [3.4, 3.5, 2.6, 4.7, 4.1, 4.0, 5.1, 4.0, 3.7, 2.1]})
data
```

Out[3]:

	Gender	TV
0	f	3.4
1	f	3.5
2	m	2.6
3	f	4.7
4	m	4.1
5	m	4.0
6	f	5.1
7	m	4.0
8	f	3.7
9	m	2.1

In [4]:
```
# Génération de variables aléatoires gaussiennes
xx = randn(100,100)
y = mean(xx,0)
# Graphique
plot(y)
show()
```




```
In [5]: help(randn)

Help on built-in function randn:

randn(...) method of mtrand.RandomState instance
    randn(d0, d1, ..., dn)

    Return a sample (or samples) from the "standard normal" distribution.

    If positive, int_like or int-convertible arguments are provided,
    `randn` generates an array of shape ``(d0, d1, ..., dn)``, filled
    with random floats sampled from a univariate "normal" (Gaussian)
    distribution of mean 0 and variance 1 (if any of the :math:`d_i` are
    floats, they are first converted to integers by truncation). A single
    float randomly sampled from the distribution is returned if no
    argument is provided.

    This is a convenience function.  If you want an interface that takes a
    tuple as the first argument, use `numpy.random.standard_normal` instead.

    Parameters
    ----------
    d0, d1, ..., dn : int, optional
        The dimensions of the returned array, should be all positive.
        If no argument is given a single Python float is returned.
```

```
    Returns
    -------
    Z : ndarray or float
        A ``(d0, d1, ..., dn)``-shaped array of floating-point samples from
        the standard normal distribution, or a single such float if
        no parameters were supplied.

    See Also
    --------
    standard_normal : Similar, but takes a tuple as its argument.

    Notes
    -----
    For random samples from :math:`N(\mu, \sigma^2)`, use:

    ``sigma * np.random.randn(...) + mu``

    Examples
    --------
    >>> np.random.randn()
    2.1923875335537315 #random

    Two-by-four array of samples from N(3, 6.25):

    >>> 2.5 * np.random.randn(2, 4) + 3
    array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],  #random
           [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]]) #random
```

2.3 3. Types de données

2.3.1 3.1 Scalaires et chaînes

La déclaration des variables est implicite et la syntaxe est très proche de celle de R (toutefois il n'y a pas de type `factor`).

```
In [6]: a=3  # est un entier
        b=1. # est un flottant

In [7]: # Comparaison
        a==b

Out[7]: False

In [8]: #  Affichage et type des variables
        type(a)

Out[8]: int
```

```
In [9]: # Chaîne de caractère
        a='bonjour '
        b='le '
        c='monde'
        a+b+c

Out[9]: 'bonjour le monde'
```

2.3.2 3.2 Structures de base

Listes Les listes permettent des combinaisons de types. **Attention**, le premier élément d'une liste ou d'un tableau est indicé par **0**, pas par 1.

```
In [10]: # Exemples de listes
         liste_A = [0,3,2,'hi']
         liste_B = [0,3,2,4,5,6,1]
         liste_C = [0,3,2,'hi',[1,2,3]]

         # Accès au deuxième d'une liste
         liste_A[1]

Out[10]: 3

In [11]: # Accès au dernier élément
         liste_C[-1]

Out[11]: [1, 2, 3]

In [12]: # Accès au premier élément du dernier élément
         liste_C[-1][0]

Out[12]: 1

In [13]: liste_C[-2]

Out[13]: 'hi'

In [14]: liste_B[0:2] #  Sous-liste

Out[14]: [0, 3]

In [15]: liste_B[0:5:2]  # début:fin:pas

Out[15]: [0, 2, 5]

In [16]: # Fonctions de listes
         List=[3,2,4,1]
         List.sort()
         print(List)
```

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

```
In [17]: List.append('hi')
         print(List)

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

```
In [18]: List.extend([7,8,9])
         print(List)

[1, 2, 3, 4, 'hi', 7, 8, 9]
```

```
In [19]: List.append([10,11,12])
         print(List)

[1, 2, 3, 4, 'hi', 7, 8, 9, [10, 11, 12]]
```

Dictionnaire Un dictionnaire est similaire à une liste mais chaque entrée est assignée par une clé / un nom, il est défini avec des accolades. Cet objet est utilisé pour la construction de l'index des colonnes (variables) du type *DataFrame* de la librairie `pandas`.

```
In [20]: months = {'Jan':31 , 'Fev': 28, 'Mar':31}
         months['Jan']

Out[20]: 31

In [21]: months.keys()

Out[21]: dict_keys(['Jan', 'Fev', 'Mar'])

In [22]: months.values()

Out[22]: dict_values([31, 28, 31])

In [23]: months.items()

Out[23]: dict_items([('Jan', 31), ('Fev', 28), ('Mar', 31)])
```

2.4 4. Syntaxe de Python

2.4.1 4.1 Structures de contrôle élémentaires

Un bloc de commandes ou de codes est défini par *deux points suivis d'une indentation fixe*. La fin d'indentation signifie la fin d'un bloc de commandes.

Structure conditionnelle

```
In [24]: # si alors sinon
         a=1
         if a>0:
             b=0
             print(b)
         else:
             b=-1
         print(b)
```

```
0
0
```

Structure itérative

```
In [25]: for i in range(1,8,2):
             print(i)
```

```
1
3
5
7
```

2.4.2 4.2 Fonctions

```
In [26]: # Définition d'une fonction
         def pythagorus(x,y):
             """ Calcule l'hypotenuse d'un triangle """
             r = pow(x**2+y**2,0.5)
             return x,y,r
```

```
In [27]: # Exemple d'appel
         pythagorus(x=3,y=4)
```

```
Out[27]: (3, 4, 5.0)
```

```
In [28]: # Aide
         help(pythagorus)
```

```
Help on function pythagorus in module __main__:

pythagorus(x, y)
    Calcule l'hypotenuse d'un triangle
```

2.5 5. Calcul scientifique

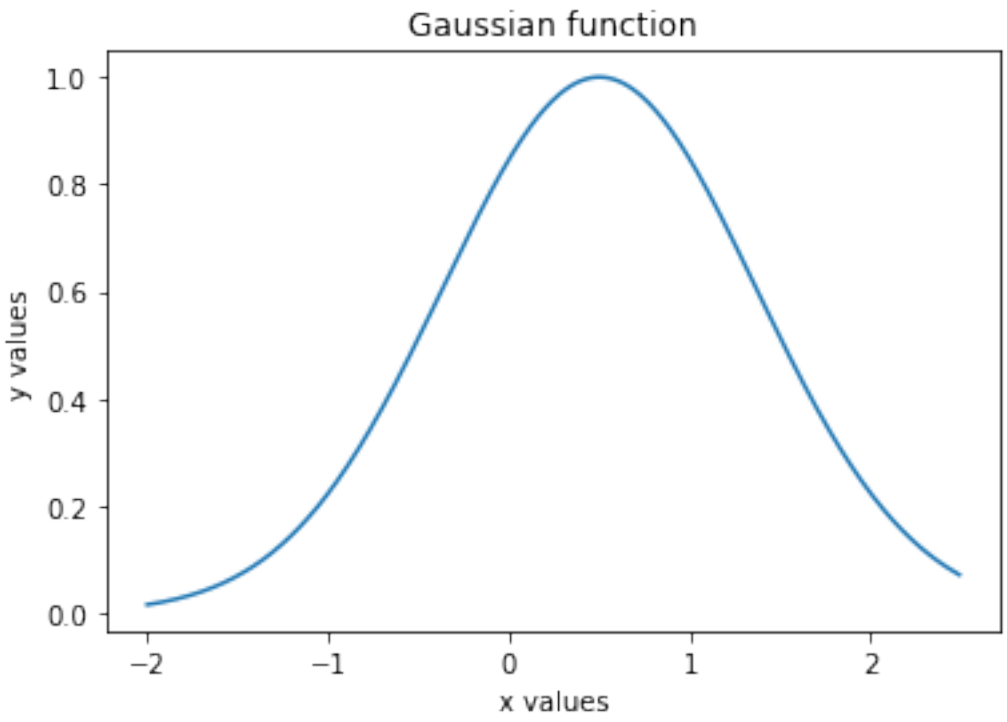
Voici quelques librairies indispensables au calcul scientifique

2.5.1 5.1 Principales librairies ou *packages*

`NumPy` Cette librairie définit le type de données `array` ainsi que les fonctions de calcul qui y sont associées. Il contient aussi quelques fonctions d'algèbre linéaire et statistiques. Il n'est finalement utilisé que pour la définition du type `array` car les fonctions numériques sont beaucoup plus développées dans `SciPy`.

`Matplotlib` Celle-ci propose des fonctions de visualisation / graphs avec des commandes proches de celles de Matlab. Aussi connue sous le nom de `pylab`. La gallerie de cette librairie propose tout un ensemble d'exemples de graphiques avec le code Python pour les générer.

```
In [29]: # Importation
         import numpy as np
         from pylab import *
         gaussian = lambda x: np.exp(-(0.5-x)**2/1.5)
         x=np.arange(-2,2.5,0.01)
         y=gaussian(x)
         plot(x,y)
         xlabel("x values")
         ylabel("y values")
         title("Gaussian function")
         show()
```



`SciPy` Cette librairie est un ensemble très complet de modules d'algèbre linéaire, statistiques et autres algorithmes numériques. Le site de la documentation en fournit la liste.

2.5.2 5.2 Type Array

C'est de loin la structure de données la plus utilisée pour le calcul scientifique sous Python. Elle décrit des tableaux ou *matrices* multi-indices de dimension $ allant de 1 à 40. Tous les éléments doivent être de même type (booléen, entier, réel, complexe).

Les tableaux ou tables de données (*data frame*), bases d'analyses statistiques et regroupant des objets de types différents sont décrits avec la librairie `pandas`.

Définition du type `array`

```
In [30]: # Importation
         import numpy as np
         my_1D_array = np.array([4,3,2])
         print(my_1D_array)

         my_2D_array = np.array([[1,0,0],[0,2,0],[0,0,3]])
         print(my_2D_array)

         my_2D_array[1]
```

```
[4 3 2]
[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

```
Out[30]: array([0, 2, 0])
```

Fonctions à valeur de type `array`

```
In [31]: np.arange(5)
```

```
Out[31]: array([0, 1, 2, 3, 4])
```

```
In [32]: np.ones(3)
```

```
Out[32]: array([1., 1., 1.])
```

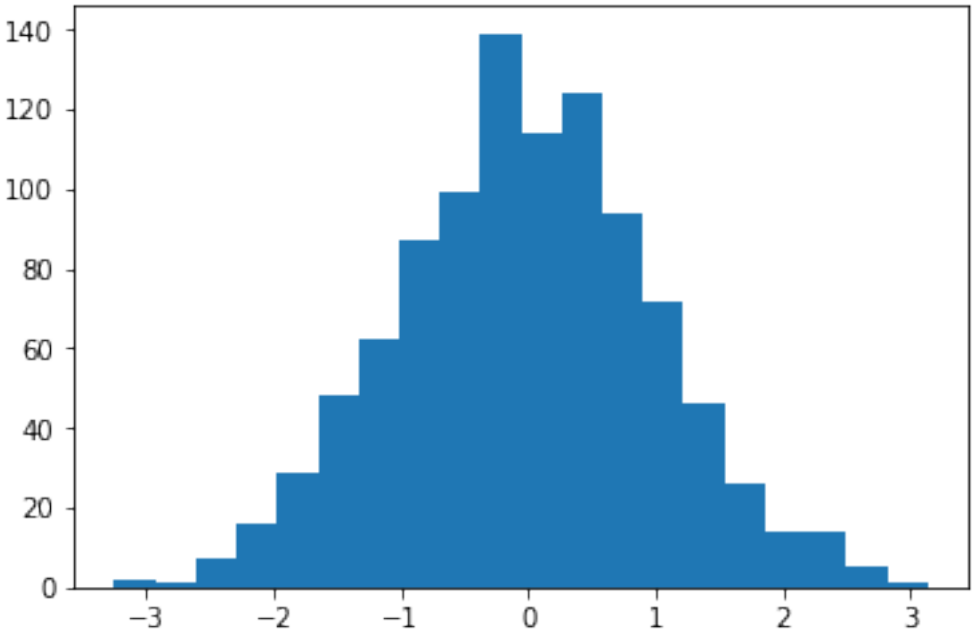
```
In [33]: np.ones((3,4))
```

```
Out[33]: array([[1., 1., 1., 1.],
                [1., 1., 1., 1.],
                [1., 1., 1., 1.]])
```

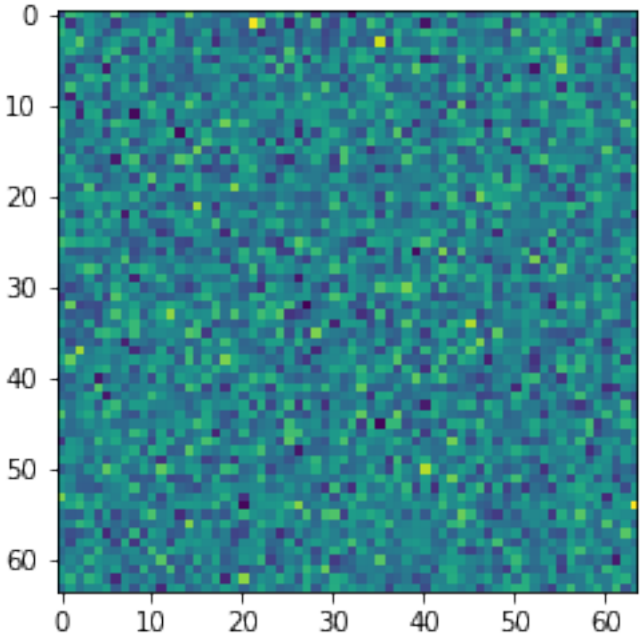
```
In [34]: np.eye(3)

Out[34]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])

In [35]: np.linspace(3, 7, 10)

Out[35]: array([3.        , 3.44444444, 3.88888889, 4.33333333, 4.77777778,
                5.22222222, 5.66666667, 6.11111111, 6.55555556, 7.        ])

In [36]: D=np.diag([1,2,4,3])
         print(D)
         print(np.diag(D))

[[1 0 0 0]
 [0 2 0 0]
 [0 0 4 0]
 [0 0 0 3]]
[1 2 4 3]
```

Le module `numpy.random` fournit toute une liste de fonctions pour la génération de matrices aléatoires.

```
In [37]: from numpy import random
         random.rand(4,2) #tirage uniforme

Out[37]: array([[0.59553642, 0.40774896],
                [0.65714043, 0.6526475 ],
                [0.35877943, 0.28705799],
                [0.74704273, 0.75534041]])

In [38]: random.randn(4,2) #tirage selon la loi N(0,1)

Out[38]: array([[-1.53281852,  0.45067726],
                [-0.11038851,  0.77104493],
                [ 0.19210964, -0.82658961],
                [ 0.8030695 ,  0.8208661 ]])

In [39]: v=random.randn(1000)
         import matplotlib.pyplot as plt
         h=plt.hist(v,20) # histogramme à 20 pas
         show()
```



In [40]:
```
A=random.randn(64,64)
plt.imshow(A,interpolation="nearest")
plt.colorbar
plt.show()
```



```
In [41]: # Sauver, écrire dans un fichier, lire un fichier
         M=random.randn(10,10)
         np.savetxt('data.csv',M,fmt='%2.2f',delimiter=',')

In [42]: # Au format propre à numpy : npy
         np.save('data.npy',M)
         np.load('data.npy')

Out[42]: array([[ 1.55988688,  1.82800708, -1.12513327,  0.15745548, -1.36148149,
                 -0.77426217,  1.14850695, -1.753012  ,  0.47124328, -0.10866965],
                [ 0.50551961, -0.2446276 , -1.18874079, -0.57990875,  0.24168502,
                 -0.32967539, -0.01819764, -0.72265048, -1.46284604,  0.61145683],
                [ 0.58002049,  0.44181077, -0.16987407, -1.57500705,  1.02269924,
                  0.3349941 , -1.30260278,  0.4902744 ,  0.48742052, -0.90857221],
                [-0.01592069, -0.56620304,  0.02004686, -1.35412522,  0.47590901,
                 -0.40592591, -1.28038165, -1.50925841, -0.73638817, -0.1841052 ],
                [ 0.98986504,  0.14567929,  0.79161938,  0.65967501,  0.58263263,
                 -0.14367809,  0.23332649,  1.02390679,  0.56627002, -0.57668016],
                [-1.59138246, -0.04217508,  0.70877171, -1.19163104,  0.06809747,
                 -0.10244072, -0.76318482,  0.21024555,  0.39632742,  0.42506331],
                [ 0.36085438, -0.86100269,  1.08986467, -0.1262439 , -1.47085244,
                 -0.90611134,  1.33750801, -0.59303609, -0.14791976, -0.95923485],
                [ 0.98167618, -0.97799586, -0.70161589, -0.09047385, -0.92448335,
                  0.65547444,  1.86579441,  0.54516528,  0.09718073, -1.06407038],
                [ 1.09234868,  0.45990172, -1.25360733,  0.11206117, -0.90504633,
                  0.31658502, -0.29623689, -0.74178912,  1.12133244, -0.59953552],
                [ 1.40481972,  0.78168326, -0.66244349,  1.04548609,  0.10049566,
                  0.68588075,  0.18269478,  0.10896863,  0.06167726,  0.09993335]])
```

Slicing

```
In [43]: v=np.array([1,2,3,4,5])
         print(v)

[1 2 3 4 5]


In [44]: v[1:4]

Out[44]: array([2, 3, 4])

In [45]: v[1:4:2]

Out[45]: array([2, 4])

In [46]: v[::]

Out[46]: array([1, 2, 3, 4, 5])

In [47]: v[ : :2] # par pas de 2
```

```
Out[47]: array([1, 3, 5])

In [48]: v[ : 3]

Out[48]: array([1, 2, 3])

In [49]: v[3 :] # à partir de l'indice 3

Out[49]: array([4, 5])

In [50]: v[-1] # dernier élément

Out[50]: 5

In [51]: v[-2 :] # deux derniers éléments

Out[51]: array([4, 5])

In [52]: M=random.rand(4,3)
         print(M)

[[0.03853487 0.94442594 0.29656618]
 [0.15282854 0.08915877 0.93003404]
 [0.4545388  0.20564332 0.98704284]
 [0.53526022 0.30220972 0.67219439]]


In [53]: ind=[1,2]
         M[ind] # lignes d'indices 1 et 2

Out[53]: array([[0.15282854, 0.08915877, 0.93003404],
                [0.4545388 , 0.20564332, 0.98704284]])

In [54]: M[:,ind] # colonnes d'indices 1 et 2

Out[54]: array([[0.94442594, 0.29656618],
                [0.08915877, 0.93003404],
                [0.20564332, 0.98704284],
                [0.30220972, 0.67219439]])

In [55]: M[[0,2],[1,2]] # renvoie M[0,1] et M[2,2]

Out[55]: array([0.94442594, 0.98704284])

In [56]: M[np.ix_([0,2],[1,2])]

Out[56]: array([[0.94442594, 0.29656618],
                [0.20564332, 0.98704284]])

In [57]: (M>0.5)

Out[57]: array([[False,  True, False],
                [False, False,  True],
                [False, False,  True],
                [ True, False,  True]])

In [58]: M[M>0.5]

Out[58]: array([0.94442594, 0.93003404, 0.98704284, 0.53526022, 0.67219439])
```

Autres fonctions

```
In [59]: a=np.array([[0,1],[2,3],[4,5]])
         np.ndim(a) # Nombre de dimensions)

Out[59]: 2

In [60]: np.size(a) # Nombre déléments

Out[60]: 6

In [61]: np.shape(a) # Tuple contenant la dimension de a

Out[61]: (3, 2)

In [62]: np.transpose(a) # Transposée

Out[62]: array([[0, 2, 4],
                [1, 3, 5]])

In [63]: a.T # autre façon de définir la transposée

Out[63]: array([[0, 2, 4],
                [1, 3, 5]])

In [64]: a.min(), np.min(a)  # Valeur min

Out[64]: (0, 0)

In [65]: a.sum(), np.sum(a)  # Somme des valeurs

Out[65]: (15, 15)

In [66]: a.sum(axis=0)  # Somme sur les colonnes

Out[66]: array([6, 9])

In [67]: a.sum(axis=1)  # sur les lignes

Out[67]: array([1, 5, 9])
```

Quelques manipulations:

```
In [68]: np.r_[1:4,10,11]  # Concaténation en ligne

Out[68]: array([ 1,  2,  3, 10, 11])

In [69]: np.c_[1:4,11:14]  # Concaténation en colonne

Out[69]: array([[ 1, 11],
                [ 2, 12],
                [ 3, 13]])
```

```
In [70]: np.arange(6).reshape(3,2)

Out[70]: array([[0, 1],
                [2, 3],
                [4, 5]])

In [71]: A=np.array([[1,2],[3,4]])
         np.tile(A,(3,2)) # Répétition de la matrice A

Out[71]: array([[1, 2, 1, 2],
                [3, 4, 3, 4],
                [1, 2, 1, 2],
                [3, 4, 3, 4],
                [1, 2, 1, 2],
                [3, 4, 3, 4]])

In [72]: A=np.array([[1,2],[3,4]])
         B=np.array([[11,12],[13,14]])
         #Concaténation en ligne
         np.concatenate((A,B),axis=0)

Out[72]: array([[ 1,  2],
                [ 3,  4],
                [11, 12],
                [13, 14]])

In [73]: # Equivalent à
         np.vstack((A,B))

Out[73]: array([[ 1,  2],
                [ 3,  4],
                [11, 12],
                [13, 14]])

In [74]: # Concaténation en colonne
         np.concatenate((A,B),axis=1)

Out[74]: array([[ 1,  2, 11, 12],
                [ 3,  4, 13, 14]])

In [75]: # Equivalent à
         np.hstack((A,B))

Out[75]: array([[ 1,  2, 11, 12],
                [ 3,  4, 13, 14]])
```

Opérations sur les arrays

```
In [76]: # somme
         a=np.arange(6).reshape(3,2)
         b=np.arange(3,9).reshape(3,2)
         c=np.transpose(b)
         a+b

Out[76]: array([[ 3,  5],
                [ 7,  9],
                [11, 13]])

In [77]: a*b # produit terme à terme

Out[77]: array([[ 0,  4],
                [10, 18],
                [28, 40]])

In [78]: np.dot(a,c) # produit matriciel

Out[78]: array([[ 4,  6,  8],
                [18, 28, 38],
                [32, 50, 68]])

In [79]: np.power(a,2)

Out[79]: array([[ 0,  1],
                [ 4,  9],
                [16, 25]])

In [80]: np.power(2,a)

Out[80]: array([[ 1,  2],
                [ 4,  8],
                [16, 32]])

In [81]: a/3

Out[81]: array([[0.        , 0.33333333],
                [0.66666667, 1.        ],
                [1.33333333, 1.66666667]])
```

Fonctions d'algèbre linéaire

```
In [82]: # Importation
         import numpy as np
         from scipy import linalg
         A = np.array([[1,2],[3,4]])
         linalg.inv(A)
```

```
Out[82]: array([[-2. ,  1. ],
                [ 1.5, -0.5]])

In [83]: linalg.det(A)

Out[83]: -2.0

In [84]: la,v = linalg.eig(A)
         l1,l2 = la
         # valeurs propres
         print(l1, l2)

(-0.3722813232690143+0j) (5.372281323269014+0j)


In [85]: # 1er vecteur propre
         print(v[:,0])

[-0.82456484  0.56576746]


In [86]: # SVD de A
         U,s,V = linalg.svd(A)
         print(s**2)

[29.86606875  0.13393125]


In [87]: linalg.eig(np.dot(np.transpose(A),A))

Out[87]: (array([ 0.13393125+0.j, 29.86606875+0.j]), array([[-0.81741556, -0.57604844],
                [ 0.57604844, -0.81741556]]))
```

2.6 6. Analyse statistique de données

On présente maintenant l'utilisation de Python pour la préparation de données qui peuvent tenir en mémoire une fois réorganisées. Cette partie est une initiation aux fonctionnalités de la librairie `pandas` et à la classe `DataFrame`.

La richesse des fonctionnalités de la librairie `pandas` est la raison principale d'utiliser Python pour extraire, préparer (et éventuellement analyser) des données. En voici un bref aperçu. - *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... - *Manipulation* de tables: concaténations, fusions, jointures, tri, gestion des types et formats...

Cette partie est basée sur la documentation en ligne qui inclut également des tutoriels à exécuter pour compléter et approfondir les possibilités de traitement de données avec la librairie `pandas`.

A titre d'illustration, on utilise des données issues d'une compétition du site Kaggle: Titanic: Machine learnic from Disaster. Le concours est terminé mais les données sont toujours disponibles sur le site avec des tutoriels utilisant Excel, Python ou R.

Une des raisons du drame, qui provoqua la mort de 1502 personnes sur les 2224 passagers et membres d'équipage, fut le manque de canots de sauvetage. Il apparaît que les chances de survie dépendaient de différents facteurs (sexe, âge, classe...). Le but du concours est de construire un modèle de prévision (classification supervisée) de survie en fonction de ces facteurs. Les données sont composées d'un échantillon d'apprentissage (891) et d'un échantillon test (418) chacun décrit par 11 variables dont la première indiquant la survie ou non lors du naufrage.

2.6.1 6.1 Les classes `Series` et `DataFrame`

De même que la librairie `Numpy` introduit le type `array` indispensable à la manipulation de matrices en calcul scientifique, celle `pandas` introduit les classes `Series` (séries chronologiques) et `DataFrame` ou table de données indispensables en statistique.

La classe `Series` est l'association de deux `arrays` unidimensionnels. Le premier est un ensemble de valeurs indexées par le 2ème qui est souvent une série temporelle. Ce type est introduit principalement pour des applications en Econométrie et Finance où Python est largement utilisé.

La classe `DataFrame` est proche de celle du même nom dans le langage R, il s'agit d'associer avec le même index de lignes des colonnes ou variables de types différents (entier, réel, booléen, caractère). C'est un tableau bi-dimensionnel avec des index de lignes et de colonnes mais il peut également être vu comme une liste de `Series` partageant le même index. L'index de colonne (noms des variables) est un objet de type `dict` (dictionnaire). C'est la classe qui sera principalement utilisée dans ce tutoriel.

```
In [88]: # Exemple de data frame
         import pandas as pd
         data = {"state": ["Ohio", "Ohio", "Ohio",
                  "Nevada", "Nevada"],
                "year": [2000, 2001, 2002, 2001, 2002],
                "pop": [1.5, 1.7, 3.6, 2.4, 2.9]}
         frame = pd.DataFrame(data)
         # ordre des colonnes
         pd.DataFrame(data, columns=["year", "state", "pop"])
```

```
Out[88]:    year   state  pop
         0  2000    Ohio  1.5
         1  2001    Ohio  1.7
         2  2002    Ohio  3.6
         3  2001  Nevada  2.4
         4  2002  Nevada  2.9
```

```
In [89]: # index des lignes et valeurs manquantes (NaN)
         frame2=pd.DataFrame(data, columns=["year", "state", "pop", "debt"],
              index=["one", "two", "three", "four", "five"])
```

```
In [ ]: # liste des colonnes
        frame.columns
```

```
Out[89]: Index(['state', 'year', 'pop'], dtype='object')
```

```
In [90]: # valeurs d'une colonnes
         frame["state"]
```

```
Out[90]: 0      Ohio
         1      Ohio
         2      Ohio
         3    Nevada
         4    Nevada
         Name: state, dtype: object
```

```
In [91]: frame.year
```

```
Out[91]: 0    2000
         1    2001
         2    2002
         3    2001
         4    2002
         Name: year, dtype: int64
```

```
In [92]: # "imputation"
         frame2["debt"] = 16.5
         frame2
```

```
Out[92]:        year   state  pop  debt
         one    2000    Ohio  1.5  16.5
         two    2001    Ohio  1.7  16.5
         three  2002    Ohio  3.6  16.5
         four   2001  Nevada  2.4  16.5
         five   2002  Nevada  2.9  16.5
```

```
In [93]: # créer une  variable
         frame2["eastern"] = frame2.state == "Ohio"
         frame2
```

```
Out[93]:        year   state  pop  debt  eastern
         one    2000    Ohio  1.5  16.5     True
         two    2001    Ohio  1.7  16.5     True
         three  2002    Ohio  3.6  16.5     True
         four   2001  Nevada  2.4  16.5    False
         five   2002  Nevada  2.9  16.5    False
```

```
In [94]: frame2.columns
```

```
Out[94]: Index(['year', 'state', 'pop', 'debt', 'eastern'], dtype='object')
```

```
In [95]: # supprimer une  variable
         del frame2[u"eastern"]
         frame2.columns
```

```
Out[95]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

2.6.2 6.2 Exemple

Les données du naufrage du Titanic illustrent l'utilisation de pandas.

```
In [96]: # Importations
         import pandas as pd
         import numpy as np
         # Chargement des données
         path='./'
         df = pd.read_csv(path+'titanic-train.csv',nrows=5)
         df
```

```
Out[96]:    PassengerId  Survived  Pclass  \
         0            1         0       3
         1            2         1       1
         2            3         1       3
         3            4         1       1
         4            5         0       3

                                                        Name     Sex  Age  SibSp  \
         0                            Braund, Mr. Owen Harris    male   22      1
         1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female   38      1
         2                             Heikkinen, Miss. Laina  female   26      0
         3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female   35      1
         4                           Allen, Mr. William Henry    male   35      0

            Parch            Ticket     Fare Cabin Embarked
         0      0         A/5 21171   7.2500   NaN        S
         1      0          PC 17599  71.2833   C85        C
         2      0  STON/O2. 3101282   7.9250   NaN        S
         3      0            113803  53.1000  C123        S
         4      0            373450   8.0500   NaN        S
```

```
In [97]: df.tail()
```

```
Out[97]:    PassengerId  Survived  Pclass  \
         0            1         0       3
         1            2         1       1
         2            3         1       3
         3            4         1       1
         4            5         0       3

                                                        Name     Sex  Age  SibSp  \
         0                            Braund, Mr. Owen Harris    male   22      1
         1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female   38      1
         2                             Heikkinen, Miss. Laina  female   26      0
         3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female   35      1
         4                           Allen, Mr. William Henry    male   35      0
```

```
   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
4      0            373450   8.0500   NaN        S
```

```
In [98]: # tout lire
         df = pd.read_csv(path+"titanic-train.csv")
         df.head()
```

```
Out[98]:    PassengerId  Survived  Pclass  \
         0            1         0       3
         1            2         1       1
         2            3         1       3
         3            4         1       1
         4            5         0       3

                                                        Name     Sex   Age  SibSp  \
         0                            Braund, Mr. Owen Harris    male  22.0      1
         1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
         2                             Heikkinen, Miss. Laina  female  26.0      0
         3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
         4                           Allen, Mr. William Henry    male  35.0      0

            Parch            Ticket     Fare Cabin Embarked
         0      0         A/5 21171   7.2500   NaN        S
         1      0          PC 17599  71.2833   C85        C
         2      0  STON/O2. 3101282   7.9250   NaN        S
         3      0            113803  53.1000  C123        S
         4      0            373450   8.0500   NaN        S
```

```
In [99]: # Des variables sont inexploitables
         # Choisir les colonnes utiles
         df=pd.read_csv(path+"titanic-train.csv",
             usecols=[1,2,4,5,6,7,9,11],nrows=5)
         df.head()
```

```
Out[99]:    Survived  Pclass     Sex  Age  SibSp  Parch     Fare Embarked
         0         0       3    male   22      1      0   7.2500        S
         1         1       1  female   38      1      0  71.2833        C
         2         1       3  female   26      0      0   7.9250        S
         3         1       1  female   35      1      0  53.1000        S
         4         0       3    male   35      0      0   8.0500        S
```

La librairie `pandas`, inclut un type `category` assez proche de celui `factor` de R. Il devrait normalement être déclaré dans un dictionnaire au moment par exemple de la lecture (`dtype={"Surv":pd.Categorical...}`) mais ce n'est pas le cas, c'est donc le type `objet` qui est

déclaré puis modifié. Il est vivement recommandé de bien affecter les bons types à chaque variable ne serait-ce que pour éviter de faire des opérations douteuses, par exemple arithmétiques sur des codes de modalités.

```
In [100]: df=pd.read_csv(path+"titanic-train.csv",skiprows=1,header=None,usecols=[1,2,4,5,9,11]
            names=["Surv","Classe","Genre","Age","Prix","Port"],dtype={"Surv":object,
              "Classe":object,"Genre":object,"Port":object})
          df.head()

Out[100]:   Surv Classe   Genre   Age     Prix Port
          0    0      3    male  22.0   7.2500    S
          1    1      1  female  38.0  71.2833    C
          2    1      3  female  26.0   7.9250    S
          3    1      1  female  35.0  53.1000    S
          4    0      3    male  35.0   8.0500    S

In [101]: df.dtypes

Out[101]: Surv       object
          Classe     object
          Genre      object
          Age       float64
          Prix      float64
          Port       object
          dtype: object
```

Redéfinition des bons types.

```
In [102]: df["Surv"]=pd.Categorical(df["Surv"],ordered=False)
          df["Classe"]=pd.Categorical(df["Classe"],ordered=False)
          df["Genre"]=pd.Categorical(df["Genre"],ordered=False)
          df["Port"]=pd.Categorical(df["Port"],ordered=False)
          df.dtypes

Out[102]: Surv      category
          Classe    category
          Genre     category
          Age        float64
          Prix       float64
          Port      category
          dtype: object
```

2.6.3 6.3 Gérer une table de données

6.3.1 Discrétisation d'une variable quantitative Pour la discrétisation d'une variable quantitative. Il est d'un bon usage de définir les bornes des classes à des quantiles, plutôt qu'également espacées, afin de construire des classes d'effectifs sensiblement égaux. Ceci est obtenu par la fonction `qcut`. La fonction `cut` propose par défaut des bornes équi-réparties à moins de fournir une liste de ces bornes.

```
In [103]: df["AgeQ"]=pd.qcut(df.Age,3,labels=["Ag1","Ag2",
             "Ag3"])
          df["PrixQ"]=pd.qcut(df.Prix,3,labels=["Pr1","Pr2",
             "Pr3"])
          df["PrixQ"].describe()

Out[103]: count     891
          unique      3
          top       Pr1
          freq      308
          Name: PrixQ, dtype: object
```

6.3.2 Modifier / regrouper des modalités Le recodage des variables qualitatives ou renommage en clair des modalités est obtenu simplement.

```
In [104]: df["Surv"]=df["Surv"].cat.rename_categories(
              ["Vnon","Voui"])
          df["Classe"]=df["Classe"].cat.rename_categories(
              ["Cl1","Cl2","Cl3"])
          df["Genre"]=df["Genre"].cat.rename_categories(
              ["Gfem","Gmas"])
          df["Port"]=df["Port"].cat.rename_categories(
              ["Pc","Pq","Ps"])

In [105]: df.head()

Out[105]:    Surv Classe Genre   Age     Prix Port AgeQ PrixQ
          0  Vnon    Cl3  Gmas  22.0   7.2500   Ps  Ag1   Pr1
          1  Voui    Cl1  Gfem  38.0  71.2833   Pc  Ag3   Pr3
          2  Voui    Cl3  Gfem  26.0   7.9250   Ps  Ag2   Pr1
          3  Voui    Cl1  Gfem  35.0  53.1000   Ps  Ag3   Pr3
          4  Vnon    Cl3  Gmas  35.0   8.0500   Ps  Ag3   Pr1
```

Il est possible d'associer recodage et regroupement des modalités en définissant un dictionnaire de transformation.

```
In [106]: data = pd.DataFrame({"food":["bacon","pulled pork",
             "bacon", "Pastrami",
             "corned beef", "Bacon", "pastrami", "honey ham",
             "nova lox"],
             "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
          data

Out[106]:          food  ounces
          0        bacon     4.0
          1  pulled pork     3.0
          2        bacon    12.0
          3     Pastrami     6.0
          4  corned beef     7.5
```

```
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

```
In [107]: meat_to_animal = {
              "bacon": "pig",
              "pulled pork": "pig",
              "pastrami": "cow",
              "corned beef": "cow",
              "honey ham": "pig",
              "nova lox": "salmon"
          }
          # Eviter les mélanges de majuscules minuscules
          # en mettant tout en minuscule
          data["animal"] = data["food"].map(
              str.lower).map(meat_to_animal)
          data
```

```
Out[107]:           food  ounces  animal
          0        bacon     4.0     pig
          1  pulled pork     3.0     pig
          2        bacon    12.0     pig
          3     Pastrami     6.0     cow
          4  corned beef     7.5     cow
          5        Bacon     8.0     pig
          6     pastrami     3.0     cow
          7    honey ham     5.0     pig
          8     nova lox     6.0  salmon
```

```
In [108]: data["food"].map(lambda x: meat_to_animal[x.lower()])
```

```
Out[108]: 0       pig
          1       pig
          2       pig
          3       cow
          4       cow
          5       pig
          6       cow
          7       pig
          8    salmon
          Name: food, dtype: object
```

```
In [109]: dfs = pd.DataFrame({"key": ["b", "b", "a", "c",
              "a", "b"],"data1": range(6)})
          print(dfs)
          pd.get_dummies(dfs["key"])
```

```
  key  data1
0   b      0
```

```
1   b      1
2   a      2
3   c      3
4   a      4
5   b      5
```

Out[109]:

	a	b	c
0	0	1	0
1	0	1	0
2	1	0	0
3	0	0	1
4	1	0	0
5	0	1	0

```
In [110]: help(pd.get_dummies)

Help on function get_dummies in module pandas.core.reshape.reshape:

get_dummies(data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, dro
    Convert categorical variable into dummy/indicator variables

    Parameters
    ----------
    data : array-like, Series, or DataFrame
    prefix : string, list of strings, or dict of strings, default None
        String to append DataFrame column names.
        Pass a list with length equal to the number of columns
        when calling get_dummies on a DataFrame. Alternatively, `prefix`
        can be a dictionary mapping column names to prefixes.
    prefix_sep : string, default '_'
        If appending prefix, separator/delimiter to use. Or pass a
        list or dictionary as with `prefix.`
    dummy_na : bool, default False
        Add a column to indicate NaNs, if False NaNs are ignored.
    columns : list-like, default None
        Column names in the DataFrame to be encoded.
        If `columns` is None then all the columns with
        `object` or `category` dtype will be converted.
    sparse : bool, default False
        Whether the dummy-encoded columns should be be backed by
        a :class:`SparseArray` (True) or a regular NumPy array (False).
    drop_first : bool, default False
        Whether to get k-1 dummies out of k categorical levels by removing the
        first level.

        .. versionadded:: 0.18.0
```

```
dtype : dtype, default np.uint8
    Data type for new columns. Only a single dtype is allowed.

    .. versionadded:: 0.23.0

Returns
-------
dummies : DataFrame

See Also
--------
Series.str.get_dummies

Examples
--------
>>> s = pd.Series(list('abca'))

>>> pd.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0

>>> s1 = ['a', 'b', np.nan]

>>> pd.get_dummies(s1)
   a  b
0  1  0
1  0  1
2  0  0

>>> pd.get_dummies(s1, dummy_na=True)
   a  b  NaN
0  1  0    0
1  0  1    0
2  0  0    1

>>> df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],
...                    'C': [1, 2, 3]})

>>> pd.get_dummies(df, prefix=['col1', 'col2'])
   C  col1_a  col1_b  col2_a  col2_b  col2_c
0  1       1       0       0       1       0
1  2       0       1       1       0       0
2  3       1       0       0       0       1

>>> pd.get_dummies(pd.Series(list('abcaa')))
```

```
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0

>>> pd.get_dummies(pd.Series(list('abcaa')), drop_first=True)
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0

>>> pd.get_dummies(pd.Series(list('abc')), dtype=float)
     a    b    c
0  1.0  0.0  0.0
1  0.0  1.0  0.0
2  0.0  0.0  1.0
```

6.3.3 Variables indicatrices Générer des indicatrices des modalités ou *dummy variables*.

```
In [111]: dummies = pd.get_dummies(dfs['key'], prefix='key')
          df_with_dummy = dfs[['data1']].join(dummies)
          df_with_dummy

Out[111]:    data1  key_a  key_b  key_c
          0      0      0      1      0
          1      1      0      1      0
          2      2      1      0      0
          3      3      0      0      1
          4      4      1      0      0
          5      5      0      1      0
```

6.3.4 Permutation et tirage aléatoires Permutation aléatoire:

```
In [112]: dfs = pd.DataFrame(np.arange(5 * 4).reshape(5, 4))
          sampler = np.random.permutation(5)
          sampler
          dfs
          dfs.take(sampler)

Out[112]:     0   1   2   3
          1   4   5   6   7
          3  12  13  14  15
```

```
4  16  17  18  19
0   0   1   2   3
2   8   9  10  11
```

Tirage aléatoire avec remplacement ou *bootstrap*.

```
In [113]: bag = np.array([5, 7, -1, 6, 4])
          sampler = np.random.randint(0, len(bag), size=10)
          draws = bag.take(sampler)
          draws

Out[113]: array([-1,  7,  7,  7, -1,  6,  7,  6,  5,  5])
```

6.3.5 Transformations, opérations Les opérations arithmétiques entre `Series` et `DataFrame` sont possibles au même titre qu'entre `array`. Si les index ne correspondent pas, des valeurs manquantes (NAN) sont créées à moins d'utiliser des méthodes d'arithmétique `flexible` (`add, sub, div, mul`) autorisant la complétion par une valeur par défaut, généralement 0.

Une fonction quelconque (`lambda`) peut être appliquée avec une même commande qu'`apply` de R.

```
In [114]: # la table de données
          frame = pd.DataFrame(np.random.randn(4,3),
              columns=list("bde"),
              index=["Utah", "Ohio", "Texas", "Oregon"])
          # une fonction
          f = lambda x: x.max() - x.min()
          frame.apply(f, axis=1)

Out[114]: Utah      2.146172
          Ohio      2.447774
          Texas     2.417317
          Oregon    1.028467
          dtype: float64
```

6.3.6 Tri et rangs Trier une table selon les valeurs d'une variable ou d'un index.

```
In [115]: frame = pd.DataFrame(np.arange(8).reshape((2,4)),
                    index=["three", "one"],
                    columns=["d", "a", "b", "c"])
          frame.sort_index()

Out[115]:        d  a  b  c
          one    4  5  6  7
          three  0  1  2  3

In [116]: frame.sort_index(axis=1)

Out[116]:        a  b  c  d
          three  1  2  3  0
          one    5  6  7  4
```

```
In [117]: frame.sort_index(axis=1, ascending=False)

Out[117]:        d  c  b  a
          three  0  3  2  1
          one    4  7  6  5

In [118]: frame.sort_values(by="b")

Out[118]:        d  a  b  c
          three  0  1  2  3
          one    4  5  6  7
```

La commande `rank` remplace les valeurs par leur rang dans l'ordre des lignes ou des colonnes.

```
In [119]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2],
              "a": [0, 1, 0, 1],"c": [-2, 5, 8, -2.5]})
          frame.rank(axis=1)

Out[119]:      b    a    c
          0  3.0  2.0  1.0
          1  3.0  1.0  2.0
          2  1.0  2.0  3.0
          3  3.0  2.0  1.0

In [120]: frame.rank(axis=0)

Out[120]:      b    a    c
          0  3.0  1.5  2.0
          1  4.0  3.5  3.0
          2  1.0  1.5  4.0
          3  2.0  3.5  1.0
```

2.6.4 6.4 Statistiques descriptives élémentaires

Continuer l'étude des données sur le naufrage du Titanic. Les commandes ci-dessous permettent des premiers diagnostics sur la qualité des données. #### 6.4.1 Description univariée

```
In [121]: df.dtypes

Out[121]: Surv      category
          Classe    category
          Genre     category
          Age        float64
          Prix       float64
          Port      category
          AgeQ      category
          PrixQ     category
          dtype: object

In [122]: df.describe()
```

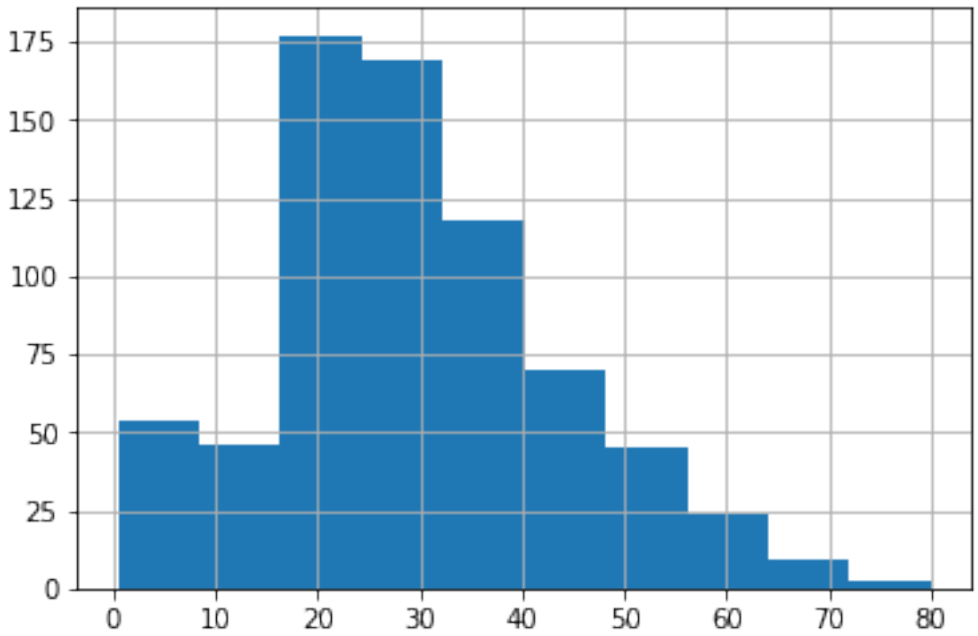
```
Out[122]:              Age        Prix
          count  714.000000  891.000000
          mean    29.699118   32.204208
          std     14.526497   49.693429
          min      0.420000    0.000000
          25%     20.125000    7.910400
          50%     28.000000   14.454200
          75%     38.000000   31.000000
          max     80.000000  512.329200
```

```
In [123]: df.head()
```

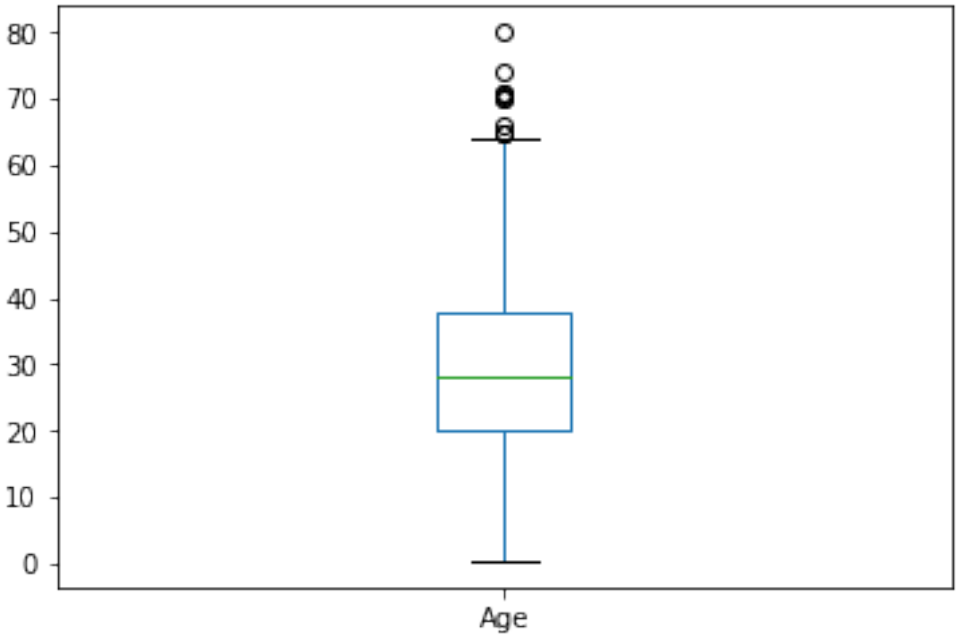
```
Out[123]:    Surv Classe Genre   Age     Prix Port AgeQ PrixQ
          0  Vnon    Cl3  Gmas  22.0   7.2500   Ps  Ag1   Pr1
          1  Voui    Cl1  Gfem  38.0  71.2833   Pc  Ag3   Pr3
          2  Voui    Cl3  Gfem  26.0   7.9250   Ps  Ag2   Pr1
          3  Voui    Cl1  Gfem  35.0  53.1000   Ps  Ag3   Pr3
          4  Vnon    Cl3  Gmas  35.0   8.0500   Ps  Ag3   Pr1
```

```
In [124]: import matplotlib.pyplot as plt
          %matplotlib inline
```

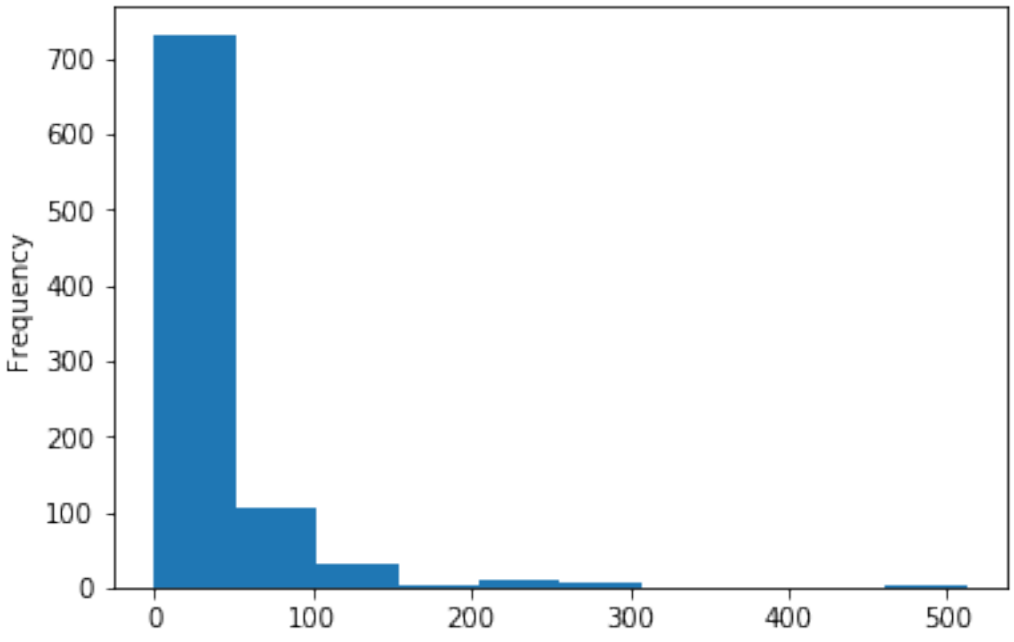
```
In [125]: df["Age"].hist()
          plt.show()
```




```
In [126]: df["Age"].plot(kind="box")
          plt.show()
```



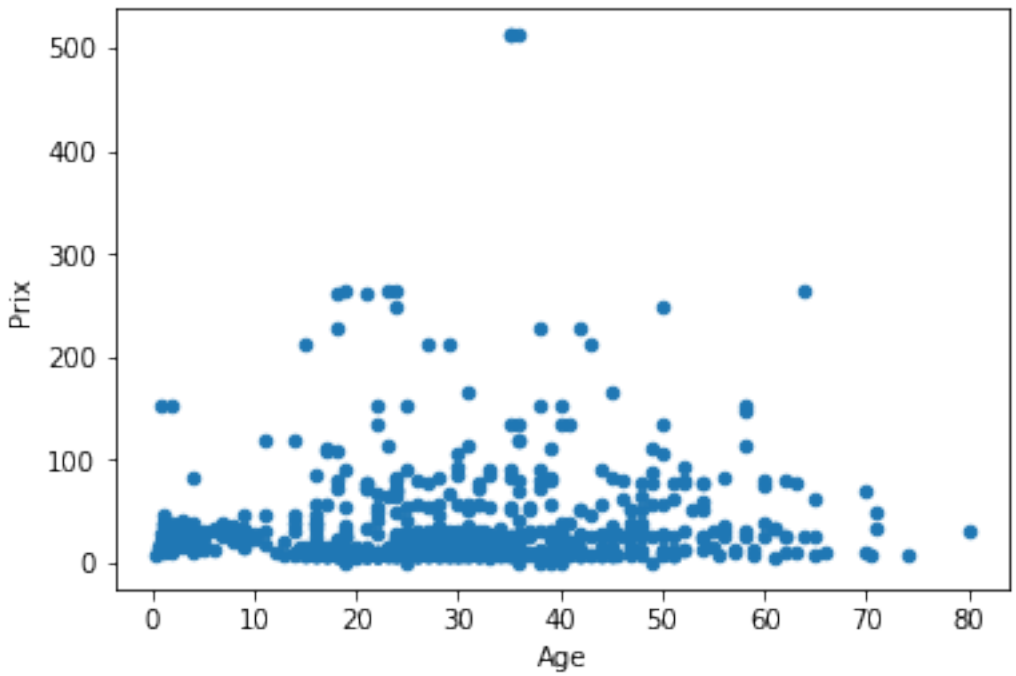

```
In [127]: df["Prix"].plot(kind="hist")
          plt.show()
```



```
In [128]: # qualitatif
          df["Surv"].value_counts()

Out[128]: Vnon    549
          Voui    342
          Name: Surv, dtype: int64

In [129]: df["Classe"].value_counts()

Out[129]: Cl3    491
          Cl1    216
          Cl2    184
          Name: Classe, dtype: int64

In [130]: df["Genre"].value_counts()

Out[130]: Gmas    577
          Gfem    314
          Name: Genre, dtype: int64

In [131]: df["Port"].value_counts()

Out[131]: Ps    644
          Pc    168
          Pq     77
          Name: Port, dtype: int64
```

6.4.2 Description bivariée

```
In [132]: df.plot(kind="scatter",x="Age",y="Prix")
          plt.show()
```

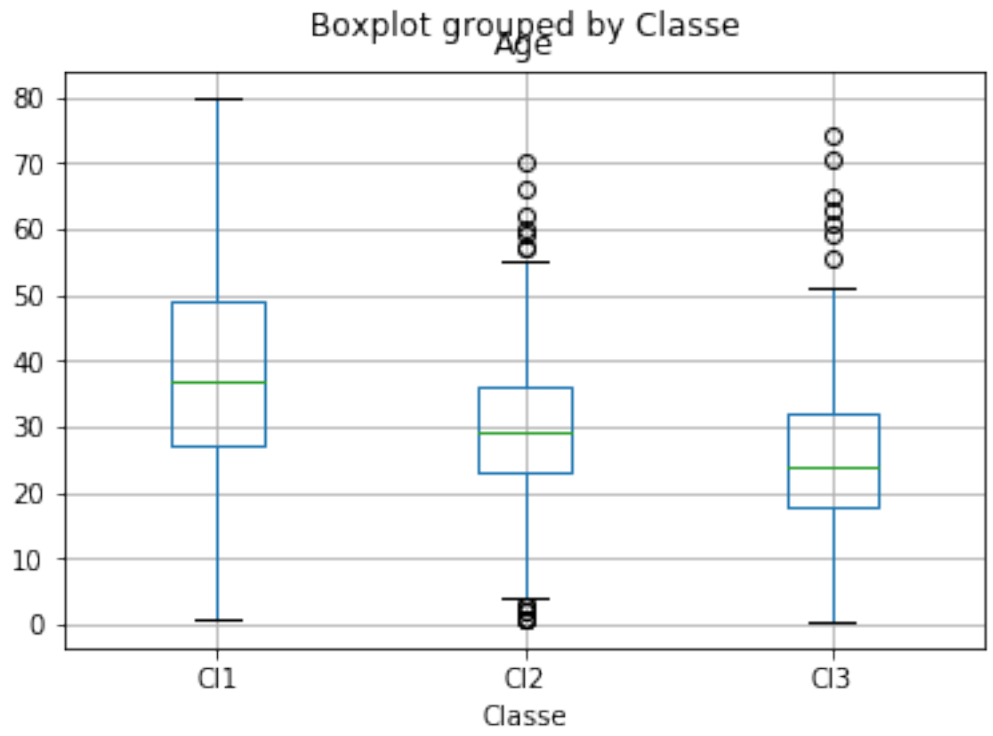


```
In [133]: # afficher une sélection
          df[df["Age"]>60][["Genre","Classe","Age","Surv"]]
```

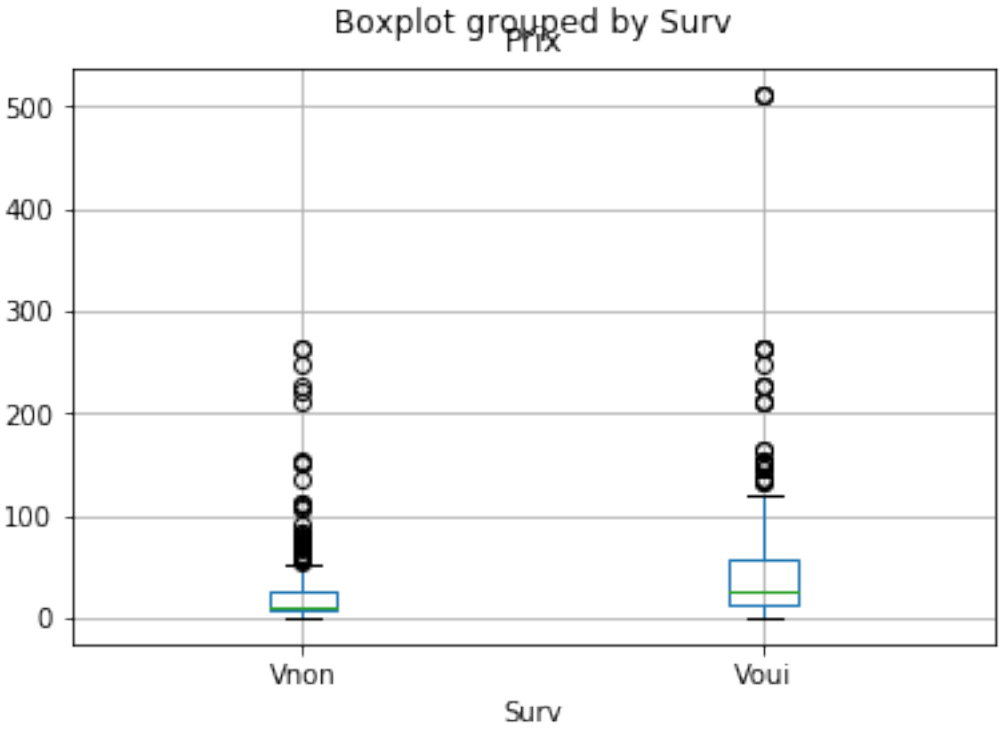
Out[133]:

	Genre	Classe	Age	Surv
33	Gmas	Cl2	66.0	Vnon
54	Gmas	Cl1	65.0	Vnon
96	Gmas	Cl1	71.0	Vnon
116	Gmas	Cl3	70.5	Vnon
170	Gmas	Cl1	61.0	Vnon
252	Gmas	Cl1	62.0	Vnon
275	Gfem	Cl1	63.0	Voui
280	Gmas	Cl3	65.0	Vnon
326	Gmas	Cl3	61.0	Vnon
438	Gmas	Cl1	64.0	Vnon
456	Gmas	Cl1	65.0	Vnon
483	Gfem	Cl3	63.0	Voui
493	Gmas	Cl1	71.0	Vnon
545	Gmas	Cl1	64.0	Vnon
555	Gmas	Cl1	62.0	Vnon
570	Gmas	Cl2	62.0	Voui
625	Gmas	Cl1	61.0	Vnon
630	Gmas	Cl1	80.0	Voui
672	Gmas	Cl2	70.0	Vnon
745	Gmas	Cl1	70.0	Vnon
829	Gfem	Cl1	62.0	Voui
851	Gmas	Cl3	74.0	Vnon

```
In [134]: df.boxplot(column="Age",by="Classe")
          plt.show()
```



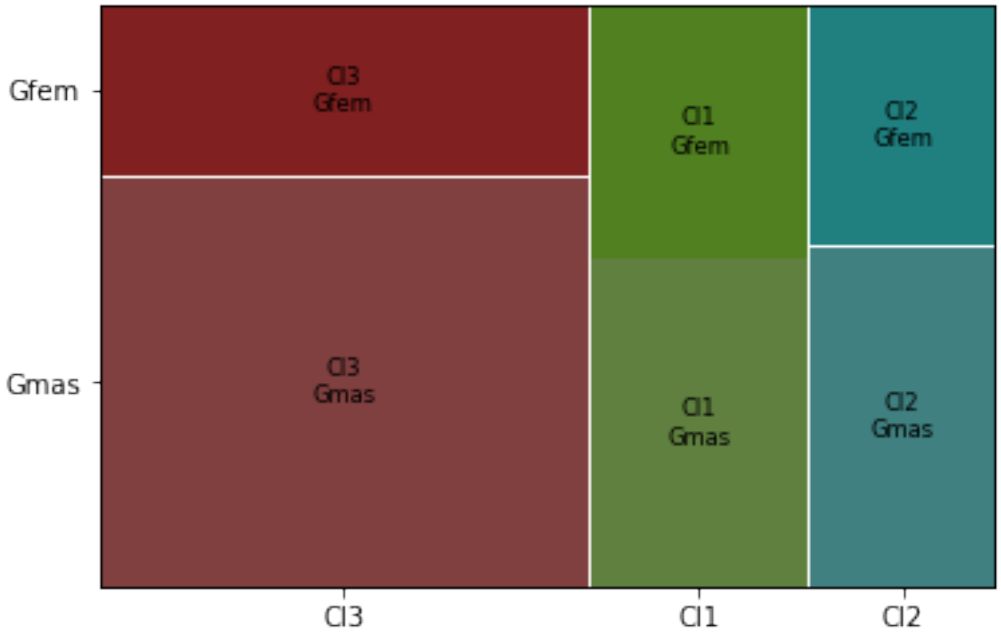

```
In [135]: df.boxplot(column="Prix",by="Surv")
          plt.show()
```



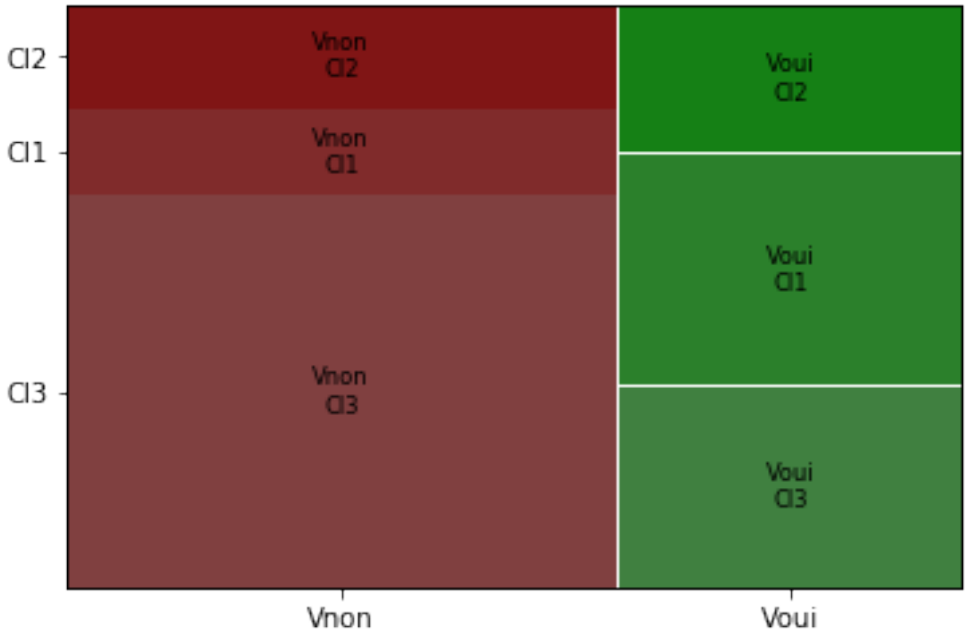
```
In [136]: # table de contingence
          table=pd.crosstab(df["Surv"],df["Classe"])
          print(table)
```

```
Classe  Cl1  Cl2  Cl3
Surv
Vnon     80   97  372
Voui    136   87  119
```

```
In [137]: # Mosaic plot
          from statsmodels.graphics.mosaicplot import mosaic
          mosaic(df,["Classe","Genre"])
          plt.show()
```




```
In [138]: mosaic(df,["Surv","Classe"])
          plt.show()
```



6.4.3 Imputation de données manquantes La gestion des données manquantes est souvent un point délicat. De nombreuses stratégies ont été élaborées, et nous ne décrivons ici que les plus élémentaires à mettre en oeuvre avec `pandas`.

Il est ainsi facile de supprimer toutes les observations présentant des données manquantes lorsque celles-ci sont peu nombreuses et majoritairement regroupées sur certaines lignes ou colonnes.

`df = df.dropna(axis=0) df = df.dropna(axis=1)`

`Pandas` permet également de faire le choix pour une variable qualitative de considérer np.nan' comme une modalité spécifique ou d'ignorer l'observation correspondante.

Autres stratégies: * Cas quantitatif: une valeur manquante est imputée par la moyenne ou la médiane. * Cas d'une série chronologique: imputation par la valeur précédente ou suivante ou par interpolation linéaire, polynomiale ou encore lissage spline. * Cas qualitatif: modalité la plus fréquente ou répartition aléatoire selon les fréquences observées des modalités.

La variable âge contient de nombreuses données manquantes. La fonction `fillna` présente plusieurs options d'imputation.

```
In [139]: # Remplacement par la médiane d'une variable quantitative
          df=df.fillna(df.median())
          df.describe()
```

```
Out[139]:              Age        Prix
          count  891.000000  891.000000
          mean    29.361582   32.204208
          std     13.019697   49.693429
          min      0.420000    0.000000
```

```
          25%     22.000000    7.910400
          50%     28.000000   14.454200
          75%     35.000000   31.000000
          max     80.000000  512.329200
```

```
In [140]: # par la modalité "médiane" de AgeQ
          df.info()
          df.AgeQ=df["AgeQ"].fillna("Ag2")
          # par le port le plus fréquent
          df["Port"].value_counts()
          df.Port=df["Port"].fillna("Ps")
          df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 8 columns):
Surv      891 non-null category
Classe    891 non-null category
Genre     891 non-null category
Age       891 non-null float64
Prix      891 non-null float64
Port      889 non-null category
AgeQ      714 non-null category
PrixQ     891 non-null category
dtypes: category(6), float64(2)
memory usage: 19.8 KB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 8 columns):
Surv      891 non-null category
Classe    891 non-null category
Genre     891 non-null category
Age       891 non-null float64
Prix      891 non-null float64
Port      891 non-null category
AgeQ      891 non-null category
PrixQ     891 non-null category
dtypes: category(6), float64(2)
memory usage: 19.8 KB
```

Ces imputations sont pour le moins très rudimentaires et d'autres sont à privilégier pour des modélisations plus soignées…

2.6.5 6.5 Manipuler des tables de données

6.5.1 Jointure Il s'agit de "jointer" deux tables partageant la même clef ou encore de concaténer horizontalement les lignes en faisant correspondre les valeurs d'une variable clef qui peuvent ne pas être uniques.

```
In [141]: # tables
          df1 = pd.DataFrame({"key": ["b", "b", "a", "c",
               "a","a", "b"],"data1": range(7)})
          df2 = pd.DataFrame({"key": ["a", "b", "d"],
              "data2": range(3)})
          pd.merge(df1,df2,on="key")

Out[141]:   key  data1  data2
          0   b      0      1
          1   b      1      1
          2   b      6      1
          3   a      2      0
          4   a      4      0
          5   a      5      0
```

La gestion des clefs manquantes est en option: entre autres, ne pas introduire de ligne (ci-dessus), insérer des valeurs manquantes ci-dessous.

```
In [142]: # valeurs manquantes
          pd.merge(df1,df2,on="key", how="outer")

Out[142]:   key  data1  data2
          0   b    0.0    1.0
          1   b    1.0    1.0
          2   b    6.0    1.0
          3   a    2.0    0.0
          4   a    4.0    0.0
          5   a    5.0    0.0
          6   c    3.0    NaN
          7   d    NaN    2.0
```

6.5.2 Concaténation selon un axe Concaténation verticale (axis=0) ou horizontales (axis=1) de tables. La concaténation horizontale est similaire à la jointure (option `outer`).

```
In [143]: # tables
          df1 = pd.DataFrame({"key": ["b", "b", "a", "c",
              "a", "a", "b"],"var": range(7)})
          df2 = pd.DataFrame({"key": ["a", "b", "d"],
              "var": range(3)})
          # concaténation verticales
          pd.concat([df1,df2],axis=0)

Out[143]:   key  var
          0   b    0
          1   b    1
          2   a    2
          3   c    3
          4   a    4
          5   a    5
```

```
6   b    6
0   a    0
1   b    1
2   d    2
```

```
In [144]: # concaténation horizontale
          pd.concat([df1,df2],axis=1)
```

```
Out[144]:   key  var  key  var
          0   b    0    a  0.0
          1   b    1    b  1.0
          2   a    2    d  2.0
          3   c    3  NaN  NaN
          4   a    4  NaN  NaN
          5   a    5  NaN  NaN
          6   b    6  NaN  NaN
```