Python et DataScience

CONSULTING BSTORM

NumPy

Introduction

NumPy (**Num**erical **Py**thon) est une librairie qui permet de charger, stocker et manipuler de manière efficace des données avec Python.

Les données peuvent prendre **beaucoup de formes** (documents texte ,des images, du son, des mesures topographiques, sismiques, etc.)

Malgré cette hétérogénéité, nous pouvons souvent considérer les données comme des **tableaux de chiffres**. E.g.: les images (noir et blanc) sont des tableaux en deux dimensions avec des pixels et une valeur associée comprise entre 0 et 255.

Les tableaux (array) dans numpy ressemblent aux listes de Python mais ils sont beaucoup plus optimisés.

Numpy est une librairie **fondamentale** pour traiter des données en Python.

Instructions de bases

```
import numpy
numpy.__version__
import numpy as np

np.<TAB>

np?
```

Typage des données en Python

Comment numpy améliore-t-il la gestion des données en Python ?

Python est un langage à typage dynamique >< typage statique comme C ou Java

```c
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

```python
# Python code
result = 0
for i in range(100):
    result += i
```

```c
/* C code */
int x = 4;
x = "four";  // FAILS
```

```python
# Python code
x = 4
x = "four"
```

Typage des données en Python

Ceci a pour conséquence que les variables en Python stockent plus que leur valeur

En effet, en plus de la valeur associée à la variable, il faut stocker le type de chaque valeur

A quoi ressemble un integer en Python ?

L'implémentation de python est écrite en C. Tous les objets Python ne sont rien d'autres que des structures C bien déguisées.

Par exemple, si on définit un int en Python comme suit: x = 10000, x n'est pas un simple integer.

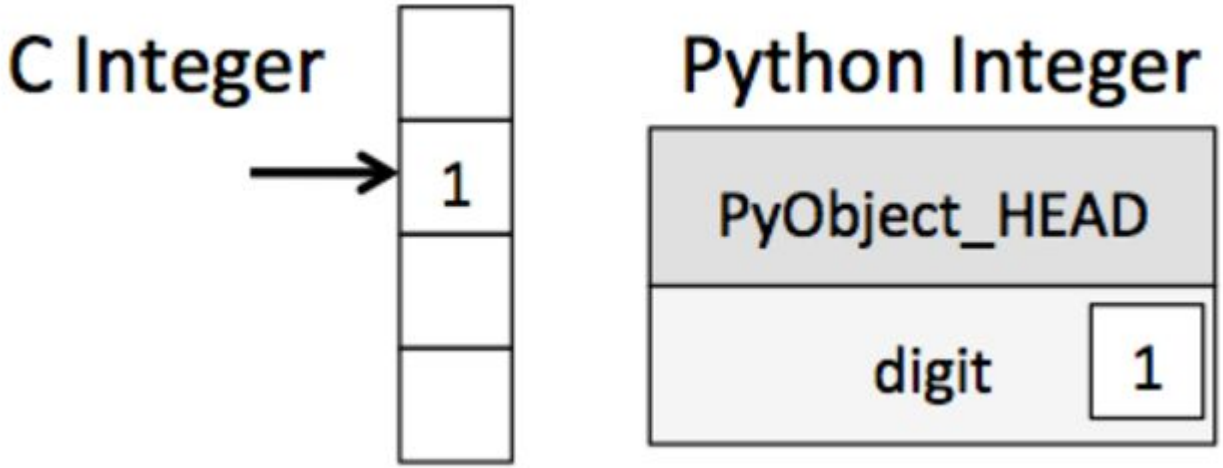
Un seul integer contient en réalité quatre parties:

- ob_refcnt, qui permet de gérer l'allocation de mémoire
- ob_type, qui encode le type de la variable
- ob_size, qui spécifie la taille des membres
- ob_digit, qui contient la valeur de la variable

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

A quoi ressemble un integer en Python ?

Donc contrairement au langage C, les integers en python possèdent une en-tête



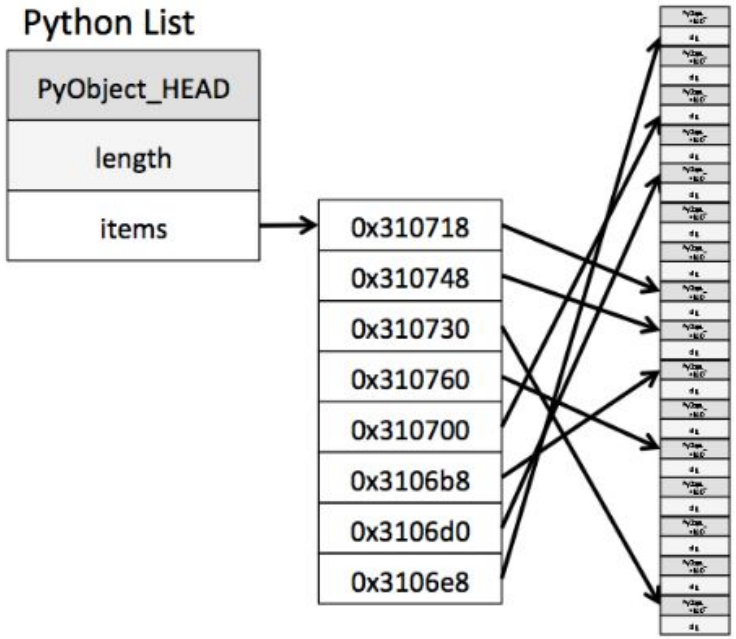
Qu’est-ce qu’une liste en Python ?

Comment Python stockent-ils plusieurs objets dans une liste ?

```
L = list(range(10))
type(L[0])
L2 = [str(c) for c in L]
```

On peut également créer des listes hétérogènes

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
```

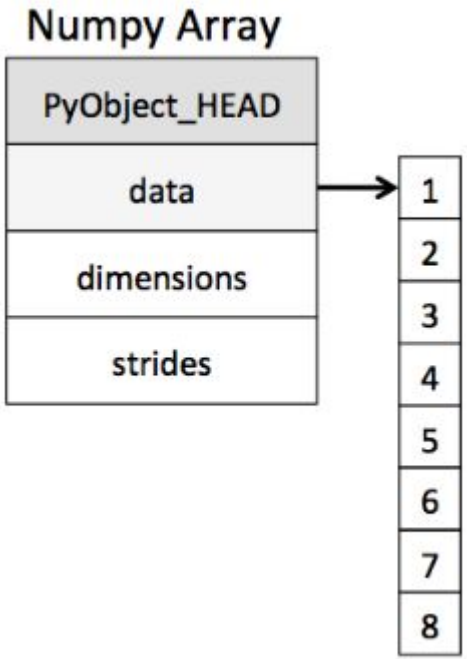


Qu’est-ce qu’un Numpy Array ?

Quand toutes les variables sont d’un même type, une partie de l’information est **redondante**

La différence entre une liste Python à typage dynamique et un numpy array (illustrée ci contre)

Les numpy array perdent en flexibilité mais gagnent en taille et donc en efficacité



Créer un NumPy Array

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

Les numpy array ne peuvent contenir qu'**un seul type**. Si ce n'est pas le cas, numpy va tenter une conversion (ici,les integers sont convertis en float):

```
np.array([3.14, 4, 2, 3])
```

On peut spécifier le type de données lorsque l'on crée un array avec le mot clef 'dtype':

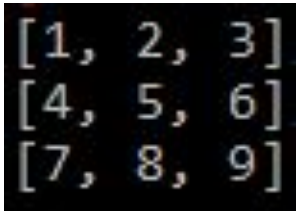
```
np.array([1, 2, 3, 4], dtype='float32')
```

Contrairement aux listes Python, les numpy array peuvent être explicitement multidimensionnels

```
# Des listes imbriquées donnent un numpy array en plusieurs dimensions
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

Exercice

1. Créer une matrice 3x3



Créer des numpy array from scratch

```
# Permet de créer un array remplit de 10 integers avec la valeur 0
np.zeros(10, dtype=int)
# Crée una matrice 3x5 remplie de 1 en float
np.ones((3, 5), dtype=float)
# Crée une matrice 3x5 remplie de 3,14
np.full((3, 5), 3.14)
# Crée un numpy array avec une séquence linéaire
# Commence à 0 jusqu'à 20 par step de 2
# Similaire à la fonction range
np.arange(0, 20, 2)
# Crée une matrice avec 5 nombres équidistants entre 0 et 1
np.linspace(0, 1, 5)
# Crée une matrice 3x3 de nombres aléatoires
np.random.random((3, 3))  // Comment faire pour voir toutes les distributions proposées par NumPy?
# Crée une matrice 3x3 suivant une distribution normale suivant les paramètres:
# d'une moyenne de 0 et d'écart type de 1
np.random.normal(0, 1, (3, 3))
```

Data Types avec NumPy

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Attributs des NumPy Array

```
np.random.seed(0)  # seed pour la reproductibilité

x1 = np.random.randint(10, size=6)  # array 1D
x2 = np.random.randint(10, size=(3, 4))  # array 2D
x3 = np.random.randint(10, size=(3, 4, 5))  # array 3D
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:", x3.dtype)
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

Array Indexing: accéder à un seul élément

Positive indexing:

`array_name[i]` ⇒ `0 <= i < array_name.shape[0]`

Negative indexing:

`array_name[i]` ⇒ `-array_name.shape[0] <= i <= -1`

Array en 2D:

```
x2[0, 0]
```

Array Slicing

```
x[start:stop:step]      # de start à stop (non inclu) par step
```

Valeur par défaut:

start = 0

stop = size

step = 1

(si step est négatif, alors les valeurs par défaut de stop et start sont inversées)

Array Slicing - 1 dimension

Exercice

1. Créer un tableau d'entier allant de 0 à 9
2. Récupérer les 5 premiers éléments
3. Récupérer les éléments après le 5
4. Récupérer les éléments de 4 à 7
5. Récupérer les paires
6. Récupérer les impaires (sans zéro)
7. Les trois derniers éléments
8. Les éléments par ordre inverse
9. Les 5 premiers en ordre inverse
10. Tout sélectionner avec la syntaxe de slicing

Array Slicing - 2 dimensions

Exercice

1. Créer la matrice suivante

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

2. Récupérer les sous-matrices suivantes

```
[9]
[6]
[3]
```

```
[8, 7]
[5, 4]
[2, 1]
```

```
[9, 8]
[6, 5]
```

3. Inverser les lignes

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

```
[9, 6, 3]
```

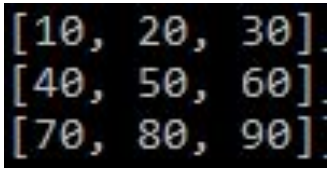
Vues

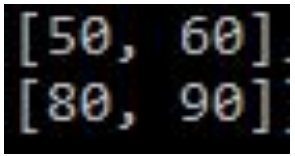
Une chose importante à savoir: le slicing retourne une vue et non une copie des valeurs

D'ailleurs, contrairement à python, le slicing sur des listes en python retourne des copies. Un petit exercice pour bien comprendre:

Exercice:

1. Créer la matrice suivante dans la variable x1:
2. Extraire dans la variable x2
3. Remplacer le 50 de x2 par 99
4. Réafficher x1



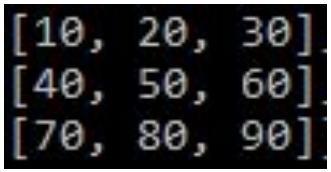


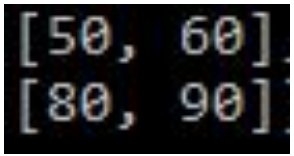
Copy

Utiliser la méthode copy() sur un NumPy Array afin de

Exercice:

1. Créer la matrice suivante dans la variable x1:
2. Extraire **une copie** de dans la variable x2
3. Remplacer le 50 de x2 par 99
4. Réafficher x1





Reshape Array

`ndarray.reshape(tuple) #no-copy`

Exemple

```
np.arange(1, 10).reshape((3, 3))
```

Transpose

x.T

Transformer vecteur en matrice colonne/ligne

```
x = np.array([1, 2, 3])
# vecteur ligne via reshape
x.reshape((1, 3))
# vecteur ligne via newaxis
x[np.newaxis, :]
x.reshape((3, 1))
# vecteur colonne via newaxis
x[:, np.newaxis]
```

np.concatenate

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

RESULT

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

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])

# concaténation sur le premier axis
np.concatenate([grid, grid])

# concaténation sur le second axis
np.concatenate([grid, grid], axis=1)
```

np.Xstack

np.hstack

```
# combine horizontalement les arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

np.vstack

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

# combine verticalement les arrays
np.vstack([x, grid])
```

np.dstack (3ème dimension (profondeur))

np.split

SPLIT:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

RESULT

```
[1 2 3] [99 99] [3 2 1]
```

VSPLIT:

```
grid = np.arange(16).reshape((4, 4))
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

RESULT

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

HSPLIT:

```
left, right =
np.hsplit(grid, [2])
print(left)
print(right)
```

RESULT

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Opérateurs arithmétiques et fonctions binaires

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Fonction unitaire

np.abs() #

np.sin(), np.cos(), np.tan()

np.exp(x) # e^x

np.exp2(x) # 2^x

np.log(x) #ln(x)

np.log2(x) #$\log_2(x)$

out parameter

Pour éviter de créer un array temporaire

```
x = np.arange(5)
np.multiply(x, 10, out=y)
```

ou

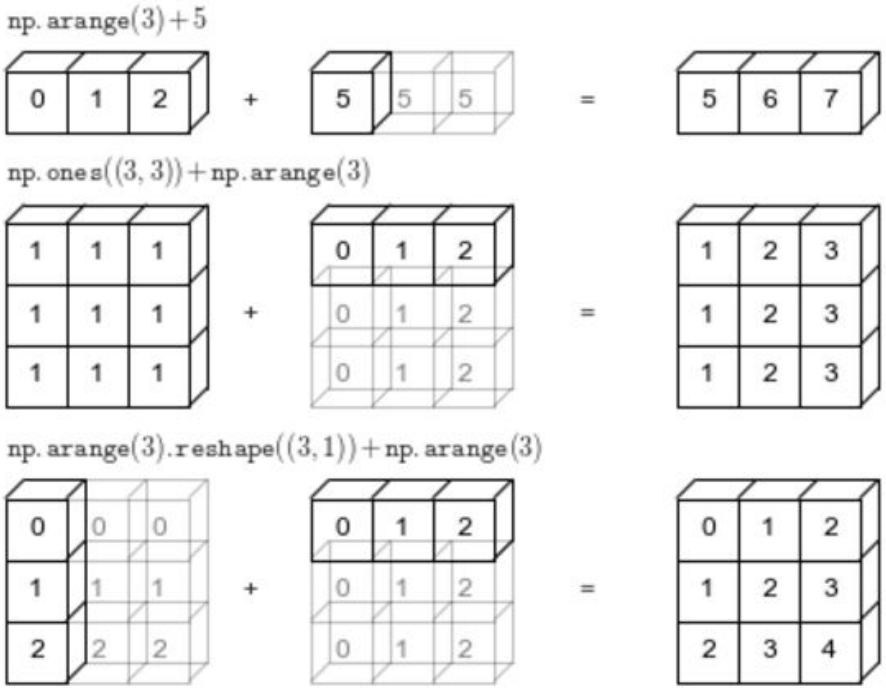
```
y = np.zeros(10)
np.power(2, x, out=y[::2]) # plus rapide que y[::2] = 2 ** x
print(y)
```

Fonction d'agrégation

On peut spécifier l'axis afin d' agréger selon tel ou tel axis

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute mean of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

Broadcasting



le broadcasting de numpy est une liste de règles strictes qui définit l'interaction entre deux arrays:

- **Règle 1:** si deux arrays n'ont pas le même nombre de dimension, on rajoute à celui avec le moins de dimension une dimension
- **Règle 2:** si deux arrays différent en dimension, on étend celui avec une seule dimension de manière à obtenir un nouvel array de même dimension
- **Règle 3:** Si les arrays n'ont pas les mêmes dimensions et qu'il y a plus d'une dimension, une erreur est générée.

Manipulation des données avec Pandas

Import du package

```
import pandas as pd
```

Structure de données avec Pandas

Series: array 1D indexé

DataFrame: Ressemble à un array en 2D mais avec plus de flexibilité. On peut par exemple ajouter des noms aux colonnes. Un DataFrame est au minimum constitué de deux Series (deux colonnes)

Index: un ou plusieurs groupes d'index pour localiser les données

Pandas Series

Construction:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
```

Simple surcouche sur base d'un numpy array:

```
data.index
data.values
```

Les indices peuvent être explicites (contrairement à numpy)

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
```

Pandas DataFrame

```
states = pd.DataFrame({'population': population, 'area': area})

states['colIndex']['rowIndex']
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Construire un DataFrame

D'une Series

```
pd.DataFrame(population, columns=['population'])
```

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

D'une liste de dictionnaires

```
pd.DataFrame([{'a': i, 'b': 2 * i} for i in range(3)])
```

D'un dictionnaire de Series

```
pd.DataFrame({'population': population, 'area': area})
```

D'un numpy array 2d

```
pd.DataFrame(np.random.rand(3, 2), columns=['foo', 'bar'], index=['a', 'b', 'c'])
```

Depuis un fichier csv

```
pd.read_csv(PATH)
```

Index

- L'index est un array immuable
  - On peut utiliser l'indexation et le slicing
  - On ne peut pas le modifier sans assignation directe
- L'index est un ensemble ordonné
  - ```
    indA = pd.Index([1, 3, 5, 7, 9])
    indB = pd.Index([2, 3, 5, 7, 11])
    ```
  - ```
    indA & indB  # intersection
    ```
  - ```
    indA | indB  # union
    ```
  - ```
    indA ^ indB  # symmetric difference >< intersection
    ```

Indexation et sélection

```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
data['b'] # 0.5
'a' in data # true
data.keys() # retourne l'indice des objets
list(data.items()) # retourne une liste de lignes sous forme de tuple
data['e'] = 1.25 # assigner une valeur à sa localisation
data['a':'c'] # slicing explicite
data[0:2] # slicing implicite avec integer
data[(data > 0.3) & (data < 0.8)] # sélection d'éléments avec masque
data[['a', 'e']] # multi indexation explicite
```

Confusion entre implicite et explicite

Que se passe-t-il ?

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data[1] # index explicite avec l'indexation
data[1:3] #index implicite avec le slicing
```

loc ⇒ Utilise des index explicites

```
data.loc[1] # 'a'
data.loc[1:3]
```

iloc ⇒ Utilise des index implicites

```
data.iloc[1]
data.iloc[1:3]
```

Ajouter une colonne

```
data['density'] = data['pop'] / data['area']
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

Opérations sur des DataFrames

- Toutes les fonctions numpy sont utilisables
- S'il manque des indices => NaN
  - ```
    A = pd.Series([2, 4, 6], index=[0, 1, 2])
    B = pd.Series([1, 3, 5], index=[1, 2, 3])
    A + B
    ```
  - ou bien: `A.add(B, fill_value=0)`

Python Operator	Pandas Method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()
    ```

Gestion des valeurs manquantes

```
df.isnull()
df.notnull()
df.dropna()
df.dropna(axis=1)
df.dropna(axis=1, how="all")
df.dropna(axis='rows', thresh=3)
# Permet de spécifier un minimum de valeurs non nulles
# afin de garder les lignes ou colonnes
df.fillna(0)
df.fillna(method='ffill', axis=1) # propagate the previous value forward
df.fillna(method='bfill', axis=0) # propagate the next values backward
```

Combiner des Datasets: Concat et Append

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])

pd.concat([ser1, ser2])
pd.concat([ser1, ser2], axis=1)
```
⇒ S'il y a des indices en double, il les double

```
pd.concat([x, y], verify_integrity=True)
```
⇒ S'il y a des indices en double, génère une erreur

```
pd.concat([x, y], ignore_index=True)
```
⇒ Ignore les indices

```
pd.concat([df5, df6], join='inner')
```
⇒ On ne prend en compte que les colonnes communes (à la place de remplir de nan)

```
pd.concat([df5, df6], join_axes=[df5.columns])
```
⇒ On ne prend en compte que les colonnes de df5

Combiner des Datasets: Merge

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.merge(df1, df2)
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Combiner des Datasets: Merge

```
# Spécifie la colonne à utiliser comme clef
pd.merge(df1, df2, on='employee')
```

```
pd.merge(df1, df3, left_on="employee", right_on="name")
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1, df3, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

Combiner des Datasets: Merge

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')

pd.merge(df1a, df2a, left_index=True, right_index=True)
```

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

	group	hire_date
employee		
Lisa	Engineering	2004
Bob	Accounting	2008
Jake	Engineering	2012
Sue	HR	2014

Combiner des Datasets: Merge

```
pd.merge(df6, df7, how='outer')

# inner, left, right join
```

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

Agrégation

.describe() => statistiques basiques

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Agrégation

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

Agrégation + GROUPBY

```
df.groupby('key').sum()
df.groupby('key').aggregate(['min', np.median, max])

La clef doit être une colonne (PAS UN INDEX)

Il faut utiliser le paramètre level pour groupby
selon l'index

Exemple: df.groupby(level=0).mean()
```

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

Visualisation avec Matplotlib

Import et Configuration

```
import matplotlib as mpl


plt.style.use('classic')
```

Afficher un graphique dans le shell

```
import matplotlib.pyplot as plt

import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--')

plt.show()
```

Sauver le graphique en image

```
fig.savefig('my_figure.png')
```

```
fig.canvas.get_supported_filetypes()
```

Matlab-like interface

```
plt.figure()  #crée un figure

# crée le premier graphique et l'indexe
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# crée le second graphique et l'indexe
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```

Line Plot

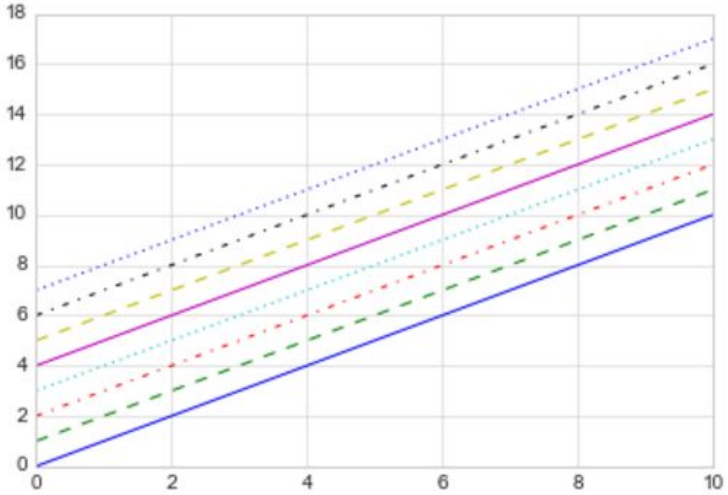
Couleur

```
plt.plot(x, np.sin(x - 0), color='blue')          # par nom
plt.plot(x, np.sin(x - 1), color='g')             # par code
plt.plot(x, np.sin(x - 2), color='0.75')          # noir et blanc (0-1)
plt.plot(x, np.sin(x - 3), color='#FFDD44')       # Hex code (RRGGBB [00-FF])
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RVB tuple (0-1)
plt.plot(x, np.sin(x - 5), color='chartreuse'); # Tous les noms de couleur
                                                # HTML sont supportés
```

Types de ligne

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

# Avec une syntaxe plus courte:
plt.plot(x, x + 4, linestyle='-')  # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':');  # dotted
```



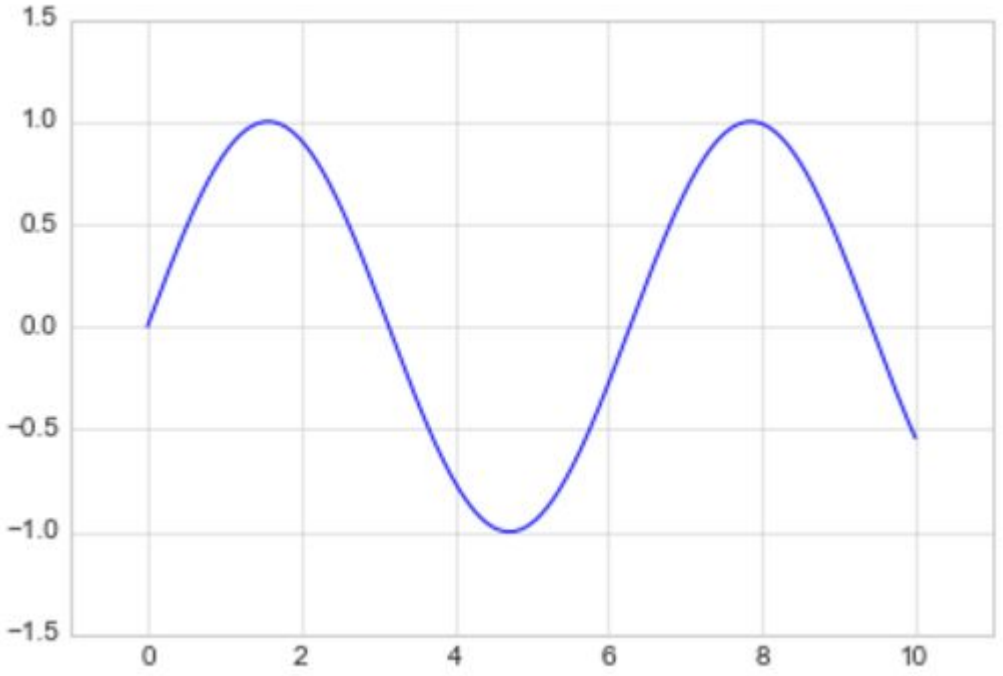
Ajuster les axis

```
plt.plot(x, np.sin(x))

plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```

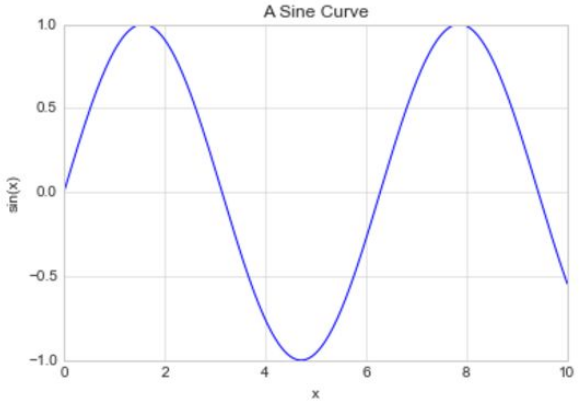
ou:

```
plt.axis([-1, 11, -1.5, 1.5]);
# [xmin, xmax, ymin, ymax]
```



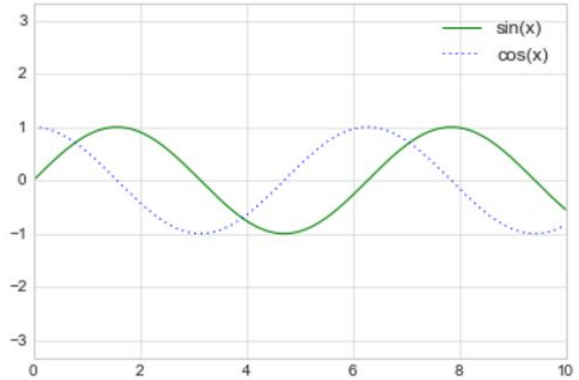
Labeling plot

```
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```




```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.axis('equal')

plt.legend();
```

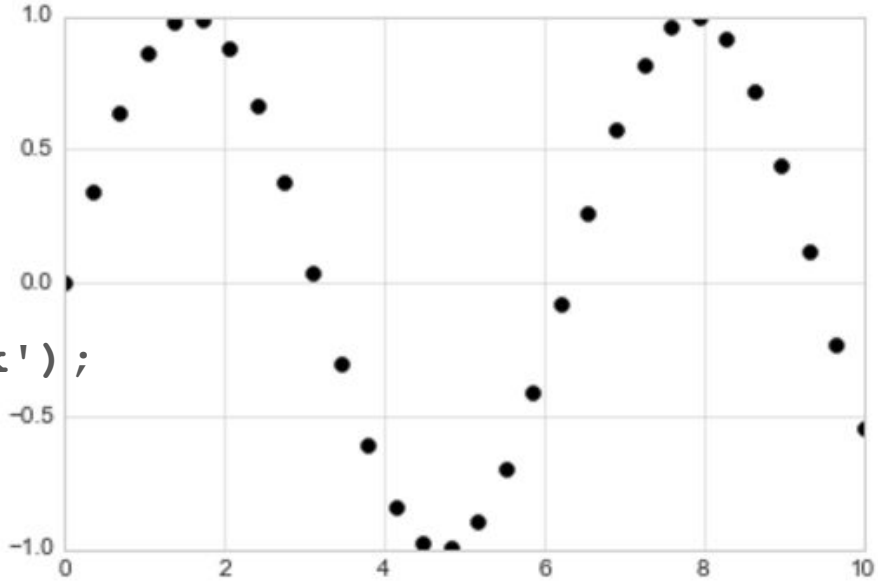


Scatter Plot

Scatter Plot

```
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```

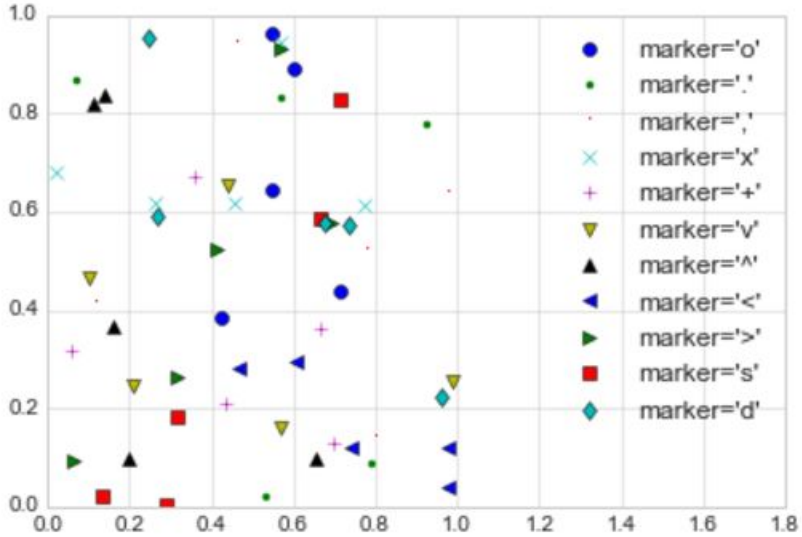


Changer les signes

```
plt.plot(rng.rand(5), rng.rand(5), 'o', label="marker='o'")

plt.plot(rng.rand(5), rng.rand(5), '+', label="marker='+'")

...
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

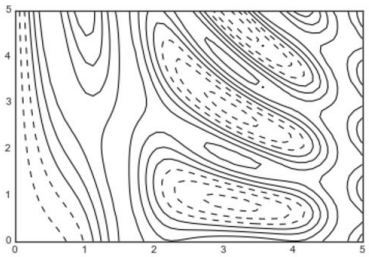


visualisations 3D

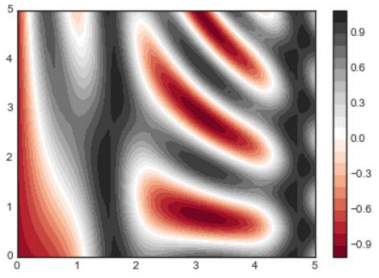
plt.contour

```
def f(x, y): return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

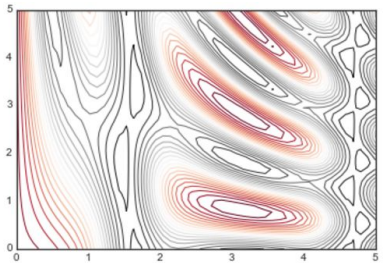
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.contour(X, Y, Z, colors='black');    #1
```



```
plt.contour(X, Y, Z, 20, cmap='RdGy');   #2
```

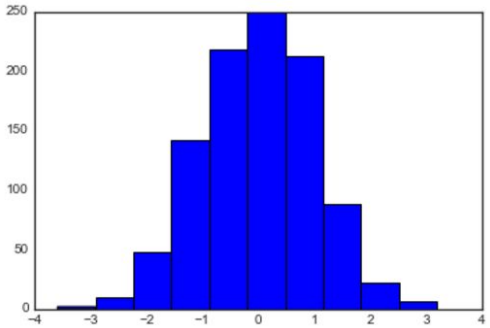


```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();                          #3
```

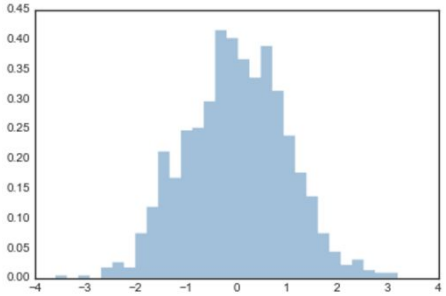


Histogrammes

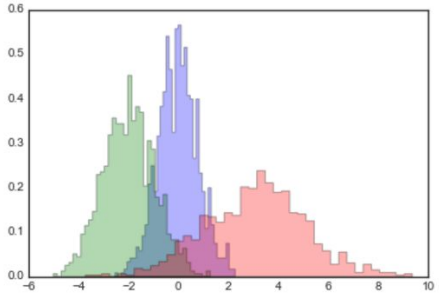
```
data = np.random.randn(1000)
plt.hist(data);
```




```
plt.hist(data, bins=30, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```



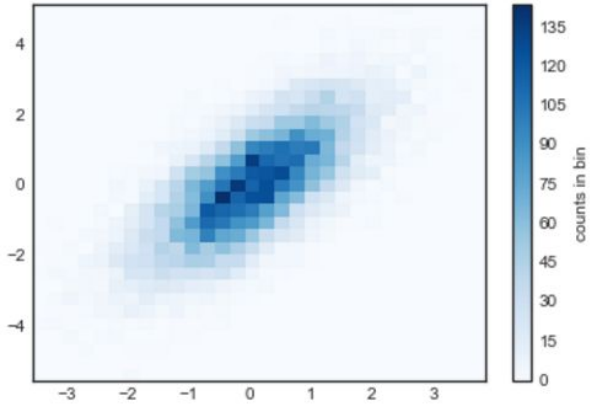

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)
kwargs = dict(histtype='stepfilled', alpha=0.3,
              normed=True, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```



histogramme 2D

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```



subplot

```
for i in range(1, 7):
    plt.subplot(2, 3, i)
    plt.text(0.5, 0.5, str((2, 3, i)),
             fontsize=18, ha='center')
```