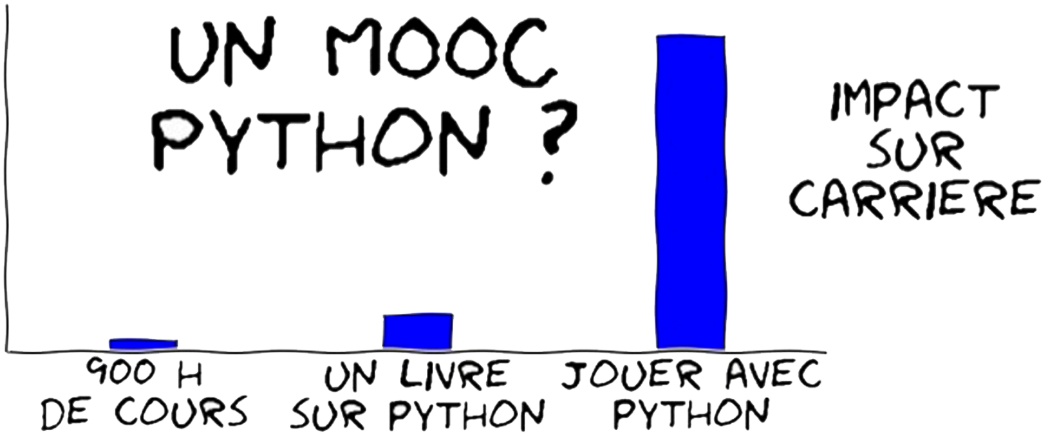
python™ 3

Des fondamentaux au concepts avancés du langage

Session 2 - 17 septembre 2018

Thierry Parmentelat Arnaud Legout



https ://www.fun-mooc.fr

Table des matières

Page

7 L'écosystème data science Python 1
- 7.1 Installations supplémentaires . . . 1
- 7.2 `numpy` en dimension 1 . . . 2
- 7.3 Type d'un tableau `numpy` . . . 6
- 7.4 Forme d'un tableau `numpy` . . . 9
- 7.5 Création de tableaux . . . 14
- 7.6 Le broadcasting . . . 19
- 7.7 Index et slices . . . 25
- 7.8 Slicing . . . 27
- 7.9 Opérations logiques . . . 32
- 7.10 Algèbre linéaire . . . 43
- 7.11 Indexation évoluée . . . 48
- 7.12 Divers . . . 57
- 7.13 Utilisation de la mémoire . . . 57
- 7.14 Types structurés pour les cellules . . . 60
- 7.15 Assemblages et découpages . . . 61
- 7.16 Exercice - niveau basique . . . 64
- 7.17 Exercice - niveau intermédiaire . . . 65
- 7.18 Exercice - niveau intermédiaire . . . 66
- 7.19 Exercice - niveau avancé . . . 68
- 7.20 Exercice - niveau intermédiaire . . . 69
- 7.21 Exercice - niveau intermédiaire . . . 70
- 7.22 Exercice - niveau intermédiaire . . . 71
- 7.23 La data science en général . . . 72
- 7.24 `Series` de `pandas` . . . 76
- 7.25 `DataFrame` de `pandas` . . . 85
- 7.26 Opération avancées en `pandas` . . . 107
- 7.27 Séries temporelles en `pandas` . . . 122
- 7.28 `matplotlib` - 2D . . . 123
- 7.29 `matplotlib` 3D . . . 131
- 7.30 Notebooks interactifs . . . 144
- 7.31 Animations interactives avec `matplotlib` . . . 153
- 7.32 Autres bibliothèques de visualisation . . . 158
- 7.33 Application à la transformée de Fourier . . . 165
- 7.34 Le théorème de Taylor illustré . . . 172
- 7.35 Coronavirus . . . 178

Chapitre 7

L'écosystème data science Python

7.1 w7-s01-c1-installation

Installations supplémentaires

7.1.1 Complément - niveau basique

Les outils que nous voyons cette semaine, bien que jouant un rôle majeur dans le succès de l'écosystème Python, ne font pas partie de la distribution standard. Cela signifie qu'il vous faut éventuellement procéder à des installations complémentaires sur votre ordinateur (évidemment vous pouvez utiliser les notebooks sans installation de votre part).

Comment savoir ?

Pour savoir si votre installation est idoine, vous devez pouvoir faire ceci dans votre interpréteur Python (par exemple, IPython) sans erreur :

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
import pandas as pd
```

Avec (ana)conda

Si vous avez installé votre Python avec conda, selon toute probabilité, toutes ces bibliothèques sont déjà accessibles pour vous. Vous n'avez rien à faire de particulier pour pouvoir faire tourner les exemples du cours sur votre ordinateur.

Distribution standard

Si vous avez installé Python à partir d'une distribution standard, vous pouvez utiliser `pip` comme ceci ; naturellement ceci doit être fait dans un terminal (sous Windows, cmd.exe avec les droits d'administrateur) et non pas dans l'interpréteur Python, ni dans IDLE :

```
$ pip3 install numpy matplotlib pandas
```

Debian/Ubuntu

Si vous utilisez Debian ou Ubuntu, et que vous avez déjà installé Python avec `apt-get`, la méthode préconisée sera :

```
$ apt-get install python3-numpy python3-matplotlib python3-pandas
```

Fedora

De manière similaire sur Fedora ou RHEL :

```
$ dnf install python3-numpy python3-matplotlib python3-pandas
```

7.2 w7-s02-c1-dimension1

numpy en dimension 1

7.2.1 Complément - niveau basique

Comme on l'a vu dans la vidéo, `numpy` est une bibliothèque qui offre un type supplémentaire par rapport aux types de base Python : le tableau, qui s'appelle en anglais `array` (en fait techniquement, `ndarray`, pour n-dimension array).

Bien que techniquement ce type ne fasse pas partie des types de base de Python, il est extrêmement puissant, et surtout beaucoup plus efficace que les types de base, dès lors qu'on manipule des données qui ont la bonne forme, ce qui est le cas dans un grand nombre de domaines.

Aussi, si vous utilisez une bibliothèque de calcul scientifique, la quasi totalité des objets que vous serez amenés à manipuler seront des tableaux `numpy`.

Dans cette première partie nous allons commencer avec des tableaux à une dimension, et voir comment les créer et les manipuler.

[1]:
```
import numpy as np
```

Création à partir de données

`np.array`

On peut créer un tableau numpy à partir d'une liste - ou plus généralement un itérable - avec la fonction `np.array` comme ceci :

[2]:
```
array = np.array([12, 25, 32, 55])
array
```

[2]:
```
array([12, 25, 32, 55])
```

Attention : une erreur commune au début consiste à faire ceci, qui ne marche pas :

[3]:
```
try:
    array = np.array(1, 2, 3, 4)
except Exception as e:
    print(f"OOPS, {type(e)}, {e}")
```

```
OOPS, <class 'TypeError'>, array() takes from 1 to 2 positional arguments b
  ut 4 were given
```

Ça marche aussi à partir d'un itérable :

[4]:
```
builtin_range = np.array(range(10))
builtin_range
```

[4]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Création d'intervalles

np.arange

Sauf que dans ce cas précis on préfèrera utiliser directement la méthode **arange** de **numpy** :

[5]:
```
numpy_range = np.arange(10)
numpy_range
```

[5]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Avec l'avantage qu'avec cette méthode on peut donner des bornes et un pas d'incrément qui ne sont pas entiers :

[6]:
```
numpy_range_f = np.arange(1.0, 2.0, 0.1)
numpy_range_f
```

[6]:
```
array([1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])
```

np.linspace

Aussi et surtout, lorsqu'on veut créer un intervalle dont on connaît les bornes, il est souvent plus facile d'utiliser **linspace**, qui crée un intervalle un peu comme **arange**, mais on lui précise un nombre de points plutôt qu'un pas :

[7]:
```
X = np.linspace(0., 10., 50)
X
```

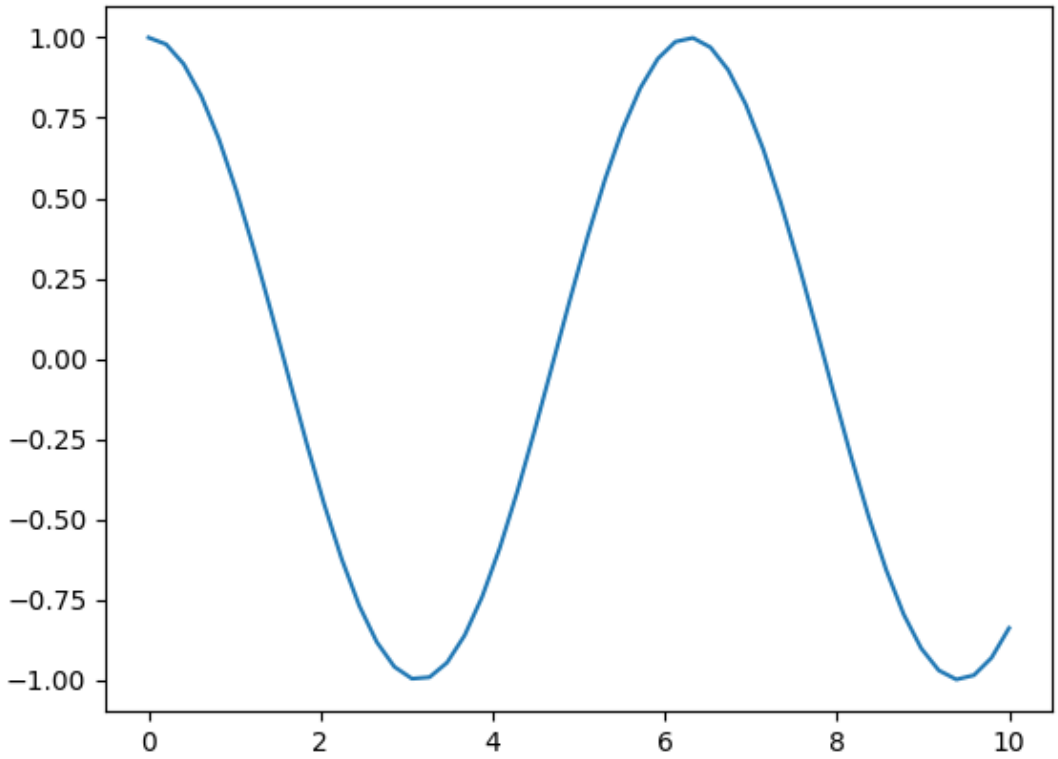
[7]:
```
array([ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

Vous remarquez que les 50 points couvrent à intervalles réguliers l'espace compris entre 0 et 10 inclusivement. Notons que 50 est aussi le nombre de points par défaut. Cette fonction est très utilisée lorsqu'on veut dessiner une fonction entre deux bornes, on a déjà eu l'occasion de le faire :

[8]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

[8]:
```
<contextlib.ExitStack at 0x1077283a0>
```

[9]:
```
# il est d'usage d'ajouter un point-virgule à la fin de la dernière ligne
# si on ne le fait pas (essayez..), on obtient l'affichage d'une ligne
# de bruit qui n'apporte rien
Y = np.cos(X)
plt.plot(X, Y);
```



Programmation vectorielle

Attardons-nous un petit peu :

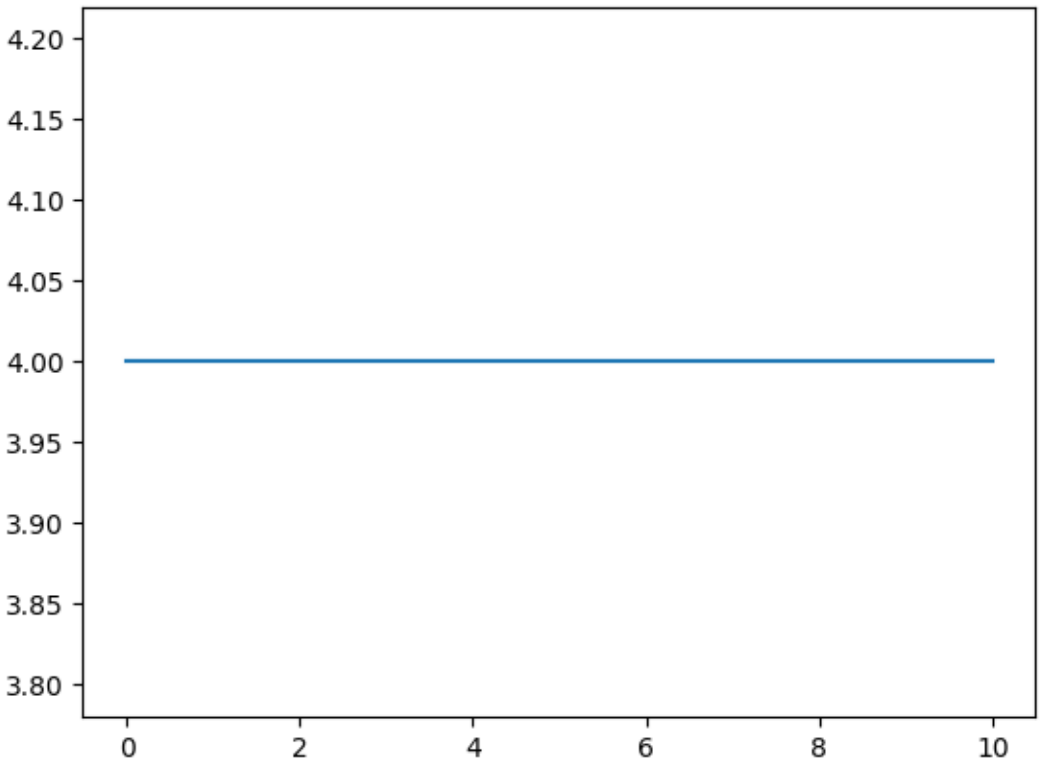
— nous avons créé un tableau X de 50 points qui couvrent l'intervalle [0..10] de manière uniforme,
— et nous avons calculé un tableau Y de 50 valeurs qui correspondent aux cosinus des valeurs de X.

Remarquez qu'on a fait ce premier calcul sans même savoir comment accéder aux éléments d'un tableau. Vous vous doutez bien qu'on va accèder aux éléments d'un tableau à base d'index, on le verra bien sûr, mais on n'en a pas eu besoin ici.

En fait en `numpy` on passe son temps à écrire des expressions dont les éléments sont des tableaux, et cela produit des opérations membre à membre, comme on vient de le voir avec cosinus.

Ainsi pour tracer la fonction $x \longrightarrow cos^2(x) + sin^2(x) + 3$ on fera tout simplement :

[10]:
```
# l'énorme majorité du temps, on écrit avec numpy
# des expressions qui impliquent des tableaux
# exactement comme si c'était des nombres
Z = np.cos(X)**2 + np.sin(X)**2 + 3

plt.plot(X, Z);
```



C'est le premier réflexe qu'il faut avoir avec les tableaux numpy : on a vu que les compréhensions et les expressions génératrices permettent de s'affranchir des boucles du genre :

```
out_data = []
for x in in_data:
    out_data.append(une_fonction(x))
```

on a vu en python natif qu'on ferait plutôt :

```
out_data = (une_fonction(x) for x in in_data)
```

Eh bien en fait, en numpy, on doit penser encore plus court :

```
out_data = une_fonction(in_data)
```

ou en tous les cas une expression qui fait intervenir `in_data` comme un tout, sans avoir besoin d'accéder à ses éléments.

`ufunc`

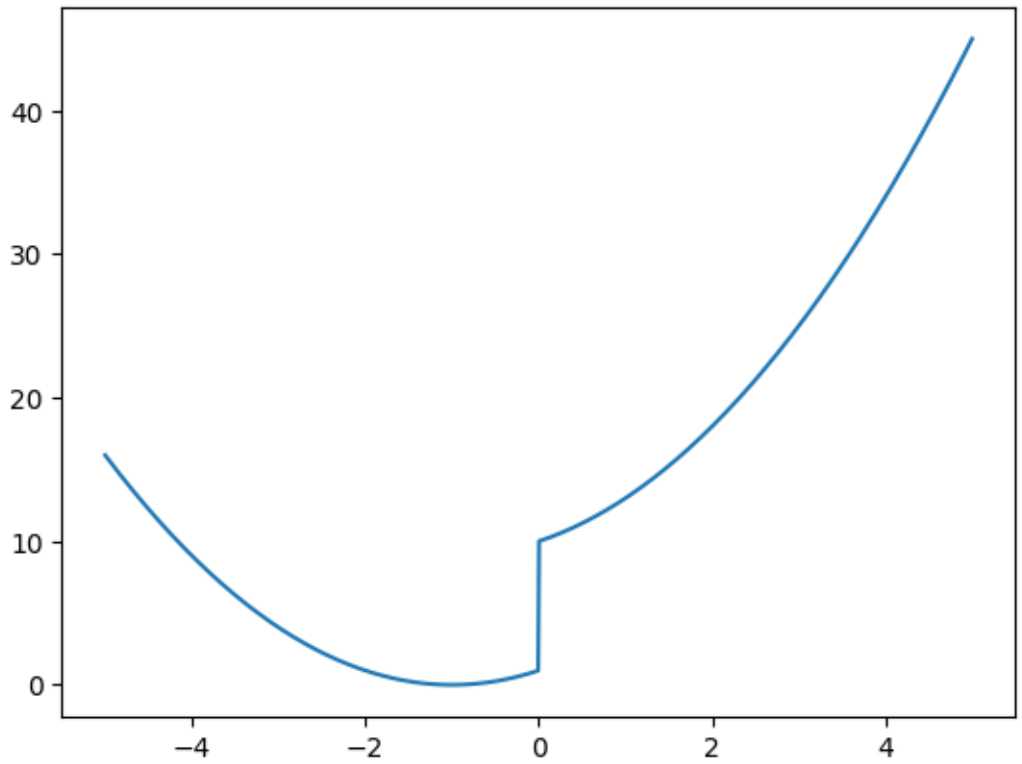
Le mécanisme général qui applique une fonction à un tableau est connu sous le terme de Universal function, ou `ufunc`, ça peut vous être utile avec les moteurs de recherche.

Voyez notamment la liste des fonctionnalités disponibles sous cette forme dans `numpy`.

Je vous signale également un utilitaire qui permet, sous forme de décorateur, de passer d'une fonction scalaire à une `ufunc` :

[11]:
```
# le décorateur np.vectorize vous permet
# de facilement transformer une opération scalaire
# en opération vectorielle
# je choisis à dessein une fonction définie par morceaux
@np.vectorize
def scalar_function(x):
    return x**2 + 2*x + (1 if x <=0 else 10)
```

[12]:
```
# je choisis de prendre beaucoup de points
# à cause de la discontinuité
X = np.linspace(-5, 5, 1000)
Y = scalar_function(X)
plt.plot(X, Y);
```



Conclusion

Pour conclure ce complément d'introduction, ce style de programmation - que je vais décider d'appeler programmation vectorielle de manière un peu impropre - est au cœur de `numpy`, et n'est bien entendu pas limitée aux tableaux de dimension 1, comme on va le voir dans la suite.

7.3 w7-s02-c2-dtype

Type d'un tableau **numpy**

7.3.1 Complément - niveau intermédiaire

Nous allons voir dans ce complément ce qu'il faut savoir sur le type d'un tableau `numpy`.

[1]:
```
import numpy as np
```

Dans ce complément nous allons rester en dimension 1 :

[2]:
```
a = np.array([1, 2, 4, 8])
```

Toutes les cellules ont le même type

Comme on l'a vu dans la vidéo, les très bonnes performances que l'on peut obtenir en utilisant un tableau `numpy` sont liées au fait que le tableau est homogène : toutes les cellules du tableau possèdent le même type :

[3]:
```
# pour accéder au type d'un tableau
a.dtype
```

[3]:
```
dtype('int64')
```

Vous voyez que dans notre cas, le système a choisi pour nous un type entier ; selon les entrées on peut obtenir :

[4]:
```
# si je mets au moins un flottant
f = np.array([1, 2, 4, 8.])
f.dtype
```

[4]:
```
dtype('float64')
```

[5]:
```
# et avec un complexe
c = np.array([1, 2, 4, 8j])
c.dtype
```

[5]:
```
dtype('complex128')
```

Et on peut préciser le type que l'on veut si cette heuristique ne nous convient pas :

[6]:
```
# je choisis explicitement mon dtype
c2 = np.array([1, 2, 4, 8], dtype=np.complex64)
c2.dtype
```

[6]:
```
dtype('complex64')
```

Pertes de précision

Une fois que le type est déterminé, on s'expose à de possibles pertes de précision, comme d'habitude :

[7]:
```
a, a.dtype
```

[7]:
```
(array([1, 2, 4, 8]), dtype('int64'))
```

[8]:
```
# a est de type entier
# je vais perdre le 0.14
a[0] = 3.14
a
```

[8]:
```
array([3, 2, 4, 8])
```

Types disponibles

Voyez la liste complète https ://docs.scipy.org/doc/numpy/user/basics.types.html.

Ce qu'il faut en retenir :

— vous pouvez choisir entre `bool`, `int`, `uint` (entier non signé), `float` et `complex` ;
— ces types ont diverses tailles pour vous permettre d'optimiser la mémoire réellement utilisée ;
— ces types existent en tant que tels (hors de tableaux).

[9]:
```
# un entier sur 1 seul octet, c'est possible !
np_1 = np.int8(1)
# l'équivalent en Python natif
py_1 = 1
```

[10]:
```
# il y a bien égalité
np_1 == py_1
```

[10]:
```
True
```

[11]:
```
# mais bien entendu ce ne sont pas les mêmes objets
np_1 is py_1
```

[11]:
```
False
```

Du coup, on peut commencer à faire de très substantielles économies de place ; imaginez que vous souhaitez manipuler une image d'un million de pixels en noir et blanc sur 256 niveaux de gris ; j'en profite pour vous montrer `np.zeros` (qui fait ce que vous pensez) :

[12]:
```
# pur Python
from sys import getsizeof
pure_py = [0 for i in range(10**6)]
getsizeof(pure_py)
```

[12]:
```
8448728
```

[13]:
```
# numpy
num_py = np.zeros(10**6, dtype=np.int8)
getsizeof(num_py)
```

[13]:
```
1000112
```

Je vous signale enfin l'attribut `itemsize` qui vous permet d'obtenir la taille en octets occupée par chacune des cellules, et qui correspond donc en gros au nombre qui apparaît dans `dtype`, mais divisé par huit :

[14]:
```
a.dtype
```

[14]:
```
dtype('int64')
```

[15]:
```
a.itemsize
```

[15]:
```
8
```

[16]:
```
c.dtype
```

[16]:
```
dtype('complex128')
```

[17]:
```
c.itemsize
```

[17]:
```
16
```

7.4 w7-s03-c1-shape

Forme d'un tableau **numpy**

Nous allons voir dans ce complément comment créer des tableaux en plusieurs dimensions et manipuler la forme (`shape`) des tableaux.

[1]:
```
import numpy as np
```

Un exemple

Nous avons vu précédemment comment créer un tableau `numpy` de dimension 1 à partir d'un simple itérable, nous allons à présent créer un tableau à 2 dimensions, et pour cela nous allons utiliser une liste imbriquée :

[2]:
```
d2 = np.array([[11, 12, 13], [21, 22, 23]])
d2
```

[2]:
```
array([[11, 12, 13],
       [21, 22, 23]])
```

Ce premier exemple va nous permettre de voir les différents attributs de tous les tableaux `numpy`.

L'attribut **shape**

Tous les tableaux `numpy` possèdent un attribut `shape` qui retourne, sous la forme d'un tuple, les dimensions du tableau :

[3]:
```
# la forme (les dimensions) du tableau
d2.shape
```

[3]:
```
(2, 3)
```

Dans le cas d'un tableau en 2 dimensions, cela correspond donc à lignes x colonnes.

On peut facilement changer de forme

Comme on l'a vu dans la vidéo, un tableau est en fait une vue vers un bloc de données. Aussi il est facile de changer la dimension d'un tableau - ou plutôt, de créer une autre vue vers les mêmes données :

[4]:
```
# l'argument qu'on passe à reshape est le tuple
# qui décrit la nouvelle *shape*
v2 = d2.reshape((3, 2))
v2
```

[4]:
```
array([[11, 12],
       [13, 21],
       [22, 23]])
```

Et donc, ces deux tableaux sont deux vues vers la même zone de données ; ce qui fait qu'une modification sur l'un se répercute dans l'autre :

[5]:
```
# je change un tableau
d2[0][0] = 100
d2
```

[5]:
```
array([[100,  12,  13],
       [ 21,  22,  23]])
```

[6]:
```
# ça se répercute dans l'autre
v2
```

[6]:
```
array([[100,  12],
       [ 13,  21],
       [ 22,  23]])
```

Les attributs liés à la forme

Signalons par commodité les attributs suivants, qui se dérivent de `shape` :

[7]:
```
# le nombre de dimensions
d2.ndim
```

[7]:
```
2
```

[8]:
```
# vrai pour tous les tableaux
len(d2.shape) == d2.ndim
```

[8]:
```
True
```

[9]:
```
# le nombre de cellules
d2.size
```

[9]:
```
6
```

[10]:
```
# vrai pour tous les tableaux
# une façon compliquée de dire
# une chose toute simple :
# la taille est le produit
# des dimensions
from operator import mul
from functools import reduce
d2.size == reduce(mul, d2.shape, 1)
```

[10]:
```
True
```

Lorsqu'on utilise `reshape`, il faut bien sûr que la nouvelle forme soit compatible :

[11]:
```
try:
    d2.reshape((3, 4))
except Exception as e:
    print(f"OOPS {type(e)} {e}")
```

```
OOPS <class 'ValueError'> cannot reshape array of size 6 into shape (3,4)
```

Dimensions supérieures

Vous pouvez donc deviner comment on construit des tableaux en dimensions supérieures à 2, il suffit d'utiliser un attribut `shape` plus élaboré :

[12]:
```
shape = (2, 3, 4)
size = reduce(mul, shape)

# vous vous souvenez de arange
data = np.arange(size)
```

[13]:
```
d3 = data.reshape(shape)
d3
```

[13]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Cet exemple vous permet de voir qu'en dimensions supérieures la forme est toujours :

n1 x n2 x … x lignes x colonnes

Enfin, ce que je viens de dire est arbitraire, dans le sens où, bien entendu, vous pouvez décider d'interpréter les tableaux comme vous voulez.

Mais en termes au moins de l'impression par `print`, il est logique de voir que l'algorithme d'impression balaye le tableau de manière mécanique comme ceci :

```
for i in range(2):
    for j in range(3):
        for k in range(4):
            array[i][j][k]
```

Et c'est pourquoi vous obtenez la présentation suivante avec des tableaux de dimensions plus grandes :

[14]:
```
# la même chose avec plus de dimensions
shape = (2, 3, 4, 5)
size = reduce(mul, shape) # le produit des 4 nombres dans shape
size
```

[14]:
```
120
```

[15]:
```
data = np.arange(size)

# ce tableau est visualisé
# à base de briques de base
# de 4 lignes et 5 colonnes
d4 = data.reshape(shape)
d4
```

[15]:
```
array([[[[  0,   1,   2,   3,   4],
         [  5,   6,   7,   8,   9],
         [ 10,  11,  12,  13,  14],
         [ 15,  16,  17,  18,  19]],

        [[ 20,  21,  22,  23,  24],
         [ 25,  26,  27,  28,  29],
         [ 30,  31,  32,  33,  34],
         [ 35,  36,  37,  38,  39]],

        [[ 40,  41,  42,  43,  44],
         [ 45,  46,  47,  48,  49],
         [ 50,  51,  52,  53,  54],
         [ 55,  56,  57,  58,  59]]],


       [[[ 60,  61,  62,  63,  64],
         [ 65,  66,  67,  68,  69],
         [ 70,  71,  72,  73,  74],
         [ 75,  76,  77,  78,  79]],

        [[ 80,  81,  82,  83,  84],
         [ 85,  86,  87,  88,  89],
         [ 90,  91,  92,  93,  94],
         [ 95,  96,  97,  98,  99]],

        [[100, 101, 102, 103, 104],
         [105, 106, 107, 108, 109],
         [110, 111, 112, 113, 114],
         [115, 116, 117, 118, 119]]]])
```

Vous voyez donc qu'avec la forme :

```
2, 3, 4, 5
```

cela vous donne l'impression que vous avez comme brique de base des tableaux qui ont :

```
4 lignes
5 colonnes
```

Et souvenez-vous que vous pouvez toujours insérer un `1` n'importe où dans la forme, puisque ça ne modifie pas la taille qui est le produit des dimensions :

[16]:
```
d2.shape
```

[16]:
```
(2, 3)
```

[17]:
```
d2
```

[17]:
```
array([[100,  12,  13],
       [ 21,  22,  23]])
```

[18]:
```
d2.reshape(2, 1, 3)
```

[18]:
```
array([[[100,  12,  13]],

       [[ 21,  22,  23]]])
```

[19]:
```
d2.reshape(2, 3, 1)
```

[19]:
```
array([[[100],
        [ 12],
        [ 13]],

       [[ 21],
        [ 22],
        [ 23]]])
```

Ou même :

[20]:
```
d2.reshape((1, 2, 3))
```

[20]:
```
array([[[100,  12,  13],
        [ 21,  22,  23]]])
```

[21]:
```
d2.reshape((1, 1, 1, 1, 2, 3))
```

[21]:
```
array([[[[[[100,  12,  13],
           [ 21,  22,  23]]]]]])
```

Résumé des attributs

Voici un résumé des attributs des tableaux `numpy` :

attribut	signification	exemple
`shape`	tuple des dimensions	`(3, 5, 7)`
`ndim`	nombre dimensions	`3`
`size`	nombre d'éléments	`3 * 5 * 7`
`dtype`	type de chaque élément	`np.float64`
`itemsize`	taille en octets d'un élément	`8`

Divers

Je vous signale enfin, à titre totalement anecdotique cette fois, l'existence de la méthode `ravel` qui vous permet d'aplatir n'importe quel tableau :

[22]:
```
d2
```

[22]:
```
array([[100,  12,  13],
       [ 21,  22,  23]])
```

[23]:
```
d2.ravel()
```

[23]:
```
array([100,  12,  13,  21,  22,  23])
```

[24]:
```
# il y a d'ailleurs aussi flatten qui fait
# quelque chose de semblable
d2.flatten()
```

[24]:
```
array([100,  12,  13,  21,  22,  23])
```

7.5 w7-s03-c2-initialisation

Création de tableaux

7.5.1 Complément - niveau basique

Passons rapidement en revue quelques méthodes pour créer des tableaux **numpy**.

[1]:
```
import numpy as np
```

Non initialisé : **np.empty**

La méthode la plus efficace pour créer un tableau **numpy** consiste à faire l'allocation de la mémoire mais sans l'initialiser :

[2]:
```
memory = np.empty(dtype=np.int8,
                  shape=(1_000, 1_000))
```

J'en profite pour attirer votre attention sur l'impression des gros tableaux où l'on s'efforce de vous montrer les coins :

[3]:
```
print(memory)
```

```
[[0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 …
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]
 [0 0 0 … 0 0 0]]
```

Il se peut que vous voyiez ici des valeurs particulières ; selon votre OS, il y a une probabilité non nulle que vous ne voyiez ici que des zéros. C'est un peu comme avec les dictionnaires qui, depuis la version 3.6, peuvent donner l'impression de conserver l'ordre dans lequel les clés ont été créées. Ici c'est un peu la même chose, vous ne devez pas écrire un programme qui repose sur le fait que **np.empty** retourne un tableau garni de zéros (utilisez alors **np.zeros**, que l'on va voir tout de suite).

Tableaux constants

On peut aussi créer et initialiser un tableau avec **np.zeros** et **np.ones** :

[4]:
```
zeros = np.zeros(dtype=np.complex128, shape=(1_000, 100))
print(zeros)
```

```
[[0.+0.j 0.+0.j 0.+0.j … 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j … 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j … 0.+0.j 0.+0.j 0.+0.j]
 …
 [0.+0.j 0.+0.j 0.+0.j … 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j … 0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j … 0.+0.j 0.+0.j 0.+0.j]]
```

[5]:
```
fours = 4 * np.ones(dtype=float, shape=(8, 8))
fours
```

[5]:
```
array([[4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.],
       [4., 4., 4., 4., 4., 4., 4., 4.]])
```

Progression arithmétique : **arange**

En guise de rappel, avec **arange** on peut créer des tableaux de valeurs espacées d'une valeur constante. Ça ressemble donc un peu au **range** de Python natif :

[6]:
```
np.arange(4)
```

[6]:
```
array([0, 1, 2, 3])
```

[7]:
```
np.arange(1, 5)
```

[7]:
```
array([1, 2, 3, 4])
```

Sauf qu'on peut y passer un pas qui n'est pas entier :

[8]:
```
np.arange(5, 7, .5)
```

[8]:
```
array([5. , 5.5, 6. , 6.5])
```

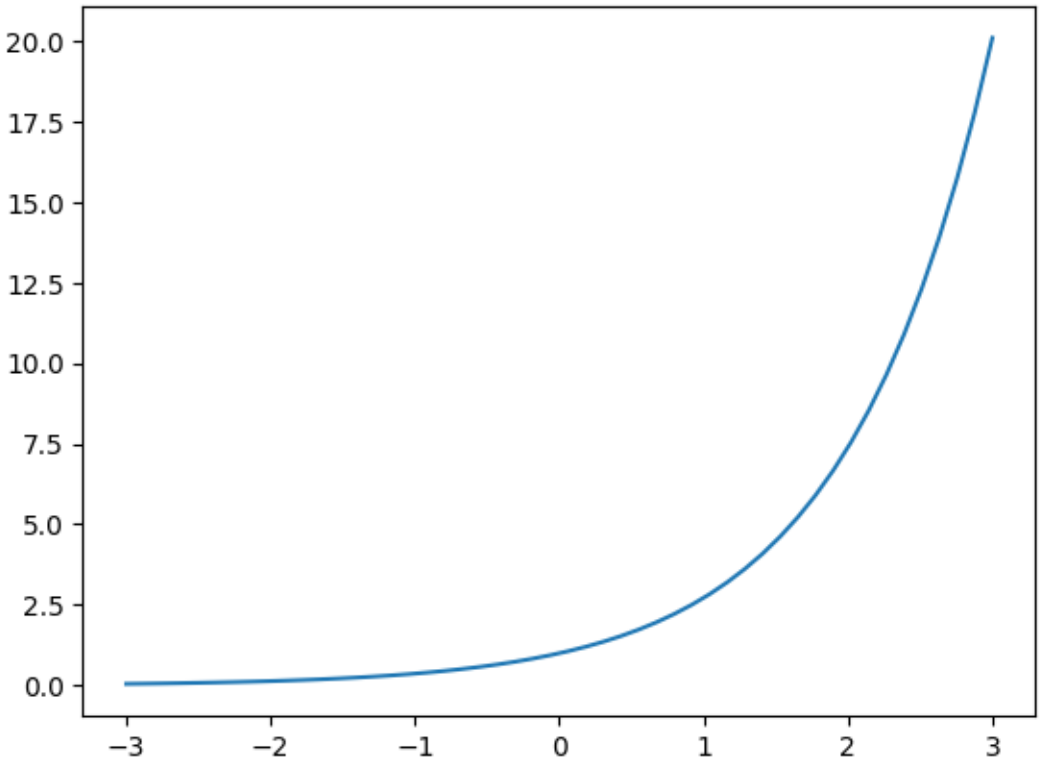
Progression arithmétique : **linspace**

Mais bien souvent, plutôt que de préciser le pas entre deux valeurs, on préfère préciser le nombre de points ; et aussi inclure la deuxième borne. C'est ce que fait **linspace**, c'est très utile pour modéliser une fonction sur un intervalle ; on a déjà vu des exemples de ce genre :

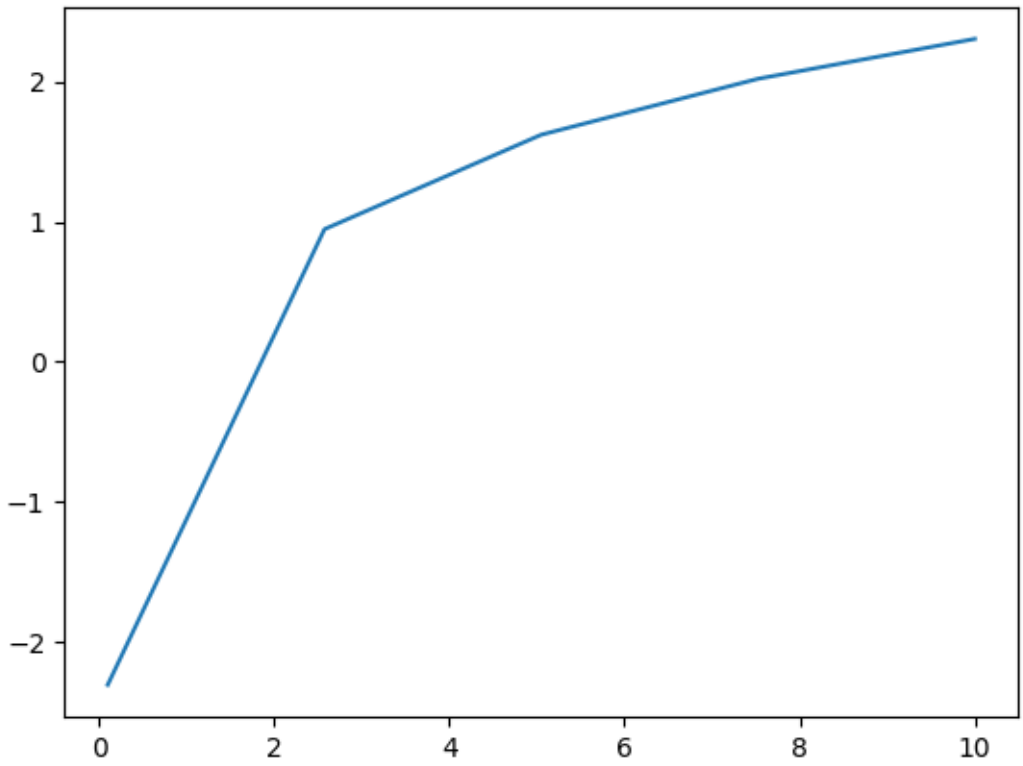
[9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.ion()
```

[9]:
```
<contextlib.ExitStack at 0x10b9804f0>
```

[10]:
```
X = np.linspace(-3., +3.)
Y = np.exp(X)

plt.plot(X, Y);
```



[11]:
```
# par défaut linspace crée 50 points
# avec moins de points

X = np.linspace(1/10, 10, num = 5)
plt.plot(X, np.log(X));
```



Pour des intervalles en progression géométrique, voyez `np.geomspace`.

Multi-dimensions : **`indices`**

La méthode `np.indices` se comporte un peu comme `arange` mais pour plusieurs directions ; voyons ça sur un exemple :

[12]:
```
ix, iy = np.indices((3, 5))
```

[13]:
```
ix
```

[13]:
```
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2]])
```

[14]:
```
iy
```

[14]:
```
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])
```

Cette fonction s'appelle `indices` parce qu'elle produit des tableaux (ici 2 car on lui a passé une `shape` à deux dimensions) qui contiennent, à la case (i, j), i (pour le premier tableau) ou j pour le second.

Ainsi, si vous voulez construire un tableau de taille (2, 4) dans lequel, par exemple :

```
tab[i, j] = 200*i + 2*j + 50
```

Vous n'avez qu'à faire :

[15]:
```
ix, iy = np.indices((2, 4))
tab = 200*ix + 2*iy + 50
tab
```

[15]:
```
array([[ 50,  52,  54,  56],
       [250, 252, 254, 256]])
```

Multi-dimensions : **meshgrid**

Si vous voulez créer un tableau un peu comme avec `linspace`, mais en plusieurs dimensions : imaginez par exemple que vous voulez tracer une fonction à deux entrées :

$f : (x, y) \longrightarrow cos(x) + cos^2(y)$

Sur un pavé délimité par :

$x \in [-\pi, +\pi], y \in [3\pi, 5\pi]$

Il vous faut donc créer un tableau, disons de 50 x 50 points, qui réalise un maillage uniforme de ce pavé, et pour ça vous pouvez utiliser `meshgrid`. Pour commencer :

[16]:
```
# on fabrique deux tableaux qui échantillonnent
# de manière uniforme les intervalles en X et en Y
# on prend un pas de 10 dans les deux sens, ça nous donnera
# 100 points pour couvrir l'espace carré qui nous intéresse

Xticks, Yticks = (np.linspace(-np.pi, np.pi, num=10),
                  np.linspace(3*np.pi, 5*np.pi, num=10))
```

Avec meshgrid, on va créer deux tableaux, qui sont respectivement les (100) X et les (100) Y de notre maillage :

[17]:
```
# avec meshgrid on les croise
# ça fait comme un produit cartésien,
# en extrayant les X et les Y du résultat

X, Y = np.meshgrid(Xticks, Yticks)

# chacun des deux est donc de taille 10 x 10
X.shape, Y.shape
```

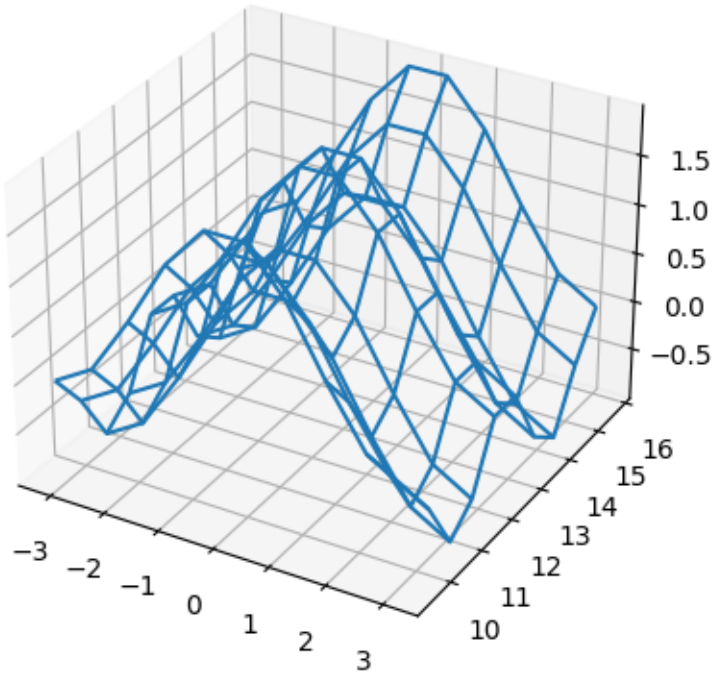
[17]:
```
((10, 10), (10, 10))
```

Que peut-on faire avec ça ? Eh bien, en fait, on a tout ce qu'il nous faut pour afficher notre fonction :

[18]:
```
# un tableau 10 x 10 qui contient les images de f()
# sur les points de la grille
Z = np.cos(X) + np.cos(Y)**2
```

[19]:
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

```
ax.plot_wireframe(X, Y, Z);
```



Je vous laisse vous convaincre qu'il est facile d'écrire `np.indices` à partir de `np.meshgrid` et `np.arange`.

7.6 w7-s05-c1-broadcasting

Le broadcasting

[1]:
```
import numpy as np
```

7.6.1 Complément - niveau intermédiaire

Lorsque l'on a parlé de programmation vectorielle, on a vu que l'on pouvait écrire quelque chose comme ceci :

[2]:
```
X = np.linspace(0, 2 * np.pi)
Y = np.cos(X) + np.sin(X) + 2
```

Je vous fais remarquer que dans cette dernière ligne on combine :

— deux tableaux de mêmes tailles - quand on ajoute `np.cos(X)` avec `np.sin(X)` ;
— un tableau avec un scalaire - quand on ajoute `2` au résultat.

En fait, le broadcasting est ce qui permet :

— d'unifier le sens de ces deux opérations ;
— de donner du sens à des cas plus généraux, où on fait des opérations entre des tableaux qui ont des tailles différentes, mais assez semblables pour que l'on puisse tout de même les combiner.

7.6.2 Exemples en 2D

Nous allons commencer par quelques exemples simples, avant de généraliser le mécanisme. Pour commencer, nous nous donnons un tableau de base :

[3]:
```
a = 100 * np.ones((3, 5), dtype=np.int32)
print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

Je vais illustrer le broadcasting avec l'opération `+`, mais bien entendu ce mécanisme est à l'œuvre dès que vous faites des opérations entre deux tableaux qui n'ont pas les mêmes dimensions.

Pour commencer, je vais donc ajouter à mon tableau de base un scalaire :

Broadcasting entre les dimensions `(3, 5)` et `(1,)`

[4]:
```
print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

[5]:
```
b = 3
print(b)
```

```
3
```

Lorsque j'ajoute ces deux tableaux, c'est comme si j'avais ajouté à `a` la différence :

[6]:
```
# pour élaborer c
c = a + b
print(c)
```

```
[[103 103 103 103 103]
 [103 103 103 103 103]
 [103 103 103 103 103]]
```

[7]:
```
# c'est comme si j'avais
# ajouté à a ce terme-ci
print(c - a)
```

```
[[3 3 3 3 3]
 [3 3 3 3 3]
 [3 3 3 3 3]]
```

C'est un premier cas particulier de broadcasting dans sa version extrême.

Le scalaire `b`, qui est en l'occurrence considéré comme un tableau dont le `shape` vaut `(1,)`, est dupliqué dans les deux directions jusqu'à obtenir ce tableau uniforme de taille `(5, 3)` et qui contient un `3` partout.

Et c'est ce tableau, qui est maintenant de la même taille que `a`, qui est ajouté à `a`.

Je précise que cette explication est du domaine du modèle pédagogique ; je ne dis pas que l'implémentation va réellement allouer un second tableau, bien évidemment on peut optimiser pour éviter cette construction inutile.

Broadcasting `(3, 5)` et `(5,)`

Voyons maintenant un cas un peu moins évident. Je peux ajouter à mon tableau de base une ligne, c'est-à-dire un tableau de taille `(5, )`. Voyons cela :

[8]:
```
print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

[9]:
```
b = np.arange(1, 6)
print(b)
```

```
[1 2 3 4 5]
```

[10]:
```
b.shape
```

[10]: `(5,)`

Ici encore, je peux ajouter les deux termes :

[11]:
```
# je peux ici encore
# ajouter les tableaux
c = a + b
print(c)
```

```
[[101 102 103 104 105]
 [101 102 103 104 105]
 [101 102 103 104 105]]
```

[12]:
```
# et c'est comme si j'avais
# ajouté à a ce terme-ci
print(c - a)
```

```
[[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
```

Avec le même point de vue que tout à l'heure, on peut se dire qu'on a d'abord transformé (broadcasté) le tableau `b` :

depuis la dimension `(5,)`

vers la dimension `(3, 5)`

[13]:
```
# départ
print(b)
```

```
[1 2 3 4 5]
```

[14]:
```
# arrivée
print(c - a)
```

```
[[1 2 3 4 5]
 [1 2 3 4 5]
 [1 2 3 4 5]]
```

Vous commencez à mieux voir comment ça fonctionne ; s'il existe une direction dans laquelle on peut "tirer" les données pour faire coincider les formes, on peut faire du broadcasting. Et ça marche dans toutes les directions, comme on va le voir tout de suite.

Broadcasting `(3, 5)` et `(3, 1)`

Au lieu d'ajouter à `a` une ligne, on peut lui ajouter une colonne, pourvu qu'elle ait la même taille que les colonnes de `a` :

[15]:
```
print(a)
```

```
[[100 100 100 100 100]
 [100 100 100 100 100]
 [100 100 100 100 100]]
```

[16]:
```
b = np.arange(1, 4).reshape(3, 1)
print(b)
```

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

Voyons comment se passe le broadcasting dans ce cas-là :

[17]:
```
c = a + b
print(c)
```

```
[[101 101 101 101 101]
 [102 102 102 102 102]
 [103 103 103 103 103]]
```

[18]:
```
print(c - a)
```

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

Vous voyez que tout se passe exactement de la même façon que lorsqu'on avait ajouté une simple ligne, on a cette fois "tiré" la colonne dans la direction des lignes, pour passer :

depuis la dimension `(3, 1)`

vers la dimension `(3, 5)`

[19]:
```
# départ
print(b)
```

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

[20]:
```
# arrivée
print(c - a)
```

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

Broadcasting `(3, 1)` et `(1, 5)`

Nous avons maintenant tous les éléments en main pour comprendre un exemple plus intéressant, où les deux tableaux ont des formes pas vraiment compatibles à première vue :

[21]:
```
col = np.arange(1, 4).reshape((3, 1))
print(col)
```

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

[22]:
```
line = 100 * np.arange(1, 6)
print(line)
```

```
[100 200 300 400 500]
```

Grâce au broadcasting, on peut additionner ces deux tableaux pour obtenir ceci :

[23]:
```
m = col + line
print(m)
```

```
[[101 201 301 401 501]
 [102 202 302 402 502]
 [103 203 303 403 503]]
```

Remarquez qu'ici les deux entrées ont été étirées pour atteindre une dimension commune.

Et donc pour illustrer le broadcasting dans ce cas, tout se passe comme si on avait :

transformé la colonne `(3, 1)`

en tableau `(3, 5)`

[24]:
```
print(col)
```

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

[25]:
```
print(col + np.zeros(5, dtype=np.int64))
```

```
[[1 1 1 1 1]
```

```
 [2 2 2 2 2]
 [3 3 3 3 3]]
```

et transformé la ligne `(1, 5)`

en tableau `(3, 5)`

[26]:
```
print(line)
```

```
[100 200 300 400 500]
```

[27]:
```
print(line + np.zeros(3, dtype=np.int64).reshape((3, 1)))
```

```
[[100 200 300 400 500]
 [100 200 300 400 500]
 [100 200 300 400 500]]
```

avant d'additionner terme à terme ces deux tableaux 3 x 5.

7.6.3 En dimensions supérieures

Pour savoir si deux tableaux peuvent être compatibles via broadcasting, il faut comparer leurs formes. Je commence par vous donner des exemples. Ici encore quand on mentionne l'addition, cela vaut pour n'importe quel opérateur binaire.

Exemples de dimensions compatibles

```
A   15 x 3 x 5
B   15 x 1 x 5
A+B 15 x 3 x 5
```

Cas de l'ajout d'un scalaire :

```
A   15 x 3 x 5
B            1
A+B 15 x 3 x 5
```

```
A   15 x 3 x 5
B        3 x 5
A+B 15 x 3 x 5
```

```
A   15 x 3 x 5
B        3 x 1
A+B 15 x 3 x 5
```

Exemples de dimensions non compatibles

Deux lignes de longueurs différentes :

```
A  3
B  4
```

Un cas plus douteux :

```
A       2 x 1
B   8 x 4 x 3
```

Comme vous le voyez sur tous ces exemples :

— on peut ajouter A et B lorsqu'il existe une dimension C qui "étire" à la fois celle de A et celle de B ;
— on le voit sur le dernier exemple, mais on ne peut broadcaster que de 1 vers n ; lorsque $p > 1$ divise n, on ne peut pas broadcaster de p vers n, comme on pourrait peut-être l'imaginer.

Comme c'est un cours de Python, plutôt que de formaliser ça sous une forme mathématique - je vous le laisse en exercice - je vais vous proposer plutôt une fonction Python qui détermine si deux tuples sont des `shape` compatibles de ce point de vue.

[28]:
```
# le module broadcasting n'est pas standard
# c'est moi qui l'ai écrit pour illustrer le cours
from broadcasting import compatible, compatible2
```

[29]:
```
# on peut dupliquer selon un axe
compatible((15, 3, 5), (15, 1, 5))
```

[29]:
```
(15, 3, 5)
```

[30]:
```
# ou selon deux axes
compatible((15, 3, 5), (5,))
```

[30]:
```
(15, 3, 5)
```

[31]:
```
# c'est bien clair que non
compatible((2,), (3,))
```

[31]:
```
False
```

[32]:
```
# on ne peut pas passer de 2 à 4
compatible((1, 2), (2, 4))
```

[32]:
```
False
```

7.7 w7-s05-c2-indexing-slicing

Index et slices

7.7.1 Complément - niveau basique

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

[1]:
```
<contextlib.ExitStack at 0x1128c0790>
```

J'espère que vous êtes à présent convaincus qu'il est possible de faire énormément de choses avec `numpy` en faisant des opérations entre tableaux, et sans aller référencer un par un les éléments des tableaux, ni faire de boucle `for`.

Il est temps maintenant de voir que l'on peut aussi manipuler les tableaux `numpy` avec des index.

Indexation par des entiers et tuples

La façon la plus naturelle d'utiliser un tableau est habituellement à l'aide des indices. On peut aussi bien sûr accéder aux éléments d'un tableau `numpy` par des indices :

[2]:
```
# une fonction qui crée un tableau
# tab[i, j] = i + 10 * j
def background(n):
    i = np.arange(n)
    j = i.reshape((n, 1))
    return i + 10 * j
```

[3]:
```
a5 = background(5)
print(a5)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

Avec un seul index on obtient naturellement une ligne :

[4]:
```
a5[1]
```

[4]:
```
array([10, 11, 12, 13, 14])
```

[5]:
```
# que l'on peut à nouveau indexer
a5[1][2]
```

[5]:
```
12
```

[6]:
```
# ou plus simplement indexer par un tuple
a5[1, 2]
```

[6]:
```
12
```

[7]:
```
# naturellement on peut affecter une case
# individuellement
a5[2][1] = 221
a5[3, 2] += 300
print(a5)
```

```
[[  0   1   2   3   4]
 [ 10  11  12  13  14]
 [ 20 221  22  23  24]
 [ 30  31 332  33  34]
 [ 40  41  42  43  44]]
```

[8]:
```
# ou toute une ligne
a5[1] = np.arange(100, 105)
print(a5)
```

```
[[  0   1   2   3   4]
 [100 101 102 103 104]
 [ 20 221  22  23  24]
 [ 30  31 332  33  34]
 [ 40  41  42  43  44]]
```

[9]:
```
# et on on peut aussi changer
# toute une ligne par broadcasting
a5[4] = 400
print(a5)
```

```
[[  0   1   2   3   4]
 [100 101 102 103 104]
 [ 20 221  22  23  24]
 [ 30  31 332  33  34]
 [400 400 400 400 400]]
```

7.8 w7-s05-c2-indexing-slicing

Slicing

Grâce au slicing on peut aussi référencer une colonne :

[10]:
```
a5 = background(5)
print(a5)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]
 [40 41 42 43 44]]
```

[11]:
```
a5[:, 3]
```

[11]:
```
array([ 3, 13, 23, 33, 43])
```

C'est un tableau à une dimension, mais vous pouvez tout de même modifier la colonne par une affectation :

[12]:
```
a5[:, 3] = range(300, 305)
print(a5)
```

```
[[  0   1   2 300   4]
 [ 10  11  12 301  14]
 [ 20  21  22 302  24]
 [ 30  31  32 303  34]
 [ 40  41  42 304  44]]
```

Ou, ici également bien sûr, par broadcasting :

[13]:
```
# on affecte un scalaire à une colonne
a5[:, 2] = 200
print(a5)
```

```
[[  0   1 200 300   4]
 [ 10  11 200 301  14]
 [ 20  21 200 302  24]
 [ 30  31 200 303  34]
 [ 40  41 200 304  44]]
```

[14]:
```
# ou on ajoute un scalaire à une colonne
a5[:, 4] += 400
print(a5)
```

```
[[  0   1 200 300 404]
 [ 10  11 200 301 414]
 [ 20  21 200 302 424]
 [ 30  31 200 303 434]
 [ 40  41 200 304 444]]
```

Les slices peuvent prendre une forme générale :

[15]:
```
a8 = background(8)
print(a8)
```

```
[[ 0  1  2  3  4  5  6  7]
 [10 11 12 13 14 15 16 17]
 [20 21 22 23 24 25 26 27]
 [30 31 32 33 34 35 36 37]
 [40 41 42 43 44 45 46 47]
 [50 51 52 53 54 55 56 57]
 [60 61 62 63 64 65 66 67]
 [70 71 72 73 74 75 76 77]]
```

[16]:
```
# toutes les lignes de rang 1, 4, 7
a8[1::3]
```

[16]:
```
array([[10, 11, 12, 13, 14, 15, 16, 17],
       [40, 41, 42, 43, 44, 45, 46, 47],
       [70, 71, 72, 73, 74, 75, 76, 77]])
```

[17]:
```
# toutes les colonnes de rang 1, 5, 9
a8[:, 1::4]
```

[17]:
```
array([[ 1,  5],
       [11, 15],
       [21, 25],
       [31, 35],
       [41, 45],
       [51, 55],
       [61, 65],
       [71, 75]])
```

[18]:
```
# et on peut bien sûr les modifier
a8[:, 1::4] = 0
print(a8)
```

```
[[ 0  0  2  3  4  0  6  7]
 [10  0 12 13 14  0 16 17]
 [20  0 22 23 24  0 26 27]
 [30  0 32 33 34  0 36 37]
 [40  0 42 43 44  0 46 47]
 [50  0 52 53 54  0 56 57]
 [60  0 62 63 64  0 66 67]
 [70  0 72 73 74  0 76 77]]
```

Du coup, le slicing peut servir à extraire des blocs :

[19]:
```
# un bloc au hasard dans a8
print(a8[5:8, 2:5])
```

```
[[52 53 54]
 [62 63 64]
 [72 73 74]]
```

newaxis

On peut utiliser également le symbole spécial `np.newaxis` en conjonction avec un slice pour "décaler" les dimensions :

[20]:
```
X = np.arange(1, 7)
print(X)
```

```
[1 2 3 4 5 6]
```

[21]:
```
X.shape
```

[21]: (6,)

[22]:
```
Y = X[:, np.newaxis]
print(Y)
```

```
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
```

[23]:
```
Y.shape
```

[23]: (6, 1)

Et ainsi de suite :

[24]:
```
Z = Y[:, np.newaxis]
Z
```

[24]:
```
array([[[1]],

       [[2]],

       [[3]],

       [[4]],

       [[5]],

       [[6]]])
```

[25]:
```
Z.shape
```

[25]:
```
(6, 1, 1)
```

De cette façon, par exemple, en combinant le slicing pour créer X et Y, et le broadcasting pour créer leur somme, je peux créer facilement la table de tous les tirages de 2 dés à 6 faces :

[26]:
```
dice2 = X + Y
print(dice2)
```

```
[[ 2  3  4  5  6  7]
 [ 3  4  5  6  7  8]
 [ 4  5  6  7  8  9]
 [ 5  6  7  8  9 10]
 [ 6  7  8  9 10 11]
 [ 7  8  9 10 11 12]]
```

Ou tous les tirages à trois dés :

[27]:
```
dice3 = X + Y + Z
print(dice3)
```

```
[[[ 3  4  5  6  7  8]
  [ 4  5  6  7  8  9]
  [ 5  6  7  8  9 10]
  [ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]]

 [[ 4  5  6  7  8  9]
  [ 5  6  7  8  9 10]
  [ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]]

 [[ 5  6  7  8  9 10]
  [ 6  7  8  9 10 11]
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]]

 [[ 6  7  8  9 10 11]
```

```
  [ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]
  [11 12 13 14 15 16]]

 [[ 7  8  9 10 11 12]
  [ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]
  [11 12 13 14 15 16]
  [12 13 14 15 16 17]]

 [[ 8  9 10 11 12 13]
  [ 9 10 11 12 13 14]
  [10 11 12 13 14 15]
  [11 12 13 14 15 16]
  [12 13 14 15 16 17]
  [13 14 15 16 17 18]]]
```

J'en profite pour introduire un utilitaire qui n'a rien à voir, mais avec `np.unique`, vous pourriez calculer le nombre d'occurrences dans le tableau, et ainsi calculer les probabilités d'apparition de tous les nombres entre 3 et 18 :

[28]:
```
np.unique(dice3, return_counts=True)
```

[28]:
```
(array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18]),
 array([ 1,  3,  6, 10, 15, 21, 25, 27, 27, 25, 21, 15, 10,  6,  3,  1]))
```

Différences avec les listes

Avec l'indexation et le slicing, on peut créer des tableaux qui sont des vues sur des fragments d'un tableau ; on peut également déformer leur dimension grâce à `newaxis` ; on peut modifier ces fragments, en utilisant un scalaire, un tableau, ou une slice sur un autre tableau. Les possibilités sont infinies.

Il est cependant utile de souligner quelques différences entre les tableaux `numpy` et, les listes natives, pour ce qui concerne les indexations et le slicing.

On ne peut pas changer la taille d'un tableau avec le slicing La taille d'un objet `numpy` est par définition constante ; cela signifie qu'on ne peut pas, par exemple, modifier sa taille totale avec du slicing ; c'est à mettre en contraste avec, si vous vous souvenez :

Listes

[29]:
```
# on peut faire ceci
liste = [0, 1, 2]
liste[1:2] = [100, 102, 102]
liste
```

[29]:
```
[0, 100, 102, 102, 2]
```

Tableaux

[30]:
```
# on ne peut pas faire cela
array = np.array([0, 1, 2])
try:
    array[1:2] = np.array([100, 102, 102])
```

```
except Exception as e:
    print(f"OOPS, {type(e)}, {e}")
```

```
OOPS, <class 'ValueError'>, could not broadcast input array from shape (3,)
   into shape (1,)
```

On peut modifier un tableau en modifiant une slice

Une slice sur un objet `numpy` renvoie une vue sur un extrait du tableau, et en changeant la vue on change le tableau ; ici encore c'est à mettre en contraste avec ce qui se passe sur les listes :

Listes

[31]:
```
# une slice d'une liste est une shallow copy
liste = [0, 1, 2]
liste[1:2]
```

[31]:
```
[1]
```

[32]:
```
# en modifiant la slice,
# on ne modifie pas la liste
liste[1:2][0] = 999999
liste
```

[32]:
```
[0, 1, 2]
```

Tableaux

[33]:
```
# une slice d'un tableau numpy est un extrait du tableau
array = np.array([0, 1, 2])
array[1:2]
```

[33]:
```
array([1])
```

[34]:
```
array[1:2][0] = 100
array
```

[34]:
```
array([  0, 100,   2])
```

7.9 w7-s05-c3-operations-logiques

Opérations logiques

7.9.1 Complément - niveau basique

Même si les tableaux contiennent habituellement des nombres, on peut être amenés à faire des opérations logiques et du coup à manipuler des tableaux de booléens. Nous allons voir quelques éléments à ce sujet.

[1]:
```
import numpy as np
```

Opérations logiques

On peut faire des opérations logiques entre tableaux exactement comme on fait des opérations arithmétiques.

On va partir de deux tableaux presque identiques. J'en profite pour vous signaler qu'on peut copier un tableau avec, tout simplement, `np.copy` :

[2]:
```
a = np.arange(25).reshape(5, 5)
print(a)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
```

[3]:
```
b = np.copy(a)
b[2, 2] = 1000
print(b)
```

```
[[   0    1    2    3    4]
 [   5    6    7    8    9]
 [  10   11 1000   13   14]
 [  15   16   17   18   19]
 [  20   21   22   23   24]]
```

Dans la lignée de ce qu'on a vu jusqu'ici en matière de programmation vectorielle, une opération logique va ici aussi nous retourner un tableau de la même taille :

[4]:
```
# la comparaison par == ne nous
# retourne pas directement un booléen
# mais un tableau de la même taille que a et b
print(a == b)
```

```
[[ True  True  True  True  True]
 [ True  True  True  True  True]
 [ True  True False  True  True]
 [ True  True  True  True  True]
 [ True  True  True  True  True]]
```

`all` et `any`

Si votre intention est de vérifier que les deux tableaux sont entièrement identiques, utilisez `np.all` - et non pas le built-in natif `all` de Python - qui va vérifier que tous les éléments du tableau sont vrais :

[5]:
```
# oui
np.all(a == a)
```

[5]: `True`

[6]:
```
# oui
np.all(a == b)
```

[6]: `False`

[7]:
```
# oui
# on peut faire aussi bien
#   np.all(x)
# ou
```

```
#   x.all()
(a == a).all()
```

[7]: True

[8]:
```
# par contre : non !
# ceci n'est pas conseillé
# même si ça peut parfois fonctionner
try:
    all(a == a)
except Exception as e:
    print(f'OOPS {type(e)} {e}')
```

```
OOPS <class 'ValueError'> The truth value of an array with more than one el
  ement is ambiguous. Use a.any() or a.all()
```

C'est bien sûr la même chose pour **any** qui va vérifier qu'il y a au moins un élément vrai. Comme en Python natif, un nombre qui est nul est considéré comme faux :

[9]:
```
np.zeros(5).any()
```

[9]: False

[10]:
```
np.ones(5).any()
```

[10]: True

Masques

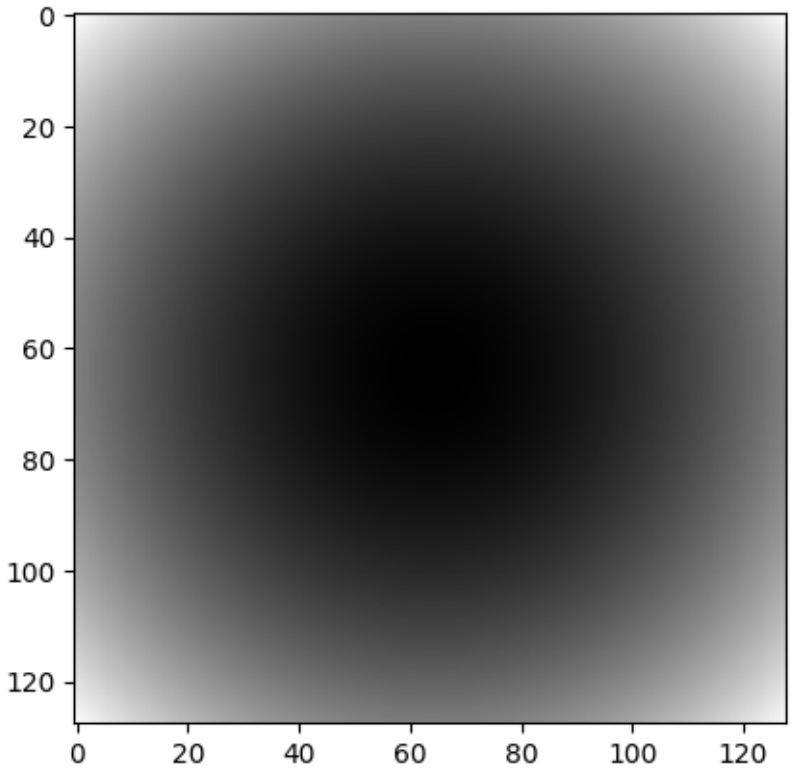
Mais en général, c'est rare qu'on ait besoin de consolider de la sorte un booléen sur tout un tableau, on utilise plutôt les tableaux logiques comme des masques, pour faire ou non des opérations sur un autre tableau.

J'en profite pour introduire une fonction de **matplotlib** qui s'appelle **imshow** et qui permet d'afficher une image :

[11]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

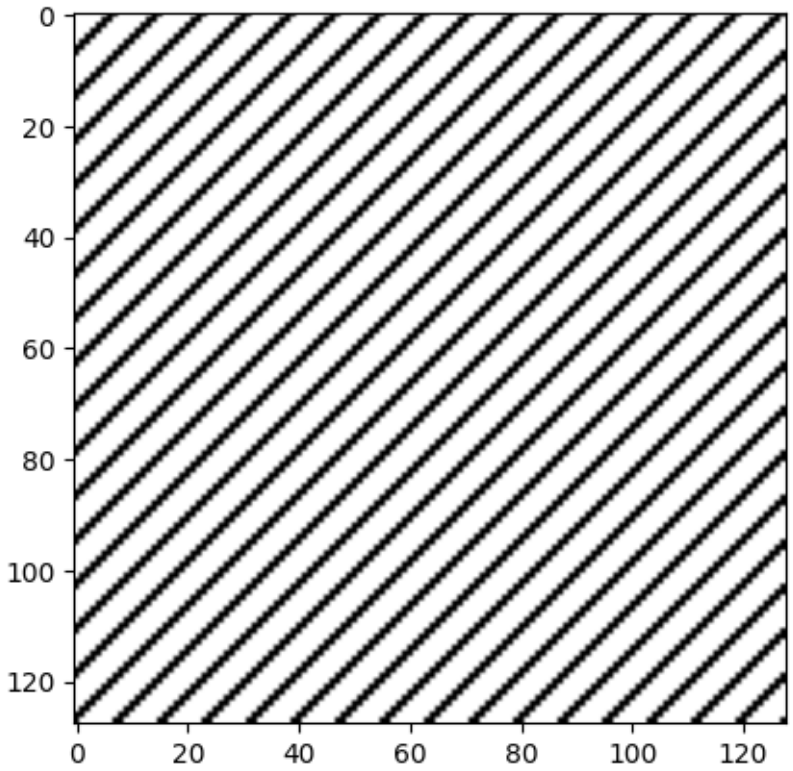
[11]: <contextlib.ExitStack at 0x1063d0490>

[12]:
```
# construisons un disque centré au milieu de l'image

width = 128
center = width / 2

ix, iy = np.indices((width, width))
image = (ix-center)**2 + (iy-center)**2
# pour afficher l'image en niveaux de gris
plt.imshow(image, cmap='gray');
```



Maintenant je peux créer un masque qui produise des rayures en diagonale, donc selon la valeur de `(i+j)`. Par exemple :

[13]:
```
# pour faire des rayures
# de 6 pixels de large
rayures = (ix + iy) % 8 <= 5
plt.imshow(rayures, cmap='gray');
```



[14]:
```
# en fait c'est bien sûr
# un tableau de booléens
print(rayures)
```

```
[[ True  True  True …  True False False]
 [ True  True  True … False False  True]
 [ True  True  True … False  True  True]
 …
 [ True False False …  True  True  True]
 [False False  True …  True  True  True]
 [False  True  True …  True  True False]]
```

je vous montre aussi comment inverser un masque parce que c'est un peu abscons :

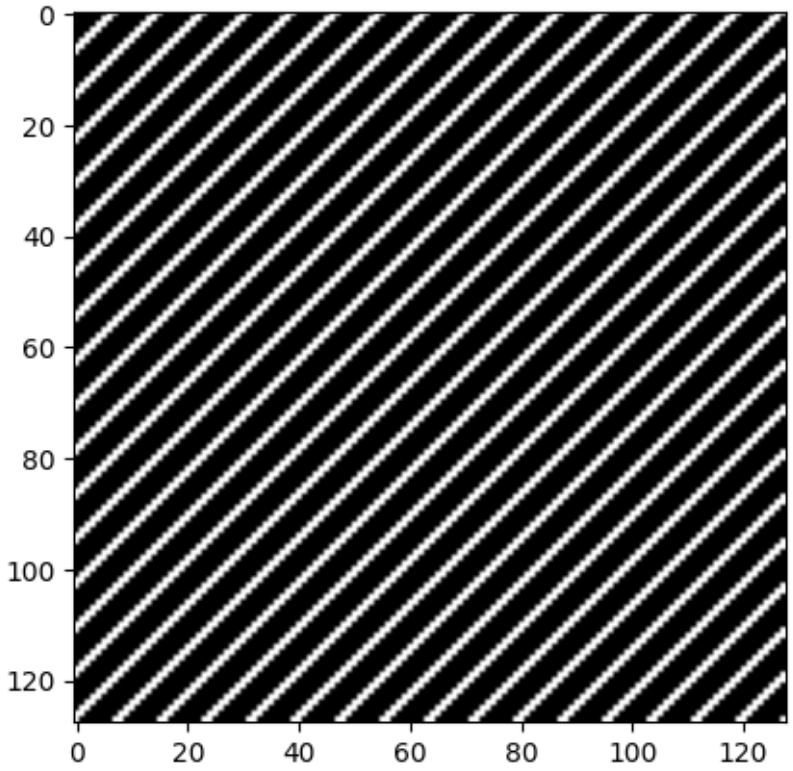
[15]:
```
# on ne peut pas faire
try:
    anti_rayures = not rayures
except Exception as e:
    print(f"OOPS - {type(e)} - {e}")
```

```
OOPS - <class 'ValueError'> - The truth value of an array with more than on
  e element is ambiguous. Use a.any() or a.all()
```

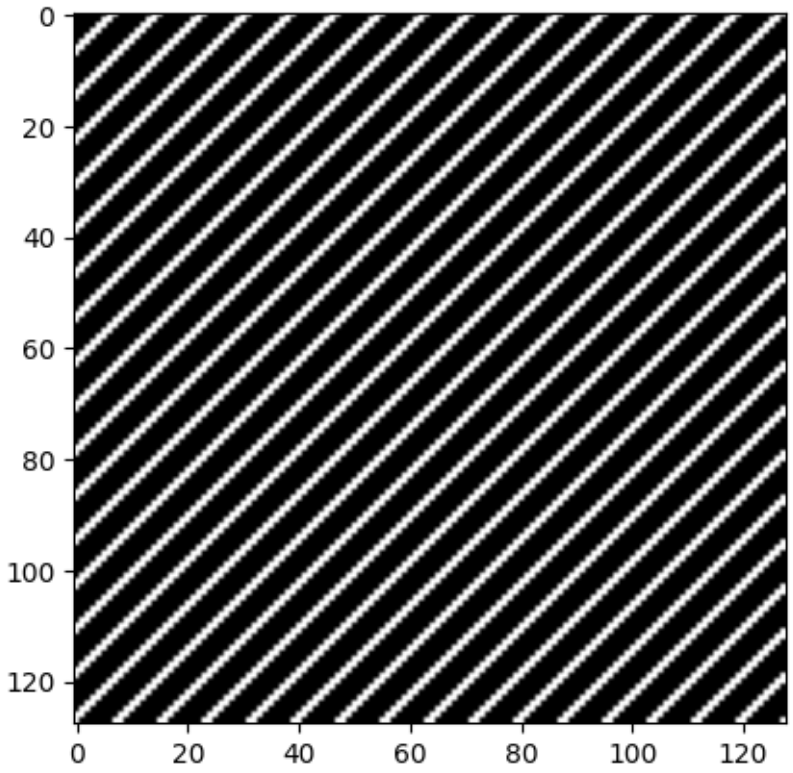
on ne peut pas non plus faire `rayures.not()`, parce `not` est un mot clé

[16]:
```
# on a le choix entre utiliser
#   rayures.logical_not()
```

```
anti_rayures = np.logical_not(rayures)
plt.imshow(anti_rayures,
           cmap='gray');
```

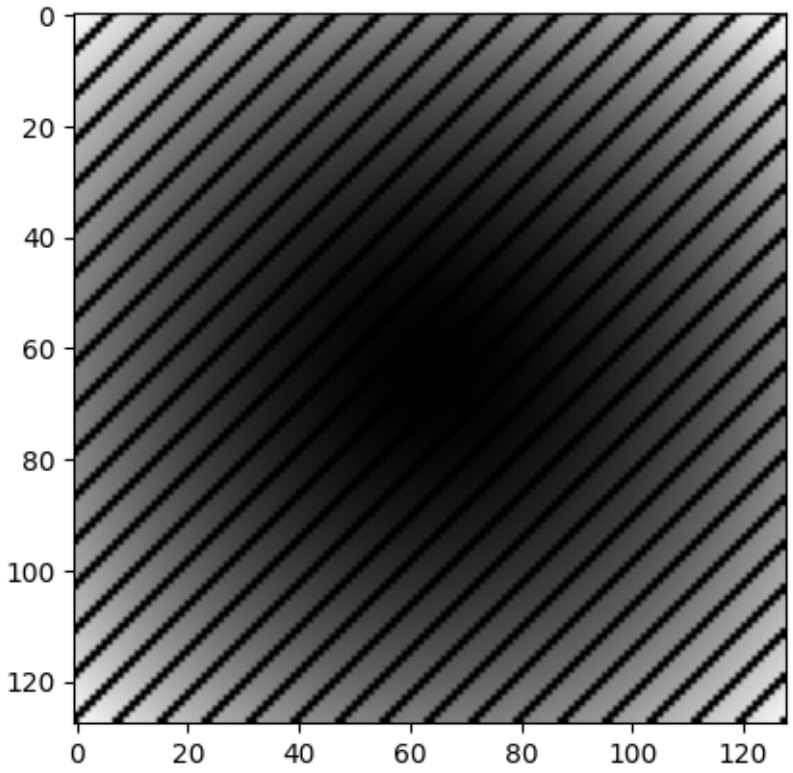


[17]:
```
# ou encore l'opérateur ~
# qui fait un not bitwise

anti_rayures = ~rayures
plt.imshow(anti_rayures,
           cmap='gray');
```

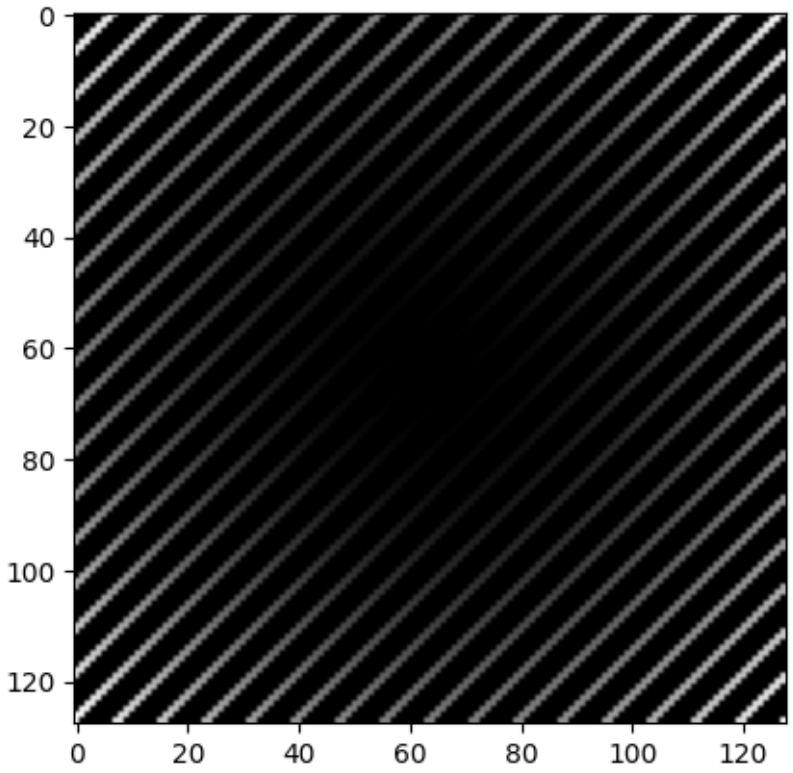


Maintenant je peux utiliser le masque `rayures` pour faire des choses sur l'image. Par exemple simplement :

[18]:
```
# pour effacer les rayures
plt.imshow(image*rayures, cmap='gray');
```



[19]:
```
# ou garder l'autre moitié
plt.imshow(image*anti_rayures, cmap='gray');
```



[20]:
```
image
```

[20]:
```
array([[8192., 8065., 7940., …, 7817., 7940., 8065.],
       [8065., 7938., 7813., …, 7690., 7813., 7938.],
       [7940., 7813., 7688., …, 7565., 7688., 7813.],
       …,
       [7817., 7690., 7565., …, 7442., 7565., 7690.],
       [7940., 7813., 7688., …, 7565., 7688., 7813.],
       [8065., 7938., 7813., …, 7690., 7813., 7938.]])
```

[21]:
```
np.logical_not(image)
```

[21]:
```
array([[False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       …,
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False]])
```

Expression conditionnelle et **np.where**

Nous avons vu en Python natif l'expression conditionnelle :

[22]:
```
3 if True else 2
```

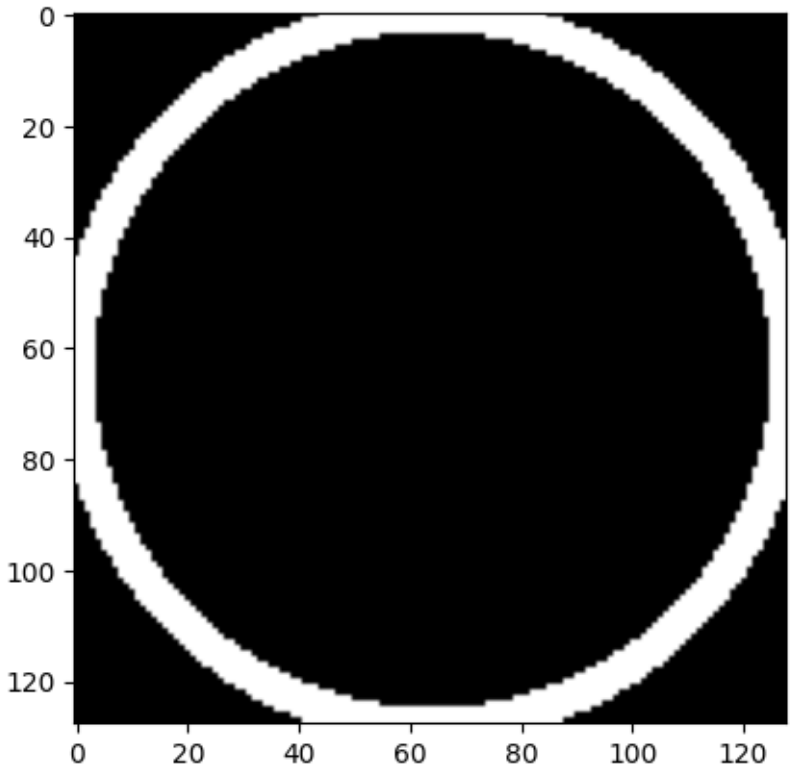
[22]:
```
3
```

Pour reproduire cette construction en **numpy** vous avez à votre disposition **np.where**. Pour l'illustrer nous allons construire deux images facilement discernables. Et, pour cela, on va utiliser **np.isclose**, qui est très utile pour comparer que deux nombres sont suffisamment proches, surtout pour les calculs flottants en fait, mais ça nous convient très bien ici aussi :

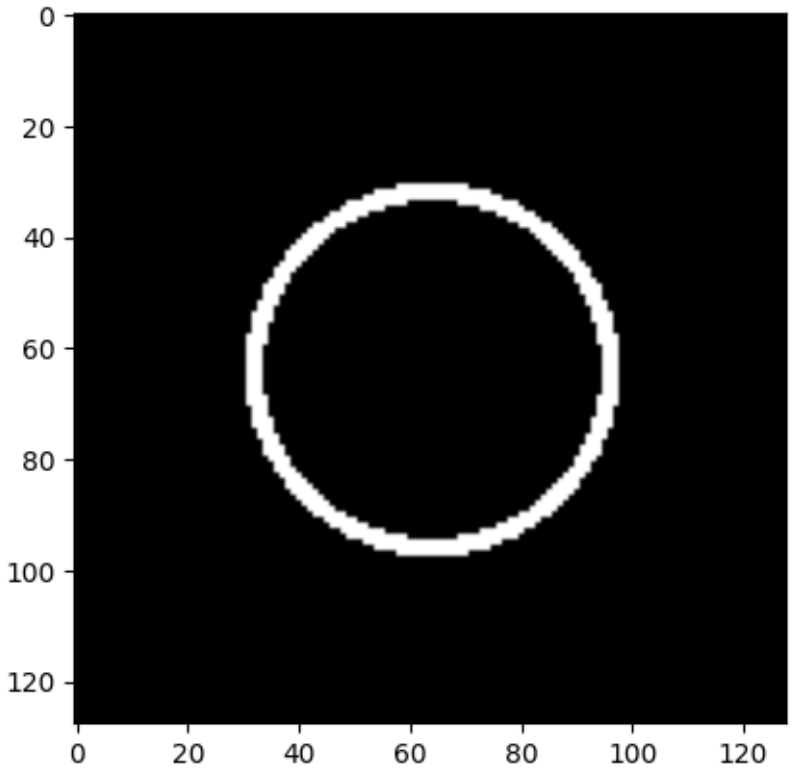
[23]:
```
np.isclose?
```

Pour élaborer une image qui contient un grand cercle, je vais dire que la distance au centre (je rappelle que c'est le contenu de **image**) est suffisamment proche de 64^2, ce que vaut **image** au milieu de chaque bord :

[24]:
```
big_circle = np.isclose(image, 64 **2, 10/100)
plt.imshow(big_circle, cmap='gray');
```



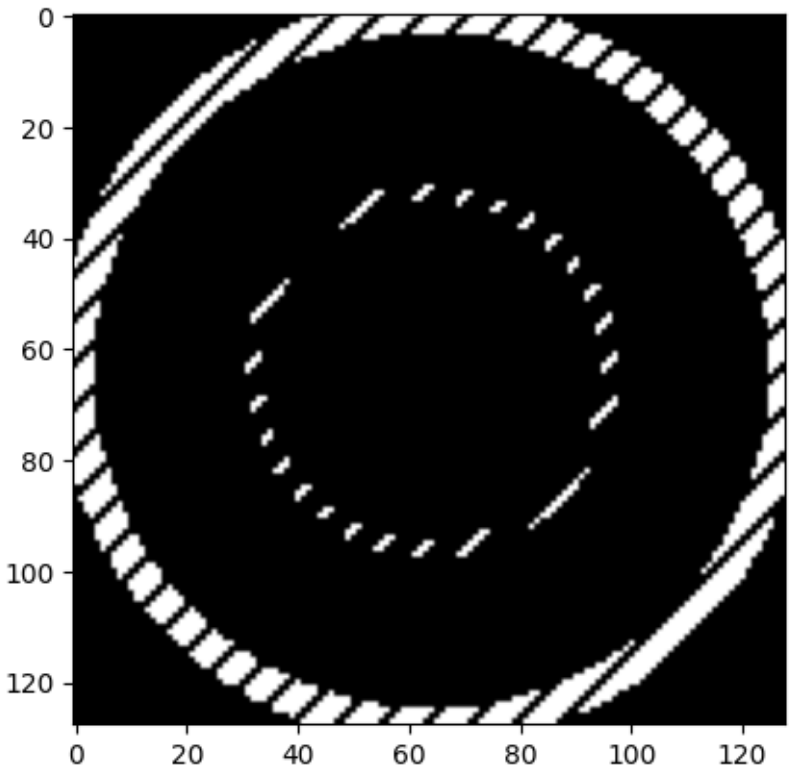
[25]:
```
small_circle = np.isclose(image, 32 **2, 10/100)
plt.imshow(small_circle, cmap='gray');
```



En utilisant `np.where`, je peux simuler quelque chose comme ceci :

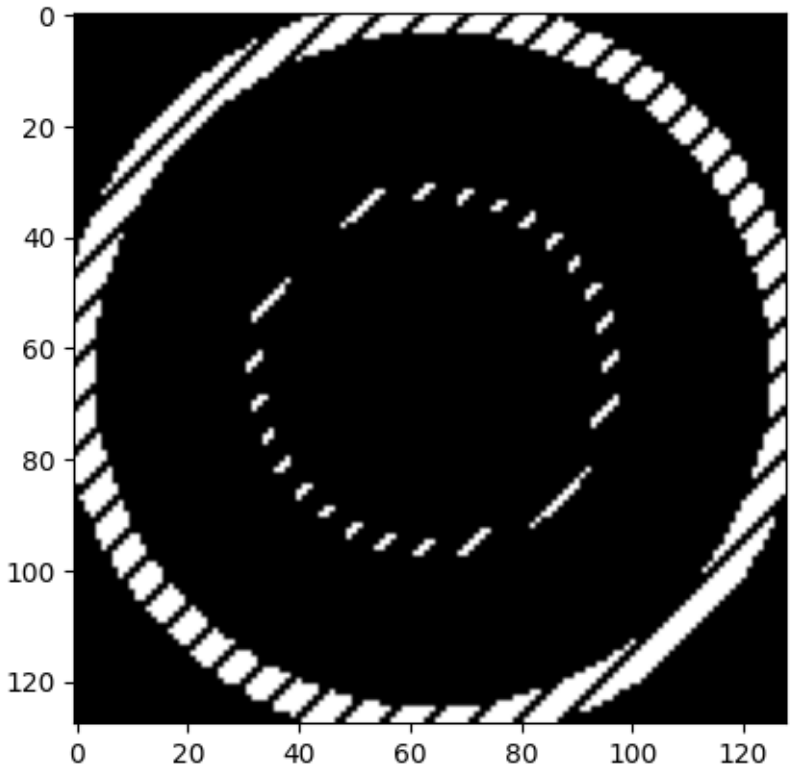
```
mixed = big_circle if rayures else small_circle
```

[26]:
```
# sauf que ça se présente en fait comme ceci :
mixed = np.where(rayures, big_circle, small_circle)
plt.imshow(mixed, cmap='gray');
```



Remarquez enfin qu'on peut aussi faire la même chose en tirant profit que `True` == 1 et `False` == 0 :

[27]:
```
mixed2 = rayures * big_circle + (1-rayures) * small_circle
plt.imshow(mixed2, cmap='gray');
```



7.10 w7-s05-c4-algebre-lineaire

Algèbre linéaire

7.10.1 Complément - niveau basique

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

[1]:
```
<contextlib.ExitStack at 0x10ef7cb20>
```

Un aspect important de l'utilisation de `numpy` consiste à manipuler des matrices et vecteurs. Voici une rapide introduction à ces fonctionnalités.

Produit matriciel - **`np.dot`**

Rappel : On a déjà vu que `*` entre deux tableaux faisait une multiplication terme à terme.

[2]:
```
ligne = 1 + np.arange(3)
print(ligne)
```

```
[1 2 3]
```

[3]:
```
colonne = 1 + np.arange(3).reshape(3, 1)
print(colonne)
```

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

Ce n'est pas ce que l'on veut ici !

[4]:
```
# avec le broadcasting, numpy me laisse écrire ceci
# mais **ce n'est pas** un produit matriciel
print(ligne * colonne)
```

```
[[1 2 3]
 [2 4 6]
 [3 6 9]]
```

L'opération de produit matriciel s'appelle `np.dot` :

[5]:
```
m1 = np.array([[1, 1],
               [2, 2]])
print(m1)
```

```
[[1 1]
 [2 2]]
```

[6]:
```
m2 = np.array([[10, 20],
               [30, 40]])
print(m2)
```

```
[[10 20]
 [30 40]]
```

[7]:
```
# comme fonction
np.dot(m1, m2)
```

[7]:
```
array([[ 40,  60],
       [ 80, 120]])
```

[8]:
```
# comme méthode
m1.dot(m2)
```

[8]:
```
array([[ 40,  60],
       [ 80, 120]])
```

Je vous signale aussi un opérateur spécifique, noté `@`, qui permet également de faire le produit matriciel.

[9]:
```
m1 @ m2
```

[9]:
```
array([[ 40,  60],
       [ 80, 120]])
```

[10]:
```
m2 @ m1
```

[10]:
```
array([[ 50,  50],
       [110, 110]])
```

C'est un opérateur un peu ad hoc pour **numpy**, puisqu'il ne fait pas de sens avec les types usuels de Python :

[11]:
```
for x, y in ( (10, 20), (10., 20.), ([10], [20]), ((10,), (20,))):
    try:
        x @ y
    except Exception as e:
        print(f"OOPS - {type(e)} - {e}")
```

```
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'int' and '
  int'
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'float' and
   'float'
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'list' and
  'list'
OOPS - <class 'TypeError'> - unsupported operand type(s) for @: 'tuple' and
   'tuple'
```

Produit scalaire - **np.dot** ou **@**

Ici encore, vous pouvez utiliser `dot` qui va intelligemment transposer le second argument :

[12]:
```
v1 = np.array([1, 2, 3])
print(v1)
```

```
[1 2 3]
```

[13]:
```
v2 = np.array([4, 5, 6])
print(v2)
```

```
[4 5 6]
```

[14]:
```
np.dot(v1, v2)
```

[14]: 32

[15]:
```
v1 @ v2
```

[15]: 32

Transposée

Vous pouvez accéder à une matrice transposée de deux façons :

— soit sous la forme d'un attribut `m.T` :

[16]:
```
m = np.arange(4).reshape(2, 2)
print(m)
```

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

[17]:
```
print(m.T)
```

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

— soit par la méthode `transpose()` :

[18]:
```
print(m)
```

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

[19]:
```
m.transpose()
```

[19]:
```
array([[0, 2],
       [1, 3]])
```

Matrice identité - **np.eye**

[20]:
```
np.eye(4, dtype=np.int8)
```

[20]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]], dtype=int8)
```

Matrices diagonales - **np.diag**

Avec `np.diag`, vous pouvez dans les deux sens :

— extraire la diagonale d'une matrice ;
— construire une matrice à partir de sa diagonale.

[21]:
```
M = np.arange(4) + 10 * np.arange(4)[:, np.newaxis]
print(M)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]]
```

[22]:
```
D = np.diag(M)
print(D)
```

```
[ 0 11 22 33]
```

[23]:
```
M2 = np.diag(D)
print(M2)
```

```
[[ 0  0  0  0]
 [ 0 11  0  0]
 [ 0  0 22  0]
 [ 0  0  0 33]]
```

Déterminant - **np.linalg.det**

Avec la fonction np.linalg.det :

[24]:
```
# une isométrie
M = np.array([[0, -1], [1, 0]])
print(M)
```

```
[[ 0 -1]
 [ 1  0]]
```

[25]:
```
# et donc
np.linalg.det(M) == 1
```

[25]:
```
True
```

Valeurs propres - **np.linalg.eig**

Vous pouvez obtenir valeurs propres et vecteurs propres d'une matrice avec np.eig :

[26]:
```
# la symétrie par rapport à x=y
S = np.array([[0, 1], [1, 0]])
```

[27]:
```
values, vectors = np.linalg.eig(S)
```

[28]:
```
# pas de déformation
values
```

[28]:
```
array([ 1., -1.])
```

[29]:
```
# les deux diagonales
vectors
```

[29]:
```
array([[ 0.70710678, -0.70710678],
       [ 0.70710678,  0.70710678]])
```

Systèmes d'équations - **np.linalg.solve**

Fabriquons-nous un système d'équations :

[30]:
```
x, y, z = 1, 2, 3
```

[31]:
```
3*x + 2*y + z
```

[31]:
```
10
```

[32]:
```
2*x + 3*y +4*z
```

[32]:
```
20
```

[33]:
```
5*x + 2*y + 6*z
```

[33]:
```
27
```

On peut le résoudre tout simplement comme ceci :

[34]:
```
coefficients= np.array([
    [3, 2, 1],
    [2, 3, 4],
    [5, 2, 6],
])
```

[35]:
```
constants = [
    10,
    20,
    27,
]
```

[36]:
```
X, Y, Z = np.linalg.solve(coefficients, constants)
```

Par contre bien sûr on est passé par les flottants, et donc on a le souci habituel avec la précision des arrondis :

[37]:
```
Z
```

[37]:
```
3.0000000000000004
```

Résumé

En résumé, ce qu'on vient de voir :

outil	propos
`np.dot`	produit matriciel
`np.dot`	produit scalaire
`np.transpose`	transposée
`np.eye`	matrice identité
`np.diag`	extrait la diagonale
`np.diag`	ou construit une matrice diagonale
`np.linalg.det`	déterminant
`np.linalg.eig`	valeurs propres
`np.linalg.solve`	résout système équations

Pour en savoir plus

Voyez la documentation complète sur l'algèbre linéaire.

7.11 w7-s05-c5-indexation-evoluee

Indexation évoluée

7.11.1 Complément - niveau avancé

Nous allons maintenant voir qu'il est possible d'indexer un tableau `numpy` avec, non pas des entiers ou des tuples comme on l'a vu dans un complément précédent, mais aussi avec d'autres types d'objets qui permettent des manipulations très puissantes :

— indexation par une liste ;

— indexation par un tableau ;
— indexation multiple (par un tuple) ;
— indexation par un tableau de booléens.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

[1]:
```
<contextlib.ExitStack at 0x10e939270>
```

Pour illustrer ceci, on va réutiliser la fonction `background` que l'on avait vue pour les indexations simples :

[2]:
```
# une fonction qui crée un tableau
# tab[i, j] = i + 10 * j
def background(n):
    i = np.arange(n)
    j = i.reshape((n, 1))
    return i + 10 * j
```

Indexation par une liste

On peut indexer par une liste d'entiers, cela constitue une généralisation des slices.

[3]:
```
b = background(6)
print(b)
```

```
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
```

Si je veux référencer les lignes 1, 3 et 4, je ne peux pas utiliser un slice ; mais je peux utiliser une liste à la place :

[4]:
```
# il faut lire ceci comme
# j'indexe b, avec comme indice la liste [1, 3, 4]
b[[1, 3, 4]]
```

[4]:
```
array([[10, 11, 12, 13, 14, 15],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45]])
```

[5]:
```
# pareil pour les colonnes, en combinant avec un slice
b[:, [1, 3, 4]]
```

[5]:
```
array([[ 1,  3,  4],
       [11, 13, 14],
       [21, 23, 24],
       [31, 33, 34],
       [41, 43, 44],
       [51, 53, 54]])
```

[6]:
```
# et comme toujours on peut faire du broadcasting
b[:, [1, 3, 4]] = np.arange(1000, 1006).reshape((6, 1))
print(b)
```

```
[[   0 1000    2 1000 1000    5]
 [  10 1001   12 1001 1001   15]
 [  20 1002   22 1002 1002   25]
 [  30 1003   32 1003 1003   35]
 [  40 1004   42 1004 1004   45]
 [  50 1005   52 1005 1005   55]]
```

Indexation par un tableau

On peut aussi indexer un tableau A … par un tableau ! Pour que cela ait un sens :

— le tableau d'index doit contenir des entiers ;
— ces derniers doivent être tous plus petits que la première dimension de A.

Le cas simple : l'entrée et l'index sont de dimension 1.

[7]:
```
# le tableau qu'on va indexer
cubes = np.arange(10) ** 3
print(cubes)
```

```
[  0   1   8  27  64 125 216 343 512 729]
```

[8]:
```
# et un index qui est un tableau numpy
# doit contenir des entiers entre 0 et 9
tab = np.array([1, 7, 2])
print(cubes[tab])
```

```
[  1 343   8]
```

[9]:
```
# donne - logiquement - le même résultat que
# si l'index était une liste Python
lis = [1, 7, 2]
print(cubes[lis])
```

```
[  1 343   8]
```

De manière générale Dans le cas général, le résultat de `A[index]` :

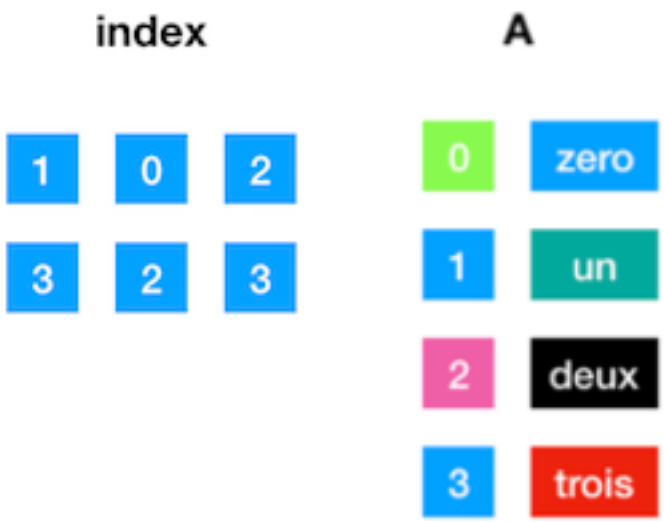
— a la même forme "externe" que `index` ;
— où l'on a remplacé `i` par `A[i]` ;
— qui peut donc être un tableau si `A` est de dimension > 1

[10]:
```
A = np.array([[0, 'zero'], [1, 'un'], [2, 'deux'], [3, 'trois']])
print(A)
```

```
[['0' 'zero']
 ['1' 'un']
 ['2' 'deux']
 ['3' 'trois']]
```

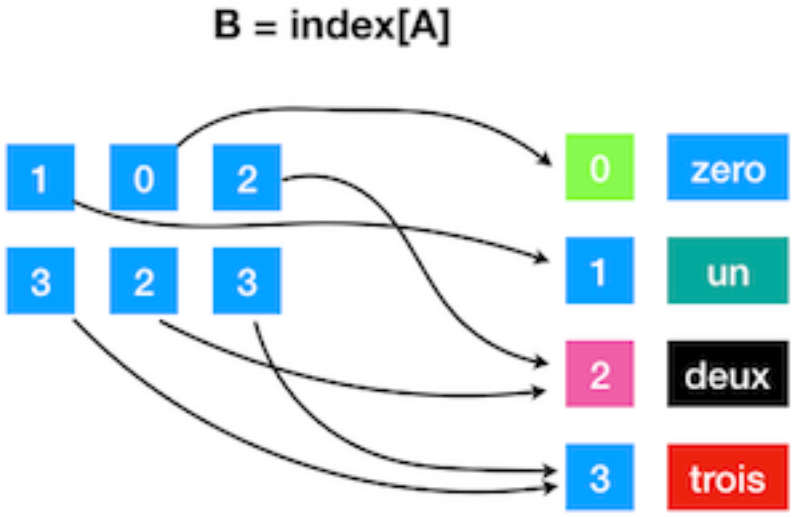
[11]:
```
index = np.array([[1, 0, 2], [3, 2, 3]])
print(index)
```

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



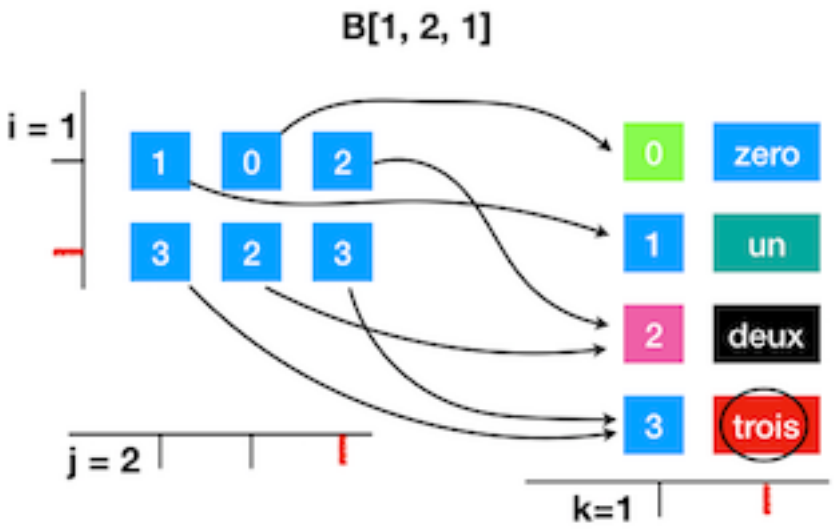
[12]:
```
B = A[index]
print(B)
```

```
[[['1' 'un']
  ['0' 'zero']
  ['2' 'deux']]

 [['3' 'trois']
  ['2' 'deux']
  ['3' 'trois']]]
```



[13]:
```
B[1, 2, 1]
```

[13]:
```
'trois'
```



Et donc si :

- `index` est de dimension `(i, j, k)` ;
- `A` est de dimension `(a, b)`.

Alors :

- `A[index]` est de dimension `(i, j, k, b)` ;
- il faut que les éléments dans `index` soient dans `[0 .. a[`.

Ce que l'on vérifie ici :

[14]:
```
# l'entrée
print(A.shape)
```

```
(4, 2)
```

[15]:
```
# l'index
print(index.shape)
```

```
(2, 3)
```

[16]:
```
# le résultat
print(A[index].shape)
```

```
(2, 3, 2)
```

Cas particulier : entrée de dimension 1, **`index`** de dim. > 1 Lorsque l'entrée `A` est de dimension 1, alors la sortie a exactement la même forme que l'`index`.

C'est comme si `A` était une fonction que l'on applique aux indices dans `index`.

[17]:
```
print(cubes)
```

```
[  0   1   8  27  64 125 216 343 512 729]
```

[18]:
```
i2 = np.array([[2, 4], [8, 9]])
print(i2)
```

```
[[2 4]
 [8 9]]
```

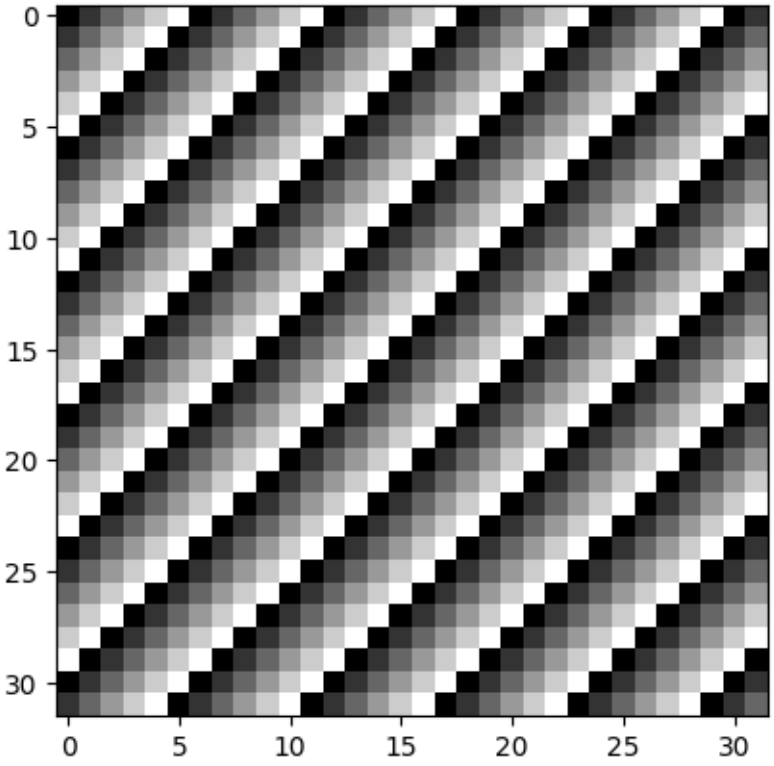
[19]:
```
print(cubes[i2])
```

```
[[  8  64]
 [512 729]]
```

Application au codage des couleurs dans une image

[20]:
```
# je crée une image avec 6 valeurs disposées en diagonale
N = 32
colors = 6

image = np.empty((N, N), dtype = np.int32)
for i in range(N):
    for j in range(N):
        image[i, j] = (i+j) % colors
```

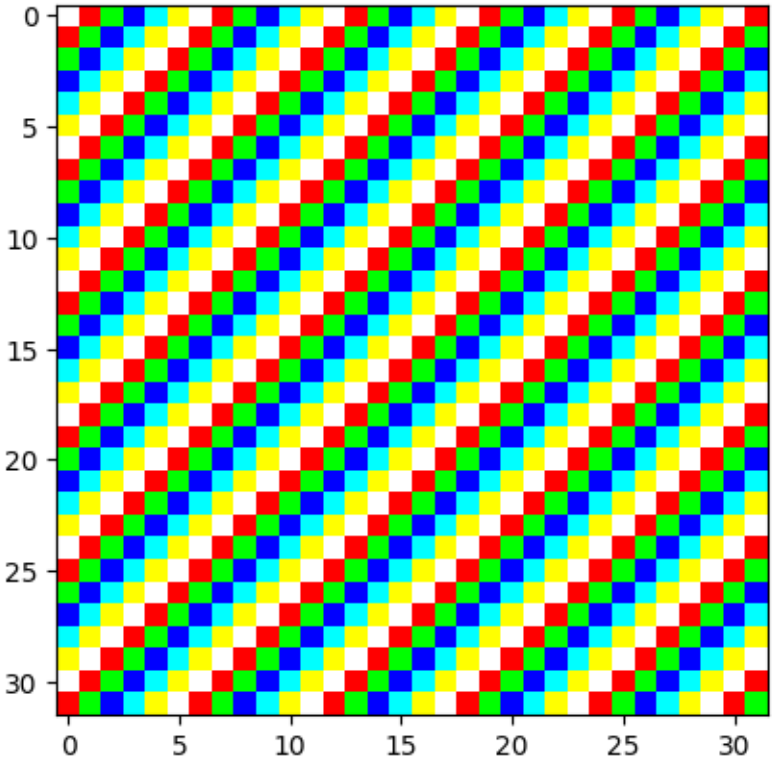
[21]:
```
plt.imshow(image, cmap='gray');
```



Les couleurs ne sont pas significatives, ce sont des valeurs entières dans `range(colors)`. On voudrait pouvoir choisir la vraie couleur correspondant à chaque valeur. Pour cela on peut utiliser une simple indexation par tableau :

[22]:
```
# une palette de couleurs
palette = np.array([
  [255, 255, 255], # 0 -> blanc
  [255, 0, 0],     # 1 -> rouge
  [0, 255, 0],     # 2 -> vert
  [0, 0, 255],     # 3 -> bleu
  [0, 255, 255],   # 4 -> cyan
  [255, 255, 0],   # 5 -> magenta
 ], dtype=np.uint8)
```

[23]:
```
plt.imshow(palette[image]);
```



Remarquez que la forme générale n'a pas changé, mais le résultat de l'indexation a une dimension supplémentaire de 3 couleurs :

[24]:
```
image.shape
```

[24]:
```
(32, 32)
```

[25]:
```
palette[image].shape
```

[25]:
```
(32, 32, 3)
```

Indexation multiple (par tuple)

Une fois que vous avez compris ce mécanisme d'indexation par un tableau, on peut encore généraliser pour définir une indexation par deux (ou plus) tableaux de formes identiques.

Ainsi, lorsque `index1` et `index2` ont la même forme :

- on peut écrire `A[index1, index2]`
- qui a la même forme externe que les `index`
- où on a remplacé `i, j` par `A[i][j]`
- qui peut donc être un tableau si `A` est de dimension > 2.

[26]:
```
# un tableau à indexer
ix, iy = np.indices((4, 3))
A = 10 * ix + iy
print(A)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

[27]:
```
# les deux tableaux d'indices sont carrés 2x2
index1 = [[3, 1], [0, 1]]  # doivent être < 4
index2 = [[2, 0], [0, 2]]  # doivent être < 3
# le résultat est donc carré 2x2
print(A[index1, index2])
```

```
[[32 10]
 [ 0 12]]
```

Et donc si :

- `index1` et `index2` sont de dimension `(i, j, k)`
- et `A` est de dimension `(a, b, c)`

Alors :

- le résultat est de dimension `(i, j, k, c)`
- il faut alors que les éléments de `index1` soient dans `[0 .. a[`
- et les éléments de `index2` dans `[0 .. b[`

Application à la recherche de maxima Imaginons que vous avez des mesures pour plusieurs instants :

[28]:
```
times = np.linspace(1000, 5000, num=5, dtype=int)
print(times)
```

```
[1000 2000 3000 4000 5000]
```

[29]:
```
# on aurait 3 mesures à chaque instant
series = np.array([
    [10, 25, 32, 23, 12],
    [12, 8, 4, 10, 7],
    [100, 80, 90, 110, 120]])
print(series)
```

```
[[ 10  25  32  23  12]
 [ 12   8   4  10   7]
 [100  80  90 110 120]]
```

Avec la fonction `np.argmax` on peut retrouver les indices des points maxima dans `series` :

[30]:
```
max_indices = np.argmax(series, axis=1)
print(max_indices)
```

```
[2 0 4]
```

Pour trouver les maxima en question, on peut faire :

[31]:
```
# les trois maxima, un par serie
maxima = series[ range(series.shape[0]), max_indices ]
print(maxima)
```

```
[ 32  12 120]
```

[32]:
```
# et ils correspondent à ces instants-ci
times[max_indices]
```

[32]:
```
array([3000, 1000, 5000])
```

Indexation par un tableau de booléens

Une forme un peu spéciale d'indexation consiste à utiliser un tableau de booléens, qui agit comme un masque :

[33]:
```
suite = np.array([1, 2, 3, 4, 5, 4, 3, 2, 1])
```

Je veux filtrer ce tableau et ne garder que les valeurs < 4 :

[34]:
```
# je construis un masque
hauts = suite >= 4
print(hauts)
```

```
[False False False  True  True  True False False False]
```

[35]:
```
# je peux utiliser ce masque pour calculer les indices qui sont vrais
suite[hauts]
```

[35]:
```
array([4, 5, 4])
```

[36]:
```
# et utiliser maintenant ceci par un index de tableau
# par exemple pour annuler ces valeurs
suite[hauts] = 0
print(suite)
```

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

7.12 w7-s05-c6-divers

Divers

7.12.1 Complément - niveau avancé

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

[1]:
```
<contextlib.ExitStack at 0x10df98880>
```

Pour finir notre introduction à `numpy`, nous allons survoler à très grande vitesse quelques traits plus annexes mais qui peuvent être utiles. Je vous laisse approfondir de votre côté les parties qui vous intéressent.

7.13 w7-s05-c6-divers

Utilisation de la mémoire

Références croisées, vues, shallow et deep copies

Pour résumer ce qu'on a vu jusqu'ici :

- un tableau `numpy` est un objet mutable ;
- une slice sur un tableau retourne une vue, on est donc dans le cas d'une référence partagée ;
- dans tous les cas que l'on a vus jusqu'ici, comme les cases des tableaux sont des objets atomiques, il n'y a pas de différence entre shallow et deep copie ;
- pour créer une copie, utilisez `np.copy()`.

Et de plus :

[2]:
```
# un tableau de base
a = np.arange(3)
```

[3]:
```
# une vue
v = a.view()
```

[4]:
```
# une slice
s = a[:]
```

Les deux objets ne sont pas différentiables :

[5]:
```
v.base is a
```

[5]:
```
True
```

[6]:
```
s.base is a
```

[6]:
```
True
```

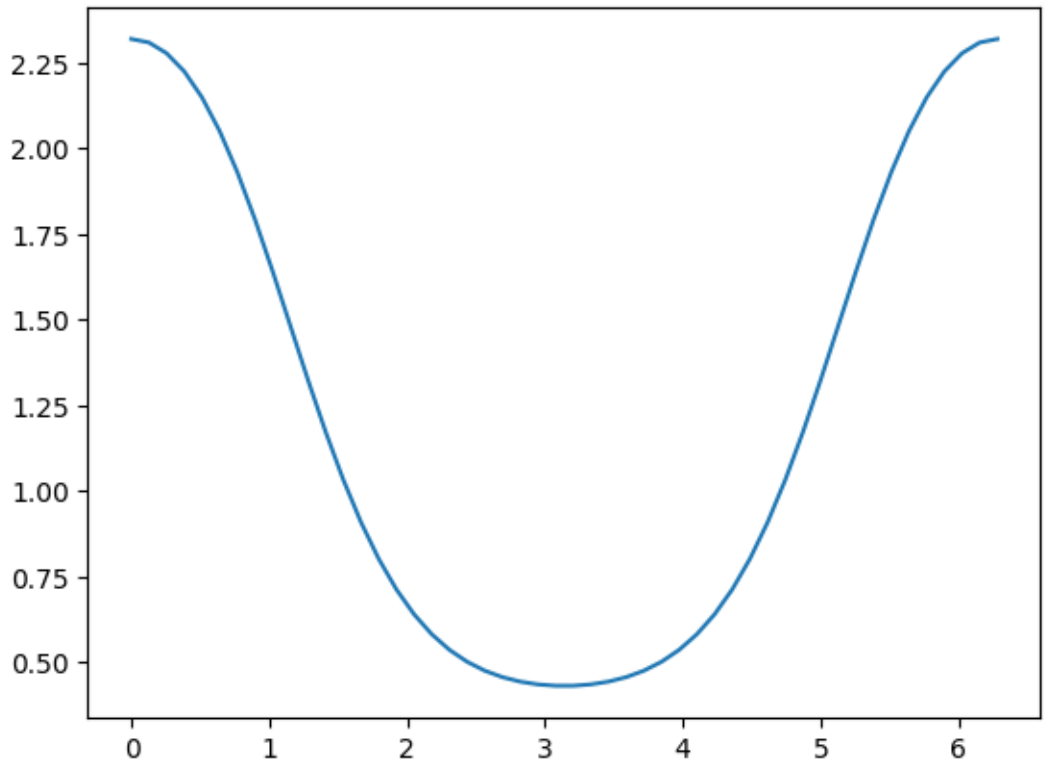
L'option `out=`

Lorsque l'on fait du calcul vectoriel, on peut avoir tendance à créer de nombreux tableaux intermédiaires qui coûtent cher en mémoire. Pour cette raison, presque tous les opérateurs `numpy` proposent un paramètre optionnel `out=` qui permet de spécifier un tableau déjà alloué, dans lequel ranger le résultat.

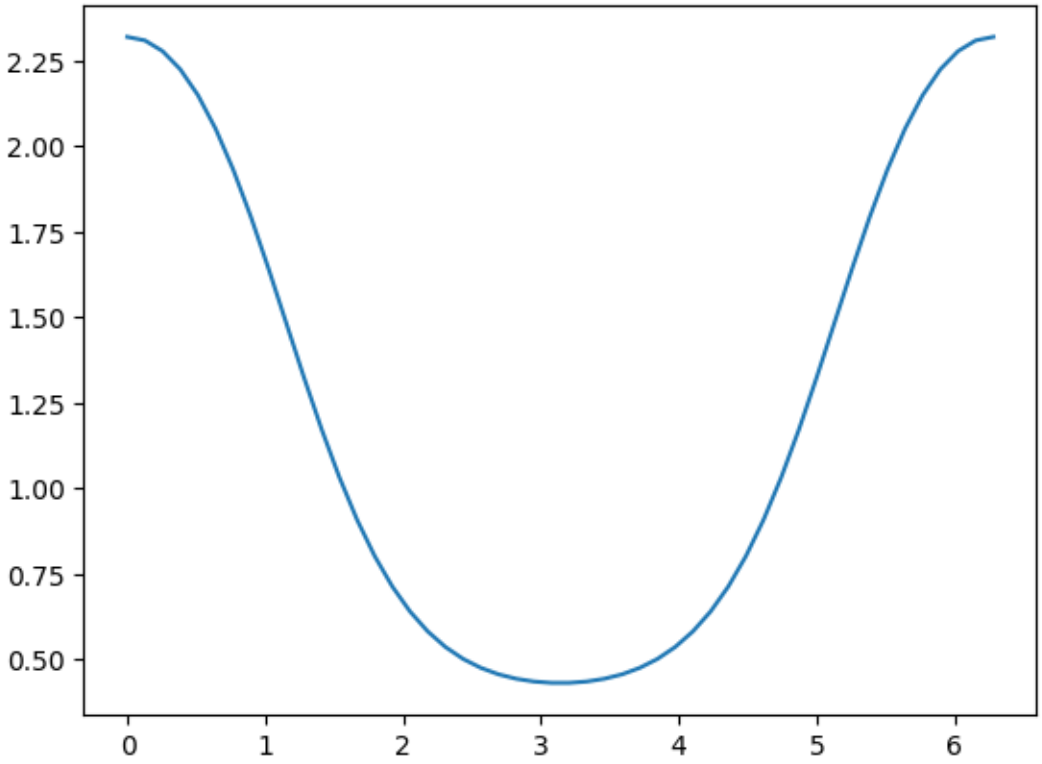
Prenons l'exemple un peu factice suivant, ou on calcule $e^{sin(cos(x))}$ sur l'intervalle $[0, 2\pi]$:

[7]:
```
# le domaine
X = np.linspace(0, 2*np.pi)
```

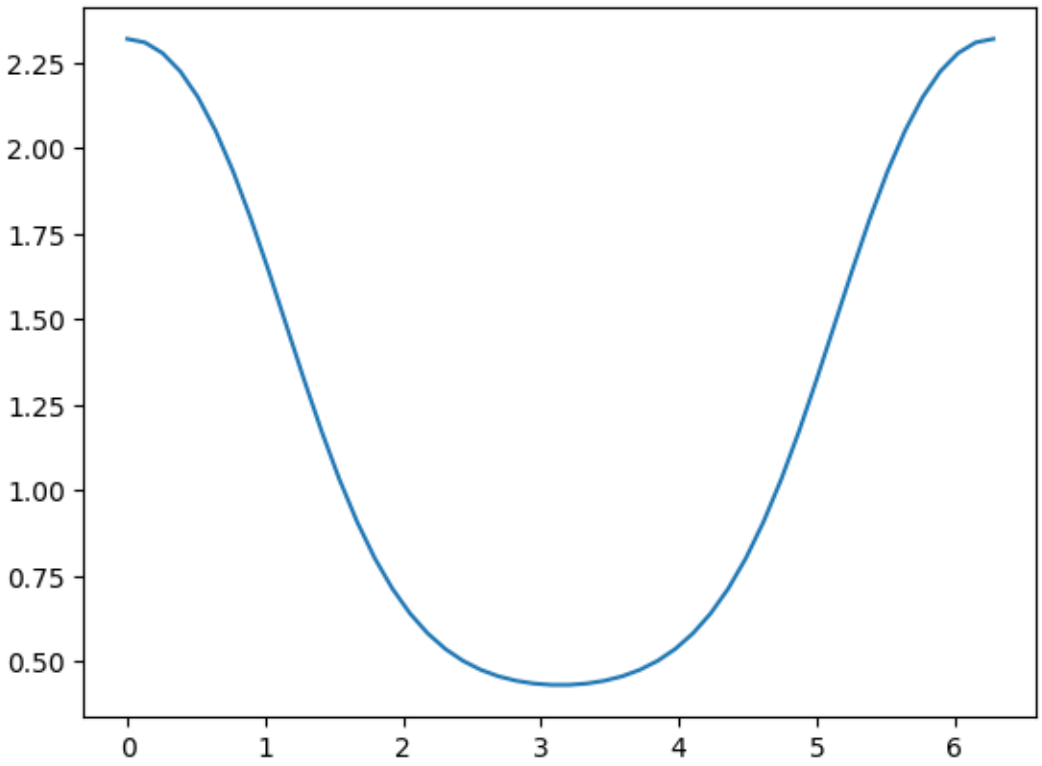
[8]:
```
Y = np.exp(np.sin(np.cos(X)))
plt.plot(X, Y);
```



[9]:
```
# chaque fonction alloue un tableau pour ranger ses résultats,
# et si je décompose, ce qui se passe en fait c'est ceci
Y1 = np.cos(X)
Y2 = np.sin(Y1)
Y3 = np.exp(Y2)
# en tout en comptant X et Y j'aurai créé 4 tableaux
plt.plot(X, Y3);
```



[10]:
```
# Mais moi je sais qu'en fait je n'ai besoin que de X et de Y
# ce qui fait que je peux optimiser comme ceci :

# je ne peux pas récrire sur X parce que j'en aurai besoin pour le plot
X1 = np.cos(X)
# par contre ici je peux recycler X1 sans souci
np.sin(X1, out=X1)
# etc ...
np.exp(X1, out=X1)
plt.plot(X, X1);
```



Et avec cette approche je n'ai créé que 2 tableaux en tout.

Notez bien : je ne vous recommande pas d'utiliser ceci systématiquement, car ça défigure nettement le code. Mais il faut savoir que ça existe, et savoir y penser lorsque la création de tableaux intermédiaires a un coût important dans l'algorithme.

`np.add` et similaires

Si vous vous mettez à optimiser de cette façon, vous utiliserez par exemple `np.add` plutôt que `+`, qui ne vous permet pas de choisir la destination du résultat.

7.14 w7-s05-c6-divers

Types structurés pour les cellules

Sans transition, jusqu'ici on a vu des tableaux atomiques, où chaque cellule est en gros un seul nombre.

En fait, on peut aussi se définir des types structurés, c'est-à-dire que chaque cellule contient l'équivalent d'un struct en C.

Pour cela, on peut se définir un `dtype` élaboré, qui va nous permettre de définir la structure de chacun de ces enregistrements.

Exemple

[11]:
```
# un dtype structuré
my_dtype = [
    # prenom est un string de taille 12
    ('prenom', '|S12'),
    # nom est un string de taille 15
```

```
    ('nom', '|S15'),
    # age est un entier
    ('age', int)
]

# un tableau qui contient des cellules de ce type
classe = np.array(
    # le contenu
    [ ( 'Jean', 'Dupont', 32),
      ( 'Daniel', 'Durand', 18),
      ( 'Joseph', 'Delapierre', 54),
      ( 'Paul', 'Girard', 20)],
    # le type
    dtype = my_dtype)
classe
```

[11]:
```
array([(b'Jean', b'Dupont', 32), (b'Daniel', b'Durand', 18),
       (b'Joseph', b'Delapierre', 54), (b'Paul', b'Girard', 20)],
      dtype=[('prenom', 'S12'), ('nom', 'S15'), ('age', '<i8')])
```

Je peux avoir l'impression d'avoir créé un tableau de 4 lignes et 3 colonnes ; cependant pour **numpy** ce n'est pas comme ça que cela se présente :

[12]:
```
classe.shape
```

[12]:
```
(4,)
```

Rien ne m'empêcherait de créer des tableaux de ce genre en dimensions supérieures, bien entendu :

[13]:
```
# ça n'a pas beaucoup d'intérêt ici, mais si on en a besoin
# on peut bien sûr avoir plusieurs dimensions
classe.reshape((2, 2))
```

[13]:
```
array([[(b'Jean', b'Dupont', 32), (b'Daniel', b'Durand', 18)],
       [(b'Joseph', b'Delapierre', 54), (b'Paul', b'Girard', 20)]],
      dtype=[('prenom', 'S12'), ('nom', 'S15'), ('age', '<i8')])
```

Comment définir **dtype** ?

Il existe une grande variété de moyens pour se définir son propre **dtype**.

Je vous signale notamment la possibilité de spécifier à l'intérieur d'un **dtype** des cellules de type **object**, qui est l'équivalent d'une référence Python (approximativement, un pointeur dans un struct C) ; c'est un trait qui est utilisé par **pandas** que nous allons voir très bientôt.

Pour la définition de types structurés, voir la documentation complète ici.

7.15 w7-s05-c6-divers

Assemblages et découpages

Enfin, toujours sans transition, et plus anecdotique : jusqu'ici nous avons vu des fonctions qui préservent la taille. Le stacking permet de créer un tableau plus grand en (juxta/super)posant plusieurs tableaux. Voici rapidement quelques fonctions qui permettent de faire des tableaux plus petits ou plus grands.

Assemblages : **hstack** et **vstack** (tableaux 2D)

[14]:
```
a = np.arange(1, 7).reshape(2, 3)
print(a)
```

```
[[1 2 3]
 [4 5 6]]
```

[15]:
```
b = 10 * np.arange(1, 7).reshape(2, 3)
print(b)
```

```
[[10 20 30]
 [40 50 60]]
```

[16]:
```
print(np.hstack((a, b)))
```

```
[[ 1  2  3 10 20 30]
 [ 4  5  6 40 50 60]]
```

[17]:
```
print(np.vstack((a, b)))
```

```
[[ 1  2  3]
 [ 4  5  6]
 [10 20 30]
 [40 50 60]]
```

Assemblages : **np.concatenate** (3D et au delà)

[18]:
```
a = np.ones((2, 3, 4))
print(a)
```

```
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
```

[19]:
```
b = np.zeros((2, 3, 2))
print(b)
```

```
[[[0. 0.]
  [0. 0.]
  [0. 0.]]

 [[0. 0.]
  [0. 0.]
  [0. 0.]]]
```

[20]:
```
print(np.concatenate((a, b), axis = 2))
```

```
[[[1. 1. 1. 1. 0. 0.]
  [1. 1. 1. 1. 0. 0.]
```

```
  [1. 1. 1. 1. 0. 0.]]

 [[1. 1. 1. 1. 0. 0.]
  [1. 1. 1. 1. 0. 0.]
  [1. 1. 1. 1. 0. 0.]]]
```

Pour conclure :

— `hstack` et `vstack` utiles sur des tableaux 2D ;
— au-delà, préférez `concatenate` qui a une sémantique plus claire.

Répétitions : **`np.tile`**

Cette fonction permet de répéter un tableau dans toutes les directions :

[21]:
```
motif = np.array([[0, 1], [2, 10]])
print(motif)
```

```
[[ 0  1]
 [ 2 10]]
```

[22]:
```
print(np.tile(motif, (2, 3)))
```

```
[[ 0  1  0  1  0  1]
 [ 2 10  2 10  2 10]
 [ 0  1  0  1  0  1]
 [ 2 10  2 10  2 10]]
```

Découpage : **`np.split`**

Cette opération, inverse du stacking, consiste à découper un tableau en parties plus ou moins égales :

[23]:
```
complet = np.arange(24).reshape(4, 6); print(complet)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

[24]:
```
h1, h2 = np.hsplit(complet, 2)
print(h1)
```

```
[[ 0  1  2]
 [ 6  7  8]
 [12 13 14]
 [18 19 20]]
```

[25]:
```
print(h2)
```

```
[[ 3  4  5]
 [ 9 10 11]
 [15 16 17]
 [21 22 23]]
```

[26]:
```
complet = np.arange(24).reshape(4, 6)
print(complet)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

[27]:
```
v1, v2 = np.vsplit(complet, 2)
print(v1)
```

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

[28]:
```
print(v2)
```

```
[[12 13 14 15 16 17]
 [18 19 20 21 22 23]]
```

7.16 w7-s05-x1-checkers

Exercice - niveau basique

7.16.1 construire un tableau en damier

On vous demande d'écrire une fonction `checkers` qui crée un tableau `numpy`.

La fonction prend en argument :

- un entier `size` >= 1
- et un booléen `corner_0_0` - qui vaut par défaut `True`

Elle construit et retourne alors un tableau carré de taille `size` x `size`, qui est rempli comme un damier avec des entiers 0 et 1 ; la valeur de la cellule d'indice 0 x 0 est correspond au paramètre `corner_0_0`.

On rappelle par ailleurs que `False == 0` et `True == 1`.

[1]:
```
import numpy as np

from corrections.exo_checkers import exo_checkers

# voici deux exemples pour la fonction checkers
exo_checkers.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:medium;"\'>appel</span
   >', _dom_classes=('header',)), HTML…
```

[2]:
```
# à vous de jouer
def checkers(size, corner_0_0=True):
    return "votre code"

# NOTE:
# auto-exec-for-latex has used hidden code instead
```

[ ]:
```
# pour corriger votre code
exo_checkers.correction(checkers)

# NOTE
```

```
# auto-exec-for-latex has skipped execution of this cell
```

Visualisation

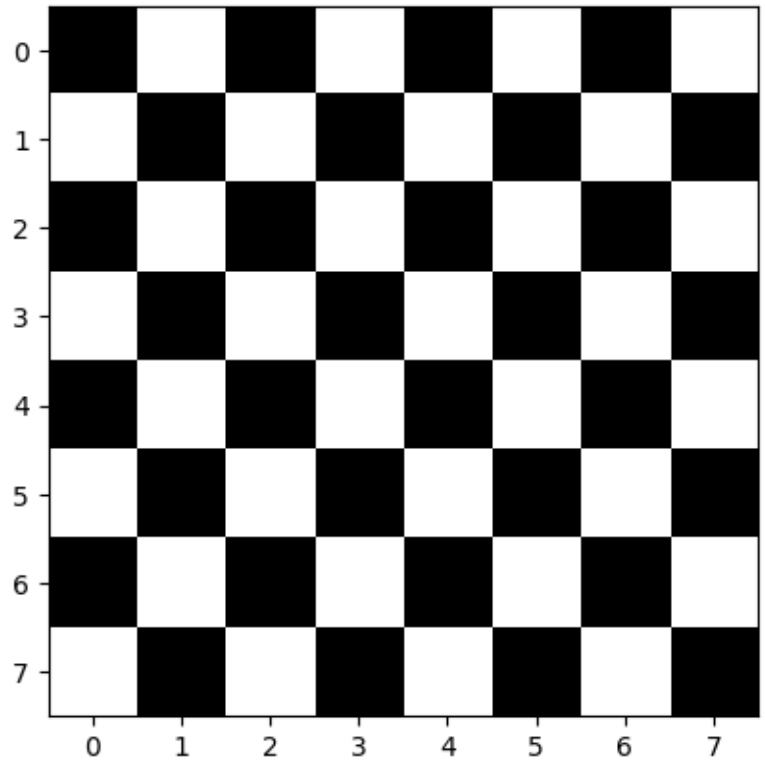
[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion();
```

L'exercice est terminé, mais si vous avez réussi et que vous voulez visualisez le résultat, voici comment vous pouvez aussi voir ce type de tableau :

[4]:
```
checkerboard = checkers(8, False)
```

Pour le voir comme une image :

[5]:
```
# convertir en flottant pour imshow
checkerboard = checkerboard.astype(float)
# afficher avec une colormap 'gray' pour avoir du noir et blanc
plt.imshow(checkerboard, cmap='gray');
```



7.17 w7-s05-x2-hundreds

Exercice - niveau intermédiaire

7.17.1 construire un tableau $100 * i + 10 * j$

On vous demande d'écrire une fonction `hundreds` qui crée un tableau `numpy`.

La fonction prend en argument :

— deux entiers `lines, columns`
— un nombre entier offset

Le résultat doit être un tableau de taille `lines` x `columns`, composé d'entiers, et on veut qu'en une case de coordonnées (i, j) la valeur du tableau soit égale à

$$result[i,j] = 100 * i + 10 * j + offset$$

[1]:
```
import numpy as np
from corrections.exo_hundreds import exo_hundreds

# voici deux exemples pour la fonction hundreds
exo_hundreds.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:medium;"\'>appel</span
  >', _dom_classes=('header',)), HTML…
```

[2]:
```
# à vous de jouer
def hundreds(lines, columns, offset):
    return "votre code"

# NOTE:
# auto-exec-for-latex has used hidden code instead
```

[ ]:
```
# pour corriger votre code
exo_hundreds.correction(hundreds)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.17.2 Plusieurs angles possibles

— la première idée peut-être, consiste à faire deux boucles imbriquées
c'est facile à écrire, ça fonctionne, mais ce n'est pas très élégant
et surtout très inefficace, je vous invite à éviter cette approche

— vous pouvez aussi penser à utiliser du broadcasting
en fabricant par exemple la souche des lignes et des colonnes
à la main avec `np.arange()`
— si vous regardez `np.indices()`, vous trouverez sans doute une inspiration
— et sans doute d'autres auxquelles je n'ai pas pensé :)

7.18 w7-s05-x3-stairs

Exercice - niveau intermédiaire

7.18.1 construire un tableau en escalier

On vous demande d'écrire une fonction `stairs` qui crée un tableau `numpy`.

La fonction prend en argument un entier `taille` et construit un tableau carré de taille $2 * taille + 1$.

Aux quatre coins du tableau on trouve la valeur 0. Dans la case centrale on trouve la valeur $2 * taille$.

Si vous partez de n'importe quelle case et que vous vous déplacez d'une case horizontalement ou verticalement vers une cas plus proche du centre, vous incrémentez la valeur du tableau de 1.

[1]:
```
import numpy as np

from corrections.exo_stairs import exo_stairs

# voici deux exemples pour la fonction stairs
exo_stairs.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:medium;"\'>appel</span
   >', _dom_classes=('header',)), HTML…
```

[2]:
```
# à vous de jouer
def stairs(taille):
    return "votre code"

# NOTE:
# auto-exec-for-latex has used hidden code instead
```

[ ]:
```
# pour corriger votre code
exo_stairs.correction(stairs)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

Visualisation

[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.ion()
```

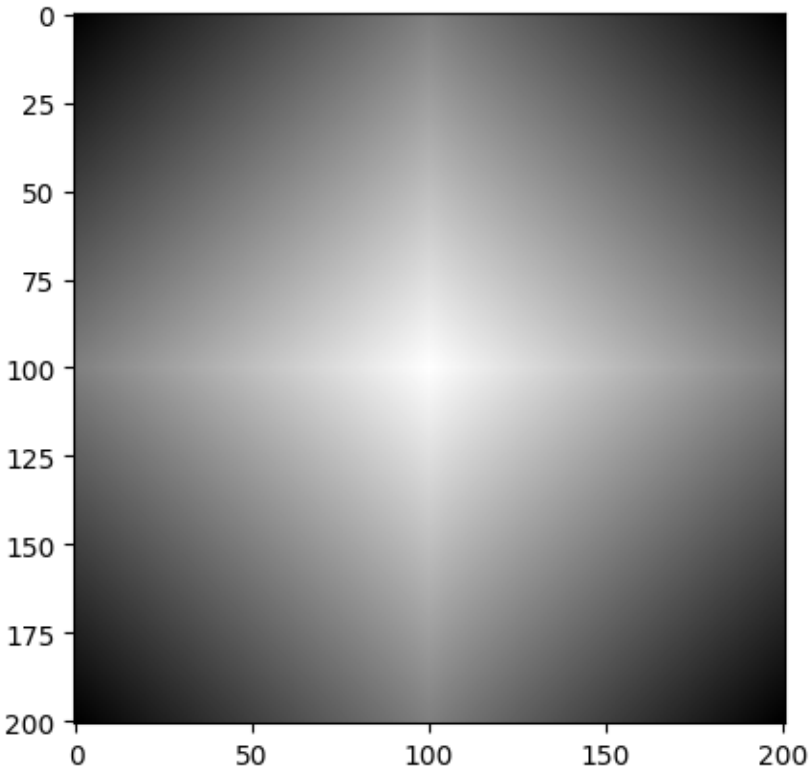
[3]:
```
<contextlib.ExitStack at 0x10c1e0730>
```

L'exercice est terminé, voyons à nouveau notre résultat sous forme d'image :

[4]:
```
squares = stairs(100)
```

Pour le voir comme une image avec un niveau de gris comme code de couleurs (noir = 0, blanc = maximum = 201 dans notre cas) :

[5]:
```
# convertir en flottant pour imshow
squares = squares.astype(float)
# afficher avec une colormap 'gray'
plt.imshow(squares, cmap='gray');
```



7.19 w7-s05-x4-dice

Exercice - niveau avancé

7.19.1 construire un tableau en dimension variable

On étudie les probabilités d'obtenir une certaine somme avec plusieurs dés.

Tout le monde connaît le cas classique avec deux dés à 6 faces, ou l'on construit mentalement la grille suivante :

+	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Imaginons que vous êtes étes étudiant, vous venez de faire un exercice de maths qui vous a mené à une formule qui permet de calculer, pour un jeu à `nb_dice` dés, chacun à `nb_sides` faces, le nombre de tirages qui donnent une certaine somme `target`.

Vous voulez vérifer votre formule, en appliquant une méthode de force brute.

C'est l'objet de cet exercice. Vous devez écrire une fonction `dice` qui prend en paramètres :

— `target` : la somme cible à atteindre,

— `nb_dice` : le nombre de dés,
— `nb_sides` : le nombre de faces sur chaque dé.

On convient que par défaut `nb_dice`=2 et `nb_sides`=6, qui correspond au cas habituel.

Dans ce cas-là par exemple, on voit, en comptant la longueur des diagonales sur la figure, que `dice(7)` doit valoir 6, puisque le tableau comporte 6 cases contenant 7 sur la diagonale.

[1]:
```
import numpy as np

from corrections.exo_dice import exo_dice

# voici quelques exemples pour la fonction dice
exo_dice.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:medium;"\'>appel</span
  >', _dom_classes=('header',)), HTML…
```

À nouveau, on demande explicitement ici un parcours de type force brute.

Pour devancer les remarques sur le forum de discussion :

— ce n'est pas parce cette semaine on étudie numpy que vous devez vous sentir obligé de le faire en numpy.
— vous pouvez même vous donner comme objectif de le faire deux fois, avec et sans numpy :)

[2]:
```
# à vous de jouer
def dice(target, nb_dice=2, nb_sides=6):
    return "votre code"

# NOTE:
# auto-exec-for-latex has used hidden code instead
```

[ ]:
```
# pour corriger votre code
exo_dice.correction(dice)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.20 w7-s05-x5-matdiag

Exercice - niveau intermédiaire

7.20.1 construire une matrice diagonale

On vous demande d'écrire une fonction `matdiag` qui

1. accepte un paramètre qui est une liste de nombres $[x_1, x_2, ..., x_n]$
2. retourne une matrice carrée diagonale dont les éléments valent

$$m_{ii} = x_i$$
$$m_{ij} = 0 \text{ pour } ij$$

Quelques précisions :

— il est raisonnable de retourner toujours un tableau de type `float64`
— vous n'avez pas besoin de vérifier que l'appelant passe au moins un paramètre, ou dit autrement, les jeux de tests n'essaient pas d'appeler la fonction sans argument.

[1]:
```
import numpy as np

# c'est ce qu'on voit sur cet exemple

from corrections.exo_matdiag import exo_matdiag

exo_matdiag.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:medium;"\'>appel</span
  >', _dom_classes=('header',)), HTML…
```

[2]:
```
# à vous de jouer
def matdiag(liste):
    ...
```

[ ]:
```
exo_matdiag.correction(matdiag)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.20.2 Indices

Vous trouverez dans les solutions 3 façons d'implémenter cette fonction ; elles utilisent respectivement :

une approche naïve, une approche à base de slicing, et une approche à base d'une fonction prédéfinie dans numpy.

7.21 w7-s05-x6-xixj

Exercice - niveau intermédiaire

7.21.1 remplir une matrice : $m(i,j) = xi * xj$

On vous demande d'écrire une fonction `xixj` qui

1. accepte un nombre quelconque de paramètres, x_1, x_2, ..., x_n, tous des flottants
2. retourne une matrice carrée symétrique M dont les termes valent

$$m_{ij} = x_i.x_j$$

Vous n'avez pas besoin de vérifier que l'appelant passe au moins un paramètre, ou dit autrement, les jeux de tests n'essaient pas d'appeler la fonction sans argument.

[1]:
```
import numpy as np

# c'est ce qu'on voit sur cet exemple

from corrections.exo_xixj import exo_xixj

exo_xixj.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:medium;"\'>appel</span
  >', _dom_classes=('header',)), HTML…
```

[2]:
```
# à vous de jouer
# n'oubliez pas de déclarer les paramètres de votre fonction
def xixj():
    ...
```

[ ]:
```
exo_xixj.correction(xixj)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.21.2 Indices

Vous trouverez dans les solutions 3 façons d'implémenter cette fonction ; elles utilisent respectivement : l'opérateur `@`, la méthode `array.dot()`, le broadcasting.
Souvenez vous que la transposée d'une matrice peut être obtenue en numpy avec l'attribut `.T` :

[3]:
```
ligne = np.array([1, 2, 3])
ligne.reshape(3, 1)
```

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

[4]:
```
ligne.T
```

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

7.22 w7-s05-x7-npsearch

Exercice - niveau intermédiaire

7.22.1 chercher une sous-matrice

On vous demande d'écrire une fonction `npsearch`

1. qui accepte accepte en entrée deux paramètres

— une matrice `world` (un tableau numpy de dimension 2)
— un tableau `needle` à chercher dans la matrice ; `needle` peut être de dimension 1 ou 2

1. `npsearch` est un générateur (i.e. une fonction génératrice), il doit énumérer tous les tuples d'indices `(i, j)` correspondant aux endroits de `world` qui coincident avec `needle`

[1]:
```
import numpy as np

# c'est ce qu'on voit sur cet exemple

from corrections.exo_npsearch import exo_npsearch

exo_npsearch.example()
```

[1]:
```
GridBox(children=(HTML(value='<span style="font-size:150%;"\'>appel</span>'
   , _dom_classes=('header',)), HTML(v…
```

[2]:
```
# à vous de jouer
# n'oubliez pas de déclarer les paramètres de votre fonction
def npsearch(world, needle):
    # souvenez-vous aussi que vous devez définir un générateur
    yield 0
```

[ ]:
```
exo_npsearch.correction(npsearch)

# NOTE
# auto-exec-for-latex has skipped execution of this cell
```

7.22.2 Indices

— je vous invite autant que possible, comme toujours avec `numpy` :
  — à éviter les boucles faites à la main
  — et à préférer des méthodes toutes faites pour faire des recherches

— essayez par exemple
  — de définir une condition nécessaire sur `world[i, j]` lorsque `(i, j)` fait partie des solutions
  — et c'est peut-être l'occasion de jeter un coup d'oeil à `numpy.argwhere`

— méfiez-vous aussi des expressions du genre `tableau_a == tableau_b` en ce sens que, si les deux tailles ne coincident pas, `numpy` va essayer de faire du broadcasting pour réconcilier les deux tailles, et ici clairement, ce n'est pas ce qu'on veut…

[3]:
```
# enfin pour transformer une ligne en tableau 2D on a le choix entre

a = np.array([1, 2, 3])
```

[4]:
```
# version un peu poussive
n, = a.shape; a.reshape((1, n))
```

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

[5]:
```
# version plus concise
a[np.newaxis, :]
```

[5]:
```
array([[1, 2, 3]])
```

7.23 w7-s06-c1-data-science

La data science en général

7.23.1 et en Python en particulier

7.23.2 Complément - niveau intermédiaire

Qu'est-ce qu'un data scientist ?

J'aimerais commencer cette séquence par quelques réflexions générales sur ce qu'on appelle data science. Ce mot valise, récemment devenu à la mode, et que tout le monde veut ajouter à son CV, est un domaine qui regroupe tous les champs de l'analyse scientifique des données. Cela demande donc, pour être fait sérieusement, de maîtriser :

1. un large champ de connaissances scientifiques, notamment des notions de statistiques appliquées ;
2. les données que vous manipulez ;
3. un langage de programmation pour automatiser les traitements.

Statistiques appliquées Pour illustrer le premier point, pour quelque chose d'aussi simple qu'une moyenne, il est déjà possible de faire des erreurs. Quel intérêt de considérer une moyenne d'une distribution bimodale ?

Par exemple, j'ai deux groupes de personnes et je veux savoir lequel a le plus de chance de gagner à une épreuve de tir à la corde. L'âge moyen de mon groupe A est de 55 ans, l'âge moyen de mon groupe B est de 30 ans. Il me semble alors pouvoir affirmer que le groupe B a plus de chances de gagner. Seulement, dans le groupe B il y a 10 enfants de 5 ans et 10 personnes de 55 ans et dans le groupe A j'ai une population homogène de 20 personnes ayant 55 ans. Finalement, ça sera sans doute le groupe A qui va gagner.

Quelle erreur ai-je faite ? J'ai utilisé un outil statistique qui n'était pas adapté à l'analyse de mes groupes de personnes. Cette erreur peut vous paraître stupide, mais ces erreurs peuvent être très subtiles voire extrêmement difficiles à identifier.

Connaissance des données C'est une des parties les plus importantes, mais largement sous estimées : analyser des données sur lesquelles on n'a pas d'expertise est une aberration. Le risque principal est d'ignorer l'existence d'un facteur caché, ou de supposer à tort l'indépendance des données (sachant que nombre d'outils statistiques ne fonctionnent que sur des données indépendantes). Sans rentrer plus dans le détail, je vous conseille de lire cet article de David Louapre sur le paradoxe de Simpson et la vidéo associée, pour vous donner l'intuition que travailler sur des données qu'on ne maîtrise pas peut conduire à d'importantes erreurs d'interprétation.

Maîtrise d'un langage de programmation Comme vous l'avez sans doute compris, le succès grandissant de la data science est dû à la démocratisation d'outils informatiques comme R, ou la suite d'outils disponibles dans Python, dont nous abordons certains aspects cette semaine.

Il y a ici cependant de nouveau des difficultés. Comme nous allons le voir il est très facile de faire des erreurs qui seront totalement silencieuses, par conséquent, vous obtiendrez presque toujours un résultat, mais totalement faux. Sans une profonde compréhension des mécanismes et des implémentations, vous avez la garantie de faire n'importe quoi.

Vous le voyez, je ne suis pas très encourageant, pour faire de la data science vous devrez maîtriser la bases des outils statistiques, comprendre les données que vous manipulez et maîtriser parfaitement les outils que vous utilisez. Beaucoup de gens pensent qu'en faisant un peu de R ou de Python on peut s'affirmer data scientist, c'est faux, et si vous êtes, par exemple, journaliste ou économiste et que vos résultats ont un impact politique, vous avez une vraie responsabilité et vos erreurs peuvent avoir d'importantes conséquences.

Présentation de `pandas`

`numpy` est l'outil qui permet de manipuler des tableaux en Python, et `pandas` est l'outil qui permet d'ajouter des index à ces tableaux. Par conséquent, `pandas` repose entièrement sur `numpy` et toutes les données que vous manipulez en `pandas` sont des tableaux `numpy`.

`pandas` est un projet qui évolue régulièrement, on vous recommande donc d'utiliser au moins `pandas` dans sa version 0.21. Voici les versions que l'on utilise ici.

[1]:
```
import numpy as np
print(f"numpy version {np.__version__}")

import pandas as pd
print(f"pandas version {pd.__version__}")
```

```
numpy version 1.23.5
pandas version 1.5.2
```

Il est important de comprendre que le monde de la data science en Python suit un autre paradigme que Python. Là où Python favorise la clarté, la simplicité et l'uniformité, `numpy` and `pandas` favorisent l'efficacité. La conséquence est une augmentation de la complexité et une moins bonne uniformité. Aussi, personne ne joue le rôle de BDFL dans la communauté data science comme le fait Guido van Rossum pour Python. Nous entrons donc largement dans une autre philosophie que celle de Python.

Les structures de données en `pandas` Il y a deux structures de données principales en `pandas`, la classe `Series` et la classe `DataFrame`. Une `Series` est un tableau à une dimension où chaque élément est indexé avec essentiellement un autre array (souvent de chaînes de caractères), et une `DataFrame` est un tableau à deux dimensions où les lignes et les colonnes sont indexées. La clef ici est de comprendre que l'intérêt de `pandas` est de pouvoir manipuler les tableaux `numpy` qui sont indexés, et le travail de `pandas` est de rendre les opérations sur ces index très efficaces.

Vous pouvez bien sûr vous demander à quoi cela sert, alors regardons un petit exemple. Nous allons revenir sur les notions utilisées dans cet exemple, notre but ici est de vous montrer l'utilité de `pandas` sur un exemple.

[2]:
```
# seaborn est un module pour dessiner des courbes qui améliore
# sensiblement matplotlib, mais ça n'est pas ce qui nous intéresse ici.
# seaborn vient avec quelques jeux de données sur lesquels on peut jouer.
import seaborn as sns

# chargeons un jeu de données qui représente des pourboires
tips = sns.load_dataset('tips')
```

`load_dataset` retourne une `DataFrame`.

[3]:
```
type(tips)
```

[3]:
```
pandas.core.frame.DataFrame
```

Regardons maintenant à quoi ressemble une `DataFrame` :

[4]:
```
# voici à quoi ressemblent ces données. On a la note totale (total_bill),
# le pourboire (tip), le sexe de la personne qui a donné le pourboire,
# si la personne est fumeur ou non fumeur (smoker), le jour du repas,
# le moment du repas (time) et le nombre de personnes à table (size)
tips.head()
```

[4]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

On voit donc un exemple de `DataFrame` qui représente des données indexées, à la fois par des labels sur les colonnes, et par un rang entier sur les lignes. C'est l'utilisation de ces index qui va nous permettre de faire des requêtes expressives sur ces données.

[5]:
```
# commençons par une rapide description statistique de ces données
tips.describe()
```

[5]:
```
       total_bill         tip        size
count  244.000000  244.000000  244.000000
mean    19.785943    2.998279    2.569672
std      8.902412    1.383638    0.951100
min      3.070000    1.000000    1.000000
25%     13.347500    2.000000    2.000000
50%     17.795000    2.900000    2.000000
75%     24.127500    3.562500    3.000000
max     50.810000   10.000000    6.000000
```

[6]:
```
# prenons la moyenne par sexe
tips.groupby('sex').mean()
```

```
/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50721/3603461209
  .py:2: FutureWarning: The default value of numeric_only in DataFrameGrou
  pBy.mean is deprecated. In a future version, numeric_only will default t
  o False. Either specify numeric_only or select only columns which should
   be valid for the function.
  tips.groupby('sex').mean()
```

[6]:
```
        total_bill       tip      size
sex
Male     20.744076  3.089618  2.630573
Female   18.056897  2.833448  2.459770
```

[7]:
```
# et maintenant la moyenne par jour
tips.groupby('day').mean()
```

```
/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50721/241362509.
  py:2: FutureWarning: The default value of numeric_only in DataFrameGroup
  By.mean is deprecated. In a future version, numeric_only will default to
   False. Either specify numeric_only or select only columns which should
  be valid for the function.
  tips.groupby('day').mean()
```

[7]:
```
      total_bill       tip      size
day
Thur   17.682742  2.771452  2.451613
Fri    17.151579  2.734737  2.105263
Sat    20.441379  2.993103  2.517241
Sun    21.410000  3.255132  2.842105
```

[8]:
```
# et pour finir la moyenne par moment du repas
tips.groupby('time').mean()
```

```
/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50721/4178470357
  .py:2: FutureWarning: The default value of numeric_only in DataFrameGrou
  pBy.mean is deprecated. In a future version, numeric_only will default t
  o False. Either specify numeric_only or select only columns which should
   be valid for the function.
  tips.groupby('time').mean()
```

[8]:
```
        total_bill       tip      size
time
Lunch    17.168676  2.728088  2.411765
```

```
Dinner   20.797159  3.102670  2.630682
```

Vous voyez qu'en quelques requêtes simples et intuitives (nous reviendrons bien sûr sur ces notions) on peut grâce à la notion d'index, obtenir des informations précieuses sur nos données. Vous voyez qu'en l'occurrence, travailler directement sur le tableau `numpy` aurait été beaucoup moins aisé.

Conclusion

Nous avons vu que la data science est une discipline complexe qui demande de nombreuses compétences. Une de ces compétences est la maîtrise d'un langage de programmation, et à cet égard la suite data science de Python qui se base sur `numpy` et `pandas` offre une solution très performante.

Il nous reste une dernière question à aborder : R ou la suite data science de Python ?

Notre préférence va bien évidemment à la suite data science de Python parce qu'elle bénéficie de toute la puissance de Python. R est un langage dédié à la statistique qui n'offre pas la puissance d'un langage générique comme Python. Mais dans le contexte de la data science, R et la suite data science de Python sont deux excellentes solutions. À très grosse maille, la syntaxe de R est plus complexe que celle de Python, par contre, R est très utilisé par les statisticiens, il peut donc avoir une implémentation d'un nouvel algorithme de l'état de l'art plus rapidement que la suite data science de Python.

7.24 w7-s06-c2-Series

`Series` de `pandas`

7.24.1 Complément - niveau intermédiaire

Création d'une `Series`

Un objet de type `Series` est un tableau `numpy` à une dimension avec un index, par conséquent, une `Series` a une certaine similarité avec un dictionnaire, et peut d'ailleurs être directement construite à partir de ce dictionnaire. Notons que, comme pour un dictionnaire, l'accès ou la modification est en $O(1)$, c'est-à-dire à temps constant indépendamment du nombre d'éléments dans la `Series`.

[1]:
```
# Regardons la construction d'une Series
import numpy as np
import pandas as pd

# à partir d'un itérable
s = pd.Series([x**2 for x in range(10)])
print(s)
```

```
0     0
1     1
2     4
3     9
4    16
5    25
6    36
7    49
8    64
9    81
dtype: int64
```

[2]:
```
# en contrôlant maintenant le type
s = pd.Series([x**2 for x in range(10)], dtype='int8')
print(s)
```

```
0     0
1     1
2     4
3     9
4    16
5    25
6    36
7    49
8    64
9    81
dtype: int8
```

[3]:
```
# en définissant un index, par défaut l'index est un rang démarrant à 0
s = pd.Series([x**2 for x in range(10)],
              index=[x for x in 'abcdefghij'],
              dtype='int8',
             )
print(s)
```

```
a     0
b     1
c     4
d     9
e    16
f    25
g    36
h    49
i    64
j    81
dtype: int8
```

[4]:
```
# et directement à partir d'un dictionnaire,
# les clefs forment l'index
d = {k:v**2 for k, v in zip('abcdefghij', range(10))}
print(d)
```

```
{'a': 0, 'b': 1, 'c': 4, 'd': 9, 'e': 16, 'f': 25, 'g': 36, 'h': 49, 'i': 6
  4, 'j': 81}
```

[5]:
```
s = pd.Series(d, dtype='int8')
print(s)
```

```
a     0
b     1
c     4
d     9
e    16
f    25
g    36
h    49
i    64
j    81
dtype: int8
```

Évidemment, l'intérêt d'un index est de pouvoir accéder à un élément par son index, comme nous aurons l'occasion de le revoir :

[6]:
```
print(s['f'])
```

```
25
```

Index

L'index d'une `Series` est un objet implémenté sous la forme d'un `ndarray` de `numpy`, mais qui ne peut contenir que des objets hashables (pour garantir la performance de l'accès).

[7]:
```
# pour accéder à l'index d'un objet Series
# attention, index est un attribut, pas une fonction
print(s.index)
```

```
Index(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype='object')
```

L'index va également supporter un certain nombre de méthodes qui vont faciliter son utilisation. Pour plus de détails, voyez la documentation de l'objet Index et de ses sous-classes.

L'autre moitié de l'objet `Series` est accessible via l'attribut `values`. ATTENTION à nouveau ici, c'est un attribut de l'objet et non pas une méthode, ce qui est très troublant par rapport à l'interface d'un dictionnaire.

[8]:
```
# regardons les valeurs de ma Series
# ATTENTION !! values est un attribut, pas une fonction
print(s.values)
```

```
[ 0  1  4  9 16 25 36 49 64 81]
```

Mais une `Series` a également une interface de dictionnaire à laquelle on accède de la manière suivante :

[9]:
```
# les clefs correspondent à l'index
k = s.keys() # attention ici c'est un appel de fonction !
print(f"Les clefs: {k}")
```

```
Les clefs: Index(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], dtype=
  'object')
```

[10]:
```
# et les couples (clefs, valeurs) sous forme d'un objet zip
for k,v in s.items(): # attention ici aussi c'est un appel de fonction !
    print(k, v)
```

```
a 0
b 1
c 4
d 9
e 16
f 25
g 36
h 49
i 64
j 81
```

[11]:
```
# pour finir remarquons que le test d'appartenance est possible sur les index
print(f"Est-ce que a est dans s ? {'a' in s}")
print(f"Est-ce que z est dans s ? {'z' in s}")
```

```
Est-ce que a est dans s ? True
Est-ce que z est dans s ? False
```

Vous remarquez ici qu'alors que `values` et `index` sont des attributs de la `Series`, `keys()` et `items()` sont des méthodes. Voici un exemple des nombreuses petites incohérences de `pandas` avec lesquelles il faut vivre.

Pièges à éviter

Avant d'aller plus loin, il faut faire attention à la gestion du type des objets contenus dans notre `Series` (on aura le même problème avec les `DataFrame`). Alors qu'un `ndarray` de `numpy` a un type qui ne change pas, une `Series` peut implicitement changer le type de ses valeurs lors d'opérations d'affectations.

[12]:
```
# créons une Series et regardons le type de ses valeurs
s = pd.Series({k:v**2 for k, v in zip('abcdefghij', range(10))})
print(s.values.dtype)
```

```
int64
```

[13]:
```
# On a déjà vu que l'on ne pouvait pas modifier lors d'une affectation le
# type d'un ndarray numpy

try:
    s.values[2] = 'spam'
except ValueError as e:
    print(f"On ne peut pas affecter une str à un ndarray de int64:\n{e}")
```

```
On ne peut pas affecter une str à un ndarray de int64:
invalid literal for int() with base 10: 'spam'
```

[14]:
```
# Par contre, on peut le faire sur une Series
s['c'] = 'spam'

# et maintenant le type des valeurs de la Series a changé
print(s.values.dtype)
```

```
object
```

C'est un point extrêment important puisque toutes les opérations vectorisées vont avoir leur performance impactée et le résultat obtenu peut même être faux. Regardons cela :

[15]:
```
s = pd.Series(range(10_000))
print(s.values.dtype)
```

```
int64
```

[16]:
```
# combien de temps prend le calcul du carré des valeurs
%timeit s**2
```

```
99.1 µs ± 1.67 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[17]:
```
# ajoutons 'spam' à la fin de la Series
s[10_000] = 'spam'

# oups, je me suis trompé, enlevons cet élément
```

```
del s[10_000]

# calculons de nouveau le temps de calcul pour obtenir le carré des valeurs
%timeit s**2
```

```
2.38 ms ± 17.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[18]:
```
# que se passe-t-il, pourquoi le calcul est maintenant plus long
s.values.dtype
```

[18]:
```
dtype('O')
```

Maintenant, les opérations vectorisées le sont sur des objets Python et non plus sur des int64, il y a donc un impact sur la performance.

Et on peut même obtenir un résultat carrément faux. Regardons cela :

[19]:
```
# créons une series de trois entiers
s = pd.Series([1, 2, 3])
print(s)
```

```
0    1
1    2
2    3
dtype: int64
```

[20]:
```
# puis ajoutons un nouvel élément, mais ici je me trompe, c'est une str
# au lieu d'un entier
s[3] = '4'

# à part le type qui pourrait attirer mon attention, rien dans l'affichage
# ne distingue les entiers de la str, à part le dtype
print(s)
```

```
0    1
1    2
2    3
3    4
dtype: object
```

[21]:
```
# seulement si j'additionne, les entiers sont additionnés,
# mais les chaînes de caractères concaténées.
print(s+s)
```

```
0     2
1     4
2     6
3    44
dtype: object
```

Alignement d'index

Un intérêt majeur de `pandas` est de faire de l'alignement d'index sur les objets que l'on manipule. Regardons un exemple :

[22]:
```
argent_poche_janvier = pd.Series([30, 35, 20],
                                 index=['alice', 'bob', 'julie'])
argent_poche_février = pd.Series([30, 35, 20],
                                 index=['alice', 'julie', 'sonia'])
argent_poche_janvier + argent_poche_février
```

[22]:
```
alice    60.0
bob       NaN
julie    55.0
sonia     NaN
dtype: float64
```

On voit que les deux `Series` ont bien été alignées, mais on a un problème. Lorsqu'une valeur n'est pas définie, elle vaut `NaN` et si on ajoute `NaN` à une autre valeur, le résultat est `NaN`. On peut corriger ce problème avec un appel explicite de la fonction add qui accepte un argument `fill_value` qui sera la valeur par défaut en cas d'absence d'une valeur lors de l'opération.

[23]:
```
argent_poche_janvier.add(argent_poche_février, fill_value=0)
```

[23]:
```
alice    60.0
bob      35.0
julie    55.0
sonia    20.0
dtype: float64
```

Accés aux éléments d'une **`Series`**

Comme les `Series` sont basées sur des `ndarray` de `numpy`, elles supportent les opérations d'accès aux éléments des `ndarray`, notamment la notion de masque et les broadcasts, tout ça en conservant évidemment les index.

[24]:
```
s = pd.Series([30, 35, 20], index=['alice', 'bob', 'julie'])

# qui a plus de 25 ans
print(s[s>25])
```

```
alice    30
bob      35
dtype: int64
```

[25]:
```
# regardons uniquement 'alice' et 'julie'
print(s[['alice', 'julie']])
```

```
alice    30
julie    20
dtype: int64
```

[26]:
```
# et affectons sur un masque
s[s<=25] = np.NaN
print(s)
```

```
alice    30.0
bob      35.0
julie     NaN
dtype: float64
```

[27]:
```
# notons également, que naturellement les opérations de broadcast
# sont supportées
s = s + 10
print(s)
```

```
alice    40.0
bob      45.0
julie     NaN
dtype: float64
```

Slicing sur les **Series**

L'opération de slicing sur les **Series** est une source fréquente d'erreur qui peut passer inaperçue pour les raisons suivantes :

- on peut slicer sur les labels des index, mais aussi sur la position (l'indice) d'un élément dans la **Series** ;
- les opérations de slices sur les positions et les labels se comportent différemment, un slice sur les positions exclut la borne de droite (comme tous les slices en Python), mais un slice sur un label inclut la borne de droite ;
- il peut y avoir ambiguïté entre un label et la position d'un élément lorsque le label est un entier.

Nous allons détailler chacun de ces cas, mais sachez qu'il existe une solution qui évite toute ambiguïté, c'est d'utiliser les interfaces **loc** et **iloc** que nous verrons un peu plus loin.

Regardons maintenant ces différents problèmes :

[28]:
```
s = pd.Series([30, 35, 20, 28], index=['alice', 'bob', 'julie', 'sonia'])
print(s)
```

```
alice    30
bob      35
julie    20
sonia    28
dtype: int64
```

[29]:
```
# on peut accéder directement à la valeur correspondant à alice
print(s['alice'])

# mais aussi par la position d'alice dans l'index
print(s[0])
```

```
30
30
```

[30]:
```
# On peut faire un slice sur les labels, dans ce cas la borne
# de droite est incluse
s['alice':'julie']
```

[30]:
```
alice    30
bob      35
julie    20
dtype: int64
```

[31]:
```
# et on peut faire un slice sur les positions, mais dans ce cas
# la borne de droite est exclue, comme un slice normal en Python
s[0:2]
```

[31]:
```
alice    30
bob      35
dtype: int64
```

Ce comportement mérite quelques explications. On voit bien qu'exclure la borne de droite peut se comprendre sur une position (si on exclut `i` on prend `i-1`), par contre, c'est mal défini pour un label.

En effet, l'ordre d'un index est défini au moment de sa création et le label venant juste avant un autre label `L` ne peut pas être trouvé uniquement avec la connaissance de `L`.

C'est pour cette raison que les concepteurs de **pandas** ont préféré inclure la borne de droite.

Regardons maintenant plus en détail cette notion d'ordre sur les index.

[32]:
```
# Regardons le slice sur un index avec un ordre particulier
s = pd.Series([30, 35, 20, 28], index=['alice', 'bob', 'julie', 'sonia'])
print(s['alice':'julie'])
```

```
alice    30
bob      35
julie    20
dtype: int64
```

[33]:
```
# Si on change l'ordre de l'index, ça change la signification du slice
s = pd.Series([30, 35, 20, 28], index=['alice', 'bob', 'sonia', 'julie'])
print(s['alice':'julie'])
```

```
alice    30
bob      35
sonia    20
julie    28
dtype: int64
```

Vous devez peut-être vous demander si un slice sur l'index est toujours défini. La réponse est non ! Pour qu'un slice soit défini sur un index, il faut que l'index ait une croissance monotone ou qu'il n'y ait pas de label dans l'index qui soit dupliqué.

Donc la croissance monotonique n'est pas nécessaire tant qu'il n'y a pas de duplication de labels. Regardons cela.

[34]:
```
# mon index a des labels dupliqués, mais a une croissance monotonique
s = pd.Series([30, 35, 20, 12], index=['a', 'a', 'b', 'c'])
# le slice est défini
s['a': 'b']
```

[34]:
```
a    30
a    35
b    20
dtype: int64
```

[35]:
```
# mon index a des labels dupliqués et n'a pas de croissance monotonique
s = pd.Series([30, 35, 20, 12], index=['a', 'b', 'c', 'a'])
# le slice n'est plus défini
try:
    s['a': 'b']
except KeyError as e:
    print(f"Je n'arrive pas à extraire un slice :\n{e}")
```

```
Je n'arrive pas à extraire un slice :
"Cannot get left slice bound for non-unique label: 'a'"
```

Pour finir sur les problèmes que l'on peut rencontrer avec les slices, que se passe-t-il si on a un index qui a pour label des entiers ?

Lorsque l'on va faire un slice, il va y avoir ambiguïté entre la position du label et le label lui-même. Dans ce cas, `pandas` donne la priorité à la position, mais ce qui est troublant, c'est que lorsqu'on accède à un seul élément en dehors d'un slice, `pandas` donne la priorité à l'index.

Encore une petite incohérence :

[36]:
```
s = pd.Series(['a', 'b', 'c'], index=[2, 0, 1])
print(f"Si on accède directement à un élément, priorité au label : {s[0]}")
print(f"Si on calcule un slice, priorité à la position : {s[0:1]}")
```

```
Si on accède directement à un élément, priorité au label : b
Si on calcule un slice, priorité à la position : 2    a
dtype: object

/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50738/735991246.
  py:3: FutureWarning: The behavior of `series[i:j]` with an integer-dtype
   index is deprecated. In a future version, this will be treated as *labe
  l-based* indexing, consistent with e.g. `series[i]` lookups. To retain t
  he old behavior, use `series.iloc[i:j]`. To get the future behavior, use
   `series.loc[i:j]`.
  print(f"Si on calcule un slice, priorité à la position : {s[0:1]}")
```

`loc` et `iloc`

La solution à tous ces problèmes est de dire explicitement ce que l'on veut faire. On peut en effet dire explicitement si l'on veut utiliser les labels ou les positions, c'est ce qu'on vous recommande de faire pour éviter les comportements implicites.

Pour utiliser les labels il faut utiliser `s.loc[]` et pour utiliser les positions if faut utiliser `s.iloc[]` (le `i` est pour localisation implicite, c'est-à-dire la position). Regardons cela :

[37]:
```
# prenons un cas plus usuel, où les labels sont plutôt des chaines
# notez que la logique est la même quel que soit le type de l'index

s = pd.Series([1000, 2000, 3000, 4000], index=['deux', 'zero', 'un', 'quatre'])
print(s)
```

```
deux      1000
zero      2000
un        3000
quatre    4000
dtype: int64
```

[38]:
```
# accès au label
print(s.loc['zero'])
```

```
2000
```

[39]:
```
# accès à la position
print(s.iloc[0])
```

```
1000
```

[40]:
```
# slice sur les labels, ATTENTION, il inclut la borne de droite
print(s.loc['deux':'zero'])
```

```
deux    1000
zero    2000
dtype: int64
```

[41]:
```
# slice sur les positions, ATTENTION, il exclut la borne de droite
print(s.iloc[1:3])
```

```
zero    2000
un      3000
dtype: int64
```

Pour allez plus loin, vous pouvez lire la documentation officielle :

http://pandas.pydata.org/pandas-docs/stable/indexing.html

Conclusion

Nous avons vu que les `Series` forment une extension des ndarray de dimension 1, en leur ajoutant un index qui permet une plus grande expressivité pour accéder aux éléments. Seulement cette expressivité vient au prix de quelques subtilités (conversions implicites de type, accès aux labels ou aux positions) qu'il faut maîtriser.

Nous verrons dans le prochain complément la notion de `DataFrame` qui est sans doute la plus utile et la plus puissante structure de données de `pandas`. Tous les pièges que nous avons vus pour les `Series` sont valables pour les `DataFrames`.

7.25 w7-s07-c1-DataFrame

`DataFrame` de `pandas`

7.25.1 Complément - niveau intermédiaire

Création d'une `DataFrame`

Une `DataFrame` est un tableau `numpy` à deux dimensions avec un index pour les lignes et un index pour les colonnes. Il y a de nombreuses manières de construire une `DataFrame`.

[1]:
```
# Regardons la construction d'une DataFrame
import numpy as np
import pandas as pd

# Créons une Series pour définir des âges
age = pd.Series([30, 20, 50], index=['alice', 'bob', 'julie'])
```

```
# et une Series pour définir des tailles
height = pd.Series([150, 170, 168], index=['alice', 'marc', 'julie'])

# On peut maintenant combiner ces deux Series en DataFrame,
# chaque Series définissant une colonne, une manière de le faire est
# de définir un dictionnaire qui contient pour clef le nom de la colonne
# et pour valeur la Series correspondante
stat = pd.DataFrame({'age': age, 'height': height})
print(stat)
```

```
        age  height
alice  30.0   150.0
bob    20.0     NaN
julie  50.0   168.0
marc    NaN   170.0
```

On remarque que `pandas` fait automatiquement l'alignement des index, lorsqu'une valeur n'est pas présente, elle est automatiquement remplacée par `NaN`. `Panda` va également broadcaster une valeur unique définissant une colonne sur toutes les lignes. Regardons cela :

[2]:
```
stat = pd.DataFrame({'age': age, 'height': height, 'city': 'Nice'})
print(stat)
```

```
        age  height  city
alice  30.0   150.0  Nice
bob    20.0     NaN  Nice
julie  50.0   168.0  Nice
marc    NaN   170.0  Nice
```

[3]:
```
# On peut maintenant accéder aux index des lignes et des colonnes

# l'index des lignes
print(stat.index)
```

```
Index(['alice', 'bob', 'julie', 'marc'], dtype='object')
```

[4]:
```
# l'index des colonnes
print(stat.columns)
```

```
Index(['age', 'height', 'city'], dtype='object')
```

Il y a de nombreuses manières d'accéder aux éléments de la `DataFrame`, certaines sont bonnes et d'autres à proscrire, commençons par prendre de bonnes habitudes. Comme il s'agit d'une structure à deux dimensions, il faut donner un indice de ligne et de colonne :

[5]:
```
# Quel est l'âge de alice
a = stat.loc['alice', 'age']
```

[6]:
```
# a est un flottant
type(a), a
```

[6]:
```
(numpy.float64, 30.0)
```

[7]:
```
# Quel est la moyenne de tous les âges
c = stat.loc[:, 'age']
m = c.mean()
print(f"L'âge moyen est de {m:.1f} ans.")
```

```
L'âge moyen est de 33.3 ans.
```

[8]:
```
# c est une Series
type(c)
```

[8]:
```
pandas.core.series.Series
```

[9]:
```
# et m est un flottant
type(m)
```

[9]:
```
numpy.float64
```

On peut déjà noter plusieurs choses intéressantes :

— On peut utiliser `.loc[]` et `.iloc` comme pour les `Series`. Pour les `DataFrame` c'est encore plus important parce qu'il y a plus de risques d'ambiguïtés (notamment entre les lignes et les colonnes, on y reviendra) ;

— la méthode `mean` calcule la moyenne, ça n'est pas surprenant, mais ignore les `NaN`. C'est en général ce que l'on veut. Si vous vous demandez comment savoir si la méthode que vous utilisez ignore ou pas les `NaN`, le mieux est de regarder l'aide de cette méthode. Il existe pour un certain nombre de méthodes deux versions : une qui ignore les `NaN` et une autre qui les prend en compte ; on en reparlera.

Une autre manière de construire une `DataFrame` est de partir d'un `array` de `numpy`, et de spécifier les index pour les lignes et les colonnes avec les arguments `index` et `columns` :

[10]:
```
a = np.random.randint(1, 20, 9).reshape(3, 3)
p = pd.DataFrame(a, index=['a', 'b', 'c'], columns=['x', 'y', 'z'])
print(p)
```

```
    x   y   z
a  14  16  11
b  12   6  10
c   7  10   2
```

Importation et exportation de données

En pratique, il est très fréquent que les données qu'on manipule soient stockées dans un fichier ou une base de données. Il existe en `pandas` de nombreux utilitaires pour importer et exporter des données et les convertir automatiquement en `DataFrame`. Vous pouvez importer ou exporter du CSV, JSON, HTML, Excel, HDF5, SQL, Python pickle, etc.

À titre d'illustration écrivons la `DataFrame p` dans différents formats.

[11]:
```
# écrivons notre DataFrame dans un fichier CSV
p.to_csv('my_data.csv')
!cat my_data.csv
```

```
,x,y,z
a,14,16,11
```

```
b,12,6,10
c,7,10,2
```

[12]:
```
# et dans un fichier JSON
p.to_json('my_data.json')
!cat my_data.json
```

```
{"x":{"a":14,"b":12,"c":7},"y":{"a":16,"b":6,"c":10},"z":{"a":11,"b":10,"c"
  :2}}
```

[13]:
```
# on peut maintenant recharger notre fichier
# la conversion en DataFrame est automatique
new_p = pd.read_json('my_data.json')
print(new_p)
```

```
   x   y   z
a  14  16  11
b  12   6  10
c   7  10   2
```

Pour la gestion des autres formats, comme il s'agit de quelque chose de très spécifique et sans difficulté particulière, je vous renvoie simplement à la documentation :

http://pandas.pydata.org/pandas-docs/stable/io.html

Manipulation d'une **DataFrame**

[14]:
```
# construisons maintenant une DataFrame jouet

# voici une liste de prénoms
names = ['alice', 'bob', 'marc', 'bill', 'sonia']

# créons trois Series qui formeront les trois colonnes
age = pd.Series([12, 13, 16, 11, 16], index=names)
height = pd.Series([130, 140, 176, 120, 165], index=names)
sex = pd.Series(list('fmmmf'), index=names)

# créons maintenant la DataFrame
p = pd.DataFrame({'age': age, 'height': height, 'sex': sex})
print(p)
```

```
       age  height sex
alice   12     130   f
bob     13     140   m
marc    16     176   m
bill    11     120   m
sonia   16     165   f
```

[15]:
```
# et chargeons le jeux de données sur les pourboires de seaborn
import seaborn as sns
tips = sns.load_dataset('tips')
```

`pandas` offre de nombreuses possibilités d'explorer les données. Attention, dans mes exemples je vais alterner entre le `DataFrame p` et le `DataFrame tips` suivant les besoins de l'explication.

[16]:
```
# afficher les premières lignes
tips.head()
```

[16]:
```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

[17]:
```
# et les dernière lignes
tips.tail()
```

[17]:
```
     total_bill   tip     sex smoker   day    time  size
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2
```

[18]:
```
# l'index des lignes
p.index
```

[18]:
```
Index(['alice', 'bob', 'marc', 'bill', 'sonia'], dtype='object')
```

[19]:
```
# et l'index des colonnes
p.columns
```

[19]:
```
Index(['age', 'height', 'sex'], dtype='object')
```

[20]:
```
# et afficher uniquement les valeurs
p.values
```

[20]:
```
array([[12, 130, 'f'],
       [13, 140, 'm'],
       [16, 176, 'm'],
       [11, 120, 'm'],
       [16, 165, 'f']], dtype=object)
```

[21]:
```
# échanger lignes et colonnes
# cf. la transposition de matrices
p.T
```

[21]:
```
        alice  bob marc bill sonia
age        12   13   16   11    16
height    130  140  176  120   165
sex         f    m    m    m     f
```

Pour finir, il y a la méthodes `describe` qui permet d'obtenir des premières statistiques sur un `DataFrame`. `describe` permet de calculer des statistiques sur des type numériques, mais aussi sur des types chaînes de caractères.

[22]:
```
# par défaut describe ne prend en compte que les colonnes numériques
p.describe()
```

[22]:
```
             age      height
count   5.000000    5.000000
mean   13.600000  146.200000
std     2.302173   23.605084
min    11.000000  120.000000
25%    12.000000  130.000000
50%    13.000000  140.000000
75%    16.000000  165.000000
max    16.000000  176.000000
```

[23]:
```
# mais on peut le forcer à prendre en compte toutes les colonnes
p.describe(include='all')
```

[23]:
```
              age      height  sex
count    5.000000    5.000000    5
unique        NaN         NaN    2
top           NaN         NaN    m
freq          NaN         NaN    3
mean    13.600000  146.200000  NaN
std      2.302173   23.605084  NaN
min     11.000000  120.000000  NaN
25%     12.000000  130.000000  NaN
50%     13.000000  140.000000  NaN
75%     16.000000  165.000000  NaN
max     16.000000  176.000000  NaN
```

Requêtes sur une **DataFrame**

On peut maintenant commencer à faire des requêtes sur les `DataFrames`. Les `DataFrame` supportent la notion de masque que l'on a vue pour les `ndarray` de `numpy` et pour les `Series`.

[24]:
```
# p.loc prend soit un label de ligne
print(p.loc['sonia'])
```

```
age        16
height    165
sex         f
Name: sonia, dtype: object
```

[25]:
```
# ou alors un label de ligne ET de colonne
print(p.loc['sonia', 'age'])
```

```
16
```

On peut mettre à la place d'une label :

- une liste de labels ;
- un slice sur les labels ;
- un masque (c'est-à-dire un tableau de booléens) ;
- un callable qui retourne une des trois premières possibilités.

Noter que l'on peut également utiliser la notation `.iloc[]` avec les mêmes règles, mais elle est moins utile.

Je recommande de toujours utiliser la notation `.loc[lignes, colonnes]` pour éviter toute ambiguïté. Nous verrons que les notations `.loc[lignes]` ou pire seulement `[label]` sont sources d'erreurs.

Regardons maintenant d'autres exemples plus sophistiqués :

[26]:
```
# un masque sur les femmes
p.loc[:, 'sex'] == 'f'
```

[26]:
```
alice     True
bob      False
marc     False
bill     False
sonia     True
Name: sex, dtype: bool
```

[27]:
```
# si bien que pour construire un tableau
# avec uniquement les femmes
p.loc[p.loc[:, 'sex'] == 'f', :]
```

[27]:
```
        age  height sex
alice   12     130   f
sonia   16     165   f
```

[28]:
```
# si on veut ne garder uniquement
# que les femmes de plus de 14 ans
p.loc[(p.loc[:, 'sex'] == 'f') & (p.loc[:, 'age'] > 14), :]
```

[28]:
```
        age  height sex
sonia   16     165   f
```

[29]:
```
# quelle est la moyenne de 'total_bill' pour les femmes
addition_f = tips.loc[tips.loc[:, 'sex'] == 'Female', 'total_bill'].mean()
print(f"addition moyenne des femmes : {addition_f:.2f}")
```

```
addition moyenne des femmes : 18.06
```

[30]:
```
# quelle est la note moyenne des hommes
addition_h = tips.loc[tips.loc[:, 'sex'] == 'Male', 'total_bill'].mean()
print(f"addition moyenne des hommes : {addition_h:.2f}")
```

```
addition moyenne des hommes : 20.74
```

[31]:
```
# qui laisse le plus grand pourcentage de pourboire :
# les hommes ou les femmes ?

pourboire_f = tips.loc[tips.loc[:, 'sex'] == 'Female', 'tip'].mean()
pourboire_h = tips.loc[tips.loc[:, 'sex'] == 'Male', 'tip'].mean()

print(f"Les femmes laissent {pourboire_f/addition_f:.2%} de pourboire")
print(f"Les hommes laissent {pourboire_h/addition_h:.2%} de pourboire")
```

```
Les femmes laissent 15.69% de pourboire
Les hommes laissent 14.89% de pourboire
```

Erreurs fréquentes et ambiguïtés sur les requêtes

Nous avons vu une manière simple et non ambiguë de faire des requêtes sur les `DataFrame`. Nous allons voir qu'il existe d'autres manières qui ont pour seul avantage d'être plus concises, mais sources de nombreuses erreurs.

Souvenez-vous, utilisez toujours la notation **`.loc[lignes, colonnes]`** sinon, soyez sûr de savoir ce qui est réellement calculé.

[32]:
```
# commençons par la notation la plus classique
p['sex']  # prend forcément un label de colonne
```

[32]:
```
alice    f
bob      m
marc     m
bill     m
sonia    f
Name: sex, dtype: object
```

[33]:
```
# mais par contre, si on passe un slice, c'est forcément des lignes,
# assez perturbant et source de confusion.
p['alice': 'marc']
```

[33]:

	age	height	sex
alice	12	130	f
bob	13	140	m
marc	16	176	m

[34]:
```
# on peut même directement accéder à une colonne par son nom
p.age
```

[34]:
```
alice    12
bob      13
marc     16
bill     11
sonia    16
Name: age, dtype: int64
```

Mais c'est fortement déconseillé parce que si un attribut de même nom existe sur une `DataFrame`, alors la priorité est donnée à l'attribut, et non à la colonne :

[35]:
```
# ajoutons une colonne qui a pour nom une méthode qui existe sur
# les DataFrame
p['mean'] = 1
print(p)
```

```
       age  height sex  mean
alice   12     130   f     1
bob     13     140   m     1
marc    16     176   m     1
bill    11     120   m     1
sonia   16     165   f     1
```

[36]:
```
# je peux bien accéder
# à la colonne sex
p.sex
```

[36]:
```
alice    f
bob      m
marc     m
bill     m
sonia    f
Name: sex, dtype: object
```

[37]:
```
# mais pas à la colonne mean
p.mean
```

[37]:
```
<bound method NDFrame._add_numeric_operations.<locals>.mean of        age
   height sex  mean
alice   12     130   f     1
bob     13     140   m     1
marc    16     176   m     1
bill    11     120   m     1
sonia   16     165   f     1>
```

[38]:
```
# à nouveau, la seule méthode non ambiguë est d'utiliser .loc
p.loc[:, 'mean']
```

[38]:
```
alice    1
bob      1
marc     1
bill     1
sonia    1
Name: mean, dtype: int64
```

[39]:
```
# supprimons maintenant la colonne mean *en place* (par défaut,
# drop retourne une nouvelle DataFrame)
p.drop(columns='mean', inplace=True)
print(p)
```

```
       age  height sex
alice   12     130   f
bob     13     140   m
marc    16     176   m
bill    11     120   m
sonia   16     165   f
```

Pour aller plus loin, vous pouvez lire la documentation officielle :

http://pandas.pydata.org/pandas-docs/stable/indexing.html

Universal functions et **pandas**

Ça n'est pas une surprise, les `Series` et `DataFrame` de `pandas` supportent les `ufunc` de `numpy`. Mais il y a une subtilité. Il est parfaitement légitime et correct d'appliquer une `ufunc` de `numpy` sur les éléments d'une `DataFrame` :

[40]:
```
d = pd.DataFrame(np.random.randint(
    1, 10, 9).reshape(3, 3), columns=list('abc'))
print(d)
```

```
   a  b  c
0  4  5  7
1  9  6  8
2  1  5  1
```

[41]:
```
np.log(d)
```

[41]:

	a	b	c
0	1.386294	1.609438	1.945910
1	2.197225	1.791759	2.079442
2	0.000000	1.609438	0.000000

Nous remarquons que comme on s'y attend, la `ufunc` a été appliquée à chaque élément de la `DataFrame` et que les labels des lignes et colonnes ont été préservés.

Par contre, si l'on a besoin d'alignement de labels, c'est le cas avec toutes les opérations qui s'appliquent sur deux objets comme une addition, alors les `ufunc` de `numpy` ne vont pas faire ce à quoi on s'attend. Elles vont faire les opérations sur les tableaux `numpy` sans prendre en compte les labels.

Pour avoir un alignement des labels, il faut utiliser les `ufunc` de `pandas`.

[42]:
```
# prenons deux Series
s1 = pd.Series([10, 20, 30],
               index=list('abc'))
print(s1)
```

```
a    10
b    20
c    30
dtype: int64
```

[43]:
```
#
s2 = pd.Series([12, 22, 32],
               index=list('acd'))
print(s2)
```

```
a    12
c    22
d    32
dtype: int64
```

[44]:
```
# la ufunc numpy fait la somme
# des arrays sans prendre en compte
# les labels, donc sans alignement
np.add(s1, s2)
```

[44]:
```
a    22.0
b     NaN
c    52.0
d     NaN
dtype: float64
```

[45]:
```
# la ufunc pandas va faire
# un alignement des labels
# cet appel est équivalent à s1 + s2
s1.add(s2)
```

[45]:
```
a    22.0
b     NaN
c    52.0
d     NaN
dtype: float64
```

[46]:
```
# comme on l'a vu sur le complément précédent, les valeurs absentes sont
# remplacées par NaN, mais on peut changer ce comportement lors de
# l'appel de .add
s1.add(s2, fill_value=0)
```

[46]:
```
a    22.0
b    20.0
c    52.0
d    32.0
dtype: float64
```

[47]:
```
# regardons un autre exemple sur des DataFrame
# on affiche tout ça dans les cellules suivantes
names = ['alice', 'bob', 'charle']

bananas = pd.Series([10, 3, 9], index=names)
oranges = pd.Series([3, 11, 6], index=names)
fruits_jan = pd.DataFrame({'bananas': bananas, 'orange': oranges})

bananas = pd.Series([6, 1], index=names[:-1])
apples = pd.Series([8, 5], index=names[1:])
fruits_feb = pd.DataFrame({'bananas': bananas, 'apples': apples})
```

[48]:
```
# ce qui donne
fruits_jan
```

[48]:

	bananas	orange
alice	10	3
bob	3	11
charle	9	6

[49]:
```
# et
fruits_feb
```

[49]:

	bananas	apples
alice	6.0	NaN
bob	1.0	8.0
charle	NaN	5.0

[50]:
```
# regardons maintenant la somme des fruits mangés
eaten_fruits = fruits_jan + fruits_feb
print(eaten_fruits)
```

```
        apples  bananas  orange
alice      NaN     16.0     NaN
bob        NaN      4.0     NaN
charle     NaN      NaN     NaN
```

[51]:
```
# On a bien un alignement des labels, mais il y a beaucoup de valeurs
# manquantes. Corrigeons cela on remplaçant les valeurs manquantes par 0
eaten_fruits = fruits_jan.add(fruits_feb, fill_value=0)
print(eaten_fruits)
```

```
        apples  bananas  orange
alice      NaN     16.0     3.0
bob        8.0      4.0    11.0
charle     5.0      9.0     6.0
```

Notons que lorsqu'une valeur est absente dans toutes les `DataFrame`, `NaN` est conservé.

Un dernière subtilité à connaître lors de l'alignement des labels intervient lorsque vous faites une opération sur une `DataFrame` et une `Series`. `pandas` va considérer la `Series` comme une ligne et va la broadcaster sur les autres lignes. Par conséquent, l'index de la `Series` va être considéré comme des colonnes et aligné avec les colonnes de la `DataFrame`.

[52]:
```
dataframe = pd.DataFrame(
    np.random.randint(1, 10, size=(3, 3)),
    columns=list('abc'), index=list('xyz'))
dataframe
```

[52]:
```
   a  b  c
x  6  3  2
y  8  4  3
z  2  8  1
```

[53]:
```
series_row = pd.Series(
    [100, 200, 300],
    index=list('abc'))
series_row
```

[53]:
```
a    100
b    200
c    300
dtype: int64
```

[54]:
```
series_col = pd.Series(
    [400, 500, 600],
    index=list('xyz'))
series_col
```

[54]:
```
x    400
y    500
z    600
dtype: int64
```

[55]:
```
# la Series est considérée comme une ligne et son index
# s'aligne sur les colonnes de la DataFrame
# la Series va être broadcastée
```

```
# sur les autres lignes de la DataFrame

dataframe + series_row
```

[55]:
```
     a    b    c
x  106  203  302
y  108  204  303
z  102  208  301
```

[56]:
```
# du coup si les labels ne correspondent pas,
# le résultat sera le suivant

dataframe + series_col
```

[56]:
```
    a   b   c   x   y   z
x NaN NaN NaN NaN NaN NaN
y NaN NaN NaN NaN NaN NaN
z NaN NaN NaN NaN NaN NaN
```

[57]:
```
# on peut dans ce cas, changer le comportement par défaut en forçant
# l'alignement de la Series suivant un autre axe avec l'argument axis

dataframe.add(series_col, axis=0)
```

[57]:
```
     a    b    c
x  406  403  402
y  508  504  503
z  602  608  601
```

Ici, `axis=0` signifie que la `Series` est considérée comme une colonne est qu'elle va être broadcastée sur les autres colonnes (le long de l'axe de ligne).

Opérations sur les chaînes de caractères

Nous allons maintenant parler de la vectorisation des opérations sur les chaînes de caractères. Il y a plusieurs choses importantes à savoir :

- les méthodes sur les chaînes de caractères ne sont disponibles que pour les `Series` et les `Index`, mais pas pour les `DataFrame` ;
- ces méthodes ignorent les `NaN` et remplacent les valeurs qui ne sont pas des chaînes de caractères par `NaN` ;
- ces méthodes retournent une copie de l'objet (`Series` ou `Index`), il n'y a pas de modification en place ;
- la plupart des méthodes Python sur le type `str` existe sous forme vectorisée ;
- on accède à ces méthodes avec la syntaxe :
  - `Series.str.<vectorized method name>`
  - `Index.str.<vectorized method name>`

Regardons quelques exemples :

[58]:
```
# Créons une Series avec des noms ayant une capitalisation inconsistante
# et une mauvaise gestion des espaces
names = ['alice ', '  bOB', 'Marc', 'bill', 3, ' JULIE ', np.NaN]
age = pd.Series(names)
```

[59]:
```
# nettoyons maintenant ces données

# on met en minuscule
a = age.str.lower()

# on enlève les espaces
a = a.str.strip()
a
```

[59]:
```
0    alice
1      bob
2     marc
3     bill
4      NaN
5    julie
6      NaN
dtype: object
```

[60]:
```
# comme les méthodes vectorisées retournent un objet de même type, on
# peut les chaîner comme ceci

[x for x in age.str.lower().str.strip()]
```

[60]:
```
['alice', 'bob', 'marc', 'bill', nan, 'julie', nan]
```

On peut également utiliser l'indexation des `str` de manière vectorisée :

[61]:
```
print(a)
```

```
0    alice
1      bob
2     marc
3     bill
4      NaN
5    julie
6      NaN
dtype: object
```

[62]:
```
print(a.str[-1])
```

```
0      e
1      b
2      c
3      l
4    NaN
5      e
6    NaN
dtype: object
```

Pour aller plus loin vous pouvez lire la documentation officielle :

http://pandas.pydata.org/pandas-docs/stable/text.html

Gestion des valeurs manquantes

Nous avons vu que des opérations sur les `DataFrame` pouvaient générer des valeurs `NaN` lors de l'alignement. Il est également possible d'avoir de telles valeurs manquantes dans votre jeu de données original. `pandas` offre plusieurs possibilités pour gérer correctement ces valeurs manquantes.

Avant de voir ces différentes possibilités, définissons cette notion de valeur manquante.

Une valeur manquante peut-être représentée avec `pandas` soit par `np.NaN` soit par l'objet Python `None`.

- `np.NaN` est un objet de type `float`, par conséquent il ne peut apparaître que dans un array de `float` ou un array d'`object`. Notons que `np.NaN` apparaît avec `pandas` comme simplement `NaN` et que dans la suite on utilise de manière indifférente les deux notations, par contre, dans du code, il faut obligatoirement utiliser `np.NaN` ;
  - si on ajoute un `NaN` dans un array d'entier, ils seront convertis en `float64` ;
  - si on ajoute un `NaN` dans un array de booléens, ils seront convertis en `object` ;
- `NaN` est contaminant, toute opération avec un `NaN` a pour résultat `NaN` ;
- lorsque l'on utilise `None`, il est automatiquement converti en `NaN` lorsque le type de l'array est numérique.

Illustrons ces propriétés :

[63]:
```
# une Series d'entiers
s = pd.Series([1, 2])
s
```

[63]:
```
0    1
1    2
dtype: int64
```

[64]:
```
# on insère un NaN, la Series est alors convertie en float64
s[0] = np.NaN
s
```

[64]:
```
0    NaN
1    2.0
dtype: float64
```

[65]:
```
# on réinitialise
s = pd.Series([1, 2])
s
```

[65]:
```
0    1
1    2
dtype: int64
```

[66]:
```
# et on insère None
s[0] = None

# Le résultat est le même
# None est converti en NaN
s
```

[66]:
```
0    NaN
1    2.0
```

```
dtype: float64
```

Regardons maintenant, les méthodes de `pandas` pour gérer les valeurs manquantes (donc `NaN` ou `None`) :

— `isna()` retourne un masque mettant à `True` les valeurs manquantes (il y a un alias `isnull()`) ;
— `notna()` retourne un masque mettant à `False` les valeurs manquantes (il y a un alias `notnull()`) ;
— `dropna()` retourne un nouvel objet sans les valeurs manquantes ;
— `fillna()` retourne un nouvel objet avec les valeurs manquantes remplacées.

On remarque que l'ajout d'alias pour les méthodes est de nouveau une source de confusion avec laquelle il faut vivre.

On remarque également qu'alors que `isnull()` et `notnull()` sont des méthodes simples, `dropna()` et `fillna()` impliquent l'utilisation de stratégies. Regardons cela :

[67]:
```
# créons une DataFrame avec quelques valeurs manquantes
names = ['alice', 'bob', 'charles']
bananas = pd.Series([6, 1], index=names[:-1])
apples = pd.Series([8, 5], index=names[1:])
fruits_feb = pd.DataFrame({'bananas': bananas, 'apples': apples})
print(fruits_feb)
```

```
         bananas  apples
alice        6.0     NaN
bob          1.0     8.0
charles      NaN     5.0
```

[68]:
```
fruits_feb.isna()
```

[68]:

	bananas	apples
alice	False	True
bob	False	False
charles	True	False

[69]:
```
fruits_feb.notna()
```

[69]:

	bananas	apples
alice	True	False
bob	True	True
charles	False	True

Par défaut, `dropna()` va enlever toutes les lignes qui contiennent au moins une valeur manquante. Mais on peut changer ce comportement avec des arguments :

[70]:
```
p = pd.DataFrame([[1, 2, np.NaN], [3, np.NaN, np.NaN], [7, 5, np.NaN]])
print(p)
```

```
   0    1   2
0  1  2.0 NaN
1  3  NaN NaN
2  7  5.0 NaN
```

[71]:
```
# comportement par défaut, j'enlève toutes les lignes avec au moins
# une valeur manquante; il ne reste rien !
```

```
p.dropna()
```

[71]:
```
Empty DataFrame
Columns: [0, 1, 2]
Index: []
```

[72]:
```
# maintenant, je fais l'opération par colonne
p.dropna(axis=1)
```

[72]:
```
   0
0  1
1  3
2  7
```

[73]:
```
# je fais l'opération par colonne si toute la colonne est manquante
p.dropna(axis=1, how='all')
```

[73]:
```
   0    1
0  1  2.0
1  3  NaN
2  7  5.0
```

[74]:
```
# je fais l'opération par ligne si au moins 2 valeurs sont manquantes
p.dropna(thresh=2)
```

[74]:
```
   0    1   2
0  1  2.0 NaN
2  7  5.0 NaN
```

Par défaut, `fillna()` remplace les valeurs manquantes avec un argument pas défaut. Mais on peut ici aussi changer ce comportement. Regardons cela :

[75]:
```
print(p)
```

```
   0    1   2
0  1  2.0 NaN
1  3  NaN NaN
2  7  5.0 NaN
```

[76]:
```
# je remplace les valeurs manquantes par -1
p.fillna(-1)
```

[76]:
```
   0    1    2
0  1  2.0 -1.0
1  3 -1.0 -1.0
2  7  5.0 -1.0
```

[77]:
```
# je remplace les valeurs manquantes avec la valeur suivante sur la colonne
# bfill est pour back fill, c'est-à-dire remplace en arrière à partir des
# valeurs existantes
p.fillna(method='bfill')
```

[77]:
```
   0    1   2
0  1  2.0 NaN
1  3  5.0 NaN
2  7  5.0 NaN
```

[78]:
```
# je remplace les valeurs manquantes avec la valeur précédente sur la ligne
# ffill est pour forward fill, remplace en avant à partir des valeurs
# existantes
p.fillna(method='ffill', axis=1)
```

[78]:
```
     0    1    2
0  1.0  2.0  2.0
1  3.0  3.0  3.0
2  7.0  5.0  5.0
```

Regardez l'aide de ces méthodes pour aller plus loin.

[79]:
```
p.dropna?
```

[80]:
```
p.fillna?
```

Analyse statistique des données

Nous n'avons pas le temps de couvrir les possibilités d'analyse statistique de la suite data science de Python. `pandas` offre quelques possibilités basiques avec des calculs de moyennes, d'écarts types ou de covariances que l'on peut éventuellement appliquer par fenêtres à un jeux de données. Pour avoir plus de détails dessus vous pouvez consulter cette documentation :

http://pandas.pydata.org/pandas-docs/stable/computation.html

Dans la suite data science de Python, il a aussi des modules spécialisés dans l'analyse statistique comme :

— StatsModels
— ScikitLearn

ou des outils de calculs scientifiques plus génériques comme SciPy.

De nouveau, il s'agit d'outils appliqués à des domaines spécifiques et ils se basent tous sur le couple `numpy`/`pandas`.

7.25.2 Complément - niveau avancé

Les MultiIndex

`pandas` avait historiquement d'autres structures de données en plus des `Series` et des `DataFrame` permettant d'exprimer des dimensionnalités supérieures à 2, comme par exemple les `Panel`. Mais pour des raisons de maintenance du code et d'optimisation, les développeurs ont décidé de ne garder que les `Series` et les `DataFrame`. Alors, comment exprimer des données avec plus de deux dimensions ?

On utilise pour cela des `MultiIndex`. Un `MultiIndex` est un index qui peut être utilisé partout où l'on utilise un index (dans une `Series`, ou comme ligne ou colonne d'une `DataFrame`) et qui a pour caractéristique d'avoir plusieurs niveaux.

Comme tous types d'index, et parce qu'un `MultiIndex` est une sous classe d'`Index`, `pandas` va correctement aligner les `Series` et les `DataFrame` avec des `MultiIndex`.

Regardons tout de suite un exemple :

[81]:
```
# construisons une DataFrame jouet

# voici une liste de prénoms
names = ['alice', 'bob', 'sonia']

# créons trois Series qui formeront trois colonnes
age = pd.Series([12, 13, 16], index=names)
height = pd.Series([130, 140, 165], index=names)
sex = pd.Series(list('fmf'), index=names)

# créons maintenant la DataFrame
p = pd.DataFrame({'age': age, 'height': height, 'sex': sex})
print(p)
```

```
       age  height sex
alice   12     130   f
bob     13     140   m
sonia   16     165   f
```

[82]:
```
# unstack, en première approximation, permet de passer d'une DataFrame à
# une Series avec un MultiIndex
s = p.unstack()
print(s)
```

```
age     alice     12
        bob       13
        sonia     16
height  alice    130
        bob      140
        sonia    165
sex     alice      f
        bob        m
        sonia      f
dtype: object
```

[83]:
```
# et voici donc l'index de cette Series
s.index
```

[83]:
```
MultiIndex([(   'age', 'alice'),
            (   'age',   'bob'),
            (   'age', 'sonia'),
            ('height', 'alice'),
            ('height',   'bob'),
            ('height', 'sonia'),
            (   'sex', 'alice'),
            (   'sex',   'bob'),
            (   'sex', 'sonia')],
           )
```

Il existe évidemment des moyens de créer directement un `MultiIndex` et ensuite de le définir comme index d'une `Series` ou comme index de ligne ou colonne d'une `DataFrame` :

[84]:
```
# on peut créer un MultiIndex à partir d'une liste de liste
names = ['alice', 'alice', 'alice', 'bob', 'bob', 'bob']
age = [2014, 2015, 2016, 2014, 2015, 2016]
```

```
s_list = pd.Series([40, 42, 45, 38, 40, 40], index=[names, age])
print(s_list)
```

```
alice  2014    40
       2015    42
       2016    45
bob    2014    38
       2015    40
       2016    40
dtype: int64
```

[85]:
```
# ou à partir d'un dictionnaire de tuples
s_tuple = pd.Series({('alice', 2014): 40,
                     ('alice', 2015): 42,
                     ('alice', 2016): 45,
                     ('bob', 2014): 38,
                     ('bob', 2015): 40,
                     ('bob', 2016): 40})

print(s_tuple)
```

```
alice  2014    40
       2015    42
       2016    45
bob    2014    38
       2015    40
       2016    40
dtype: int64
```

[86]:
```
# ou avec la méthode from_product()
name = ['alice', 'bob']
year = [2014, 2015, 2016]
i = pd.MultiIndex.from_product([name, year])
s = pd.Series([40, 42, 45, 38, 40, 40], index=i)
print(s)
```

```
alice  2014    40
       2015    42
       2016    45
bob    2014    38
       2015    40
       2016    40
dtype: int64
```

On peut même nommer les niveaux d'un `MultiIndex`.

[87]:
```
name = ['alice', 'bob']
year = [2014, 2015, 2016]
i = pd.MultiIndex.from_product([name, year], names=['name', 'year'])
s = pd.Series([40, 42, 45, 38, 40, 40], index=i)
print(s)
```

```
name   year
alice  2014    40
       2015    42
       2016    45
```

```
bob    2014    38
       2015    40
       2016    40
dtype: int64
```

[88]:
```
# on peut changer le nom des niveaux du MultiIndex
s.index.names = ['NAMES', 'YEARS']
print(s)
```

```
NAMES  YEARS
alice  2014     40
       2015     42
       2016     45
bob    2014     38
       2015     40
       2016     40
dtype: int64
```

Créons maintenant une `DataFrame` jouet avec des `MultiIndex` pour étudier comment accéder aux éléments de la `DataFrame`.

[89]:
```
index = pd.MultiIndex.from_product([[2013, 2014],
                                    [1, 2, 3]],
                                   names=['year',
                                          'visit'])

columns = pd.MultiIndex.from_product([['Bob', 'Sue'],
                                      ['avant', 'arrière']],
                                     names=['client',
                                            'pression'])

# on crée des pressions de pneus factices
data = 2 + np.random.rand(6, 4)

# on crée la DataFrame
mecanics_data = pd.DataFrame(data, index=index, columns=columns)
print(mecanics_data)
```

```
client           Bob                 Sue
pression       avant   arrière     avant   arrière
year visit
2013 1      2.489671  2.949029  2.383188  2.293936
     2      2.501769  2.684582  2.826446  2.064549
     3      2.739685  2.530770  2.804261  2.780559
2014 1      2.682081  2.859026  2.739915  2.297523
     2      2.136760  2.403622  2.945736  2.140737
     3      2.445869  2.444920  2.156667  2.642346
```

Il y a plusieurs manières d'accéder aux éléments, mais une seule que l'on recommande :

utilisez la notation **`.loc[ligne, colonne]`**, **`.iloc[ligne, colonne]`**.

[90]:
```
# pression en 2013 pour Bob
mecanics_data.loc[2013, 'Bob']
```

[90]:
```
pression     avant   arrière
visit
1         2.489671  2.949029
2         2.501769  2.684582
3         2.739685  2.530770
```

[91]:
```
# pour accéder aux sous niveaux du MultiIndex, on utilise des tuples
mecanics_data.loc[(2013, 2), ('Bob', 'avant')]
```

[91]:
```
2.501769486202295
```

Le slice sur le `MultiIndex` est un peu délicat. On peut utiliser la notation `:` si on veut slicer sur tous les éléments d'un `MultiIndex`, sans prendre en compte un niveau. Si on spécifie les niveaux, il faut utiliser un objet `slice` ou `pd.IndexSlice` :

[92]:
```
# slice(None) signifie tous les éléments du niveau
print(mecanics_data.loc[slice((2013, 2), (2014, 1)), ('Sue', slice(None))])
```

```
client           Sue
pression       avant   arrière
year visit
2013 2      2.826446  2.064549
     3      2.804261  2.780559
2014 1      2.739915  2.297523
```

[93]:
```
# on peut utiliser la notation : si on ne distingue par les niveaux
print(mecanics_data.loc[(slice(None), slice(1, 2)), :])
```

```
client           Bob                 Sue
pression       avant   arrière     avant   arrière
year visit
2013 1      2.489671  2.949029  2.383188  2.293936
     2      2.501769  2.684582  2.826446  2.064549
2014 1      2.682081  2.859026  2.739915  2.297523
     2      2.136760  2.403622  2.945736  2.140737
```

[94]:
```
# on peut aussi utiliser pd.IndexSlice pour slicer avec une notation
# un peu plus concise
idx = pd.IndexSlice
print(mecanics_data.loc[idx[:, 1:2], idx['Sue', :]])
```

```
client           Sue
pression       avant   arrière
year visit
2013 1      2.383188  2.293936
     2      2.826446  2.064549
2014 1      2.739915  2.297523
     2      2.945736  2.140737
```

Pour aller plus loin, regardez la documentation des `MultiIndex` :

http://pandas.pydata.org/pandas-docs/stable/advanced.html

7.25.3 Conclusion

La `DataFrame` est la structure de données la plus souple et la plus puissante de `pandas`. Nous avons vu comment créer des `DataFrame` et comment accéder aux éléments. Nous verrons dans le prochain complément les techniques permettant de faire des opérations complexes (et proches dans l'esprit de ce que l'on peut faire avec une base de données) comme les opérations de `merge` ou de `groupby`.

7.26 w7-s08-c1-operations-avancees-pandas

Opération avancées en **pandas**

7.26.1 Complément - niveau intermédiaire

Introduction

`pandas` supporte des opérations de manipulation des `Series` et `DataFrame` qui sont similaires dans l'esprit à ce que l'on peut faire avec une base de données et le langage SQL, mais de manière plus intuitive et expressive et beaucoup plus efficacement puisque les opérations se déroulent toutes en mémoire.

Vous pouvez concaténer (`concat`) des `DataFrame`, faire des jointures (`merge`), faire des regroupements (`groupby`) ou réorganiser les index (`pivot`).

Nous allons dans la suite développer ces différentes techniques.

[1]:
```
import numpy as np
import pandas as pd
```

Concaténations avec `concat`

`concat` est utilisé pour concaténer des `Series` ou des `DataFrame`. Regardons un exemple.

[2]:
```
s1 = pd.Series([30, 35], index=['alice', 'bob'])
s2 = pd.Series([32, 22, 29], index=['bill', 'alice', 'jo'])
```

[3]:
```
s1
```

[3]:
```
alice    30
bob      35
dtype: int64
```

[4]:
```
s2
```

[4]:
```
bill     32
alice    22
jo       29
dtype: int64
```

[5]:
```
pd.concat([s1, s2])
```

[5]:
```
alice    30
bob      35
bill     32
alice    22
jo       29
dtype: int64
```

On remarque, cependant, que par défaut il n'y a pas de contrôle sur les labels d'index dupliqués. On peut corriger cela avec l'argument `verify_integrity`, qui va produire une exception s'il y a des labels d'index communs. Évidemment, cela a un coût de calcul supplémentaire, ça n'est donc à utiliser que si c'est nécessaire.

[6]:
```
try:
    pd.concat([s1, s2], verify_integrity=True)
except ValueError as e:
    print(f"erreur de concaténation:\n{e}")
```

```
erreur de concaténation:
Indexes have overlapping values: Index(['alice'], dtype='object')
```

[7]:
```
# créons deux Series avec les index sans recouvrement
s1 = pd.Series(range(1000), index=[chr(x) for x in range(1000)])
s2 = pd.Series(range(1000), index=[chr(x+2000) for x in range(1000)])
```

[8]:
```
# temps de concaténation avec vérification des recouvrements
%timeit pd.concat([s1, s2], verify_integrity=True)
```

```
401 µs ± 5.21 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[9]:
```
# temps de concaténation sans vérification des recouvrements
%timeit pd.concat([s1, s2])
```

```
289 µs ± 4.82 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Par défaut, `concat` concatène les lignes, c'est-à-dire que `s2` sera sous `s1`, mais on peut changer ce comportement en utilisant l'argument `axis` :

[10]:
```
p1 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                  columns=list('ab'), index=list('xy'))
p2 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                  columns=list('ab'), index=list('zt'))
```

[11]:
```
p1
```

[11]:

	a	b
x	6	3
y	2	2

[12]:
```
p2
```

[12]:

	a	b
z	2	3
t	7	7

[13]:
```
# équivalent à pd.concat([p1, p2], axis=0)
# concaténation des lignes
pd.concat([p1, p2])
```

[13]:

	a	b
x	6	3

```
y  2  2
z  2  3
t  7  7
```

[14]:
```
p1 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                  columns=list('ab'), index=list('xy'))
p2 = pd.DataFrame(np.random.randint(1, 10, size=(2,2)),
                  columns=list('cd'), index=list('xy'))
```

[15]:
```
p1
```

[15]:
```
   a  b
x  5  5
y  7  8
```

[16]:
```
p2
```

[16]:
```
   c  d
x  7  7
y  3  3
```

[17]:
```
# concaténation des colonnes
pd.concat([p1, p2], axis=1)
```

[17]:
```
   a  b  c  d
x  5  5  7  7
y  7  8  3  3
```

Regardons maintenant ce cas :

[18]:
```
pd.concat([p1, p2])
```

[18]:
```
     a    b    c    d
x  5.0  5.0  NaN  NaN
y  7.0  8.0  NaN  NaN
x  NaN  NaN  7.0  7.0
y  NaN  NaN  3.0  3.0
```

Vous remarquez que lors de la concaténation, on prend l'union des tous les labels des index de `p1` et `p2`, il y a donc des valeurs absentes qui sont mises à `NaN`. On peut contrôler ce comportement de plusieurs manières comme nous allons le voir ci-dessous.

Par défaut (ce que l'on a fait ci-dessus), join utilise la stratégie dite `outer`, c'est-à-dire qu'on prend l'union des labels.

[19]:
```
# on concatène les lignes, l'argument join décide quels labels on garde
# sur l'autre axe  (ici sur les colonnes).

# si on spécifie 'inner' on prend l'intersection des labels
# du coup il ne reste rien ..
pd.concat([p1, p2], join='inner')
```

[19]:
```
Empty DataFrame
Columns: []
Index: [x, y, x, y]
```

Avec `reindex`, on peut spécifier les labels qu'on veut garder dans l'index des lignes (`axis=0`, c'est la valeur par défaut) ou des colonnes (`axis=1`) :

[20]:
```
# on peut passer à reindex une liste de labels...
pd.concat([p1, p2], axis=1).reindex(['x'])
```

[20]:
```
   a  b  c  d
x  5  5  7  7
```

[21]:
```
# ou un objet Index
# Pour les colonnes je spécifie un reindex avec axis=1
pd.concat([p1, p2], axis=1).reindex(p2.columns, axis=1)
```

[21]:
```
   c  d
x  7  7
y  3  3
```

[22]:
```
pd.concat([p1, p2], axis=1).reindex(['a', 'b'], axis=1)
```

[22]:
```
   a  b
x  5  5
y  7  8
```

Notons que les `Series` et `DataFrame` ont une méthode `append` qui est un raccourci vers `concat`, mais avec moins d'options.

Pour aller plus loin, voici la documentation officielle :

http://pandas.pydata.org/pandas-docs/stable/merging.html#concatenating-objects

Jointures avec `merge`

`merge` est dans l'esprit similaire au `JOIN` en SQL. L'idée est de combiner deux `DataFrame` en fonction d'un critère d'égalité sur des colonnes. Regardons un exemple :

[23]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Sue'],
                    'hire_date': [2004, 2008, 2014]})
```

[24]:
```
df1
```

[24]:
```
  employee        group
0      Bob   Accounting
1     Lisa  Engineering
2      Sue           HR
```

[25]:
```
df2
```

[25]:
```
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2      Sue       2014
```

On souhaite ici combiner `df1` et `df2` de manière à ce que les lignes contenant le même employee soient alignées. Notre critère de merge est donc l'égalité des labels sur la colonne employee.

[26]:
```
pd.merge(df1, df2)
```

[26]:
```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Lisa  Engineering       2004
2      Sue           HR       2014
```

Par défaut, `merge` fait un inner join (ou jointure interne) en utilisant comme critère de jointure les colonnes de même nom (ici `employee`). inner join veut dire que pour joindre deux lignes il faut que le même `employee` apparaisse dans les deux `DataFrame`.

Il existe trois type de merges :

— one-to-one, c'est celui que l'on vient de voir. C'est le merge lorqu'il n'y a pas de labels dupliqués dans les colonnes utilisées comme critère de merge ;
— many-to-one, c'est le merge lorsque l'une des deux colonnes contient des labels dupliqués, dans ce cas, on applique la stratégie one-to-one pour chaque label dupliqué, donc les entrées dupliquées sont préservées ;
— many-to-many, c'est la stratégie lorsqu'il y a des entrées dupliquées dans les deux colonnes. Dans ce cas, on fait un produit cartésien des lignes.

D'une manière générale, gardez en tête que `pandas` fait essentiellement ce à quoi on s'attend. Regardons cela sur des exemples :

[27]:
```
df1 = pd.DataFrame({'patient': ['Bob', 'Lisa', 'Sue'],
                    'repas': ['SS', 'SS', 'SSR']})
df2 = pd.DataFrame({'repas': ['SS', 'SSR'],
                    'explication': ['sans sel', 'sans sucre']})
```

[28]:
```
df1
```

[28]:
```
  patient repas
0     Bob    SS
1    Lisa    SS
2     Sue   SSR
```

[29]:
```
df2
```

[29]:
```
  repas explication
0    SS    sans sel
1   SSR  sans sucre
```

[30]:
```
# la colonne commune pour le merge est 'repas' et dans une des colonnes
# (sur df1), il y a des labels dupliqués, on applique la stratégie many-to-one
pd.merge(df1, df2)
```

[30]:
```
  patient repas explication
0     Bob    SS    sans sel
1    Lisa    SS    sans sel
2     Sue   SSR  sans sucre
```

[31]:
```
df1 = pd.DataFrame({'patient': ['Bob', 'Lisa', 'Sue'],
                    'repas': ['SS', 'SS', 'SSR']})
df2 = pd.DataFrame({'repas': ['SS', 'SS', 'SSR'],
                    'explication': ['sans sel', 'légumes', 'sans sucre']})
```

[32]:
```
df1
```

[32]:
```
  patient repas
0     Bob    SS
1    Lisa    SS
2     Sue   SSR
```

[33]:
```
df2
```

[33]:
```
  repas explication
0    SS    sans sel
1    SS     légumes
2   SSR  sans sucre
```

[34]:
```
# la colonne commune pour le merge est 'repas' et dans les deux colonnes
# il y a des labels dupliqués, on applique la stratégie many-to-many
pd.merge(df1,df2)
```

[34]:
```
  patient repas explication
0     Bob    SS    sans sel
1     Bob    SS     légumes
2    Lisa    SS    sans sel
3    Lisa    SS     légumes
4     Sue   SSR  sans sucre
```

Dans un merge, on peut contrôler les colonnes à utiliser comme critère de merge. Regardons ces différents cas sur des exemples :

[35]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Sue'],
                    'hire_date': [2004, 2008, 2014]})
```

[36]:
```
df1
```

[36]:
```
  employee        group
0      Bob   Accounting
1     Lisa  Engineering
2      Sue           HR
```

[37]:
```
df2
```

[37]:
```
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2      Sue       2014
```

[38]:
```
# on décide d'utiliser la colonne 'employee' comme critère de merge
pd.merge(df1, df2, on='employee')
```

[38]:
```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Lisa  Engineering       2004
2      Sue           HR       2014
```

[39]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'HR']})
df2 = pd.DataFrame({'name': ['Lisa', 'Bob', 'Sue'],
                    'hire_date': [2004, 2008, 2014]})
```

[40]:
```
df1
```

[40]:
```
  employee        group
0      Bob   Accounting
1     Lisa  Engineering
2      Sue           HR
```

[41]:
```
df2
```

[41]:
```
   name  hire_date
0  Lisa       2004
1   Bob       2008
2   Sue       2014
```

[42]:
```
# mais on peut également définir un nom de colonne différent
# à gauche et à droite
m = pd.merge(df1,df2, left_on='employee', right_on='name')
m
```

[42]:
```
  employee        group  name  hire_date
0      Bob   Accounting   Bob       2008
1     Lisa  Engineering  Lisa       2004
2      Sue           HR   Sue       2014
```

[43]:
```
# dans ce cas, comme on garde les colonnes utilisées comme critère dans
# le résultat du merge, on peut effacer la colonne inutile ainsi
m.drop('name', axis=1)
```

[43]:
```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Lisa  Engineering       2004
2      Sue           HR       2014
```

`merge` permet également de contrôler la stratégie à appliquer lorsqu'il y a des valeurs dans une colonne utilisée comme critère de merge qui sont absentes dans l'autre colonne. C'est ce que l'on appelle jointure à gauche, jointure à droite, jointure interne (comportement par défaut) et jointure externe. Pour ceux qui ne sont pas familiers avec ces notions, regardons des exemples :

[44]:
```
df1 = pd.DataFrame({'name': ['Bob', 'Lisa', 'Sue'],
                    'pulse': [70, 63, 81]})
df2 = pd.DataFrame({'name': ['Eric', 'Bob', 'Marc'],
                    'weight': [60, 100, 70]})
```

[45]:
```
df1
```

[45]:
```
   name  pulse
0   Bob     70
1  Lisa     63
2   Sue     81
```

[46]:
```
df2
```

[46]:
```
   name  weight
0  Eric      60
1   Bob     100
2  Marc      70
```

[47]:
```
# la colonne 'name' est le critère de merge dans les deux DataFrame.
# Seul Bob existe dans les deux colonnes. Dans un inner join
# (le cas par défaut) on ne garde que les lignes pour lesquelles il y a une
# même valeur présente à gauche et à droite
pd.merge(df1, df2) # équivalent à pd.merge(df1, df2, how='inner')
```

[47]:
```
  name  pulse  weight
0  Bob     70     100
```

[48]:
```
# le outer join va au contraire faire une union des lignes et compléter ce
# qui manque avec NaN
pd.merge(df1, df2, how='outer')
```

[48]:
```
   name  pulse  weight
0   Bob   70.0   100.0
1  Lisa   63.0     NaN
2   Sue   81.0     NaN
3  Eric    NaN    60.0
4  Marc    NaN    70.0
```

[49]:
```
# le left join ne garde que les valeurs de la colonne de gauche
pd.merge(df1, df2, how='left')
```

[49]:
```
   name  pulse  weight
0   Bob     70   100.0
1  Lisa     63     NaN
2   Sue     81     NaN
```

[50]:
```
# et le right join ne garde que les valeurs de la colonne de droite
pd.merge(df1, df2, how='right')
```

[50]:
```
   name  pulse  weight
0  Eric    NaN      60
1   Bob   70.0     100
2  Marc    NaN      70
```

Pour aller plus loin, vous pouvez lire la documentation. Vous verrez notamment que vous pouvez merger sur les index (au lieu des colonnes) ou le cas où vous avez des colonnes de même nom qui ne font pas partie du critère de merge :

http://pandas.pydata.org/pandas-docs/stable/merging.html#database-style-dataframe-joining-merging

Regroupement avec **groupby**

Regardons maintenant cette notion de groupement. Il s'agit d'une notion très puissante avec de nombreuses options que nous ne couvrirons que partiellement. La logique derrière `groupby` est de créer des groupes dans une `DataFrame` en fonction des valeurs d'une (ou plusieurs) colonne(s), toutes les lignes contenant la même valeur sont dans le même groupe. On peut ensuite appliquer à chaque groupe des opérations qui sont :

- soit des calculs sur chaque groupe ;
- soit un filtre sur chaque groupe qui peut garder ou supprimer un groupe ;
- soit une transformation qui va modifier tout le groupe (par exemple, pour centrer les valeurs sur la moyenne du groupe).

Regardons quelques exemples :

[51]:
```
d = pd.DataFrame({'key': list('ABCABC'), 'val': range(6)})
d
```

[51]:
```
  key  val
0   A    0
1   B    1
2   C    2
3   A    3
4   B    4
5   C    5
```

[52]:
```
# utilisons comme colonne de groupement 'key'
g = d.groupby('key')
g
```

[52]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x1194d88b0>
```

`groupby` produit un nouvel objet, mais ne fait aucun calcul. Les calculs seront effectués lors de l'appel d'une fonction sur ce nouvel objet. Par exemple, calculons la somme pour chaque groupe.

[53]:
```
g.sum()
```

[53]:
```
     val
key
A      3
B      5
```

```
C      7
```

`groupby` peut utiliser comme critère de groupement une colonne, une liste de colonnes, ou un index (c'est notamment utile pour les `Series`).

Une particularité de `groupby` est que le critère de groupement devient un index dans le nouvel objet généré. L'avantage est que l'on a maintenant un accès optimisé sur ce critère, mais l'inconvénient est que sur certaines opérations qui détruisent l'index on peut perdre ce critère. On peut contrôler ce comportement avec `as_index`.

[54]:
```
g = d.groupby('key', as_index=False)
g.sum()
```

[54]:
```
  key  val
0   A    3
1   B    5
2   C    7
```

L'objet produit par `groupby` pemet de manipuler les groupes, regardons cela :

[55]:
```
d = pd.DataFrame({'key': list('ABCABC'),
                  'val1': range(6),
                  'val2' : range(100, 106)})
d
```

[55]:
```
  key  val1  val2
0   A     0   100
1   B     1   101
2   C     2   102
3   A     3   103
4   B     4   104
5   C     5   105
```

[56]:
```
g = d.groupby('key')

# g.groups donne accès au dictionnaire des groupes,
# les clefs sont le nom du groupe
# et les valeurs les index des lignes
# appartenant au groupe
g.groups
```

[56]:
```
{'A': [0, 3], 'B': [1, 4], 'C': [2, 5]}
```

[57]:
```
# pour accéder directement au groupe, on peut utiliser get_group
g.get_group('A')
```

[57]:
```
  key  val1  val2
0   A     0   100
3   A     3   103
```

[58]:
```
# on peut également filtrer un groupe par colonne
# lors d'une opération
g.sum()['val2']
```

[58]:
```
key
A    203
B    205
C    207
Name: val2, dtype: int64
```

[59]:
```
# ou directement sur l'objet produit par groupby
g['val2'].sum()
```

[59]:
```
key
A    203
B    205
C    207
Name: val2, dtype: int64
```

On peut également itérer sur les groupes avec un boucle `for` classique :

[60]:
```
import seaborn as sns
# on charge le fichier de données des pourboires
tips = sns.load_dataset('tips')

# pour rappel
tips.head()
```

[60]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

[61]:
```
# on groupe le DataFrame par jours
g = tips.groupby('day')

# on calcule la moyenne du pourboire par jour
for (group, index) in g:
    print(f"On {group} the mean tip is {index['tip'].mean():.3}")
```

```
On Thur the mean tip is 2.77
On Fri the mean tip is 2.73
On Sat the mean tip is 2.99
On Sun the mean tip is 3.26
```

L'objet produit par `groupby` supporte ce que l'on appelle le dispatch de méthodes. Si une méthode n'est pas directement définie sur l'objet produit par `groupby`, elle est appelée sur chaque groupe (il faut donc qu'elle soit définie sur les `DataFrame` ou les `Series`). Regardons cela :

[62]:
```
# on groupe par jour et on extrait uniquement la colonne 'total_bill'
# pour chaque groupe
g = tips.groupby('day')['total_bill']

# on demande à pandas d'afficher les float avec seulement deux chiffres
# après la virgule
pd.set_option('display.float_format', '{:.2f}'.format)
```

```
# on appelle describe() sur g, mais elle n'est pas définie sur cet objet,
# elle va donc être appelée (dispatch) sur chaque groupe
g.describe()
```

[62]:
```
       count  mean  std  min   25%   50%   75%   max
day
Thur  62.00 17.68 7.89 7.51 12.44 16.20 20.16 43.11
Fri   19.00 17.15 8.30 5.75 12.09 15.38 21.75 40.17
Sat   87.00 20.44 9.48 3.07 13.91 18.24 24.74 50.81
Sun   76.00 21.41 8.83 7.25 14.99 19.63 25.60 48.17
```

[63]:
```
# Mais, il y a tout de même un grand nombre de méthodes
# définies directement sur l'objet produit par le groupby

methods = [x for x in dir(g) if not x.startswith('_')]
f"Le type {type(g).__name__} expose {len(methods)} méthodes."
```

[63]:
```
'Le type SeriesGroupBy expose 70 méthodes.'
```

[64]:
```
# profitons de la mise en page des dataframes
# pour afficher ces méthodes sur plusieurs colonnes
# on fait un peu de gymnastique
# il y a d'ailleurs sûrement plus simple..
columns = 7
nb_methods = len(methods)
nb_pad = (columns - nb_methods % columns) % columns

array = np.array(methods + nb_pad * ['']).reshape((columns, -1))
```

[65]:
```
pd.DataFrame(data=array.transpose())
```

[65]:
```
           0          1          2                          3          4  \
0        agg   cumcount      ffill                    indices      ngroup
1  aggregate     cummax     fillna  is_monotonic_decreasing     ngroups
2        all     cummin     filter  is_monotonic_increasing    nlargest
3        any    cumprod      first                       last   nsmallest
4      apply     cumsum  get_group                        mad         nth
5   backfill   describe     groups                        max     nunique
6      bfill       diff       head                       mean        ohlc
7       corr      dtype       hist                     median         pad
8      count        ewm     idxmax                        min  pct_change
9        cov  expanding     idxmin                       ndim        pipe

          5            6
0      plot         skew
1      prod          std
2  quantile          sum
3      rank         tail
4  resample         take
5   rolling    transform
6    sample       tshift
7       sem       unique
8     shift  value_counts
9      size          var
```

Nous allons regarder la méthode `aggregate` (dont l'alias est `agg`). Cette méthode permet d'appliquer une fonction (ou liste de fonctions) à chaque groupe avec la possibilité d'appliquer une fonction à une colonne spécifique du groupe.

Une subtilité de `aggregate` est que l'on peut passer soit un objet fonction, soit un nom de fonction sous forme d'une `str`. Pour que l'utilisation du nom de la fonction marche, il faut que la fonction soit définie sur l'objet produit par le `groupby` ou qu'elle soit définie sur les groupes (donc avec dispatching).

[66]:
```
# calculons la moyenne et la variance pour chaque groupe
# et chaque colonne numérique
tips.groupby('day').agg(['mean', 'std'])
```

```
/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50790/3904838133
  .py:3: FutureWarning: ['sex', 'smoker', 'time'] did not aggregate succes
  sfully. If any error is raised this will raise in a future version of pa
  ndas. Drop these columns/ops to avoid this warning.
  tips.groupby('day').agg(['mean', 'std'])
```

[66]:

	total_bill		tip		size	
	mean	std	mean	std	mean	std
day						
Thur	17.68	7.89	2.77	1.24	2.45	1.07
Fri	17.15	8.30	2.73	1.02	2.11	0.57
Sat	20.44	9.48	2.99	1.63	2.52	0.82
Sun	21.41	8.83	3.26	1.23	2.84	1.01

[67]:
```
# de manière équivalente avec les objets fonctions
tips.groupby('day').agg([np.mean, np.std])
```

```
/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50790/3172960551
  .py:2: FutureWarning: ['sex', 'smoker', 'time'] did not aggregate succes
  sfully. If any error is raised this will raise in a future version of pa
  ndas. Drop these columns/ops to avoid this warning.
  tips.groupby('day').agg([np.mean, np.std])
```

[67]:

	total_bill		tip		size	
	mean	std	mean	std	mean	std
day						
Thur	17.68	7.89	2.77	1.24	2.45	1.07
Fri	17.15	8.30	2.73	1.02	2.11	0.57
Sat	20.44	9.48	2.99	1.63	2.52	0.82
Sun	21.41	8.83	3.26	1.23	2.84	1.01

[68]:
```
# en appliquant une fonction différente pour chaque colonne,
# on passe alors un dictionnaire qui a pour clef le nom de la
# colonne et pour valeur la fonction à appliquer à cette colonne
tips.groupby('day').agg({'tip': np.mean, 'total_bill': np.std})
```

[68]:

	tip	total_bill
day		
Thur	2.77	7.89
Fri	2.73	8.30
Sat	2.99	9.48
Sun	3.26	8.83

La méthode `filter` a pour but de filtrer les groupes en fonction d'un critère. Mais attention, `filter` retourne un sous ensemble des données originales dans lesquelles les éléments appartenant aux groupes filtrés ont été enlevés.

[69]:
```
d = pd.DataFrame({'key': list('ABCABC'),
                  'val1': range(6),
                  'val2' : range(100, 106)})
d
```

[69]:
```
  key  val1  val2
0   A     0   100
1   B     1   101
2   C     2   102
3   A     3   103
4   B     4   104
5   C     5   105
```

[70]:
```
# regardons la somme par groupe
d.groupby('key').sum()
```

[70]:
```
      val1  val2
key
A        3   203
B        5   205
C        7   207
```

[71]:
```
# maintenant gardons dans les données originales toutes les lignes
# pour lesquelles la somme de leur groupe est supérieure à 3
# (ici les groupes B et C)
d.groupby('key').filter(lambda x: x['val1'].sum() > 3)
```

[71]:
```
  key  val1  val2
1   B     1   101
2   C     2   102
4   B     4   104
5   C     5   105
```

La méthode `transform` a pour but de retourner un sous ensemble des données originales dans lesquelles une fonction a été appliquée par groupe. Un usage classique est de centrer des valeurs par groupe, ou de remplacer les `NaN` d'un groupe par la valeur moyenne du groupe.

Attention, `transform` ne doit pas faire de modifications en place, sinon le résultat peut être faux. Faites donc bien attention de ne pas appliquer des fonctions qui font des modications en place.

[72]:
```
r = np.random.normal(0.5, 2, 4)
d = pd.DataFrame({'key': list('ab'*2), 'data': r,'data2': r*2})
d
```

[72]:
```
  key  data  data2
0   a  0.03   0.07
1   b  3.27   6.54
2   a -0.92  -1.83
3   b  1.14   2.28
```

[73]:
```
# je groupe sur la colonne 'key'
g = d.groupby('key')
```

[74]:
```
# maintenant je centre chaque groupe par rapport à sa moyenne
g.transform(lambda x: x - x.mean())
```

[74]:
```
   data  data2
0  0.48   0.95
1  1.06   2.13
2 -0.48  -0.95
3 -1.06  -2.13
```

Notez que la colonne `key` a disparu, ce comportement est expliqué ici :

http://pandas.pydata.org/pandas-docs/stable/groupby.html#automatic-exclusion-of-nuisance-columns

Pour aller plus loin sur `groupby` vous pouvez lire la documentation :

http://pandas.pydata.org/pandas-docs/stable/groupby.html

Réorganisation des indexes avec `pivot`

Un manière de voir la notion de pivot est de considérer qu'il s'agit d'une extension de `groupy` à deux dimensions. Pour illustrer cela, prenons un exemple en utilisant le jeu de données seaborn sur les passagers du Titanic.

[75]:
```
titanic = sns.load_dataset('titanic')
```

[76]:
```
# regardons le format de ce jeu de données
titanic.head()
```

[76]:
```
   survived  pclass     sex   age  sibsp  parch  fare embarked  class    wh
   o  \
0         0       3    male 22.00      1      0  7.25        S  Third    ma
   n
1         1       1  female 38.00      1      0 71.28        C  First  woma
   n
2         1       3  female 26.00      0      0  7.92        S  Third  woma
   n
3         1       1  female 35.00      1      0 53.10        S  First  woma
   n
4         0       3    male 35.00      0      0  8.05        S  Third    ma
   n

   adult_male deck  embark_town alive  alone
0        True  NaN  Southampton    no  False
1       False    C    Cherbourg   yes  False
2       False  NaN  Southampton   yes   True
3       False    C  Southampton   yes  False
4        True  NaN  Southampton    no   True
```

[77]:
```
# regardons maintenant le taux de survie par classe et par sex
titanic.pivot_table('survived', index='class', columns='sex')
```

[77]:
```
sex     female  male
class
First     0.97  0.37
Second    0.92  0.16
Third     0.50  0.14
```

Je ne vais pas entrer plus dans le détail, mais vous voyez qu'il s'agit d'un outil très puissant.

Pour aller plus loin, vous pouvez regarder la documentation officielle :

http://pandas.pydata.org/pandas-docs/stable/reshaping.html

mais vous aurez des exemples beaucoup plus parlants en regardant ici :

https://github.com/jakevdp/PythonDataScienceHandbook/blob/master/notebooks/03.09-Pivot-Tables.ipynb

7.27 w7-s09-c1-TimeSeries

Séries temporelles en **pandas**

7.27.1 Complément - niveau intermédiaire

Parsing des dates et gestion des erreurs

Lorsqu'il y a des erreurs de parsing des dates, pandas offre la possibilité de lancer une exception, ou de produire un objet `NaT` pour Not a Time qui se manipule ensuite comme un `NaN`.

[1]:
```
import pandas as pd
date = '100/06/2018' # cette date ne peut pas être parsée

try:
    pd.to_datetime(date) # comportement pas défaut qui lance une exception
except ValueError as e:
    print(e)
```

```
Unknown string format: 100/06/2018 present at position 0
```

[2]:
```
# retourne l'input en cas d'erreur
pd.to_datetime(date, errors='ignore')
```

[2]:
```
'100/06/2018'
```

[3]:
```
# retourne NaT en cas d'erreur
pd.to_datetime(date, errors='coerce')
```

[3]:
```
NaT
```

[4]:
```
# la dernière date n'est pas valide
d = pd.to_datetime(['jun 2018', '10/12/1980',
                    '25 january 2000', '100 june 1900'],
                   errors='coerce')
print(d)
```

```
DatetimeIndex(['2018-06-01', '1980-10-12', '2000-01-25', 'NaT'], dtype='dat
  etime64[ns]', freq=None)
```

[5]:
```
# on peut utiliser les méthodes pour les NaN directement sur un NaT
d.fillna(pd.to_datetime('10 june 1980'))
```

[5]:
```
DatetimeIndex(['2018-06-01', '1980-10-12', '2000-01-25', '1980-06-10'], dty
   pe='datetime64[ns]', freq=None)
```

Pour aller plus loin

Vous trouverez de nombreux exemples dans la documentation officielle de pandas

7.27.2 Conclusion

Ce notebook clôt notre survol de `numpy` et `pandas`. C'est un sujet vaste que nous avons déjà largement dégrossi. Pour aller plus loin vous avez évidemment la documentation officielle de `numpy` et `pandas` :

— reference numpy
— reference pandas

Mais vous avez aussi l'excellent livre de Jake VanderPlas "Python Data Science Handbook" qui est entièrement disponible sous forme de notebooks en ligne :

https://github.com/jakevdp/PythonDataScienceHandbook

Il s'agit d'un très beau travail (c'est rare) utilisant les dernières versions de Python, `pandas` and `numpy` (c'est encore plus rare), fait par un physicien qui fait de la data science et qui a contribué au développement de nombreux modules de data science en Python.

Je vous conseille par ailleurs, pour ceux qui sont à l'aise en anglais, une série de 10 vidéos sur YouTube publiées par le même Jake VanderPlas, où il étudie un jeu de données du début (chargement des données) à la fin (classification).

Pour finir, si vous voulez faire de la data science, il y a un livre incontournable : "An Introduction de Statistical Learning" de G. James, D. Witten, T. Hastie, R. Tibshirani. Ce livre utilise R, mais vous pouvez facilement l'appliquer en utilisant `pandas`.

Les auteurs mettent à disposition gratuitement le PDF du livre ici :

http://www-bcf.usc.edu/~gareth/ISL/

N'oubliez pas, si ces ressources vous sont utiles, d'acheter ces livres pour supporter ces auteurs. Les ressources de grande qualité sont rares, elles demandent un travail énorme à produire, elles doivent être encouragées et recompensées.

7.28 w7-s10-c1-matplotlib-2d

`matplotlib` - 2D

7.28.1 Complément - niveau basique

Plutôt que de récrire (encore) un tutorial sur `matplotlib`, je préfère utiliser les ressources disponibles en ligne en anglais :

— pour la dimension 2 : https://matplotlib.org/2.0.2/users/pyplot_tutorial.html ;
— pour la dimension 3 : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html.

Je vais essentiellement utiliser des extraits tels quels. N'hésitez pas à consulter ces documents originaux pour davantage de précisions.

[1]:
```
# les imports habituels
import numpy as np
import matplotlib.pyplot as plt
```

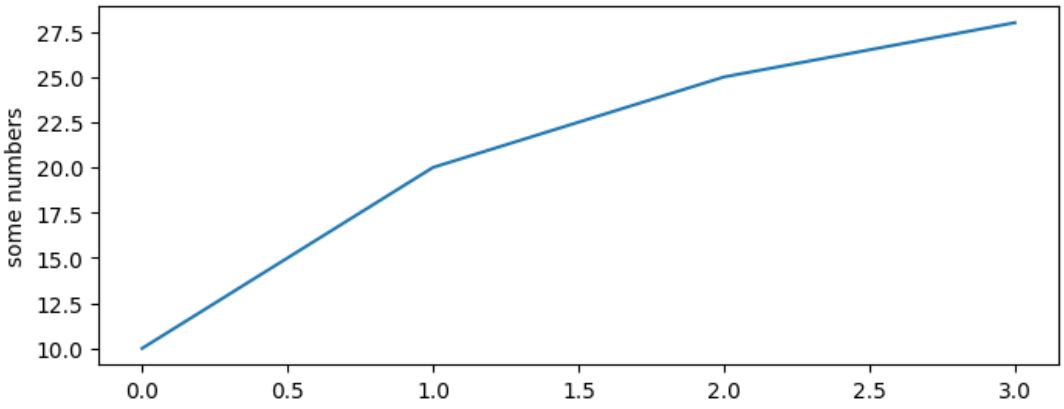
Intentionnellement dans ce notebook, on ne va pas utiliser le mode automatique de `matplotlib` dans les notebooks (pour rappel, `plt.ion()`), car on veut justement apprendre à utiliser `matplotlib` dans un contexte normal.

[2]:
```
# pour changer la taille par défaut des figures matplotlib
plt.rcParams["figure.figsize"] = (8, 3)
```

plt.plot

Nous avons déjà vu plusieurs fois comment tracer une courbe avec `matplotlib`, avec la fonction `plot`. Si on donne seulement une liste de valeurs, elles sont considérées comme les Y, les X étant les entiers en nombre suffisant et en commençant à 0.

[3]:
```
# si je ne donne qu'une seule liste à plot
# alors ce sont les Y
plt.plot([10, 20, 25, 28])
# on peut aussi facilement ajouter une légende
# ici sur l'axe des y
plt.ylabel('some numbers')

plt.show()
```

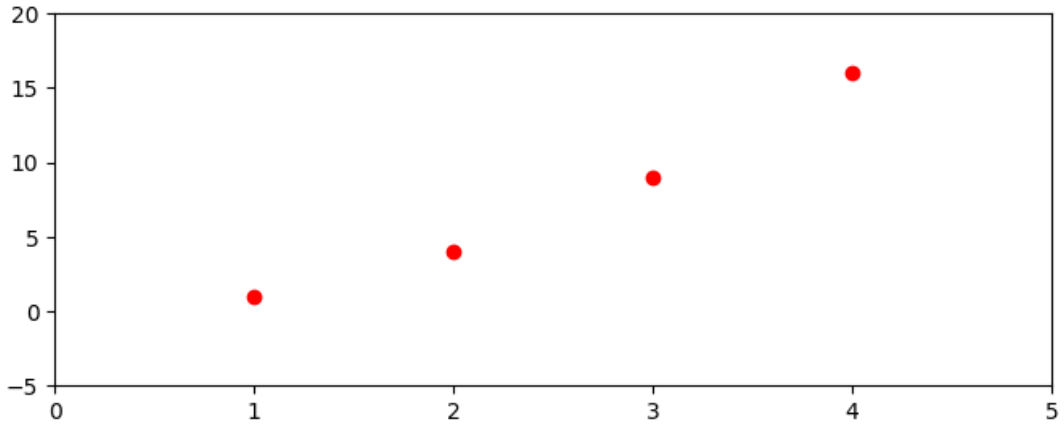


On peut changer le style utilisé par `plot` pour tracer ; ce style est spécifié sous la forme d'une chaîne de caractères, par défaut `'b-'`, qui signifie une ligne bleue (`b` pour bleu, et `-` pour ligne). Ici on va préciser à la place `ro`, `r` qui signifie rouge et `o` qui signifie cercle.

Voyez la documentation de référence de plot pour une liste complète.

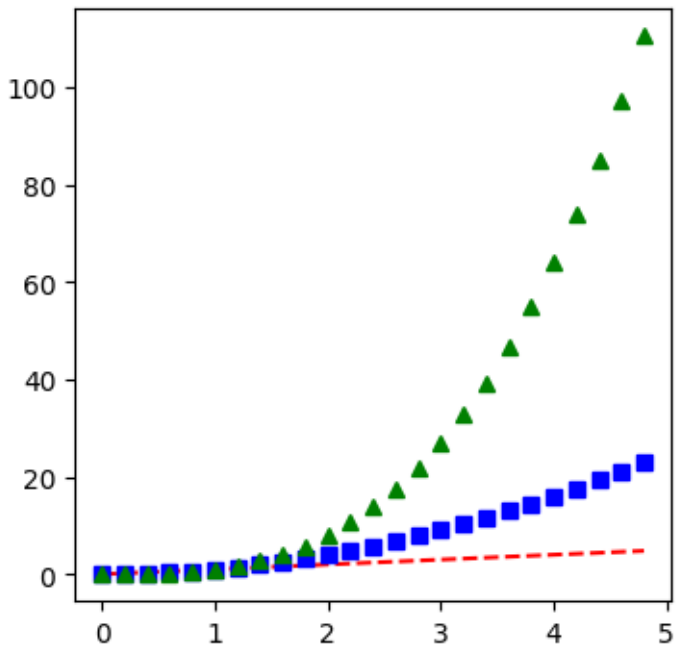
[4]:
```
# mais le plus souvent on passe à plot
# une liste de X ET une liste de Y
plt.plot([1, 2, 3, 4, 5], [1, 4, 9, 16, 25], 'ro')

# ici on veut dire d'utiliser
# pour l'axe des X : entre 0 et 5
# pour l'axe des Y : entre -5 et 20
plt.axis([0, 5, -5, 20])
```

```
plt.show()
```



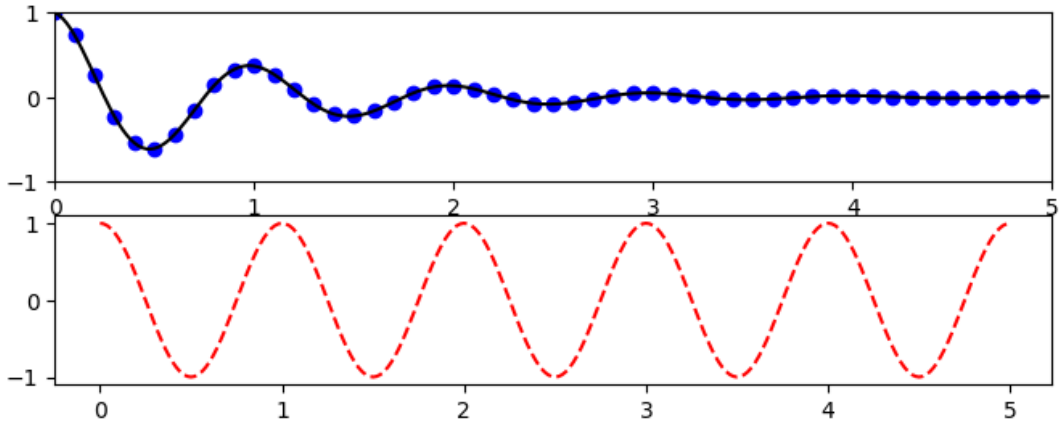
On peut très simplement dessiner plusieurs fonctions dans la même zone :

[5]:
```
# on peut changer la taille d'une figure précise
plt.figure(figsize=(4, 4))

# échantillon de points entre 0 et 5 espacés de 0.2
t = np.arange(0., 5., 0.2)

# plusieurs styles de ligne
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
# on pourrait ajouter d'autres plot bien sûr aussi
plt.show()
```



Plusieurs subplots

[6]:
```
# et ici c'est toujours la taille par défaut
# - celle fixée avec plt.rcParams - qui est utilisée

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

## deux domaines presque identiques
# celui-ci pour les points bleus
t1 = np.arange(0.0, 5.0, 0.1)
# celui-ci pour la ligne bleue
t2 = np.arange(0.0, 5.0, 0.02)

# cet appel n'est pas nécessaire
# vous pouvez vérifier qu'on pourrait l'enlever
plt.figure(1)
# on crée un 'subplot'
plt.subplot(211)
# le fonctionnement de matplotlib est dit 'stateful'
# par défaut on dessine dans le dernier objet créé
plt.axis([0, 5, -1, 1])
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

# une deuxième subplot
plt.subplot(212)
# on écrit dedans
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



C'est pour pouvoir construire de tels assemblages qu'il y a une fonction `plt.show()`, qui indique que la figure est terminée.

Il faut revenir un peu sur les arguments passés à `subplot`. Lorsqu'on écrit :

```
plt.subplot(211)
```

ce qui est par ailleurs juste un raccourci pour :

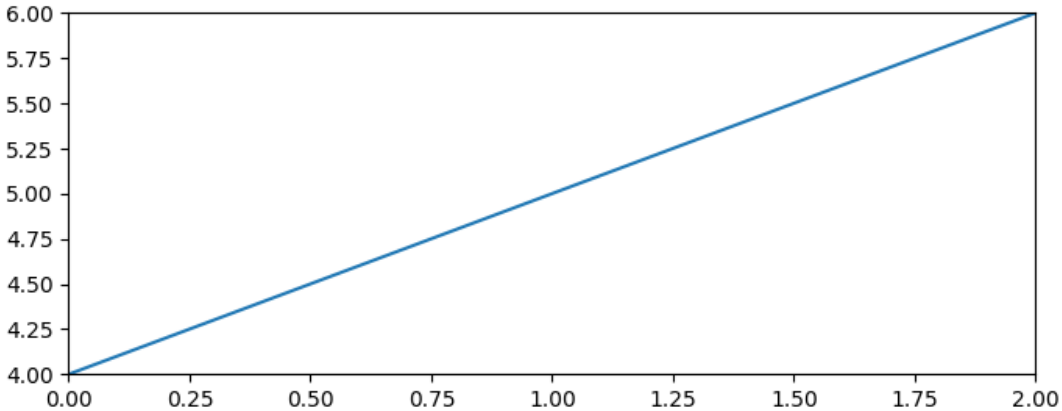
```
plt.subplot(2, 1, 1)
```

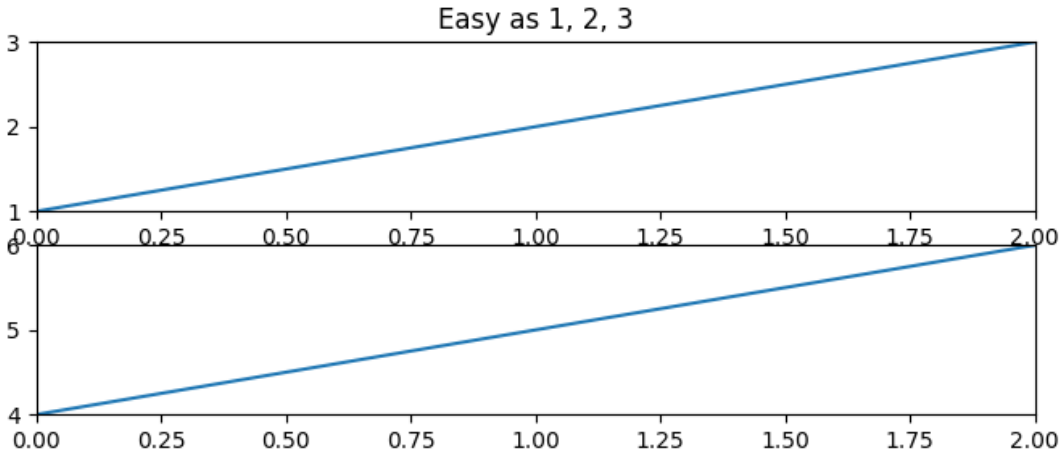
on veut dire qu'on veut créer un quadrillage de 2 lignes de 1 colonne, et que le subplot va occuper le 1er emplacement.

Plusieurs figures

En fait, on peut créer plusieurs figures, et plusieurs subplots dans chaque figure. Dans l'exemple qui suit on illustre encore mieux cette notion de statefulness. Je commence par vous donner l'exemple du tutorial tel quel :

[7]:
```
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.axis([0, 2, 1, 3])
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.axis([0, 2, 4, 6])
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.axis([0, 2, 4, 6])
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current;
                             # subplot(212) still current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
plt.show()
```

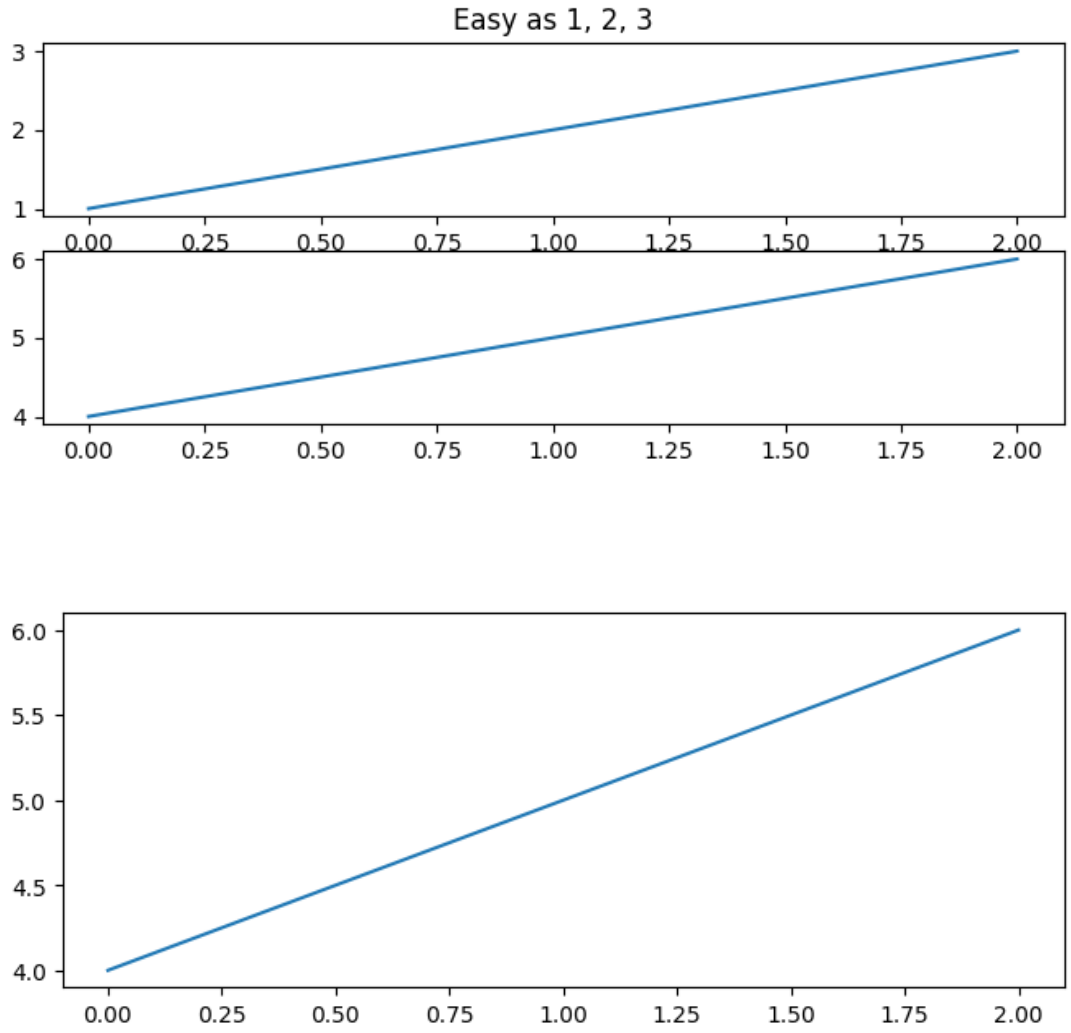




Cette façon de faire est améliorable. D'abord c'est source d'erreurs, il faut se souvenir de ce qui précède, et du coup, si on change un tout petit peu la logique, ça risque de casser tout le reste. En outre selon les environnements, on peut obtenir un vilain avertissement.

C'est pourquoi je vous conseille plutôt, pour faire la même chose que ci-dessus, d'utiliser **`plt.subplots`** qui vous retourne la figure avec ses subplots, que vous pouvez ranger dans des variables Python :

[8]:
```
# c'est assez déroutant au départ, mais
# traditionnellement les subplots sont appelés 'axes'
# c'est pourquoi ici on utilise ax1, ax2 et ax3 pour désigner
# des subplots

# ici je crée une figure et deux subplots,
# sur une grille de 2 lignes * 1 colonne
fig1, (ax1, ax2) = plt.subplots(2, 1)

# au lieu de faire plt.plot, vous pouvez envoyer
# la méthode plot à un subplot
ax1.plot([1, 2, 3])
ax2.plot([4, 5, 6])

fig2, ax3 = plt.subplots(1, 1)
ax3.plot([4, 5, 6])

# pour revenir au premier subplot
# il suffit d'utiliser la variable ax1
# attention on avait fait avec 'plt.title'
# ici c'est la méthode 'set_title'
ax1.set_title('Easy as 1, 2, 3')

plt.show()
```

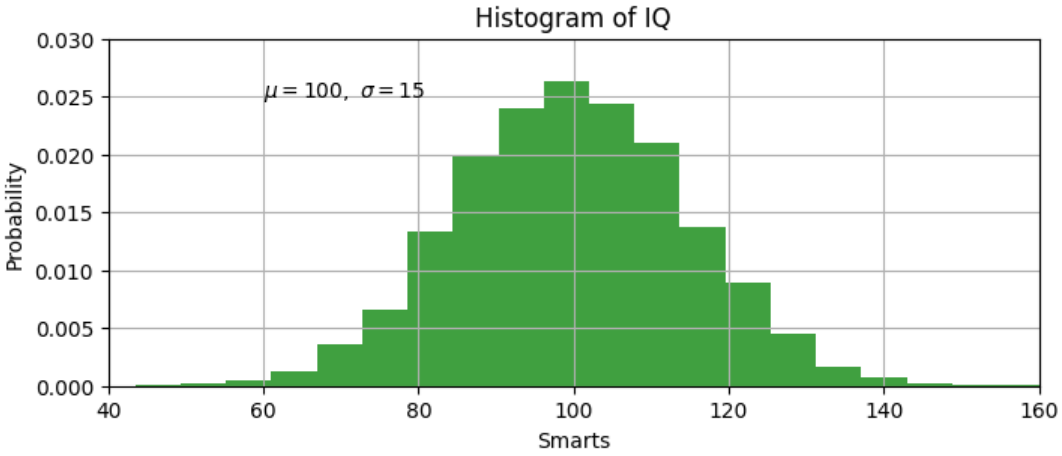


plt.hist

S'agissant de la dimension 2, voici le dernier exemple que nous tirons du tutoriel `matplotlib`, surtout pour illustrer `plt.hist`, mais qui montre aussi comment ajouter du texte :

[9]:
```
# pour être reproductible, on fixe la graine
# du générateur aléatoire
np.random.seed(19680801)

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# dessiner un histogramme
# on range les valeurs en 20 boites (bins)
n, bins, patches = plt.hist(x, 20, density=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
```

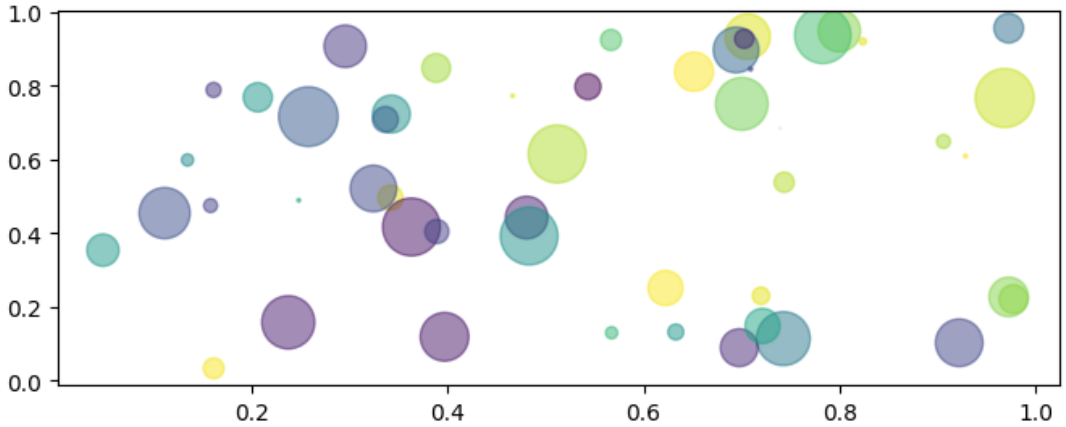
```
plt.grid(True)
plt.show()
```



plt.scatter

Je vous recommande aussi de regarder également la fonction `plt.scatter` qui permet de faire par exemple des choses comme ceci :

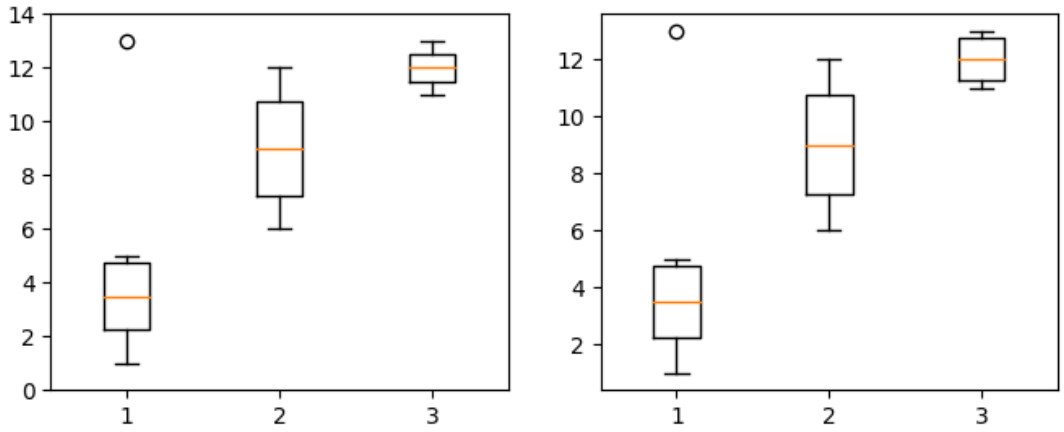
[10]:
```
# pour être reproductible, on fixe la graine
# du générateur aléatoire
np.random.seed(19680801)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radii

plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



plt.boxplot

Avec `boxplot` vous obtenez des boîtes à moustache :

[11]:
```
# la disposition des deux ssou-figures aka axes
fig, (ax1, ax2) = plt.subplots(1, 2)

# on peut passer à boxplot une liste de suites de nombres
# chaque suite donne lieu à une boite à moustache
# ici 3 suites
ax1.boxplot([[1, 2, 3, 4, 5, 13], [6, 7, 8, 10, 11, 12], [11, 12 ,13]])
ax1.set_ylim(0, 14)

# on peut aussi comme toujours lui passer un ndarray numpy
# attention c'est lu dans l'autre sens, ici aussi on a 3 suites de nombres
ax2.boxplot(np.array([[1, 6, 11],
                      [2, 7, 12],
                      [3, 8, 13],
                      [4, 10, 11],
                      [5, 11, 12],
                      [13, 12, 13]]))
plt.show()
```



7.29 w7-s10-c2-matplotlib-3d

matplotlib 3D

Nous poursuivons notre introduction à `matplotlib` avec les visualisations en 3 dimensions. Comme pour la première partie sur les fonctions en 2 dimensions, nous allons seulement paraphraser le tutoriel en ligne, avec l'avantage toutefois que nous procurent les notebooks.

[1]:
```
# la ration habituelle d'imports
import matplotlib.pyplot as plt
# et aussi numpy, même si ça n'est pas strictement nécessaire
import numpy as np
```

Pour pouvoir faire des visualisations en 3D, il vous faut importer ceci :

[2]:
```
# même si l'on n'utilise pas explicitement
# d'attributs du module Axes3D
# cet import est nécessaire pour faire
# des visualisations en 3D
from mpl_toolkits.mplot3d import Axes3D
```

Dans ce notebook nous allons utiliser un mode de visualisation un peu plus élaboré, mieux intégré à l'environnement des notebooks :

[3]:
```
# ce mode d'interaction va nous permettre de nous déplacer
# dans l'espace pour voir les courbes en 3D
# depuis plusieurs points de vue
%matplotlib notebook
```

Comme on va le voir très vite, avec ces réglages vous aurez la possibilité d'explorer interactivement les visualisations en 3D.

Un premier exemple : une courbe

Commençons par le premier exemple du tutorial, qui nous montre comment dessiner une ligne suivant une courbe définie de manière paramétrique (ici, x et y sont fonctions de z). Les points importants sont :

- la composition d'un plot (plusieurs figures, chacune composée de plusieurs subplots), reste bien entendu valide ; j'ai enrichi l'exemple initial pour mélanger un subplot en 3D avec un subplot en 2D ;
- l'utilisation du paramètre `projection='3d'` lorsqu'on crée un subplot qui va se prêter à une visualisation en 3D ;
- l'objet subplot ainsi créé est une instance de la classe `Axes3DSubplot` ;
- on peut envoyer à cet objet :
  - la méthode `plot` qu'on avait déjà vue pour la dimension 2 (c'est ce que l'on fait dans ce premier exemple) ;
  - des méthodes spécifiques à la 3D, que l'on voit dans les exemples suivants.

[4]:
```
# je choisis une taille raisonnable compte tenu de l'espace
# disponible dans fun-mooc
fig = plt.figure(figsize=(6, 3))

# voici la façon de créer un *subplot*
# qui se prête à une visualisation en 3D
ax = fig.add_subplot(121, projection='3d')

# à présent, copié de
# https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#line-plots
# on crée une courbe paramétrique
# où x et y sont fonctions de z
theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
# on fait maitenant un appel à plot normal
# mais avec un troisième paramètre
ax.plot(x, y, z, label='parametric curve')
ax.legend()

# on peut tout à fait ajouter un plot usuel
# dans un subplot, comme on l'a vu pour la 2D
```

```
ax2 = fig.add_subplot(122)
x = np.linspace(0, 10)
y = x**2
ax2.plot(x, y)
plt.show()
```

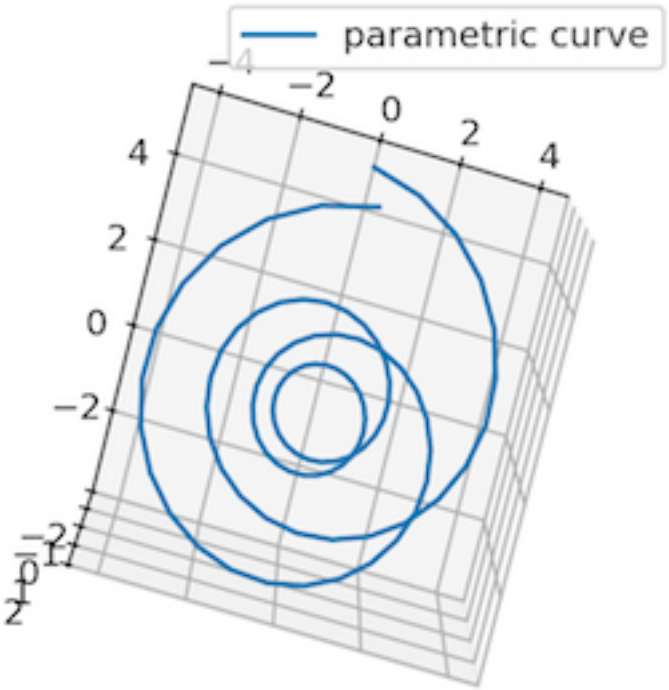
```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Un autre point à remarquer est qu'avec le mode d'interaction que nous avons choisi :

```
%matplotlib notebook
```

vous bénéficiez d'un mode d'interaction plus riche avec la figure. Par exemple, vous pouvez cliquer dans la figure en 3D, et vous déplacer pour changer de point de vue ; par exemple si vous sélectionnez l'outil `Pan/Zoom` (l'outil avec 4 flèches), vous pouvez arriver à voir ceci :



Les différents boutons d'outil sont décrits plus en détail ici. Je dois avouer ne pas arriver à tout utiliser lorsque la visualisation est faite dans un notebook, mais la possibilité de modifier le point de vue peut s'avérer intéressante pour explorer les données.

En explorant les autres exemples du tutorial, vous pouvez commencer à découvrir l'éventail des possibilités offertes par `matplotlib`.

`Axes3DSubplot.scatter`

Comme en dimension 2, `scatter` permet de montrer un nuage de points.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#scatter-plots

scatter3d_demo.py

[5]:
```
'''
==============
3D scatterplot
==============

Demonstration of a basic scatterplot in 3D.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

[6]:
```
fig = plt.figure(figsize=(4, 4))

def randrange(n, vmin, vmax):
    '''
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
    '''
    return (vmax - vmin)*np.random.rand(n) + vmin

ax = fig.add_subplot(111, projection='3d')

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for c, m, zlow, zhigh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, c=c, marker=m)

ax.set_xlabel('X Label')
plt.show()
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

Axes3DSubplot.plot_wireframe

Utilisez cette méthode pour dessiner en mode “fil de fer”.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#wireframe-plots.

wire3d_demo.py

[7]:
```
from mpl_toolkits.mplot3d import axes3d
```

[8]:
```
fig = plt.figure(figsize=(4, 4))

ax = fig.add_subplot(111, projection='3d')
```

```
# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Axes3DSubplot.plot_surface

Comme on s'en doute, `plot_surface` sert à dessiner des surfaces dans l'espace ; ces exemples montrent surtout comment utiliser des couleurs ou des patterns.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#surface-plots.

surface3d_demo.py

[9]:
```
'''
======================
3D surface (color map)
======================

Demonstrates plotting a 3D surface colored with the coolwarm color map.
The surface is made opaque by using antialiased=False.

Also demonstrates using the LinearLocator and custom formatting for the
z axis tick labels.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
```

[10]:
```
fig = plt.figure(figsize=(4, 4))

# dans le tuto on trouve
# fig, ax = plt.subplots(subplot_kw=dict(projection='3d'))

# personnellement je trouve plus facile à retenir ceci
ax = fig.add_subplot(111, projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plot the surface.
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
```

```
# Customize the z axis.
ax.set_zlim(-1.01, 1.01)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
surface3d_demo2.py
```

[11]:
```
'''
========================
3D surface (solid color)
========================

Demonstrates a very basic plot of a 3D surface using a solid color.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

[12]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

# Make data
u = np.linspace(0, 2 * np.pi, 30)
v = np.linspace(0, np.pi, 30)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))

# Plot the surface
ax.plot_surface(x, y, z, color='b')

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
surface3d_demo3.py
```

[13]:
```
'''
=========================
3D surface (checkerboard)
=========================

Demonstrates plotting a 3D surface colored in a checkerboard pattern.
'''
```

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator
import numpy as np
```

[14]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

# Make data.
X = np.arange(-5, 5, 0.25)
xlen = len(X)
Y = np.arange(-5, 5, 0.25)
ylen = len(Y)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Create an empty array of strings with the same shape as the meshgrid, and
# populate it with two colors in a checkerboard pattern.
colortuple = ('y', 'b')
colors = np.empty(X.shape, dtype=str)
for y in range(ylen):
    for x in range(xlen):
        colors[x, y] = colortuple[(x + y) % len(colortuple)]

# Plot the surface with face colors taken from the array we made.
surf = ax.plot_surface(X, Y, Z, facecolors=colors, linewidth=0)

# Customize the z axis.
ax.set_zlim(-1, 1)
ax.w_zaxis.set_major_locator(LinearLocator(6))

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/var/folders/9n/sxs31qhj1gnd6gk2v0ns8848000fn2/T/ipykernel_50867/244284003.
  py:26: MatplotlibDeprecationWarning: The w_zaxis attribute was deprecate
  d in Matplotlib 3.1 and will be removed in 3.8. Use zaxis instead.
  ax.w_zaxis.set_major_locator(LinearLocator(6))
```

Axes3DSubplot.plot_trisurf

`plot_trisurf` se prête aussi au rendu de surfaces, mais sur la base de maillages en triangles.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#tri-surface-plots.

`trisurf3d_demo.py`

[15]:
```
'''
======================
Triangular 3D surfaces
```

```
=====================

Plot a 3D surface with a triangular mesh.
'''

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

[16]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

n_radii = 8
n_angles = 36

# Make radii and angles spaces (radius r=0 omitted to eliminate duplication).
radii = np.linspace(0.125, 1.0, n_radii)
angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)

# Repeat all angles for each radius.
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)

# Convert polar (radii, angles) coords to cartesian (x, y) coords.
# (0, 0) is manually added at this stage,  so there will be no duplicate
# points in the (x, y) plane.
x = np.append(0, (radii*np.cos(angles)).flatten())
y = np.append(0, (radii*np.sin(angles)).flatten())

# Compute z to make the pringle surface.
z = np.sin(-x*y)

ax.plot_trisurf(x, y, z, linewidth=0.2, antialiased=True)

plt.show()
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

trisurf3d_demo2.py
```

[17]:
```
'''
============================
More triangular 3D surfaces
============================

Two additional examples of plotting surfaces with triangular mesh.

The first demonstrates use of plot_trisurf's triangles argument, and the
second sets a Triangulation object's mask and passes the object directly
to plot_trisurf.
'''

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.tri as mtri
```

[18]:
```
fig = plt.figure(figsize=(6, 3))

#============
# First plot
#============

# Make a mesh in the space of parameterisation variables u and v
u = np.linspace(0, 2.0 * np.pi, endpoint=True, num=50)
v = np.linspace(-0.5, 0.5, endpoint=True, num=10)
u, v = np.meshgrid(u, v)
u, v = u.flatten(), v.flatten()

# This is the Mobius mapping, taking a u, v pair and returning an x, y, z
# triple
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space to determine the triangles
tri = mtri.Triangulation(u, v)

# Plot the surface.  The triangles in parameter space determine which x, y, z
# points are connected by an edge.
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles, cmap=plt.cm.Spectral)
ax.set_zlim(-1, 1)


#============
# Second plot
#============

# Make parameter spaces radii and angles.
n_angles = 36
n_radii = 8
min_radius = 0.25
radii = np.linspace(min_radius, 0.95, n_radii)

angles = np.linspace(0, 2*np.pi, n_angles, endpoint=False)
angles = np.repeat(angles[..., np.newaxis], n_radii, axis=1)
angles[:, 1::2] += np.pi/n_angles

# Map radius, angle pairs to x, y, z points.
x = (radii*np.cos(angles)).flatten()
y = (radii*np.sin(angles)).flatten()
z = (np.cos(radii)*np.cos(angles*3.0)).flatten()

# Create the Triangulation; no triangles so Delaunay triangulation created.
triang = mtri.Triangulation(x, y)

# Mask off unwanted triangles.
xmid = x[triang.triangles].mean(axis=1)
ymid = y[triang.triangles].mean(axis=1)
mask = np.where(xmid**2 + ymid**2 < min_radius**2, 1, 0)
triang.set_mask(mask)

# Plot the surface.
```

```
ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_trisurf(triang, z, cmap=plt.cm.CMRmap)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Axes3DSubplot.contour

Pour dessiner des contours.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#contour-plots.

contour3d_demo.py

[19]:
```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

[20]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

contour3d_demo2.py

[21]:
```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

[22]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contour(X, Y, Z, extend3d=True, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

contour3d_demo3.py

[23]:
```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

[24]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contour(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Axes3DSubplot.contourf

Comme `Axes3DSubplot.contour`, mais avec un rendu plein plutôt que sous forme de lignes (le `f` provient de l'anglais filled).

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#filled-contour-plots.

contourf3d_demo.py

[25]:
```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

[26]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
cset = ax.contourf(X, Y, Z, cmap=cm.coolwarm)
ax.clabel(cset, fontsize=9, inline=1)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

contourf3d_demo2.py

[27]:
```
"""
.. versionadded:: 1.1.0
   This demo depends on new features added to contourf3d.
"""

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
```

[28]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
X, Y, Z = axes3d.get_test_data(0.05)
ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.3)
cset = ax.contourf(X, Y, Z, zdir='z', offset=-100, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='x', offset=-40, cmap=cm.coolwarm)
cset = ax.contourf(X, Y, Z, zdir='y', offset=40, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-40, 40)
ax.set_ylabel('Y')
ax.set_ylim(-40, 40)
ax.set_zlabel('Z')
ax.set_zlim(-100, 100)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Axes3DSubplot.add_collection3d

Pour afficher des polygones.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#polygon-plots.

[29]:
```
"""
==============================================
Generate polygons to fill under 3D line graph
==============================================

Demonstrate how to create polygons which fill the space under a line
graph. In this example polygons are semi-transparent, creating a sort
of 'jagged stained glass' effect.
"""

from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import PolyCollection
import matplotlib.pyplot as plt
from matplotlib import colors as mcolors
import numpy as np
```

[30]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

def cc(arg):
    return mcolors.to_rgba(arg, alpha=0.6)

xs = np.arange(0, 10, 0.4)
verts = []
zs = [0.0, 1.0, 2.0, 3.0]
for z in zs:
    ys = np.random.rand(len(xs))
    ys[0], ys[-1] = 0, 0
    verts.append(list(zip(xs, ys)))
```

```
poly = PolyCollection(verts, facecolors=[cc('r'), cc('g'), cc('b'),
                                         cc('y')])
poly.set_alpha(0.7)
ax.add_collection3d(poly, zs=zs, zdir='y')

ax.set_xlabel('X')
ax.set_xlim3d(0, 10)
ax.set_ylabel('Y')
ax.set_ylim3d(-1, 4)
ax.set_zlabel('Z')
ax.set_zlim3d(0, 1)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Axes3DSubplot.bar

Pour construire des diagrammes à barres.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#bar-plots.

bars3d_demo.py

[31]:
```
"""
=========================================
Create 2D bar graphs in different planes
=========================================

Demonstrates making a 3D plot which has 2D bar graphs projected onto
planes y=0, y=1, etc.
"""

from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
```

[32]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)

    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set will be colored cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'
    ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Axes3DSubplot.quiver

Pour afficher des champs de vecteurs sous forme de traits.

Tutoriel original : https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html#quiver.

quiver3d_demo.py

[33]:
```
'''
==============
3D quiver plot
==============

Demonstrates plotting directional arrows at points on a 3d meshgrid.
'''

from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np
```

[34]:
```
fig = plt.figure(figsize=(4, 4))
ax = fig.add_subplot(111, projection='3d')

# Make the grid
x, y, z = np.meshgrid(np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.2),
                      np.arange(-0.8, 1, 0.8))

# Make the direction data for the arrows
u = np.sin(np.pi * x) * np.cos(np.pi * y) * np.cos(np.pi * z)
v = -np.cos(np.pi * x) * np.sin(np.pi * y) * np.cos(np.pi * z)
w = (np.sqrt(2.0 / 3.0) * np.cos(np.pi * x) * np.cos(np.pi * y) *
     np.sin(np.pi * z))

ax.quiver(x, y, z, u, v, w, length=0.1, normalize=True)

plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

7.30 w7-s10-c3-notebooks-interactifs

Notebooks interactifs

7.30.1 Complément - niveau basique

Pour conclure cette série sur les outils de visualisation, nous allons voir quelques fonctionnalités disponibles uniquement dans l'environnement des notebooks, et qui offrent des possibilités supplémentaires par rapport aux visualisations que l'on a vues jusqu'à maintenant.

installation

Pour exécuter ou créer un notebook depuis votre ordinateur, il vous faut installer Jupyter, ce que se fait bien sûr depuis le terminal :

```
pip install jupyter
```

En 2020 il existe deux versions de l'interface Jupyter dites classic et lab, la seconde étant plus puissante en termes d'UI ; pour installer le tout, faire plutôt

```
pip install jupyterlab
```

Pour lancer un serveur jupyter, faire selon le mode choisi

```
jupyter notebook
# ou
jupyter lab
```

Contenus

Pour le contenu des notebooks :

- une cellule est marquée comme étant soit du code, soit du texte(markdown) ;
- pour les cellules de markdown, on peut très simplement :
  - insérer des formules mathématiques, en insérant un fragment de LaTeX entre deux simples `$`, comme $\forall x \in \mathbb{R}$, ou encore
    sur une ligne séparée en entourant entre deux doubles dollars `$$`, comme

$$\forall \epsilon > 0, \exists \alpha > 0, \forall x, |x - x_0| < \epsilon \implies |f(x) - f(x_0| < \epsilon$$

  - et bien sûr toute la panoplie des effets markdown, quoi qu'il faut se méfier car tout cela n'est pas très bien standardisé actuellement.
- un notebook choisit son kernel (en clair son langage) ; le mot Jupyter vient de Julia + Python + R, et aujourd'hui il y a moyen de faire tourner presque tous les langages, même `bash` et `C++` (mais en mode interprété bien sûr)

Courbes

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Comme on l'a déjà vu plein de fois, la bonne façon de créer un graphique matplotlib c'est avec la formule magique suivante :

[2]:
```
# ça c'est pour choisir la sortie 'notebook'
%matplotlib notebook

# et ça c'est pour dire 'interactive on'
# pour éviter de devoir plt.show() tout le temps
plt.ion()
```

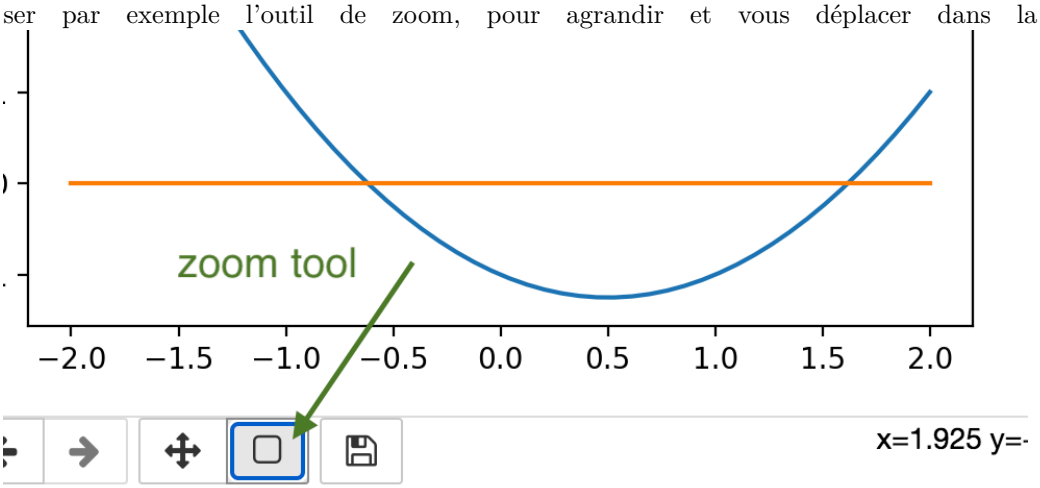
[2]:
```
<contextlib.ExitStack at 0x103bddea0>
```

Avec ces réglages - enfin surtout le premier - il y a pas mal de possibilités qui sont très pratiques :

— pour commencer on peut changer la taille de la courbe en cliquant sur le petit coin visible en bas

à droite de la figure

— les courbes apparaissent avec un barre d'outils en dessous ; entraînez-vous à utiliser par exemple l'outil de zoom, pour agrandir et vous déplacer dans la courbe



À titre d'exercice, sur cette courbe le nombre d'or correspond à une des racines du polynôme, à vous de trouver sa valeur avec une précision de

[3]:
```
plt.figure(figsize=(2, 2))
X = np.linspace(-2, 2)
ZERO = X * 0
def golden(x):
    return x**2 - x - 1
plt.plot(X, golden(X));
plt.plot(X, ZERO);
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Voici à quoi je suis arrivé de mon côté (je ne dis pas que c'est forcément la méthode la plus rapide pour trouver le nombre d'or ;-) :
Mais tous les outils de visualisation décents vons proposer des mécanismes analogues, soyez-y attentifs car ça fait parfois gagner beaucoup de temps.

Exemple de notebook interactif

Je vous signale enfin un exemple de notebook publié par la célèbre revue Nature, qui pourra vous donner une idée de ce qu'il est possible de faire avec un notebook interactif. Interactif dans le sens où on peut faire varier les paramètres d'une expérience et voir l'impact du changement se refléter immédiatement sur la visualisation.

Comme il n'est malheureusement plus actif en ligne semble-t-il, je vous invite à le faire marcher localement à partir de la version sur github ici.

7.30.2 Complément - niveau intermédiaire

Une visualisation interactive simple : **`interact`**

Pour refaire de notre coté quelque chose d'analogue, nous allons commencer par animer la fonction sinus, avec un bouton pour régler la fréquence. Pour cela nous allons utiliser la fonction `interact` ; à nouveau c'est un utilitaire qui fait partie de l'écosystème des notebooks, et plus précisément du module `ipywidgets` :

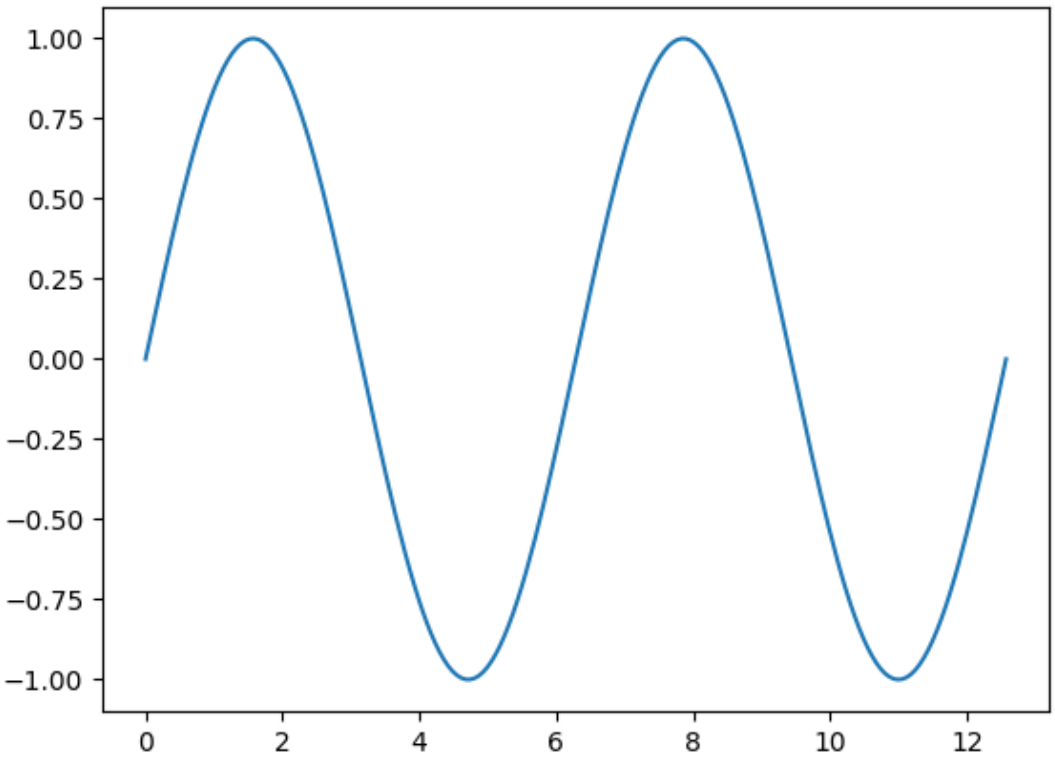
[4]:
```
# dans cette partie on a besoin de
# revenir dans un mode plus usuel
%matplotlib inline
```

[5]:
```
from ipywidgets import interact
```

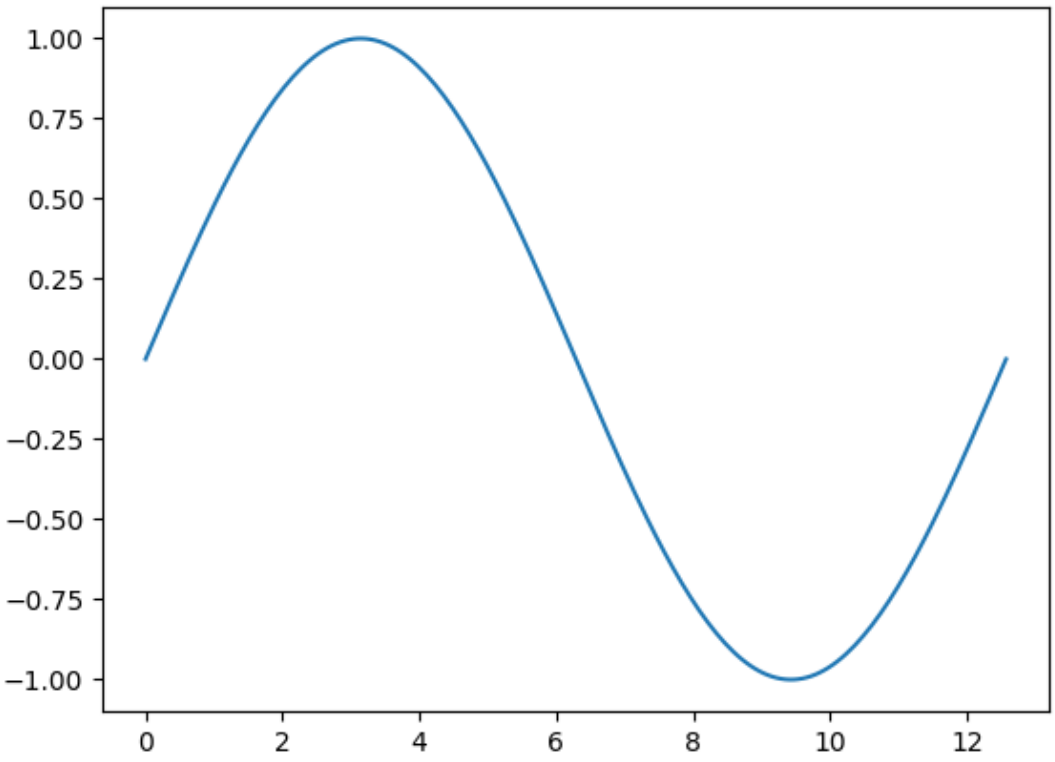
Dans un premier temps, j'écris une fonction qui prend en paramètre la fréquence, et qui dessine la fonction sinus sur un intervalle fixe de 0. à 4π :

[6]:
```
def sinus(freq):
    X = np.linspace(0., 4*np.pi, 200)
    Y = np.sin(freq*X)
    plt.plot(X, Y)
```

[7]:
```
sinus(1)
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12

[8]:
```
sinus(0.5)
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
12

Maintenant, plutôt que de tracer individuellement les courbes une à une, j'utilise `interact` qui va m'afficher une réglette pour changer le paramètre `freq`. Ça se présente comme ceci :

[9]:
```
# je change maintenant la taille des visualisations
plt.rcParams["figure.figsize"] = (12, 4)
```

[10]:
```
interact(sinus, freq=(0.5, 10., 0.25));
```

```
interactive(children=(FloatSlider(value=5.25, description='freq', max=10.0, min=0.5,␣
↪step=0.25), Output()), _d…
```

Mécanisme d'`interact`

La fonction `interact` s'attend à recevoir :

— en premier argument : une fonction `f` ;
— et ensuite autant d'arguments nommés supplémentaires que de paramètres attendus par `f`.

Comme dans mon cas la fonction `sinus` attend un paramètre nommé `freq`, le deuxième argument de `interact` lui est passé aussi avec le nom `freq`.

Les objets `Slider`

Chacun des arguments à `interact` (en plus de la fonction) correspond à un objet de type `Slider` (dans la ménagerie de `ipywidget`). Ici en passant juste le tuple `(0.5, 10., 0.25)` j'utilise un raccourci pour dire que je veux pouvoir régler le paramètre `freq` sur une plage allant de `0.5` à `10` avec un pas de `0.25`.

Mon premier exemple avec `interact` est en réalité équivalent à ceci :

[11]:
```
from ipywidgets import FloatSlider
```

[12]:
```
# exactement équivalent à la version ci-dessus
interact(sinus, freq=FloatSlider(min=0.5, max=10., step=0.25));
```

```
interactive(children=(FloatSlider(value=0.5, description='freq', max=10.0, min=0.5,␣
↪step=0.25), Output()), _do…
```

Mais en utilisant la forme bavarde, je peux choisir davantage d'options, comme notamment :

— mettre `continuous_update = False` ; l'effet de ce réglage, c'est que l'on met à jour la figure seulement lorsque je lâche la réglette ; c'est utile lorsque les calculs sont un peu lents, comme ici avec l'infrastructure notebook qui est à distance ;
— mettre `value=1.` pour choisir la valeur initiale :

[13]:
```
# exactement équivalent à la version ci-dessus
# sauf qu'on ne redessine que lorsque la réglette
# est relâchée
interact(sinus, freq=FloatSlider(min=0.5, max=10.,
                                 step=0.25, value=1.,
                                 continuous_update=False));
```

```
interactive(children=(FloatSlider(value=1.0, continuous_update=False,␣
↪description='freq', max=10.0, min=0.5, s…
```

Plusieurs paramètres

Voyons tout de suite un exemple avec deux paramètres, je vais écrire maintenant une fonction qui me permet de changer aussi la phase :

[14]:
```
def sinus2(freq, phase):
    X = np.linspace(0., 4*np.pi, 200)
    Y = np.sin(freq*(X+phase))
    plt.plot(X, Y)
```

Et donc maintenant je passe à `interact` un troisième paramètre :

[15]:
```
interact(sinus2,
         freq=FloatSlider(min=0.5, max=10., step=0.5,
                          continuous_update=False),
         phase=FloatSlider(min=0., max=2*np.pi, step=np.pi/6,
                           continuous_update=False),
        );
```

```
interactive(children=(FloatSlider(value=0.5, continuous_update=False,␣
↪description='freq', max=10.0, min=0.5, s…
```

Bouche-trou : **`fixed`**

Si j'ai une fonction qui prend plus de paramètres que je ne veux montrer de réglettes, je peux fixer un des paramètres par exemple comme ceci :

[16]:
```
from ipywidgets import fixed
```

[17]:
```
# avec une fonction à deux argument,
# je peux en fixer un, et n'avoir qu'une réglette
# pour fixer celui qui est libre
interact(sinus2, freq=fixed(1.),
         phase=FloatSlider(min=0., max=2*np.pi, step=np.pi/6),
        );
```

```
interactive(children=(FloatSlider(value=0.0, description='phase', max=6.
↪283185307179586, step=0.52359877559829…
```

7.30.3 Widgets

Il existe toute une famille de widgets, dont `FloatSlider` est l'exemple le plus courant, mais vous pouvez aussi :

- créer des radio bouton pour entrer un paramètre booléen ;
- ou une saisie de texte pour entre un paramètre de type `str` ;
- ou une liste à choix multiples…

Bref, vous pouvez créer une mini interface-utilisateur avec des objets graphiques simples choisis dans une palette assez complète pour ce type d'application.

Voyez les détails complets sur `readthedocs.io`

[18]:
```
# de même qu'un tuple était ci-dessus un raccourci pour un FloatSlider
# une liste ou un dictionnaire est transformé(e) en un Dropdown
interact(sinus, freq={'rapide': 10., 'moyenne': 1., 'lente': 0.1});
```

```
interactive(children=(Dropdown(description='freq', options={'rapide': 10.0, 'moyenne':
↪ 1.0, 'lente': 0.1}, val…
```

Voyez la liste complète des widgets ici.

Dashboards

Lorsqu'on a besoin de faire une interface un peu plus soignée, on peut créer sa propre disposition de boutons et autres réglages.

Voici un exemple de dashboard, uniquement pour vous donner une meilleure idée, qui pour changer agit sur une visualisation réalisée avec plot.ly plutôt que matplotlib :

[19]:
```
import plotly
plotly.__version__
```

[19]:
```
'5.11.0'
```

[20]:
```
# on importe la bibliothèque plot.ly
import chart_studio.plotly as py
import plotly.graph_objs as go
```

[21]:
```
# il est impératif d'utiliser plot.ly en mode 'offline'
# pour in mode interactif,
# car sinon les affichages sont beaucoup trop lents
import plotly.offline as pyoff

pyoff.init_notebook_mode()
```

[22]:
```
# les widgets pour construire le tableau de bord
from ipywidgets import (interactive_output,
                        IntSlider, Dropdown, Layout, HBox, VBox, Text)
from IPython.display import display
```

[23]:
```
# une fonction sinus à 4 réglages
# qu'on réalise pour changer avec plot.ly
# et non pas avec matplotlib
def sinus4(freq, phase, amplitude, domain):

    X = np.linspace(0., domain*np.pi, 500)
    Y = amplitude * np.sin(freq*(X+phase))

    data = [ go.Scatter(x=X, y=Y, mode='lines', name='sinus') ]
    # je fixe l'amplitude à 10 pour que les animations
    # soient plus parlantes
    layout = go.Layout(
        yaxis = {'range' : [-10, 10]},
        title="Exemple de graphique interactif avec dashboard",
        height=500,
        width=500,
    )
    figure = go.Figure(data=data, layout=layout)
    pyoff.iplot(figure)
```

[24]:
```
def my_dashboard():
    """
    create and display a dashboard
    return a dictionary name->widget suitable for interactive_output
    """
    # dashboard pieces as widgets
    l_75 = Layout(width='75%')
    l_50 = Layout(width='50%')
    l_25 = Layout(width='25%')

    w_freq = Dropdown(options=list(range(1, 10)),
                      value = 1,
                      description = "fréquence",
                      layout=l_50)
    w_phase = FloatSlider(min=0., max = 2*np.pi, step=np.pi/12,
                          description="phase",
                          value=0., layout=l_75)
    w_amplitude = Dropdown(options={"micro" : .1,
                                    "mini" : .5,
                                    "normal" : 1.,
                                    "grand" : 3.,
                                    "énorme" : 10.},
                           value = 3.,
                           description = "amplitude",
                           layout = l_25)
    w_domain = IntSlider(min=1, max=10, description="dom. n *  ", layout=l_50)

    # make up a dashboard
    dashboard = VBox([HBox([w_amplitude, w_phase]),
                      HBox([w_domain, w_freq]),
                     ])
    display(dashboard)
    return dict(freq=w_freq, phase=w_phase,
                amplitude=w_amplitude, domain=w_domain)
```

Avec tout ceci en place on peut montrer un dialogue interactif pour changer tous les paramètres de sinus4.

[25]:
```
# interactively call sinus4
# attention il reste un bug:
# au tout début rien ne s'affiche,
# il faut faire bouger au moins un réglage
interactive_output(sinus4, my_dashboard())
```

```
VBox(children=(HBox(children=(Dropdown(description='amplitude', index=3,␣
 ↪layout=Layout(width='25%'), options={…
```

[25]:
```
Output()
```

7.31 w7-s10-c4-animations-matplotlib

Animations interactives avec **matplotlib**

7.31.1 Complément - niveau avancé

Nous allons voir dans ce notebook comment créer une animation avec matplotlib et tirer parti des widgets dans un notebook pour rendre ces animations interactives.

Ce sera l'occasion de décortiquer un exemple un peu sophistiqué, où l'utilisation d'un générateur permet de rendre l'implémentation plus propre et plus élégante.

Le sujet

En guise d'illustration, nous allons créer :

- une animation matplotlib : disons que l'on veut faire défiler horizontalement une sinusoïde ;
- un widget interactif : disons que l'on veut pouvoir changer la vitesse de défilement.

Les outils

Pour fabriquer cela nous aurons besoin principalement :

- de la librairie d'animation de matplotlib, et spécifiquement le sous-package `animation`,
- et des widgets du module `ipywidgets`.

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation
```

[2]:
```
from IPython.display import display as display_widget
from ipywidgets import IntSlider
```

La logique

Dans un notebook précédent nous avions abordé la fonction `interact`, de la librairie `ipywidgets`, qui nous permettait d'appeler interactivement une fonction avec des arguments choisis interactivement via une série de widgets.

Si on essaie d'utiliser `interact` pour faire des animations, l'effet visuel, qui revient à effacer/recalculer une visualisation à chaque changement de valeur des entrées, donne un rendu peu agréable à l'oeil.

C'est pourquoi ici la logique va être un petit peu différente :

- c'est une fonction native de `matplotlib` qui implémente la boucle principale, en travaillant sur un objet `Figure`,
- et le widget est utilisé uniquement pour modifier une variable python ;
- pour simplifier notre code, l'échange d'informations entre ces deux morceaux se fait le plus simplement possible, via une variable globale.

Bien entendu cette dernière pratique n'est pas recommandée dans du code de production, et le lecteur intéressé est invité à améliorer ce point.

Version non interactive et basique

Pour commencer nous allons voir comment utiliser `matplotlib.animation` sans interactivité.

Cette version est inspirée du tutorial matplotlib sur les animations, qui montre d'ailleurs d'autres animations plus complexes et convaincantes, comme le double pendule par exemple.

Mais avant tout choisissons ce mode de rendu :

[3]:
```
%matplotlib notebook
```

Nous allons utiliser la fonction `animation.FuncAnimation` ; celle-ci s'attend à recevoir en argument, principalement :

— une figure,
— et une fonction d'affichage.

La logique est que la fonction d'affichage est appelée à intervalles réguliers par `FuncAnimation`, elle doit retourner un itérable d'objets affichable dans la figure.

Dans notre cas, nous allons créer une instance unique d'un objet `plot` ; cette instance sera modifiée à chaque frame par la fonction d'affichage, qui le renverra dans un tuple à un élément (ceci parce qu'un itérable est attendu).

Version basique dite tout-en-un

[4]:
```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

# on commence par créer une figure;
figure1 = plt.figure(figsize=(4, 2))

# en général pour une animation
# il est important de fixer les bornes en x et en y
# pour ne pas avoir d'artefacts de changement d'échelle
# pendant l'animation
ax1 = plt.axes(xlim=(0, 2), ylim=(-1.5, 1.5))

# on crée aussi un plot vide qui va être modifié à chaque frame
line1, = ax1.plot([], [], linewidth=2)

# la vitesse de défilement
speed = 1

# une globale; c'est vilain !
offset = 0.

# la fonction qui calcule et affiche chaque frame
def compute_and_display(n):
    global offset
    offset += speed / 100
    x = np.linspace(0, 2, 1000)
    y = np.sin(2 * np.pi * (x - offset))
    line1.set_data(x, y)
    return line1,


# la fonction magique pour animer une figure
# blit=True est une optimisation graphique
# pour ne raffficher que le nécessaire
anim = animation.FuncAnimation(figure1,
                               func=compute_and_display,
                               frames=50, repeat=False,
                               interval=40, blit=True)
```

```
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Séparation calcul et affichage

[5]:
```
plt.ion()
```

[5]:
```
<contextlib.ExitStack at 0x113b73f40>
```

On voit qu'on a appelé `FuncAnimation` avec `frames=50` et `interval=40` (ms) ; ce qui correspond donc à 25 images par seconde, soit une durée de deux secondes.

Profitons-en pour voir tout de suite une amélioration possible. Il serait souhaitable de séparer :

— d'une part la logique du calcul - ou de l'acquisition, si on voulait par exemple faire du postprocessing temps réel d'images vidéo,
— et d'autre part l'affichage à proprement parler.

Pour cela, remarquez que le paramètre `frames` est documenté comme pouvant être un itérateur. La logique en fait à l'oeuvre dans `FuncAnimation` est que

— `frames` est un itérateur qui va énumérer des données,
— à chaque frame cet itérateur est avancé avec `next()`, et son retour est passé à la fonction d'affichage.

En guise de commodité, lorsqu'on passe comme ci-dessus `frames=200`, la fonction transforme cela en `frames=range(200)`. C'est pourquoi d'ailleurs il est important que `compute_and_display` accepte un paramètre unique, même si nous n'en avons pas eu besoin.

Cette constatation nous amène à une deuxième version, en concevant un générateur pour le calcul, qui est passé à `FuncAnimation` comme paramètre `frames`.

Version non interactive, mais avec séparation calcul / affichage

[6]:
```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

figure2 = plt.figure(figsize=(4, 2))
ax2 = plt.axes(xlim=(0, 2), ylim=(-1.5, 1.5))
line2, = ax2.plot([], [], linewidth=2)

# la vitesse de défilement
speed = 1

# remarquez qu'on n'a plus besoin de globale ici
# une locale dans le générateur est bien plus propre

# la logique du calcul est conçue comme un générateur
def compute():
    offset = 0.
    # nous sommes dans un générateur, il n'y a pas
    # de contrindication à tourner indéfiniment
    while True:
        offset += speed / 100
```

```
        x = np.linspace(0, 2, 1000)
        y = np.sin(2 * np.pi * (x - offset))
        # on décide de retourner un tuple X, Y
        # qui est passé tel-quel à l'affichage
        yield x, y

# la fonction qui affiche
def display(frame):
    # on retrouve nos deux éléments x et y
    x, y = frame
    # il n'y a plus qu'à les afficher
    line2.set_data(x, y)
    return line2,


anim = animation.FuncAnimation(figure2,
                               func=display,
                               frames=compute(),
                               interval=40, blit=True)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
/Users/tparment/miniconda3/envs/flotpython-course/lib/python3.10/site-packa
  ges/matplotlib/animation.py:879: UserWarning: Animation was deleted with
  out rendering anything. This is most likely not intended. To prevent del
  etion, assign the Animation to a variable, e.g. `anim`, that exists unti
  l you output the Animation using `plt.show()` or `anim.save()`.
  warnings.warn(
```

Cette fois l'animation ne se termine jamais, mais dans le notebook vous pouvez cliquer le bouton bleu en haut à droite de la figure pour la faire cesser.

Avec interactivité

Pour rendre ceci interactif, nous allons simplement ajouter un widget qui nous permettra de régler la vitesse de défilement.

Version interactive avec widget pour choisir la vitesse

[7]:
```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

from IPython.display import display as display_widget
from ipywidgets import IntSlider

figure3 = plt.figure(figsize=(4, 2))
ax3 = plt.axes(xlim=(0, 2), ylim=(-1.5, 1.5))
line3, = ax3.plot([], [], linewidth=2)

# un widget pour choisir la vitesse de défilement
speed_slider = IntSlider(min=1, max=10, value=3,
                         description="Vitesse:")

def compute():
```

```
    offset = 0.
    # nous sommes dans un générateur, il n'y a pas
    # de contrindication à tourner indéfiniment
    while True:
        # on accède à la vitesse via le widget
        offset += speed_slider.value / 100
        x = np.linspace(0, 2, 1000)
        y = np.sin(2 * np.pi * (x - offset))
        # on décide de retourner un tuple X, Y
        # qui est passé tel-quel à l'affichage
        yield x, y

# la fonction qui affiche
def display(frame):
    # on retrouve nos deux éléments x et y
    x, y = frame
    # il n'y a plus qu'à les afficher
    line3.set_data(x, y)
    return line3,


anim = animation.FuncAnimation(figure3,
                               func=display,
                               frames=compute(),
                               interval=40, blit=True)

display_widget(speed_slider)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
IntSlider(value=3, description='Vitesse:', max=10, min=1)
```

Conclusion

Avec une approche de programmation plus traditionnelle, on pourrait penser avoir besoin de recourir à plusieurs threads pour implémenter ce genre de visualisation interactive.

En effet, vous remarquerez que dans cette dernière version, en termes de parallèlisme, on peut avoir l'impression que 3 choses ont lieu principalement en même temps :

— la logique de calcul, qui en substance est décrite comme un unique `while True:`,
— la logique d'affichage, qui est cadencée par `FuncAnimation`,
— et la logique interactive, qui gère le widget sur interaction de l'utilisateur.

Le point à retenir ici est que, grâce à la fois au générateur et au notebook, on n'a pas du tout besoin de gérer soi-même cet aspect de programmation parallèle.

Nous verrons d'ailleurs dans la semaine suivante comment le paradigme de programmation asynchrone de Python a été bâti, au dessus de cette capacité qu'offre le générateur, pour utiliser ce type d'approche de manière systématique, afin de faire tourner dans un seul thread et de manière transparente, un grand nombre de logiques.

Pour en savoir plus

Voyez :

— la documentation du module `animation`,
— ainsi que le tutoriel dont on s'est inspiré pour ce notebook, surtout pour voir d'autres animations plus élaborées.

7.32 w7-s10-c5-bokeh-et-al

Autres bibliothèques de visualisation

7.32.1 Complément - niveau basique

Pour conclure cette séquence sur les outils de visualisation, nous allons très rapidement évoquer des alternatives à la bibliothèque `matplotlib`, sachant que le domaine est en pleine expansion.

Le poids du passé

On a vu que `matplotlib` est un outil relativement complet. Toutefois, on peut lui reprocher deux défauts majeurs.

— D'une part, `matplotlib` a choisi d'offrir une interface aussi proche que possible de ce qui existait préalablement en MatLab. C'est un choix tout à fait judicieux dans l'optique d'attirer la communauté MatLab à des outils open source basés sur Python et numpy. Mais en contrepartie, cela implique d'adopter tels quels des choix de conception.
— Et notamment, en suivant cette approche on hérite d'un modèle mental qui est plus orienté vers la sortie vers du papier que vers la création de documents interactifs.

Ceci, ajouté à l'explosion du domaine de l'analyse et de la visualisation de données, explique la largeur de l'offre en matière de bibliothèques de visualisation alternatives.

Dans ce complément nous allons explorer notamment quelques techniques qui permettent de faire des visualisations interactives ; c'est-à-dire où l'on peut modifier la visualisation en fonction de paramètres, réglables facilement.

C'est quelque chose qui demande un peu de soin car, si on utilise `interact()` brutalement, on obtient des visualisations qui "flashent", car à chaque changement du contexte on recalcule toute une image, plutôt que de modifier l'image précédente. Ça semble un détail, mais l'oeil est très sensible à ce type d'artefact, et à l'expérience ce détail a plus d'impact qu'on ne pense.

`bokeh`

Commençons par signaler notamment la bibliothèque `bokeh`, qui est développée principalement par Anaconda, dans un modèle open source.

`bokeh` présente quelques bonnes propriétés qui nous semblent mériter d'être signalées.

Pour commencer cette bibliothèque utilise une architecture qui permet de penser la visualisation comme quelque chose d'interactif (disons une page html), et non pas de figé comme lorsqu'on pense en termes de feuille de papier. Notamment elle permet de faire collaborer du code Python avec du code JavaScript, qui offre immédiatement des possibilités bien plus pertinentes lorsqu'il s'agit de créer des interactions utilisateur qui soient attractives et efficaces. Signalons en passant, à cet égard, qu'elle utilise la librairie JavaScript `d3.js`, qui est devenu un standard de fait plus ou moins incontournable dans le domaine de la visualisation.

En tout état de cause, elle offre une interface de programmation qui tient compte d'environnements comme les notebooks, ce qui peut s'avérer un atout précieux si vous utilisez massivement ce support, comme on va le voir, précisément, dans ce notebook.

Il peut aussi être intéressant de savoir que `bokeh` offre des possibilités natives de visualisation de graphes et de données géographiques.

Par contre à ce stade du développement, la visualisation en 3D n'est sans doute pas le point fort de `bokeh`. C'est une option qui reste possible (voir par exemple ceci), mais cela est pour l'instant considéré comme une extension de la librairie, et donc n'est accessible qu'au prix de l'écriture de code javascript.

Pour une présentation plus complète, je vous renvoie à la documentation utilisateur.

`bokeh` dans les notebooks

Nous allons rapidement illustrer ici comment `bokeh` s'interface avec l'environnement des notebooks pour créer une visualisation interactive. Vous remarquerez que dans le code qui suit, on n'a pas eu besoin de mentionner de magic ipython, comme lorsqu'on avait du faire dans le complément sur les notebooks interactifs :

```
%matplotlib notebook
```

[1]:
```
import numpy as np
```

[2]:
```
# l'attirail de notebooks interactifs
from ipywidgets import interact, fixed, FloatSlider, Dropdown
```

[3]:
```
# les imports pour bokeh
from bokeh.plotting import figure, show
# dans la rubrique entrée-sortie, on trouve
# les outils pour produire du html
#  (le mode par défaut)
# ou pour interactig avec un notebook
from bokeh.io import push_notebook, output_notebook
```

[4]:
```
# c'est cette déclaration qui remplace
# si on veut la magic  '%matplotlib notebook'
output_notebook()
```

---

[5]:
```
# on crée un objet figure
figure1 = figure(
    title="fonctions trigonométriques",
    height=300, width=600,
    # c'est là notamment qu'on précise
    # l'intervalle en y
    y_range=(-5, 5),
)
```

[6]:
```
# on initialise la figure en créant
# un objet courbe
x = np.linspace(0, 2*np.pi, 2000)
y = np.sin(x)
courbe_trigo = figure1.line(x, y, color="#2222aa", line_width=3)
```

[7]:
```
# la fonction de mise à jour, qui sera connectée
# à interact
def update_trigo(function, frequency=1,
                 amplitude=1, phase=0,
                 # l'objet handle correspond
```

```
                # à une figure à mettre à jour
                *, handle):
    # c'est ici qu'on modifie les données
    # utilisées pour produire la courbe
    courbe_trigo.data_source.data['y'] = \
       amplitude * function(frequency * x + phase)
    # et c'est ici qu'on provoque la mise à jour
    push_notebook(handle=handle)
```

[8]:
```
# au moment où on matérialise l'objet figure
# on récupère une `handle` qui lui correspond
handle1 = show(figure1, notebook_handle=True)
```

[9]:
```
# maintenant on peut créer un interacteur
interact(update_trigo,
         # on peut définir les options sont des tuples (label, valeur)
         # et ici nos valeurs sont des fonctions
         function=Dropdown(options =(("sinus", np.sin),
                                     ("cosinus", np.cos),
                                     ("tangeante", np.tan))),
         frequency=(1,20),
         amplitude=[0.5, 1, 3, 5],
         phase=(0, 2*np.pi, 0.05),
         handle=fixed(handle1),
        );
```

```
interactive(children=(Dropdown(description='function', options=(('sinus', <ufunc␣
↪'sin'>), ('cosinus', <ufunc '…
```

7.32.2 Complément : niveau intermédiaire

Une classe pour ce genre d'usages

En termes de conception, notre approche jusqu'ici est améliorable.
En effet par construction, nous devons partager des données entre l'initialisation et la mise à jour - cf. les variables globales comme `handle1` - et c'est, comme toujours, une pratique qu'on cherche à éviter.

Voici une approche qui va réaliser exactement la même fonction, mais basée sur une classe ; on va tirer profit de l'instance pour ranger proprement toutes les données.

[10]:
```
# première version d'une classe d'animation

class Animation:

    # la fonction doit être vectorisée
    def display(self, function, title, *,
                y_range=(-5, 5), height=300, width=600):
        self.figure = figure(
            title=title, y_range=y_range,
            height=height, width=width)
        self.x = np.linspace(0, 2*np.pi, 200)
        y = function(self.x)
        self.courbe = self.figure.line(self.x, y, color="#2222aa", line_width=3)
        self.handle = show(self.figure, notebook_handle=True)

    # on passe directement la fonction en paramètre
    def update(self, function, frequency, amplitude, phase):
```

```
        new_y = amplitude * function(frequency * self.x + phase)
        self.courbe.data_source.data['y'] = new_y
        push_notebook(handle=self.handle)

    def interact(self):
        # interact nous impose de passer une simple fonction
        # pour passer 'self' à cette fonction on crée une cloture
        def closure(function, frequency, amplitude, phase):
            self.update(function, frequency, amplitude, phase)
        interact(closure,
                 function = Dropdown(
                     options=(('sinus', np.sin), ('cosinus', np.cos), ('tangeante',␣
↪np.tan))),
                 frequency=(1, 20),
                 amplitude=[0.5, 1, 3, 5],
                 phase=(0, 2*np.pi, 0.05),
                )
```

[11]:
```
a1 = Animation()
a1.display(np.sin, "fonctions trigonométriques")
```

[12]:
```
a1.interact()
```

```
interactive(children=(Dropdown(description='function', options=(('sinus', <ufunc␣
↪'sin'>), ('cosinus', <ufunc '…
```

Remarque

Je vous recommande cette pratique car, à nouveau, cela permet d'éviter les variables globales qui sont toujours une mauvaise idée ; tous les morceaux interdépendants sont regroupés, ainsi on limite la possibilité de casser le code en ne modifiant qu'un morceau ; la classe matérialise les interdépendances entre les objets `figure`, `handle` et `courbe` ; remarquez qu'en fait on n'a pas strictement besoin de `self.figure` comme attribut de l'instance.

Exemple : distribution uniforme

Voyons un deuxième exemple avec `bokeh`. Vous pouvez prendre ceci comme un exercice, et le faire de votre côté avant de lire la suite du notebook.

On veut ici écrire un outil qui déplace et déforme une distribution de points ; on part d'une distribution de N points calculée aléatoirement une bonne fois au début dans le cercle unité ; grâce aux réglages on pourra déformer ce nuage de points, qui va devenir une ellipse, grâce aux réglages suivants :

- `dx` et `dy`, les coordonnées du centre de l'ellipse,
- `rx` et `ry` les rayons en x et en y de l'ellipse,
- et enfin `alpha` l'angle de rotation de l'ellipse.

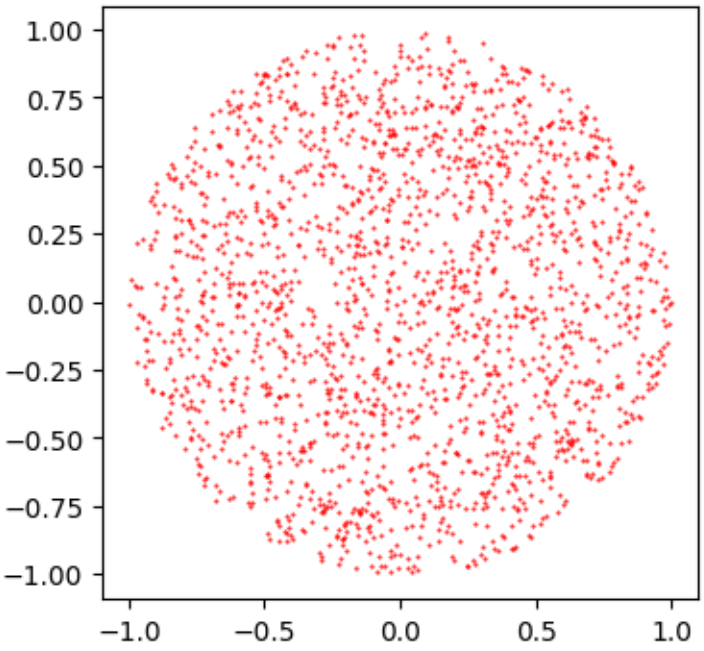
---

[13]:
```
# petit utilitaire pour calculer la distribution
# uniforme de départ
def uniform_distribution(N):
    # on tire au hasard un rho et un rayon
    rhos = 2 * np.pi * np.random.sample(N)
    rads = np.random.sample(N)
```

```
    # il faut prendre la racine carrée du rayon
    # sinon ce n'est pas uniforme dans le plan
    circle_x = np.sqrt(rads) * np.cos(rhos)
    circle_y = np.sqrt(rads) * np.sin(rhos)
    return circle_x, circle_y
```

[14]:
```
# regardons ça rapidement,  - avec matplotlib
# pour vérifier que la répartition est bien homogène
import matplotlib.pyplot as plt
```

[15]:
```
plt.figure(figsize=(4, 4))
X, Y = uniform_distribution(2000)
plt.scatter(X, Y, marker='.', s=1, color='red');
```



un peu de variété

[16]:
```
# et aussi: pour que ce soit plus joli
# et surtout plus facile à suivre visuellement
# je tire au hasard des couleurs
# et des tailles pour les points
def enhanced_uniform_distribution(N):
    # on calcule la distribution initiale
    # (celle-ci est vraiment uniforme)
    # dans le cercle de rayon 1
    x, y = uniform_distribution(N)

    # le rouge entre 50 et 250
    reds = 50 + 200 * np.random.random(size=N)
    # le vert entre 30 et 250
    greens = 30 + 220 * np.random.random(size=N)
    # la mise en forme des couleurs
    # le bleu est constant à 150
```

```
    colors = [
        f"#{int(red):02x}{int(green):02x}{150:02x}"
        for red, green in zip(reds, greens)
    ]

    # les rayons des points; entre 0.05 et 0.25
    radii = 0.05 + np.random.random(size=N) * .20

    return x, y, colors, radii
```

[17]:
```
# c'est ici qu'on commence à faire du bokeh

# j'applique la technique qu'on vient de voir
# en créant une classe
# pour éviter les variables globales

class AnimatedDistribution:

    def __init__(self, N):
        self.N = N


    def show(self):
        # les choix des bornes sont très arbitraires
        # dans une version plus élaborée tous ces détails pourraient
        # être passés en paramètre au constructeur
        self.figure = figure(
            title="distribution pseudo-uniforme",
            height=300, width=300,
            x_range=(-10, 10),
            y_range=(-10, 10),
        )

        # on range x0 et y0 dans des attributs de l'instance
        # pour pouvoir faire les mises à jour
        self.x0, self.y0, colors, radii = enhanced_uniform_distribution(self.N)

        # le paquets de cercles
        self.cloud = self.figure.circle(
            self.x0, self.y0,
            radius = radii,
            fill_color=colors, fill_alpha=0.6,
            line_color=None, line_width=.1,
        )

        # et enfin la poignée qui, à nouveau, sera nécessaire
        # pour les mises à jour
        self.handle = show(self.figure, notebook_handle=True)

    def update(self, rx, ry, dx, dy, alpha):
        # on recalcule les x et y
        # à partir des valeurs initiales
        s, c = np.sin(alpha), np.cos(alpha)
        x = dx + c * rx * self.x0 - s * ry * self.y0
        y = dy + s * rx * self.x0 + c * ry * self.y0
        self.cloud.data_source.data['x'] = x
        self.cloud.data_source.data['y'] = y
```

```
        push_notebook(handle=self.handle)

    def interact(self):
        def closure(rx, ry, dx, dy, alpha):
            self.update(rx, ry, dx, dy, alpha)
        interact(closure,
            rx=FloatSlider(min=.5, max=8,
                           step=.1, value=1.),
            ry=FloatSlider(min=.5, max=8,
                           step=.1, value=1.),
            dx=(-3, +3, .2),
            dy=(-3, +3, .2),
            alpha=FloatSlider(min=0., max=np.pi,
                              step=.05, value=0.))
```

[18]:
```
dist = AnimatedDistribution(1000)
dist.show()
```

[19]:
```
# pour déformer / déplacer
dist.interact()
```

```
interactive(children=(FloatSlider(value=1.0, description='rx', max=8.0, min=0.5),␣
↪FloatSlider(value=1.0, descr…
```

le point étant ici de montrer que toutes les modifications sont lisses, sans l'effet de flickering qu'on obtiendrait en redessinant toute l'image à chaque fois

---

Autres bibliothèques

Pour terminer cette digression sur les solutions alternatives à `matplotlib`, j'aimerais vous signaler enfin rapidement quelques autres options disponibles actuellement.

Comme on l'a dit en introduction, l'offre dans ce domaine est pléthorique, aussi si vous avez un témoignage à apporter sur une expérience que vous avez eue dans ce domaine, nous serons ravis de vous voir la partager dans le forum du cours.

plotly la bibliothèque `plotly`.

Cette bibliothèque est disponible en open source, et l'offre commerciale de plotly est tournée vers le conseil autour de cette technologie. Comme pour `bokeh`, elle est conçue comme un hybride entre Python et JavaScript, au dessus de `d3.js`. En réalité, elle présente même la particularité d'offrir une API unique disponible depuis Python, JavaScript, et R.

mpld3 https ://mpld3.github.io/

Je n'ai pas d'expérience à partager avec cette librairie, mais sur la papier l'approche semble prometteuse, puisqu'il s'agit (aussi) de conciler matplotlib avec `d3.js`.

k3d J'ai utilisé récemment la librairie k3d et j'ai trouvé le résultat assez bluffant pour les visualisations 3d. C'est un outil assez spartiate en termes de documentation, mais très performant.

Cette librairie se prête bien à la technique d'interactions que nous avons développée dans ce notebook. On en verra un autre exemple dans un prochain notebook.

7.32.3 Complément - niveau avancé (voire oiseux)

Simplement pour finir, j'aimerais revenir sur notre classe `Animation`.

On pourrait même considérer qu'une instance de notre classe `Animation` est une figure, et donc envisager de la faire hériter d'une classe `bokeh.figure` ; sauf qu'en fait `bokeh.figure` n'est pas une classe mais une fonction (une factory, c'est-à-dire une fonction qui contruit des instances) :

[20]:
```
# l'objet bokeh.figure est une factory, est pas une classe
# comme on le devine grâce aux minuscules
type(figure)
```

[20]:
```
bokeh.core.has_props.MetaHasProps
```

[21]:
```
# la classe c'est celle-ci:
type(figure())
```

[21]:
```
bokeh.plotting._figure.figure
```

[22]:
```
# qu'on pourrait importer comme ceci
# from bokeh.plotting import figure
# mais pour ne pas écraser notre variable figure
# nous allons plutôt faire
from bokeh.plotting import figure as Figure

type(figure()) is Figure
```

[22]:
```
True
```

Exercice (niveau avancé) :

vous semble-t-il possible de récrire la classe `Animation` comme une classe qui hérite cette fois de `Figure` ; quels seraient les bénéfices de cette approche ?

7.33 w7-s10-c6-fourier-k3d

Application à la transformée de Fourier

7.33.1 Complément - niveau avancé

On va appliquer ce qu'on a appris jusqu'ici, au cas de la transformée de Fourier.

Mon angle c'est d'essayer de vous faire intuiter à quoi correspond cette fameuse formule, dans le cas d'une fonction périodique en tous cas, et pourquoi dans ce cas-là on trouve un résultat non nul seulement sur les fréquences harmoniques de la fonction de base.

En guise de bonus, on va en profiter pour représenter aussi la fonction complexe en 3D, c'est surtout un prétexte pour faire au moins un exemple avec `k3d`, qui est très efficace, et qui à mon humble avis gagne à être connue.

Mais commençons par importer ce qui va nous servir.

[1]:
```
import numpy as np
# mostly we use bokeh in here, but the first glimpse is made with mpl
import matplotlib.pyplot as plt
%matplotlib notebook
```

[2]:
```
from bokeh.plotting import figure, show
from bokeh.io import push_notebook, output_notebook

output_notebook()
```

[3]:
```
# install with - unsurprisingly (from the terminal)
# pip install ipywidgets

from ipywidgets import interact, fixed
from ipywidgets import SelectionSlider, IntSlider
```

[4]:
```
# ditto w/
# pip install k3d

import k3d
from k3d.plot import Plot
```

7.33.2 Une fonction périodique

On considère donc une fonction périodique, comme celle-ci :

[5]:
```
# a vectorized function is required here

def my_periodic_2pi(t):
    '2sin(x) + sin(2x) - 3/2 sin(3x) + 2'
    return 2*np.sin(t) + np.sin(2*t) - 1.5*np.sin(3*t) + 2
```

Pour un aperçu, on la plotte rapidement avec matplotlib

[6]:
```
def plot_functions(domain, title, *functions):
    plt.figure(figsize=(4, 2))
    for function in functions:
        plt.plot(domain, function(domain))
    # notice how to retrieve the function's docstring
    plt.title(title)
    plt.show()

# period is 2 pi, let us plot between 0 and 15 with a .001 step
plot_functions(np.linspace(-1, 15, 200), "period = 2 ", my_periodic_2pi)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

changement d'échelle

Comme on le voit, la période est de 2 , évidemment ;
pour nous simplifier la vie nous allons changer l'échelle des x, pour travailler avec une période entière, ce sera plus facile pour faire les calculs mentalement ;
je choisis arbitrairement une période = 2 :

[7]:
```
# this one has a period of 2
def my_periodic(t):
    "addition of 3 sinus - period = 2"
    return my_periodic_2pi(t*np.pi)
```

[8]:
```
D1 = np.linspace(0, 6, 200)

plot_functions(D1, "now period=2", my_periodic)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

les morceaux

En ignorant la constante additive 2, on sait donc que notre fonction d'entrée est la superposition des 3 fonctions

[9]:
```
def H1(t):
    "fundamental"
    return 2*np.sin(t*np.pi)
def H2(t):
    "fundamental"
    return np.sin(2*t*np.pi)
def H3(t):
    "fundamental"
    return -1.5*np.sin(3*t*np.pi)

plot_functions(D1, "the 3 pieces", H1, H2, H3)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

La transformée de Fourier permet de retrouver ces 3 morceaux, donc par contruction de `my_periodic` on doit retrouver :

— la fondamentale (bleu) : period = 2, frequency = 1/2
— l'harmonique de 2nd ordre(orange) : period = 1, frequency = 1
— l'harmonique de 3eme ordre (vert) : period = 2/3, frequency = 3/2

les plages de fréquence

Du coup on va avoir envie de s'intéresser à plusieurs plages de fréquence :

[10]:
```
from ipywidgets import FloatSlider, Dropdown, Layout
# for building sliders
full_width = Layout(width='100%')

# la fréquence fondamentale
FUNDAMENTAL = 1/2

# un widget à large spectre, pour choisir une fréquence entre 1/4 et 3
def full_spectrum():
    return FloatSlider(min=0.25, max=3., step=0.01,
                       layout=full_width,
                       value=FUNDAMENTAL,
```

```
                        )

# quand on voudra faire un zoom autour d'une fréquence précise
def closeup_around(freq):
    return FloatSlider(min=freq * 0.98, max=freq * 1.02,
                       # 400 steps
                       step = freq/10_000,
                       layout=full_width,
                       value=FUNDAMENTAL,
                       readout_format='.4f',
                       )
```

7.33.3 La formule de fourier

Je rappelle la formule magique, la transformée de Fourier de f est la fonction F qui associe à une fréquence ϕ la valeur :

$F : \phi \rightarrow \int_{-\infty}^{\infty} f(t)e^{2i\pi\phi t}dt$

Pour un ϕ donné, il s'agit donc de calculer l'intégrale sur $\mathbb{R}$ de la fonction complexe

$F_\phi(t) = f(t)e^{2i\pi\phi t}$

On va commencer par représenter cette courbe en 3D : * sur l'axe des x, on représente le temps t * et sur les axes y et z, on représente les partie réelle et imaginaire de $F_\phi(t)$

7.33.4 représentation 3D

Voici une classe permettant de visualiser la courbe de F_ϕ en 3D.
En plus de la courbe, on matérialise par une ligne rouge l'emplacement du barycentre de la distribution complexe (plus de détails plus bas).

Une différence par rapport à ce qu'on avait pu voir avec `bokeh`, c'est qu'ici la librairie `k3d` expose une classe `Plot`, qui peut s'afficher directement dans le notebook ; du coup il semble raisonnable ici d'hériter de cette classe ; sinon l'idée générale est la même.

Vous avez sans doute déjà remarqué que chaque librairie de visualisation s'attend à recevoir les données d'entrée sous un format spécifique - ce qui a tendance à rendre l'utilisation de toutes ces techniques un peu fastidieuse parfois..

En tous cas notez que de manière opportuniste, la méthode centrale ici, à savoir `compute_dots_and_center()`, retourne ses données sous un format qui est propice pour `k3d` - qui aime les tableaux de `shape` $(n, 3)$, d'où l'appel à `np.stack()`.

[11]:
```
class FourierAnimator3D(Plot):

    DOTS_PER_UNIT = 50

    def __init__(self, function, phi,
                 domain=10, **kwds):
        self.function = function
        self.phi = phi
        self.domain = domain
        # pass along named parameters, like e.g. height
        super().__init__(**kwds)

        # returns the format expected by k3d line
        dots, center = self.compute_dots_and_center()
```

```
        # create line and add in plot
        self.line = k3d.line(dots)
        self += self.line
        # the line that materializes the barycenter
        self.center_line = k3d.line(center, color=0xff0000, width=0.5)
        self += self.center_line

    def update(self, phi):
        self.phi = phi
        new_dots, new_center = self.compute_dots_and_center()
        self.line.vertices = new_dots
        self.center_line.vertices = new_center

    def compute_dots_and_center(self):
        """
        returns an array of shape (nb_points, 3) suitable for k3d.line
        """
        nb_points = self.DOTS_PER_UNIT * self.domain
        t, dt = np.linspace(0, self.domain, nb_points, retstep=True)
        # a complex value
        rotating = self.function(t) * np.exp(2j * np.pi * self.phi * t)
        x = t
        y = np.real(rotating)
        z = np.imag(rotating)
        # the format expected by k3d line
        dots = np.stack([x, y, z], axis=1)
        # compute barycenter - as a complex average
        center_complex = np.sum(rotating) / nb_points
        # the format expected by k3d points
        # here 1 point at each end of the cylinder
        center = np.array([(0, center_complex.real, center_complex.imag),
                           (self.domain, center_complex.real, center_complex.imag)])
        return dots, center

    def interact(self, phi_widget):
        interact(lambda phi: self.update(phi), phi=phi_widget)
```

[12]:
```
a3d = FourierAnimator3D(my_periodic, phi=1.)
display(a3d)
a3d.interact(full_spectrum())
```

```
/Users/tparment/miniconda3/envs/flotpython-course/lib/python3.10/site-packa
  ges/traittypes/traittypes.py:97: UserWarning: Given trait value dtype "f
  loat64" does not match required type "float32". A coerced copy has been
  created.
 warnings.warn(
```

```
FourierAnimator3D(antialias=3, axes=['x', 'y', 'z'], axes_helper=1.0,␣
 ↪axes_helper_colors=[16711680, 65280, 255…
```

```
interactive(children=(FloatSlider(value=0.5, description='phi',␣
 ↪layout=Layout(width='100%'), max=3.0, min=0.25…
```

calculer et visualiser l'intégrale : le barycentre

Rappelez-vous que : * on commence par fixer ϕ ;
et ϕ correspond à la vitesse de rotation de la courbe de f autour de l'axe y=z=0

Et là vous vous dites, c'est bien joli mais comment je calcule l'intégrale de cette fonction complexe ?

En fait c'est assez simple à faire mentalement, et ça le truc crucial à comprendre, c'est que cette intégrale se déduit du barycentre de la courbe qu'on observe si on se met "au bout" de l'axe du temps et qu'on observe le signal tourner.

Intuitivement, pour évaluer l'intégrale d'une fonction usuelle, on peut estimer la moyenne de f entre les bornes, on n'a plus qu'à multiplier par la longueur du segment.

De la même façon, le barycentre de ce dessin - à nouveau quand on regarde le long de l'axe du temps - donne une bonne indication de la valeur de l'intégrale ; bien sûr, pour obtenir les coordonnées du barycentre il faut normaliser (diviser par la longueur du segment, comme pour la moyenne en dimension 1) ; aussi si on a N points dans notre échantillon :

$$\begin{aligned} \int_a^b F_\phi(t)dt &\approx \sum rotating[i] * dt \\ &\approx \sum \frac{rotating[i]*(b-a)}{N} \end{aligned}$$

et du coup le barycentre, qui est obtenu (souvenez-vous le cas de la dimension 1) en divisant cette valeur par la longueur du domaine $(b-a)$ s'obtient par notre unique ligne de code

```
center_complex = np.sum(rotating) / nb_points
```

On peut essayer de faire le calcul mentalement, mais c'est parfois délicat ; d'une part on ne voit pas la différence entre les points où f est positive ou négative ; d'autre part il faut noter que ça ne marche bien en fait que parce la vitesse de rotation est uniforme.
En tous cas le barycentre dans la visualisation donne, lui, une indication fiable de la valeur de l'intégrale.

Ce qu'on observe sur cette première visualisation, - on va le voir encore mieux en 2D - c'est que le barycentre est souvent nul, sauf au voisinage des fameuse fréquences de f - ouf, ça marche !

7.33.5 animation en 2D

Il ne nous reste plus qu'à représenter la même chose, mais cette fois en 2D en regardant le long de l'axe des x ; la logique est la même, sauf pour le format de retour de `compute_dots_and_center`, qui est adapté pour `bokeh` :

[13]:
```
DEFAULT_RANGE = (-6, 6)
DEFAULT_DOMAIN = 100

class FourierAnimator2D:

    DOTS_PER_UNIT = 50

    def __init__(self, function, phi, domain=DEFAULT_DOMAIN, **kwds):
        self.function = function
        self.phi = phi
        self.domain = domain

    def compute_dots_and_center(self):
        """
        returns X, Y for the curve in 2D
        and xc, yc the coordinates of the (bary)center
        """
        nb_points = self.DOTS_PER_UNIT * self.domain
        t, dt = np.linspace(0, self.domain, nb_points, retstep=True)
        # a complex value
        rotating = self.function(t) * np.exp(2j * np.pi * self.phi * t)
        X = np.real(rotating)
```

```
        Y = np.imag(rotating)
        # compute barycenter - as a complex average
        center_complex = np.sum(rotating) / nb_points
        return X, Y, center_complex.real, center_complex.imag

    def display(self, x_range=DEFAULT_RANGE, y_range=DEFAULT_RANGE):
        self.figure = figure(
            title=self.function.__name__,
            x_range=x_range, y_range=y_range)

        X, Y, xc, yc = self.compute_dots_and_center()

        self.courbe = self.figure.line(X, Y, color='blue', line_width = 1)
        self.center = self.figure.circle([xc], [yc], size=5, color="red")
        self.handle = show(self.figure, notebook_handle=True)

    def update(self, phi):
        self.phi = phi

        X, Y, xc, yc = self.compute_dots_and_center()
        self.courbe.data_source.data['x'] = X
        self.courbe.data_source.data['y'] = Y
        self.center.data_source.data['x'] = [xc]
        self.center.data_source.data['y'] = [yc]
        push_notebook(handle=self.handle)

    def interact(self, phi_widget):
        interact(lambda phi: self.update(phi), phi=phi_widget)
```

[14]:
```
a2d = FourierAnimator2D(my_periodic, FUNDAMENTAL)
a2d.display()
a2d.interact(full_spectrum())
```

```
interactive(children=(FloatSlider(value=0.5, description='phi',␣
 ↪layout=Layout(width='100%'), max=3.0, min=0.25…
```

On peut même zoomer autour des fréquences critiques :

[15]:
```
a2d.interact(closeup_around(FUNDAMENTAL))
```

```
interactive(children=(FloatSlider(value=0.5, description='phi',␣
 ↪layout=Layout(width='100%'), max=0.51, min=0.4…
```

Discussion

Vous observez la forte discontinuité de F qui vaut 0 presque partout ; vous pouvez comprendre que lorsque la fréquence ϕ n'est pas en résonance avec celle de f, le dessin qu'on obtient est presque parfaitement centré sur $(0, 0)$ et que donc le barycentre est nul.

En mode zoom autour de la fondamentale, on observe mieux la mise en résonance ; par contre cette visualisation peut donner l'illusion que F est continue, ce n'est pas le cas, c'est un artefact lié à la longueur finie de notre domaine.

On a choisi pour la 2D un domaine par défaut qui est [0..100], ce qui fait donc 50 périodes.
C'est pour cela qu'on a l'illusion qu'au voisinage d'une fréquence sensible, le barycentre s'écarte petit à petit ; en fait ce n'est pas le cas, c'est réellement une fonction discontinue, mais pour le voir il faut faire le calcul sur un domaine plus long ; lorsque vous choisissez par exemple $\phi = 0.501$, vous voyez seulement les 50 premiers pas de la figure qui commencent à diverger ; vous pouvez imaginer qu'en augmentant le domaine, on verra une décroissance plus rapide.

Par contre ce sont les vitesses de calcul qui vont commencer à nous limiter :

[16]:
```
# la discontinuité est plus forte qu'on ne pourrait le penser
# mais pour le vois il faut augmenter le domaine
# et donc les calculs sont plus lents
a2dzoom = FourierAnimator2D(my_periodic, FUNDAMENTAL, domain=500)
a2dzoom.display()
a2dzoom.interact(closeup_around(FUNDAMENTAL))
```

```
interactive(children=(FloatSlider(value=0.5, description='phi',␣
↪layout=Layout(width='100%'), max=0.51, min=0.4…
```

7.33.6 voir aussi

Une vidéo de 3BlueBrown, sur le même sujet ; bon ses animations sont autrement plus sophistiquées :-)

mais ici au moins on les a faites nous-mêmes ;)

https ://www.youtube.com/watch ?v=spUNpyF58BY

7.34 w7-s10-x1-taylor

Le théorème de Taylor illustré

7.34.1 exercice : niveau avancé

En guise d'application de ce qu'on a vu jusqu'ici, je vous invite à réaliser une visualisation du théorème de Taylor ; je vous renvoie à votre cours d'analyse, ou à wikipedia pour une présentation détaillée de ce théorème, mais ce que nous en retenons se résume à ceci.

On peut approximer une fonction "suffisamment régulière" - disons C^∞ pour fixer les idées - par un polynôme d'ordre n, dont les coefficients dépendent uniquement des n dérivées successives au voisinage d'un point :

$$f_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(0).x^i}{i!}$$

Sans perte de généralité nous avons ici fixé le point de référence comme étant égal à 0, il suffit de translater f par changement de variable pour se ramener à ce cas-là.

Le théorème de Taylor nous dit que la suite de fonctions (f_n) converge vers f.

On pourrait penser - c'était mon cas la première fois que j'ai entendu parler de ce théorème - que l'approximation est valable au voisinage de 0 seulement ; si on pense en particulier à sinus, on peut accepter l'idée que ça va nous donner une période autour de 0 peut-être.

En fait, c'est réellement bluffant de voir que ça marche vraiment incroyablement bien et loin.

7.34.2 mon implémentation

Je commence par vous montrer seulement le résultat de l'implémentation que j'ai faite.

Pour calculer les dérivées successives j'utilise la librairie `autograd`.

Ce code est relativement générique, vous pouvez visualiser l'approximation de Taylor avec une fonction que vous passez en paramètre - qui doit avoir tout de même la bonne propriété d'être vectorisée, et d'utiliser la couche `numpy` exposée par `autograd` :

[1]:
```
# to compute derivatives
import autograd
import autograd.numpy as np
```

Sinon pour les autres dépendances, j'ai utilisé les `ipywidgets` et `bokeh`

[2]:
```
from math import factorial

from ipywidgets import interact, IntSlider, Layout
```

[3]:
```
from bokeh.plotting import figure, show
from bokeh.io import push_notebook, output_notebook

output_notebook()
```

la classe **`Taylor`**

J'ai défini une classe `Taylor`, je ne vous montre pas encore le code, je vais vous inviter à en écrire une vous même ; nous allons voir tout de suite comment l'utiliser, mais pour la voir fonctionner il vous faut l'évaluer :

↓↓↓↓↓ ↓↓↓↓↓ assurez-vous de bien évaluer la cellule cachée ici ↓↓↓↓↓ ↓↓↓↓↓

[4]:
```
# @BEG@ name=taylor
class Taylor:
    """
    provides an animated view of Taylor approximation
    where one can change the degree interactively

    Taylor is applied on X=0, translate as needed
    """

    def __init__(self, function, domain):
        self.function = function
        self.domain = domain

    def display(self, y_range):
        """
        create initial drawing with degree=0

        Parameters:
          y_range: a (ymin, ymax) tuple
            for the animation to run smoothly, we need to display
            all Taylor degrees with a fixed y-axis range
        """
        # create figure
        x_range = (self.domain[0], self.domain[-1])
        self.figure = figure(title=self.function.__name__,
                             x_range=x_range, y_range=y_range)

        # each of the 2 curves is a bokeh line object
```

```
        self.figure.line(self.domain, self.function(self.domain), color='green')
        # store this in an attribute so _update can do its job
        self.line_approx = self.figure.line(
            self.domain, self._approximated(0), color='red', line_width=2)

        # needed so that push_notebook can do its job down the road
        self.handle = show(self.figure, notebook_handle=True)
# @END@

# @BEG@ name=taylor continued=true
    def _approximated(self, degree):
        """
        Computes and returns the Y array, the images of the domain
        through Taylor approximation

        Parameters:
          degree: the degree for Taylor approximation
        """
        # initialize with a constant f(0)
        # 0 * self.domain allows to create an array
        # with the right length
        result = 0 * self.domain + self.function(0.)
        # f'
        derivative = autograd.grad(self.function)
        for n in range(1, degree+1):
            # the term in f(n)(x)/n!
            result += derivative(0.)/factorial(n) * self.domain**n
            # next-order derivative
            derivative = autograd.grad(derivative)
        return result

    def _update(self, degree):
        # update the second curve only, of course
        # the 2 magic lines for bokeh updates
        self.line_approx.data_source.data['y'] = self._approximated(degree)
        push_notebook(handle=self.handle)

    def interact(self, degree_widget):
        """
        Parameters:
          degree_widget: a ipywidget, typically an IntSlider
            styled at your convenience
        """
        interact(lambda degree: self._update(degree), degree=degree_widget)
# @END@
```

↑↑↑↑↑ ↑↑↑↑↑ assurez-vous de bien évaluer la cellule cachée ici ↑↑↑↑↑ ↑↑↑↑↑

[5]:
```
# check the code was properly loaded
help(Taylor)
```

```
Help on class Taylor in module __main__:

class Taylor(builtins.object)
 |  Taylor(function, domain)
 |
 |  provides an animated view of Taylor approximation
```

```
|  where one can change the degree interactively
|
|  Taylor is applied on X=0, translate as needed
|
|  Methods defined here:
|
|  __init__(self, function, domain)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  display(self, y_range)
|      create initial drawing with degree=0
|
|      Parameters:
|        y_range: a (ymin, ymax) tuple
|          for the animation to run smoothly, we need to display
|          all Taylor degrees with a fixed y-axis range
|
|  interact(self, degree_widget)
|      Parameters:
|        degree_widget: a ipywidget, typically an IntSlider
|          styled at your convenience
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

sinus

Ma classe `Taylor` s'utilise comme ceci : d'abord on crée une instance à partir d'une fonction et d'un domaine, i.e. l'intervalle des X qui nous intéresse.

[6]:
```
# between -4  and 4
DOMAIN = np.linspace(-4*np.pi, 4*np.pi, 250)

# an instance
sinus_animator = Taylor(np.sin, DOMAIN)
```

Remarquez bien qu'ici la fonction que je passe au constructeur est en réalité **`autograd.numpy.sin`** et non pas `numpy.sin`, vu la façon dont on a défini notre symbole `np` lors des imports (et ça ne marcherait pas du tout avec `numpy.sin`).

Ensuite on crée un `ipywidget` qui va nous permettre de choisir le degré n ; dans le cas de sinus, qui est impaire, les degrés intéressants sont impairs (vous pouvez vérifier que les coefficients de Taylor pairs sont nuls lorsque f est impaire).

[7]:
```
# the widget to select a degree
sinus_widget = IntSlider(
   min=1, max=33, step=2,         # sinus being odd we skip even degrees
   layout=Layout(width='100%'))  # more convenient with the whole page width
```

Pour lancer l'interaction, on n'a plus qu'à :

— afficher le diagramme avec la méthode `display()` ; on a besoin pour cela de préciser les bornes en Y, qui resteront constantes au cours de l'animation (sinon la visualisation est vilaine)

puis lancer l'interaction en passant en paramètre le widget qui choisit le degré, ce qui donne :

[8]:
```
# fixed limits in Y
sinus_animator.display((-1.5, 1.5))

sinus_animator.interact(sinus_widget)
```

```
interactive(children=(IntSlider(value=1, description='degree',␣
↪layout=Layout(width='100%'), max=33, min=1, ste…
```

cosinus

La même chose avec cosinus nous donnerait ceci :

[9]:
```
# allows to select a degree
sinus_widget = IntSlider(
   min=0, max=34, step=2,      # only even degrees
   layout=Layout(width='100%'))

### ready to go
sinus_animator = Taylor(np.cos, DOMAIN)
sinus_animator.display((-1.5, 1.5))
sinus_animator.interact(sinus_widget)
```

```
interactive(children=(IntSlider(value=0, description='degree',␣
↪layout=Layout(width='100%'), max=34, step=2), O…
```

exponentielle

[10]:
```
# allows to select a degree
exp_widget = IntSlider(min=0, max=17,
   layout=Layout(width='100%'))

### ready to go
exp_animator = Taylor(np.exp, np.linspace(-5, 10, 200))
exp_animator.display((-15_000, 25000))
exp_animator.interact(exp_widget)
```

```
interactive(children=(IntSlider(value=0, description='degree',␣
↪layout=Layout(width='100%'), max=17), Output())…
```

7.34.3 quelques indices

affichage

Ici j'ai utilisé `bokeh`, mais on peut tout à fait arriver à quelque chose de similaire avec `matplotlib` sans aucun doute

conception

Ma classe `Taylor` s'inspire très exactement de la technique décrite dans le Complément #6 "Autres bibliothèques de visualisation", et notamment la classe `Animation`, modulo quelques renommages.

calcul de dérivées avec **autograd**

La seule fonction que j'ai utilisée de la bibliothèque `autograd` est `grad` :

[11]:
```
from autograd import grad
```

[12]:
```
# dans le cas de sinus par exemple
# les dérivées successives en 0 se retrouvent comme ceci
f = np.sin  # à nouveau cette fonction est autograd.numpy.sin
f(0.)
```

[12]:
```
0.0
```

[13]:
```
# ordre 1
f1 = grad(f)
f1(0.)
```

[13]:
```
1.0
```

[14]:
```
# ordre 2
f2 = grad(f1)
f2(0.)
```

[14]:
```
-0.0
```

7.34.4 votre implémentation

Je vous invite à écrire votre propre implémentation, qui se comporte en gros comme notre classe `Taylor`.

Vous pouvez naturellement simplifier autant que vous le souhaitez, ou modifier la signature comme vous le sentez (pensez alors à modifier aussi la cellule de test).

À titre indicatif ma classe `Taylor` fait environ 30 lignes de code utile, i.e. sans compter les lignes blanches, les docstrings et les commentaires.

[15]:
```
# à vous de jouer

class MyTaylor:
    def __init__(self, function, domain):
        ...
    def display(self, y_range):
        # là on veut créer le dessin original, c'est à dire
        # la figure, la courbe de f qui ne chagera plus,
        # et la courbe approchée avec disons n=0 (donc y=f(0))
        ...
    def _update(self, n):
        # modifier la courbe approximative avec Taylor à l'ordre n
        # je vous recommande de créer cette méthode privée
        # pour pouvoir l'appeler dans interact()
        ...
    def interact(self, widget):
        # là clairement il va y avoir un appel à
        # interact() de ipywidgets
        print("inactive for now")
```

[16]:
```
# testing MyTaylor on cosinus

sinus_widget = IntSlider(
   min=0, max=34, step=2,       # only even degrees
   layout=Layout(width='100%'))

### ready to go
sinus_animator = MyTaylor(np.cos, DOMAIN)
sinus_animator.display((-1.5, 1.5))
sinus_animator.interact(sinus_widget)
```

```
inactive for now
```

7.35 w7-s10-x2-coronavirus

Coronavirus

7.35.1 Exercice - niveau intermédiaire

Où on vous invite à visualiser des données liées au coronavirus.

Bon honnêtement à ce stade je devrais m'arrêter là, et vous laisser vous débrouiller complètement :)

Mais juste pour vous donner éventuellement des idées - voici des suggestions sur comment on peut s'y prendre.

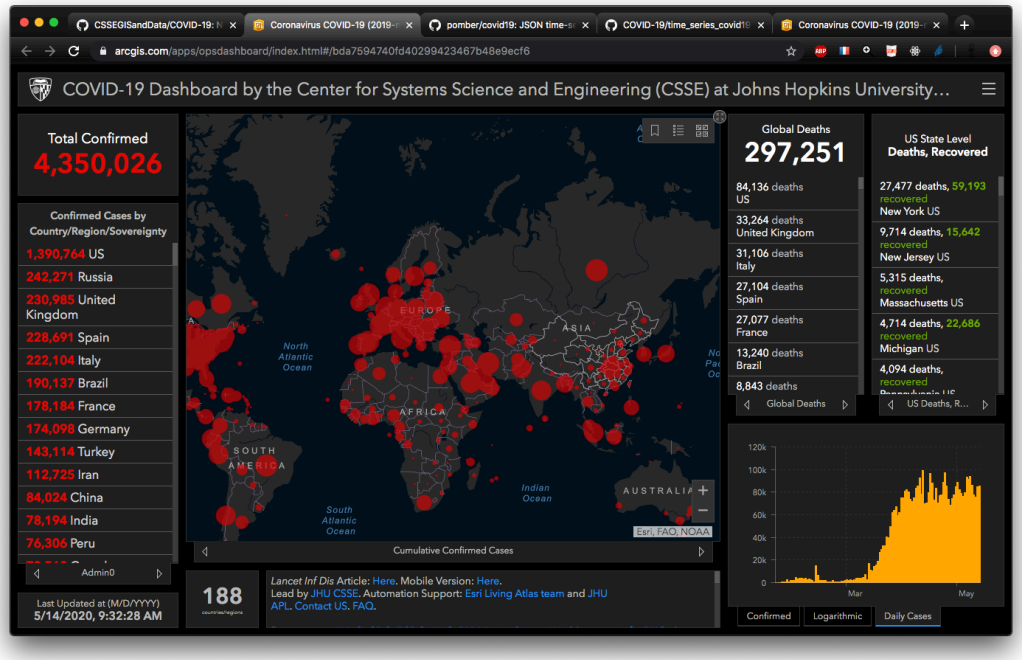
[1]:
```
import matplotlib.pyplot as plt
%matplotlib notebook
```

7.35.2 Le dashboard de Johns Hopkins

Le département Center for Systems Science and Engineering (CSSE), de l'Université Johns Hopkins, publie dans un dépôt github https://github.com/CSSEGISandData/COVID-19 les données dans un format assez brut. C'est très détaillé et touffu :

[2]:
```
# le repo github
official_url = "https://github.com/CSSEGISandData/COVID-19"
```

Le README mentionne aussi un dashboard, qui permet de visualiser les données en question. Il me semble que l'URL change tous les jours au fur et à mesure des updates, mais voici une capture d'écran pour donner une idée :



Ce qu'on vous propose de faire, pour s'amuser, c'est quelque chose d'anologue - en version beaucoup plus modeste naturellement - pour pouvoir visualiser facilement telle ou telle courbe.

7.35.3 Exercice 1 : un jeu de données intéressant

Pour ma part j'ai préféré utiliser un dépôt de seconde main, qui consolide en fait les données du CSSE, pour les exposer dans un seul fichier au jormat JSON. Cela est disponible dans ce second dépôt github https://github.com/pomber/covid19 ; la sortie de ce processus est mise à jour quotidiennement - à l'heure où j'écris ce texte en Mai 2020 - et est disponible (voir le README) à cette URL https://pomber.github.io/covid19/timeseries.json.

[3]:
```
abridged_url = "https://pomber.github.io/covid19/timeseries.json"
```

Comme c'est du JSON, on peut charger ces données en mémoire comme ceci

[4]:
```
# pour aller chercher l'URL
import requests

# pour charger le JSON en objets Python
import json
```

[5]:
```
# allons-y
req = requests.get(abridged_url)
# en utilisant la property `text` on décode en Unicode
encoded = req.text
# que l'on peut décoder
decoded = json.loads(encoded)
```

[6]:
```
## un peu de vérification
# si ceci n'est pas True, il y a un souci
```

```
# avec le réseau ou cette URL
req.ok
```

[6]: True

Les données sont indexées par pays

[7]:
```
# voici ce qu'on obtient
type(decoded)
```

[7]: dict

[8]:
```
# une clé
list(decoded.keys())[0]
```

[8]: 'Afghanistan'

Les données d'un pays sont dans un format très simple, une liste

[9]:
```
france_data = decoded['France']
type(france_data)
```

[9]: list

[10]:
```
france_data[0]
```

[10]: {'date': '2020-1-22', 'confirmed': 0, 'deaths': 0, 'recovered': 0}

[11]:
```
france_data[-1]
```

[11]: {'date': '2023-3-9', 'confirmed': 39866718, 'deaths': 166176, 'recovered':
0}

Homogénéité par pays

Ce que j'ai constaté, et je suppose qu'on peut plus ou moins compter sur cette bonne propriété, c'est que * pour chaque pays on trouve un enregistrement par jour * tous les pays ont la même plage de temps - quitte à rajouter des enregistrements à 0, comme ci-dessus pour la France le 22 janvier

[12]:
```
us_data = decoded['US']
us_data[0]
```

[12]: {'date': '2020-1-22', 'confirmed': 1, 'deaths': 0, 'recovered': 0}

[13]:
```
us_data[-1]
```

[13]: {'date': '2023-3-9', 'confirmed': 103802702, 'deaths': 1123836, 'recovered'
: 0}

[14]:
```
len(france_data) == len(us_data)
```

[14]:
```
True
```

[15]:
```
# nombre de pays
len(decoded)
```

[15]:
```
201
```

[16]:
```
# nombre de jours
len(france_data)
```

[16]:
```
1143
```

Un sujet possible (#1)

Vous pourriez interpréter ces données pour créer un dashbord dans lequel on peut choisir : * la donnée à laquelle on s'intéresse (confirmed, deaths ou recovered) * le pays auquel on s'intéresse (idéalement dans une liste triée) * la période (en version simple : les n derniers jours)

et en fonction, afficher deux courbes qui montrent sur cette période * la donnée brute (une fonction croissante donc) * sa dérivée (la différence avec le jour précédent)

Selon votre envie, toutes les variantes sont possibles, pour simplifier (commencez sans dashboard), ou complexifier, comme vous le sentez. Pour revenir sur le dashboard de CSSE, on pourrait penser à utiliser un package comme `folium` pour afficher les résultats sur une carte du Monde ; cela dit je vous recommande de bien réfléchir avant de vous lancer là-dedans, car c'est facile de se perdre, et en plus la valeur ajoutée n'est pas forcément majeure…

un mot par rapport à pandas

Il y a plein d'approches possibles, et toutes raisonnables :

- si vous êtes à l'aise avec `pandas`, vous allez avoir le réflexe de construire immédiatement une grosse dataframe avec toutes ces données, et utiliser la puissance de `pandas` pour faire tous les traitements de type tris, assemblages, moyennes, roulements, etc.. et les affichages (et si vous êtes dans ce cas, vous préférerez l'exercice #2) ;
- si au contraire vous êtes réfractaire à pandas, vous pouvez absolument tout faire sans aucune dataframe ;
- entre ces deux extrêmes, on peut facilement imaginer des hybrides, où on construit des dataframes de manière opportuniste selon les traitements.

La courbe d'apprentissage de pandas est parfois jugée un peu raide ; c'est à vous de voir ce qui vous convient le mieux. Ce qui est clair c'est que quand on maitrise bien, et une fois qu'on a construit une grosse dataframe avec toutes les données, on dispose avec d'un outil surpuissant pour faire plein de choses en très peu de lignes.

Mais pour bien maitriser il faut avoir l'occasion de pratiquer fréquemment, ce n'est pas forcément le cas de tout le monde (ce n'est pas le mien par exemple), donc à chacun de choisir son approche.

Pour illustrer une approche disons hybride, voici ce qui pourrait être un début de mise en forme des données pour un pays et une caractéristique (parmi les 3 exposées dans ce jeu de donnéees)

[17]:
```
import numpy as np
import pandas as pd

def extract_last_days(countryname, value, days):
    country = decoded[countryname]
```

```
    cropped = country[-(days):]
    dates = np.array([chunk['date'] for chunk in cropped])
    # take one more than requested for computing deltas including
    # for the first day (we need the value the day before the first day)
    cropped = country[-(days+1):]
    values = np.array([chunk[value] for chunk in cropped])
    # shift one day so we get the value from the day before
    shifted = np.roll(values, 1)
    # the daily increase; ignore first value which is wrong
    deltas = (values - shifted)[1:]
    relevant = values[1:]
    # all 3 arrays dates, deltas and relevant have the same shape
    data = {'dates': dates, value: relevant, 'daily': deltas}
    return pd.DataFrame(data=data)


df1 = extract_last_days('France', 'deaths', 45)
df1.plot();
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

7.35.4 Exercice 2 : idem mais à partir d'un autre jeu de données

Je vous signale une autre source de données, dans ce repo git https://github.com/owid/covid-19-data/tree/master/public/data ; les données cette fois-ci sont au format excel, et publiées à cette adresse

[18]:
```
alt_url = 'https://covid.ourworldindata.org/data/owid-covid-data.csv'
```

Dans ces cas-là il faut avoir le réflexe d'utiliser pandas ; voici un aperçu (ayez de la patience pour le chargement)

[19]:
```
import pandas as pd

df = pd.read_csv(alt_url)
```

[20]:
```
df.head(2)
```

[20]:
```
  iso_code continent     location        date  total_cases  new_cases  \
0      AFG      Asia  Afghanistan  2020-01-03          NaN        0.0
1      AFG      Asia  Afghanistan  2020-01-04          NaN        0.0

   new_cases_smoothed  total_deaths  new_deaths  new_deaths_smoothed  …
  \
0                 NaN           NaN         0.0                  NaN  …

1                 NaN           NaN         0.0                  NaN  …


   male_smokers  handwashing_facilities  hospital_beds_per_thousand  \
0           NaN                  37.746                         0.5
1           NaN                  37.746                         0.5

   life_expectancy  human_development_index  population  \
```

```
0            64.83                    0.511  41128772.0
1            64.83                    0.511  41128772.0

   excess_mortality_cumulative_absolute  excess_mortality_cumulative  \
0                                   NaN                          NaN
1                                   NaN                          NaN

   excess_mortality  excess_mortality_cumulative_per_million
0               NaN                                     NaN
1               NaN                                     NaN

[2 rows x 67 columns]
```

Un sujet possible (#2)

Le sujet à la base est le même bien entendu, essayer de visualiser ces données sous une forme où on y perçoit quelque chose :)

Les données sont bien entendu beaucoup plus riches, a contrario cela va demander davantage de mise en forme avant de pouvoir visualiser quoi que ce soit.

Je vous propose ce second point de vue si vous souhaitez vous entraîner avec `pandas`, puisqu'ici on a déjà une dataframe (ce qui ne veut pas dire qu'on ne peut pas traiter le premier exercice en utilisant `pandas`).

Explorons un peu

Voici quelques éléments sur la stucture de ces données :

[21]:
```
# beaucoup plus de détails
df.columns
```

[21]:
```
Index(['iso_code', 'continent', 'location', 'date', 'total_cases', 'new_cas
  es',
       'new_cases_smoothed', 'total_deaths', 'new_deaths',
       'new_deaths_smoothed', 'total_cases_per_million',
       'new_cases_per_million', 'new_cases_smoothed_per_million',
       'total_deaths_per_million', 'new_deaths_per_million',
       'new_deaths_smoothed_per_million', 'reproduction_rate', 'icu_patient
  s',
       'icu_patients_per_million', 'hosp_patients',
       'hosp_patients_per_million', 'weekly_icu_admissions',
       'weekly_icu_admissions_per_million', 'weekly_hosp_admissions',
       'weekly_hosp_admissions_per_million', 'total_tests', 'new_tests',
       'total_tests_per_thousand', 'new_tests_per_thousand',
       'new_tests_smoothed', 'new_tests_smoothed_per_thousand',
       'positive_rate', 'tests_per_case', 'tests_units', 'total_vaccination
  s',
       'people_vaccinated', 'people_fully_vaccinated', 'total_boosters',
       'new_vaccinations', 'new_vaccinations_smoothed',
       'total_vaccinations_per_hundred', 'people_vaccinated_per_hundred',
       'people_fully_vaccinated_per_hundred', 'total_boosters_per_hundred',
       'new_vaccinations_smoothed_per_million',
       'new_people_vaccinated_smoothed',
       'new_people_vaccinated_smoothed_per_hundred', 'stringency_index',
       'population_density', 'median_age', 'aged_65_older', 'aged_70_older'
  ',
```

```
       'gdp_per_capita', 'extreme_poverty', 'cardiovasc_death_rate',
       'diabetes_prevalence', 'female_smokers', 'male_smokers',
       'handwashing_facilities', 'hospital_beds_per_thousand',
       'life_expectancy', 'human_development_index', 'population',
       'excess_mortality_cumulative_absolute', 'excess_mortality_cumulative
  ',
       'excess_mortality', 'excess_mortality_cumulative_per_million'],
      dtype='object')
```

[22]:
```
# la colonne iso_code représente le pays :
df.iso_code.unique()[:5]
```

[22]:
```
array(['AFG', 'OWID_AFR', 'ALB', 'DZA', 'ASM'], dtype=object)
```

[23]:
```
# rien que sur la france, on a ce nombre d'enregistrements
df_france = df[df.iso_code == 'FRA']
len(df_france)
```

[23]:
```
1280
```

[24]:
```
# manifestement c'est un par jour
df_france.head(2)
```

[24]:
```
      iso_code continent location        date  total_cases  new_cases  \
97242      FRA    Europe   France  2020-01-03          NaN        0.0
97243      FRA    Europe   France  2020-01-04          NaN        0.0

       new_cases_smoothed  total_deaths  new_deaths  new_deaths_smoothed  .
  ..  \
97242                 NaN           NaN         0.0                  NaN  .
  ..
97243                 NaN           NaN         0.0                  NaN  .
  ..

       male_smokers  handwashing_facilities  hospital_beds_per_thousand  \
97242          35.6                     NaN                        5.98
97243          35.6                     NaN                        5.98

       life_expectancy  human_development_index  population  \
97242            82.66                   0.901  67813000.0
97243            82.66                   0.901  67813000.0

       excess_mortality_cumulative_absolute  excess_mortality_cumulative  \
97242                                   NaN                          NaN
97243                                   NaN                          NaN

       excess_mortality  excess_mortality_cumulative_per_million
97242               NaN                                     NaN
97243               NaN                                     NaN

[2 rows x 67 columns]
```

[25]:
```
df_france.tail(2)
```

[25]:
```
      iso_code continent location        date  total_cases  new_cases  \
98520      FRA    Europe   France  2023-07-04   38989382.0        0.0
98521      FRA    Europe   France  2023-07-05   38989382.0        0.0

       new_cases_smoothed  total_deaths  new_deaths  new_deaths_smoothed  .
   ..  \
98520                 0.0      167923.0         0.0                  0.0  .
   ..
98521                 0.0      167923.0         0.0                  0.0  .
   ..

       male_smokers  handwashing_facilities  hospital_beds_per_thousand  \
98520          35.6                     NaN                        5.98
98521          35.6                     NaN                        5.98

       life_expectancy  human_development_index  population  \
98520            82.66                   0.901  67813000.0
98521            82.66                   0.901  67813000.0

       excess_mortality_cumulative_absolute  excess_mortality_cumulative  \
98520                                   NaN                          NaN
98521                                   NaN                          NaN

       excess_mortality  excess_mortality_cumulative_per_million
98520               NaN                                     NaN
98521               NaN                                     NaN

[2 rows x 67 columns]
```

Pour afficher, disons les décès par jour en France depuis le début de la pandémie, on pourrait faire :

[26]:
```
df_france.plot(x='date', y='new_deaths');
```

```
<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>
```

[27]:
```
# n'hésitez pas à installer des packages
# supplémentaires au besoin
!pip install plotly-express
```

```
Collecting plotly-express
  Using cached plotly_express-0.4.1-py2.py3-none-any.whl (2.9 kB)
Collecting patsy>=0.5
  Using cached patsy-0.5.3-py2.py3-none-any.whl (233 kB)
Collecting scipy>=0.18
  Using cached scipy-1.11.1-cp310-cp310-macosx_10_9_x86_64.whl (37.2 MB)
Requirement already satisfied: numpy>=1.11 in /Users/tparment/miniconda3/en
   vs/flotpython-course/lib/python3.10/site-packages (from plotly-express)
   (1.23.5)
Requirement already satisfied: plotly>=4.1.0 in /Users/tparment/miniconda3/
   envs/flotpython-course/lib/python3.10/site-packages (from plotly-express
   ) (5.11.0)
Requirement already satisfied: pandas>=0.20.0 in /Users/tparment/miniconda3
   /envs/flotpython-course/lib/python3.10/site-packages (from plotly-expres
   s) (1.5.2)
Collecting statsmodels>=0.9.0
```

```
  Using cached statsmodels-0.14.0-cp310-cp310-macosx_10_9_x86_64.whl (9.9 M
    B)
Requirement already satisfied: pytz>=2020.1 in /Users/tparment/miniconda3/e
    nvs/flotpython-course/lib/python3.10/site-packages (from pandas>=0.20.0-
    >plotly-express) (2022.6)
Requirement already satisfied: python-dateutil>=2.8.1 in /Users/tparment/mi
    niconda3/envs/flotpython-course/lib/python3.10/site-packages (from panda
    s>=0.20.0->plotly-express) (2.8.2)
Requirement already satisfied: six in /Users/tparment/miniconda3/envs/flotp
    ython-course/lib/python3.10/site-packages (from patsy>=0.5->plotly-expre
    ss) (1.16.0)
Requirement already satisfied: tenacity>=6.2.0 in /Users/tparment/miniconda
    3/envs/flotpython-course/lib/python3.10/site-packages (from plotly>=4.1.
    0->plotly-express) (8.1.0)
Requirement already satisfied: packaging>=21.3 in /Users/tparment/miniconda
    3/envs/flotpython-course/lib/python3.10/site-packages (from statsmodels>
    =0.9.0->plotly-express) (22.0)
Installing collected packages: scipy, patsy, statsmodels, plotly-express
Successfully installed patsy-0.5.3 plotly-express-0.4.1 scipy-1.11.1 statsm
    odels-0.14.0
```

[28]:
```
# plusieurs courbes en une
# avec plotly express, pour changer un peu
import plotly.express as px

selection = ['USA', 'FRA']

start = '2020-03-15'
date_start = start
#date_start = pd.to_datetime(start)
sel = df[(df.iso_code.isin(selection)) & (df.date > date_start)]
fig1 = px.line(sel, x="date", y="total_deaths_per_million", color="location")
fig1.update_layout(height= 800, title_text="Décès Covid, cumulés (par million␣
 ↪d'habitants)")
fig1.show()
```

7.35.5 Comment partager ?

Je ne publie pas de corrigés pour cet exercice.

J'invite ceux d'entre vous qui le souhaitent à nous faire passer leur code ; le plus simple étant de les ajouter dans le repo github dit de récréation, à cet endroit https ://github.com/flotpython/recreation/tree/master/corona-dashboards.