Cours POO M1: python

19 Septembre 2023

Aide python

On peut obtenir de l'aide sur une fonction, une classe ou une constante grâce au symbole "?" (ou de *help()*). Exemples

```
import numpy as np
? np.array
? np.sin
? np.pi
help(np.pi)
```

Cellules de code (spyder)

Astuce 1 Dans l'éditeur de script de Spyder, il est possible de structurer le script en **cellules**, pouvant être compilées séparément.
- Une cellule démarre par #%% (éventuellement suivi d'un commentaire) et un retour à la ligne.
- Elle se termine au prochain #%% (nouvelle cellule).

Pour **compiler uniquement une cellule** :

1. cliquer quelque part dans la cellule
2. cliquer sur l'icone "run current cell" ou "run current cell and go to the next one" (raccourci Ctrl+return ou Shift+Return sur Windows)

Astuce 2 : On peut aussi ne compiler que la (les) ligne(s) sélectionnée(s) avec l'icone "run selection or current line" (raccourcis F9 sur Windows).

numpy (suite)

Vu jusqu'à présent

- *numpy.array* : tableaux 1D (vecteurs) et 2D (matrices)

Vu jusqu'à présent

- *numpy.array*: tableaux 1D (vecteurs) et 2D (matrices)
- *numpy.arange* et *numpy.linspace*: suites de valeurs numériques (discrétisation d'intervalles)

Vu jusqu'à présent

- *numpy.array*: tableaux 1D (vecteurs) et 2D (matrices)
- *numpy.arange* et *numpy.linspace*: suites de valeurs numériques (discrétisation d'intervalles)
- *numpy.size*, *numpy.shape*, *numpy.sum*: nb d'éléments, dimensions, somme

Vu jusqu'à présent

- *numpy.array*: tableaux 1D (vecteurs) et 2D (matrices)
- *numpy.arange* et *numpy.linspace*: suites de valeurs numériques (discrétisation d'intervalles)
- *numpy.size*, *numpy.shape*, *numpy.sum*: nb d'éléments, dimensions, somme
- *numpy.zeros*, *numpy.ones*, *numpy.eye*: initialisation

Vu jusqu'à présent

- *numpy.array*: tableaux 1D (vecteurs) et 2D (matrices)
- *numpy.arange* et *numpy.linspace*: suites de valeurs numériques (discrétisation d'intervalles)
- *numpy.size*, *numpy.shape*, *numpy.sum*: nb d'éléments, dimensions, somme
- *numpy.zeros*, *numpy.ones*, *numpy.eye*: initialisation
- *numpy.append*, *numpy.sort*, *numpy.argsort*,*numpy.unique*: ajouts d'éléments, tris

Vu jusqu'à présent

- *numpy.array*: tableaux 1D (vecteurs) et 2D (matrices)
- *numpy.arange* et *numpy.linspace*: suites de valeurs numériques (discrétisation d'intervalles)
- *numpy.size*, *numpy.shape*, *numpy.sum*: nb d'éléments, dimensions, somme
- *numpy.zeros*, *numpy.ones*, *numpy.eye*: initialisation
- *numpy.append*, *numpy.sort*, *numpy.argsort*,*numpy.unique*: ajouts d'éléments, tris
- Fonctions mathématiques de *numpy*

Vu jusqu'à présent

- *numpy.array*: tableaux 1D (vecteurs) et 2D (matrices)
- *numpy.arange* et *numpy.linspace*: suites de valeurs numériques (discrétisation d'intervalles)
- *numpy.size*, *numpy.shape*, *numpy.sum*: nb d'éléments, dimensions, somme
- *numpy.zeros*, *numpy.ones*, *numpy.eye*: initialisation
- *numpy.append*, *numpy.sort*, *numpy.argsort*,*numpy.unique*: ajouts d'éléments, tris
- Fonctions mathématiques de *numpy*

On va voir d'autres possibilités de *numpy*.

numpy.reshape

- *numpy.reshape* permet de changer les dimensions d'un tableau numpy existant, de manière cohérente

```
import numpy as np
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
b = np.reshape(a, [2, 4]); print(b)
```

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

```
c = np.reshape(a, [4, 2]); print(c)
```

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

Rappel : la transposée est obtenue par `a.T` ou `np.transpose(a)`

numpy.reshape

Un tableau 1D n'est ni *horizontal* ni *vertical*. Il a une seule dimension.

```
a = np.array([1, 2, 3]); print(a.shape)
>>> (3,)
```

On peut le convertir en vecteur colonne ou ligne

```
a=np.reshape(a,[3,1]); print(a.shape) # colonne
a=np.reshape(a,[1,3]); print(a.shape) # ligne
>>> (3,1)
>>> (1,3)
```

numpy.where

Trouver les valeurs et indices pour lesquels une condition est satisfaite.

- ```
  a = np.array([5, 7, 10, -2, 8])
  print(a>6)
  print(a[a>6]) # seulement les indices pour
     lesquels a>6 est vraie
  ```

  ```
  [False  True  True False  True]
  [7 10 8]
  ```

- *numpy.where* donne les indices du tableau numpy qui vérifient la condition en argument

  ```
  print(np.where(a>6)) # indices de a tq a>6
  ```

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

numpy.where

- *numpy.where* renvoie en sortie un tuple. On peut récupérer les indices dans un array

```
a2=np.array(np.where(a>6))[0]
print(a2)
```

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

- *numpy.where* peut prendre plusieurs arguments

```
b = np.array([12, 15, 13, 35, 5])
print(np.where(a>6, b, -1000)) # remplacement
    par -1000 dans b là où a>6 est fausse.
```

```
[-1000, 15, 13, -1000,  5]
```

numpy.all & numpy.any

- *numpy.all* renvoie un booléen:
  - *True* si toutes les valeurs du tableau en argument sont *True*,
  - False sinon
- *numpy.any* renvoie *True* si au moins une valeur est *True*

```
a = np.array([1, 5, 3])
c = np.array([3, 2, 5])
print(np.all(a<c))
print(np.any(a<c))
```

```
False
True
```

Indices dans les matrices

Pour manipuler des matrices (tableaux numpy 2D) avec des indices, le principe est le même que pour des tableaux 1D ou des listes.
Soit M un tableau numpy 2D

- $M[i] \rightarrow$ ligne i de M
- $M[i,:] \rightarrow$ idem (ligne i de M)
- $M[i_1{:}i_2,:] \rightarrow$ lignes d'indices i_1 à i_2-1 de M
- $M[i_1{:}i_2{:}2,:] \rightarrow$ lignes d'indices i_1 à i_2-1 avec pas de 2 de M

Indices dans les matrices

Pour manipuler des matrices (tableaux numpy 2D) avec des indices, le principe est le même que pour des tableaux 1D ou des listes.
Soit M un tableau numpy 2D

- $M[i] \rightarrow$ ligne i de M
- $M[i,:] \rightarrow$ idem (ligne i de M)
- $M[i_1{:}i_2,:] \rightarrow$ lignes d'indices i_1 à i_2-1 de M
- $M[i_1{:}i_2{:}2,:] \rightarrow$ lignes d'indices i_1 à i_2-1 avec pas de 2 de M
- $M[:,j] \rightarrow$ colonne j de M
- $M[:,j_1{:}j_2] \rightarrow$ colonnes d'indices j_1 à j_2-1 de M

Indices dans les matrices

Pour manipuler des matrices (tableaux numpy 2D) avec des indices, le principe est le même que pour des tableaux 1D ou des listes.
Soit M un tableau numpy 2D

- $M[i] \rightarrow$ ligne i de M
- $M[i,:] \rightarrow$ idem (ligne i de M)
- $M[i_1{:}i_2,:] \rightarrow$ lignes d'indices i_1 à i_2-1 de M
- $M[i_1{:}i_2{:}2,:] \rightarrow$ lignes d'indices i_1 à i_2-1 avec pas de 2 de M
- $M[:,j] \rightarrow$ colonne j de M
- $M[:,j_1{:}j_2] \rightarrow$ colonnes d'indices j_1 à j_2-1 de M
- $M[i_1{:}i_2,j_1{:}j_2]$ bloc de la matrice M allant des indices de ligne i_1 à i_2-1 et des indices de colonne j_1 à j_2-1

Tableaux multidimensionnels

- Exemple d'un objet à 3 dimensions

```
P = np.array([[[1,2],[3,4]],[[5,6],[7,8]]]) # dim 2,2,2
print(P[0,:,:])
```

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

```
print(P[:,:,1])
```

-
```
array([[2, 4],
       [6, 8]])
```

Initialisation d'un tableau numpy multidimensionnel

- numpy.zeros($(n_1, n_2, .., n_k)$) $\rightarrow$ création d'un tableau numpy de dimension $(n_1, n_2, .., n_k)$ ne contenant que des 0
- numpy.ones($(n_1, n_2, .., n_k)$) $\rightarrow$ création d'un tableau numpy de dimension $(n_1, n_2, .., n_k)$ ne contenant que des 1
- numpy.eye(n,m) $\rightarrow$ création d'un tableau numpy de dimension (n,m) contenant 1 sur la diagonale, 0 ailleurs

produits entre tableaux

Soient $M1$ et $M2$ deux tableaux numpy. Lorsque ces produits sont bien définis,

- `numpy.dot(M1,M2)` (ou l'opérateur `M1@M2` si version récente de python) renvoie le produit matriciel
- `M1*M2` renvoie le produit terme à terme.
- `numpy.cross(M1,M2)` renvoie le produit vectoriel dans $\mathbb{R}^3$ $M1 \wedge M2$.
- Si v est un vecteur ligne (resp. colonne) ou un tableau 1D, `M1*v` renvoie le produit terme à terme pour chaque ligne (resp. colonne) de M1.

Fonctions statistiques de numpy

Soit a un tableau *numpy* 1D à n éléments ($n = np.size(a)$)

- *numpy.mean* (moyenne): $np.mean(a) = \frac{1}{n}\sum_{i=0}^{n-1} a_i = \overline{a}$
- *numpy.var* (variance): $np.var(a) = \frac{1}{n}\sum_{i=0}^{n-1}(a_i - \overline{a})^2$
- *numpy.std* (écart-type): $np.std(a) = \sqrt{np.var(a)}$
- *numpy.median* (médiane)

```
import numpy as np
a = np.array([12, 14, 35, 22])
[np.mean(a), np.var(a), np.std(a), np.median(a)]
```

```
[20.75  81.688  9.038 18. ]
```

Nombres aléatoires avec numpy

- librairie *numpy.random*
- contient les principales lois de probabilité: normale, uniforme, Poisson, exponentielle, ...
- *numpy.random.rand* correspond à la loi uniforme $\mathcal{U}(0,1)$
- On verra plus tard *scipy.stats*, préférable à *numpy.random*!

```
import numpy as np
v = np.random.rand() #une seule valeur

t = np.random.rand(2,3) # tableau de dim (2,3)

np.random.seed(1) # initialise le générateur de
    nombres aléatoires une "graine" spécifique
```

Entrées/sorties avec numpy

- *numpy.loadtxt*: pour lire un tableau numpy depuis un fichier existant au format ascii (fichier texte)
- *numpy.savetxt*: pour écrire un tableau numpy dans un fichier ascii

```
import numpy as np
a = np.array([7, 8, 9, 10])
np.savetxt('a.txt', a) # ecriture de a dans le
   fichier a.txt
b = np.loadtxt('a.txt') #lecture de b
b == a
```

Pour ouvrir un fichier, il doit se trouver dans le répertoire courant :

- Avec jupyter, par défaut, le dossier contenant le notebook
- Avec spyder, visible et modifiable en haut à droite de l'interface ("Browse a working directory").

numpy.linalg

numpy.linalg contient beaucoup de fonctions faisant les opérations élémentaires sur des matrices

- numpy.linalg.det → calcul du déterminant
- numpy.linalg.lstsq → résolution par moindres carrés
- numpy.linalg.chol → factorisation de Cholesky
- numpy.linalg.qr → factorisation QR
- numpy.linalg.eig → calcul des valeurs et vecteurs propres
- numpy.linalg.eigvals → calcul des valeurs propres uniquement
- numpy.linalg.solve → résolution de systèmes linéaires
- numpy.linalg.pinv → calcul du pseudo-inverse
- numpy.linalg.inv → calcul de l'inverse

Toutes se retrouvent dans *scipy* qu'on verra prochainement!

numpy.linalg.solve

- Par exemple

```
M1 = np.array([[1,2,3], [4,5,6], [7,8,10]])
v1 = np.array([-1,-2, -6])
x =np.linalg.solve(M1,v1)
print(x); print(M1@x)
```

- Résultat

```
[-2.66666667  5.33333333 -3.        ]
[-1. -2. -6.]
```

matplotlib

matplotlib - Présentation

- Ensemble de fonctions permettant la représentation graphique en 2D (mplot3d existe pour le 3D)
- Affichages à la Matlab
- Traditionnellement, on importe matplotlib.pyplot comme suit:

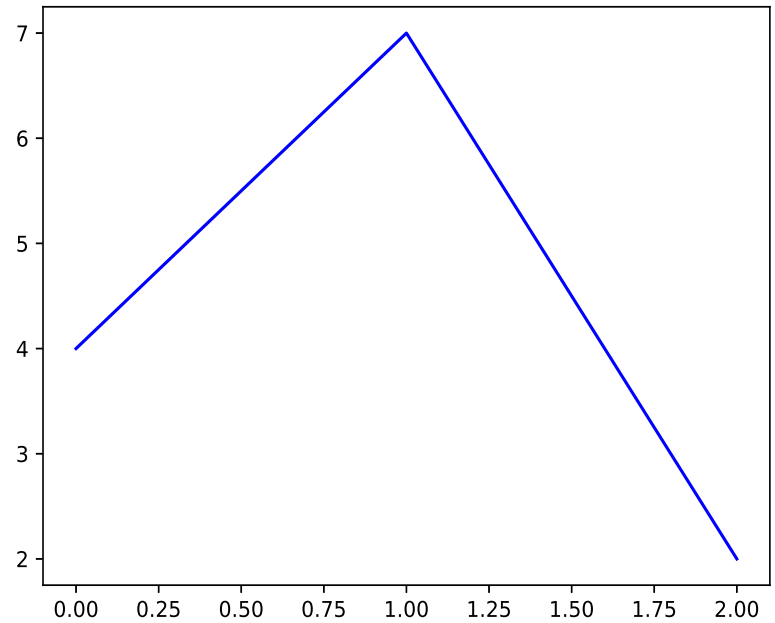
```
import matplotlib.pyplot as plt
```

- Sur Spyder, par défaut, les figures apparaissent dans l'onglet "Plots". Pour générer la figure dans une nouvelle fenêtre : Tools→Preferences→Ipython Console→Graphics→Backend : automatic.

matplotlib - Début

- Affichage de points reliés entre eux

```
import matplotlib.pyplot
    as plt
plt.plot([4,7,2])
```

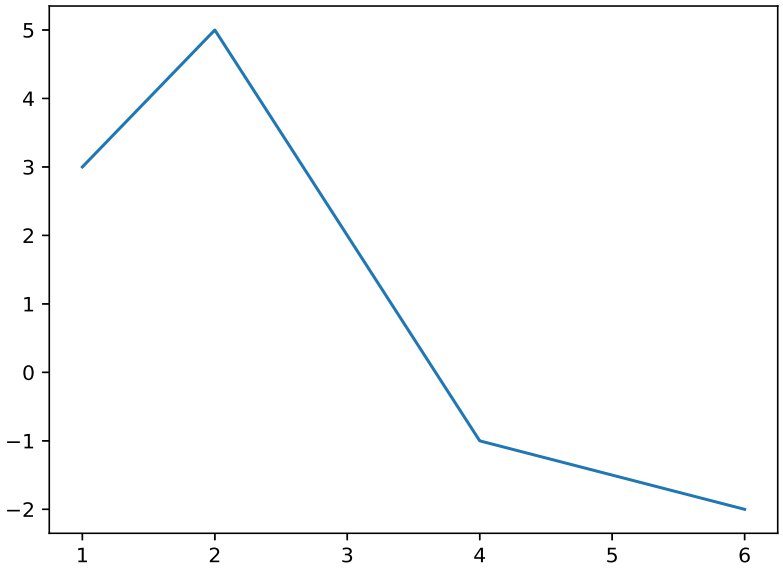


- plt.show() pour afficher (**inutile** dans un environnement interactif comme Jupyter, Spyder). Dans ce cas, l'affichage se fait en mode bloquant (l'utilisateur doit fermer la fenêtre pour reprendre la main, dans l'interpréteur python) sauf si on utilise `plt.show(block=False)`.

matplotlib - Début 2

- Sur la figure suivante, les points $(1, 3)$, $(2, 5)$, $(4, -1)$ et $(6, -2)$ sont reliés entre eux
- *plt.plot(x, y)* relie les points de coordonnées (x_i, y_i) entre eux

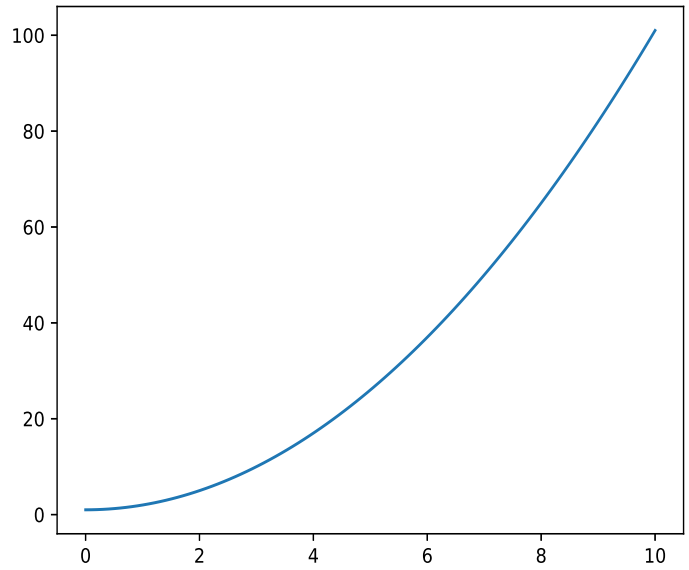
```
plt.plot([1,2,4,6], [3,5,-1,-2])
```



Plot de base: $y = f(x)$

On veut représenter le vecteur y en fonction de x, tel que $y = f(x)$, où f est une fonction mathématique.

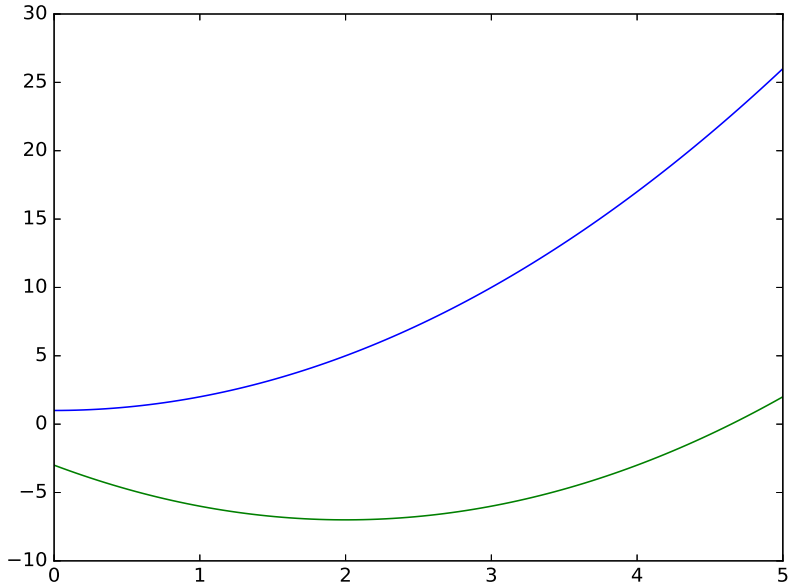
```
f = lambda x: x**2+1
x = np.linspace(0,10,100)
y = f(x)
plt.plot(x,y)
```



Superposition de deux plots

Supposons deux fonctions f et g à représenter.
La superposition est automatique!

```
f= lambda x: x**2 + 1
g= lambda x: x**2 - 4*x - 3
x = np.linspace(0, 5, 100)
plt.plot(x, f(x))
plt.plot(x, g(x))
```



matplotlib

- Il existe les commandes suivantes:

```
plt.figure()   # nouvelle figure vide
plt.figure(1)  # sélectionner la figure 1
plt.pause(2)   # figure en pause pendant 2s
plt.close()    # fermer la figure courante
plt.close('all') # fermer toutes les figures
plt.clf()      # effacer le contenu de la
   figure courante
```

- Attention, la numérotation des figures commence à 1!

Marqueurs

Par défaut : données reliées en trait plein. On peut aussi n'afficher que les données.

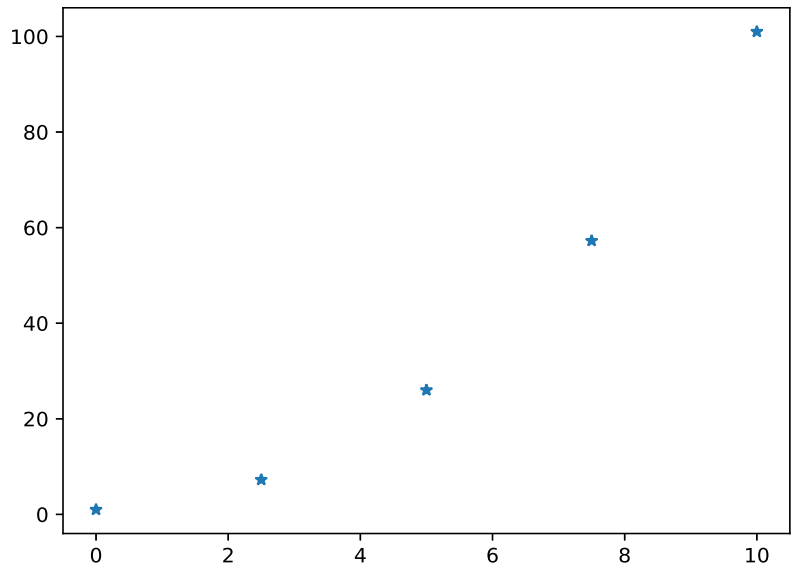
- Différents marqueurs peuvent être utilisés pour l'affichage.

'o'	'.'	'd'	'*'	'v'	's'
rond	point	diamant fin	étoile	triangle	carré

- Exemple de représentation avec des étoiles

```
f= lambda x: x**2 + 1
x = np.linspace(0, 10, 5)
plt.plot(x, f(x),
         ls=' ',marker='*')
```

Le `ls=" "` permet de ne pas relier les points entre eux.

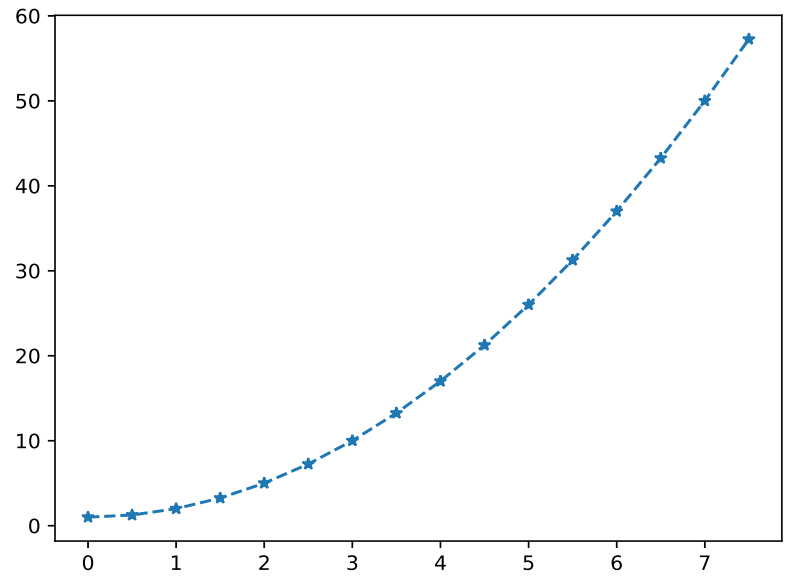


Styles de plot - *linestyle*

L'option *linestyle* (en abrégé *ls*) de *plot* permet de définir le style de trait reliant les données.

'-'	'- -'	'-.'	':'	' '
trait plein	tiret	tiret et point	pointillé	pas de trait

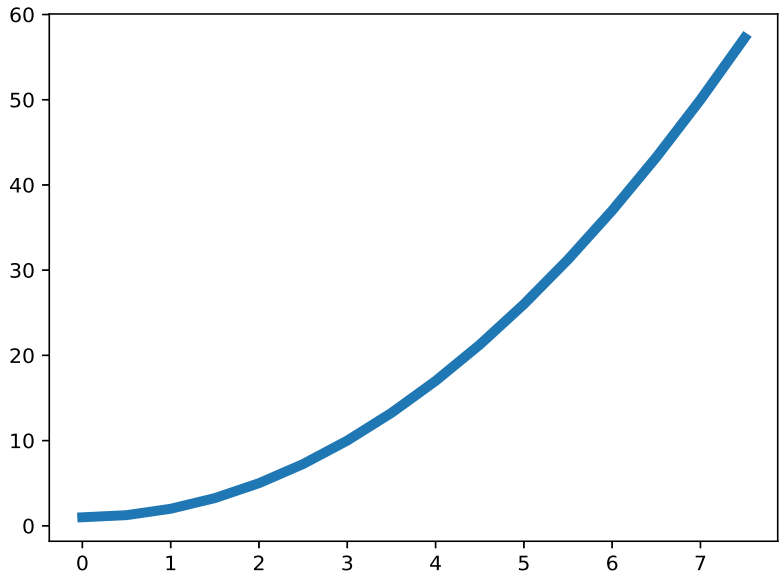
```
f= lambda x: x**2 + 1
x = np.arange(0, 8, 0.5)
y = f(x)
plt.plot(x, y,
         marker='*',ls='--')
```



Styles de plot - *linewidth*

L'option *linewidth* (en abrégé *lw*) de *plot* permet de définir l'épaisseur de la ligne de trait.

```
import matplotlib.pyplot as plt
import numpy as np
f= lambda x: x**2 + 1
x = np.arange(0, 8, 0.5)
y = f(x)
plt.plot(x, y, lw=5)
plt.savefig('f4.eps')
plt.show()
```

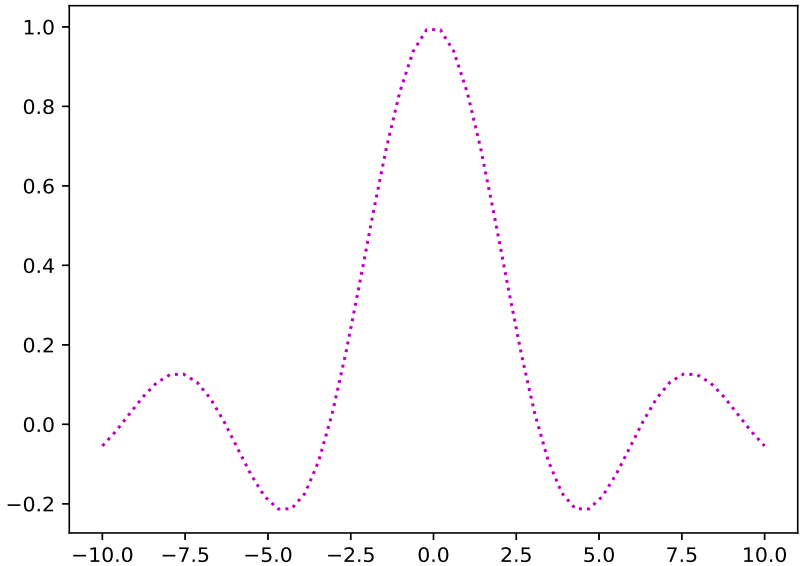


Couleurs d'un plot - matplotlib

- Couleurs de base dans matplotlib

'b'	'g'	'r'	'c'	'm'	'y'	'k'	'w'
bleu	vert	rouge	cyan	magenta	jaune	noir	blanc

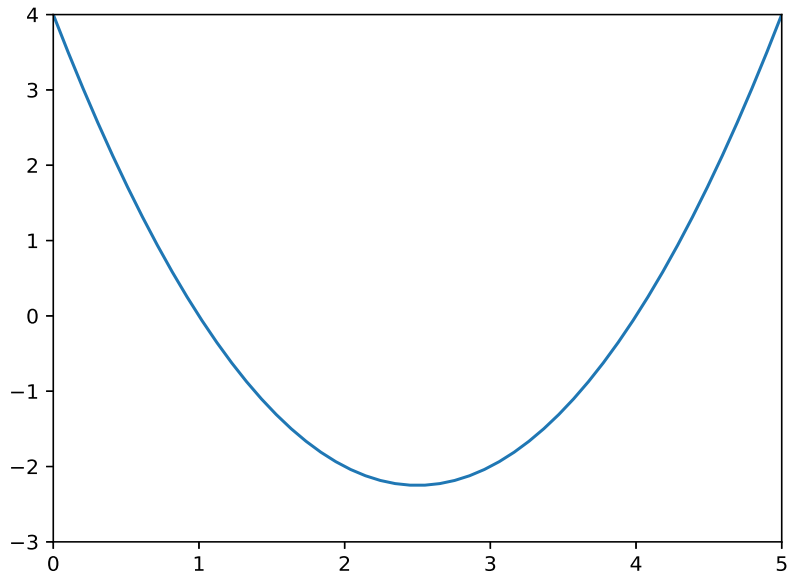
```
x = np.linspace(-10, 10)
y = np.sin(x)/x
plt.plot(x, y, color='m', ls=':')
```



Axes d'un plot

- Utiliser *axis* pour définir les abscisses et ordonnées min et max d'un plot
- *xlim* pour définir uniquement les abscisses min et max, *ylim* pour les ordonnées min et max
- Par exemple, on veut $x_{min} = 0$, $x_{max} = 5$, $y_{min} = -3$, $y_{max} = 4$

```
f= lambda x: x**2 - 5*x + 4
x = np.linspace(0, 5)
y = f(x)
plt.plot(x,y)
plt.axis([0, 5, -3, 4])
```

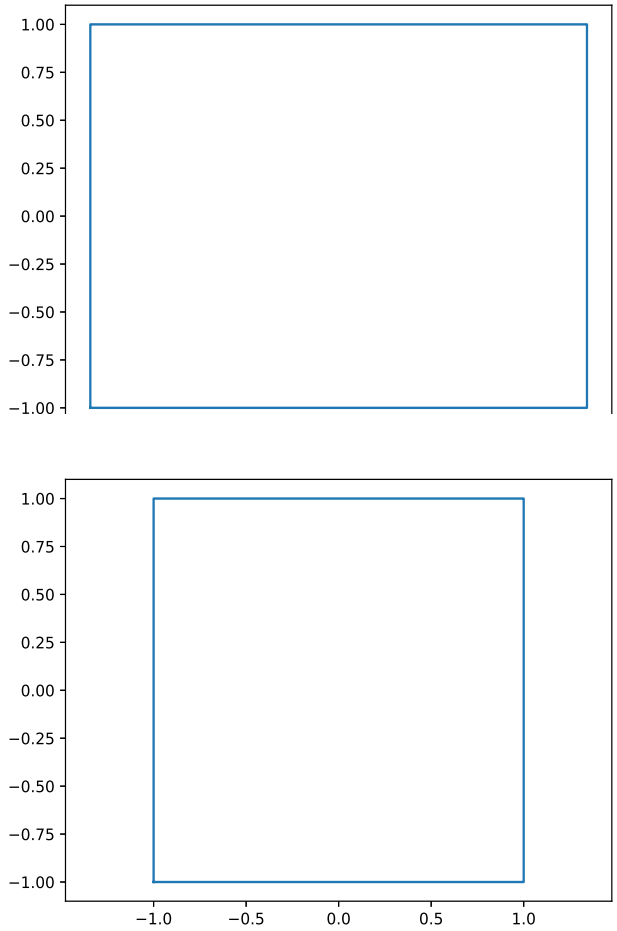


Axes d'un plot - 'axis equal'

```
plt.axis('equal') #  Même échelle pour tous les
    axes
```

```
plt.plot([-1,1,1,-1,-1],
         [-1,-1,1,1,-1])

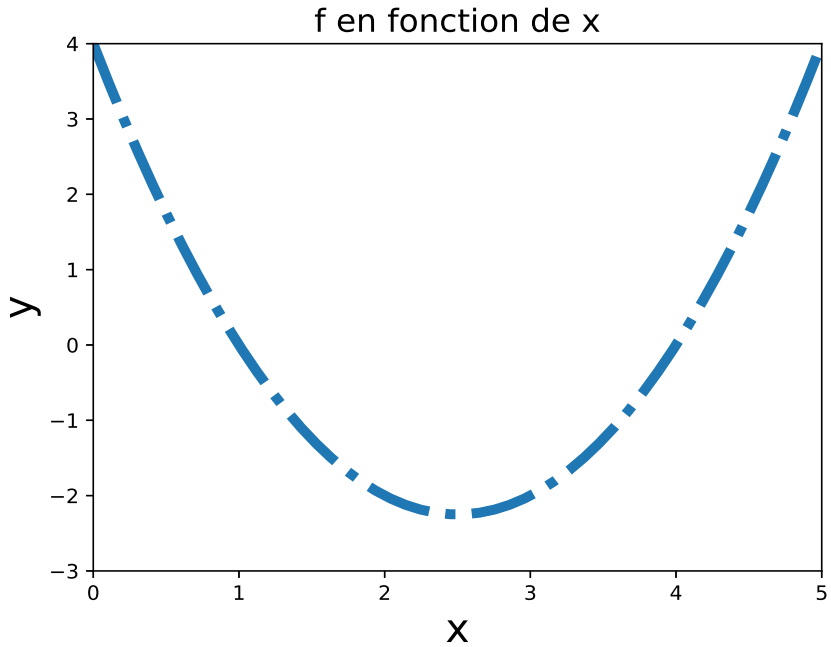
plt.figure()
plt.plot([-1,1,1,-1,-1],
         [-1,-1,1,1,-1])
plt.axis('equal')
```



labels et titre d'une figure

Utilisation de *xlabel*, *ylabel* et *title*

```
f= lambda x: x**2 - 5*x + 4
x = np.linspace(0, 5)
y = f(x)
plt.plot(x, y, ls='-.', lw=5)
plt.axis([0, 5, -3, 4])
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.title('f en fonction de x',
          fontsize=16)
```

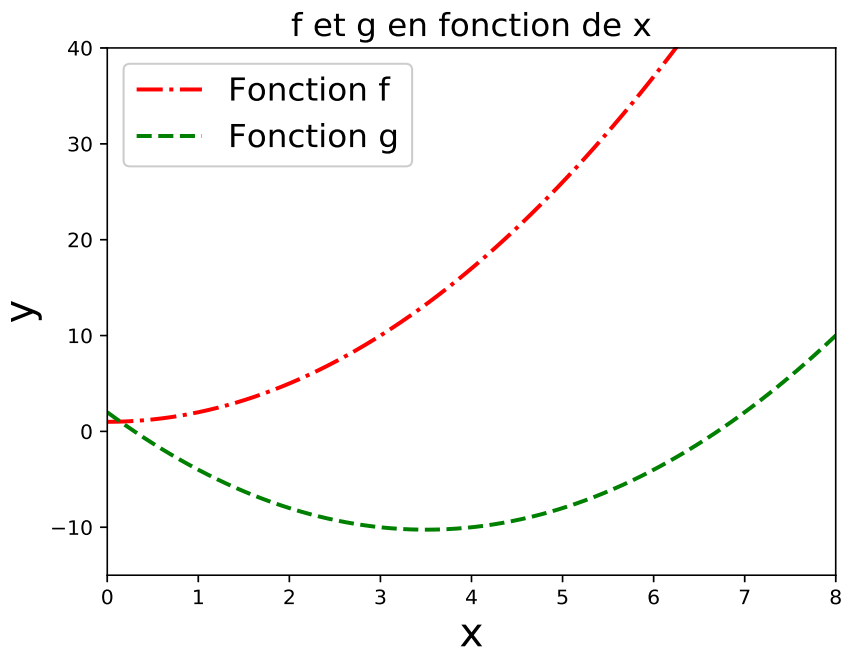


Légendes - matplotlib

Utiliser *plt.legend*.

Version 1 : nom des courbes dans le `plt.legend()`

```
f = lambda x: x**2 + 1
x1 = np.linspace(0, 10, 100)
y1 = f(x1)
g = lambda x: x**2 -7*x + 2
y2 = g(x1)
plt.plot(x1, y1, ls='-.', lw=2,
         color='r')
plt.plot(x1, y2, ls='--', lw=2,
         color='g')
plt.axis([0, 8, -15, 40])
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.title('f et g en fonction de x',
          fontsize=16)
plt.legend(['Fonction f','Fonction g'],
           fontsize=16)
```

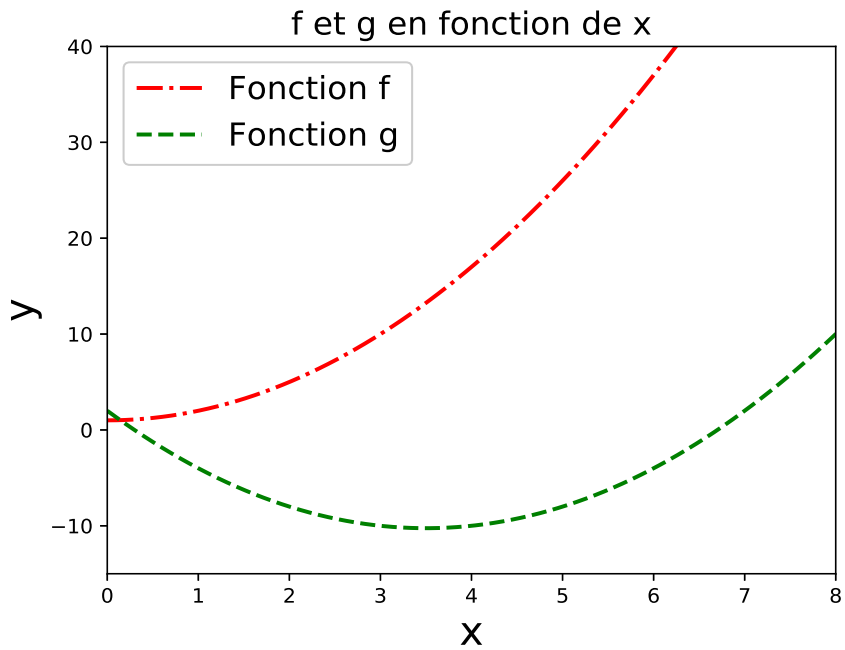


Légendes - matplotlib

Utiliser *plt.legend*

Version 2 : utiliser l'option label pour chaque plot.

```
f= lambda x: x**2 + 1
x1 = np.linspace(0, 10, 100)
y1 = f(x1)
g= lambda x: x**2 -7*x + 2
y2 = g(x1)
plt.plot(x1, y1, ls='-.', lw=2,
         color='r', label='Fonction f')
plt.plot(x1, y2, ls='--', lw=2,
         color='g', label='Fonction g')
plt.axis([0, 8, -15, 40])
plt.xlabel('x', fontsize=20)
plt.ylabel('y', fontsize=20)
plt.title('f et g en fonction de x',
           fontsize=16)
plt.legend(fontsize=16)
```

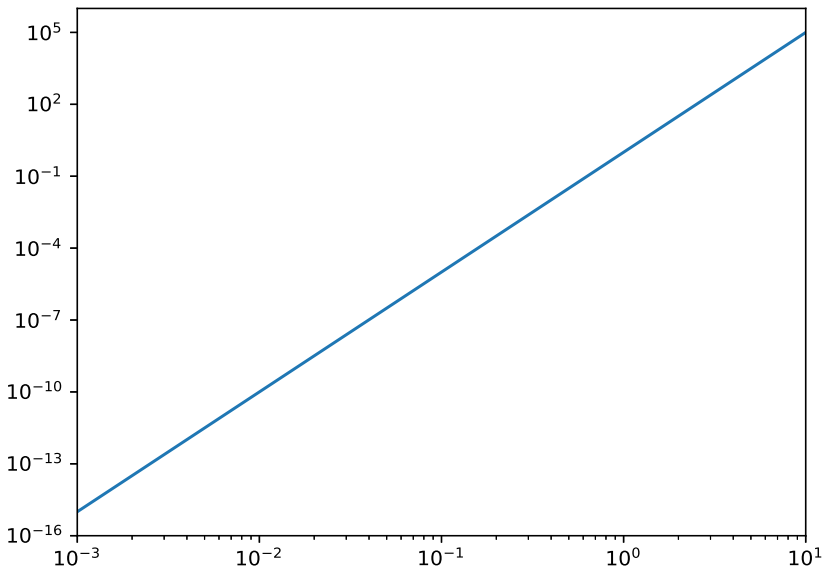


Reprendre ici

Echelles log

- *loglog*, *semilogx*, *semilogy*

```
f= lambda x: x**5
x = np.linspace(0.001, 10, 100)
y = f(x)
plt.loglog(x,y)
plt.xlim((0.001,10))
```



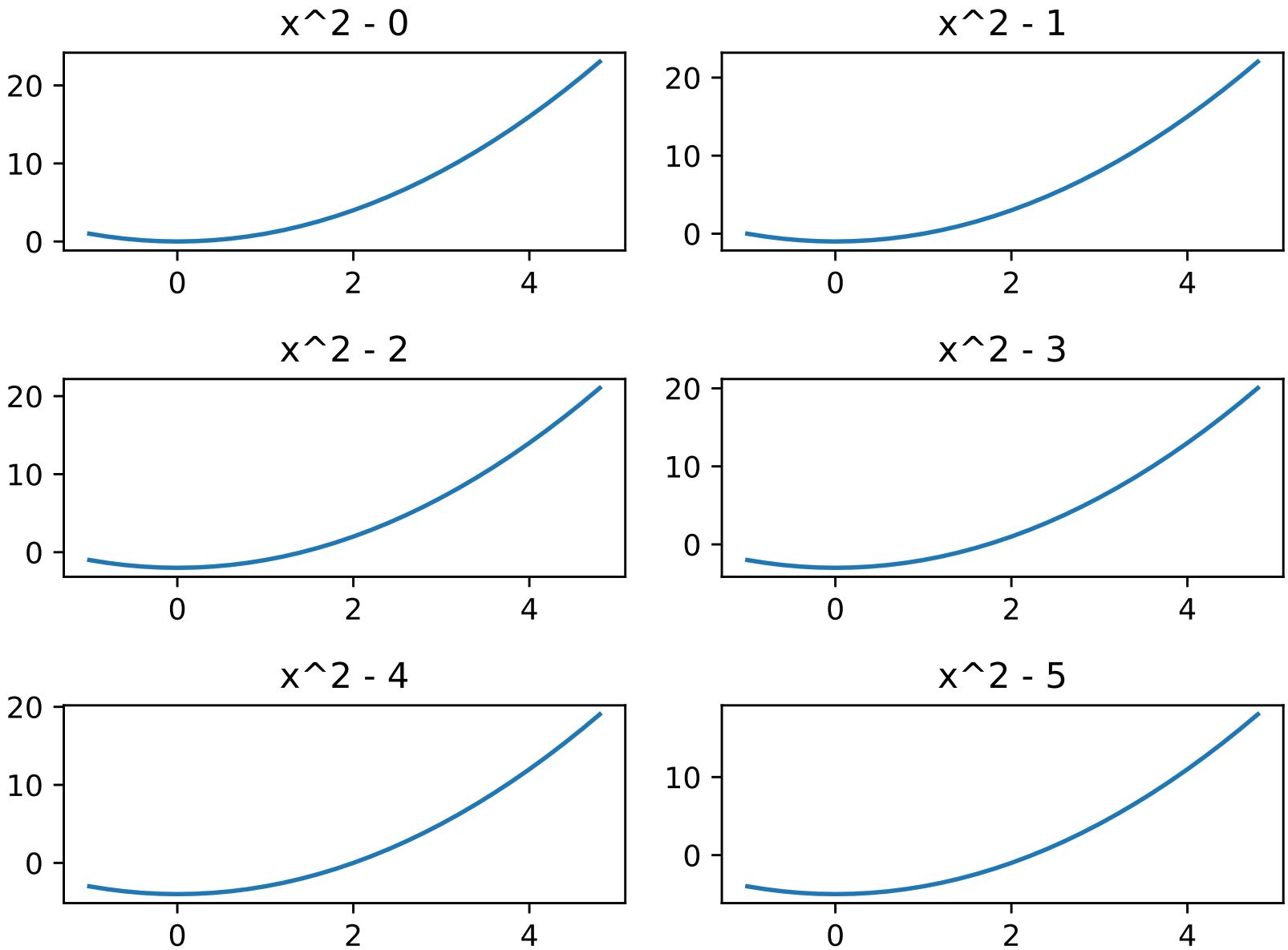
subplot

- *subplot* permet de diviser la fenêtre graphique en plusieurs figures, que l'on parcours colonne par colonne, de la 1ère à la dernière ligne.
- *plt.subplot*(rows, cols, num)

```
def f(k):
    return lambda x: x**2 - k
tx = np.arange(-1, 5, 0.2)
for i in range(1,7):
    plt.subplot(3, 2, i) #3 lignes, 2 colonnes, emplacement i
    plt.plot(tx, f(i)(tx)) # f(i) est une fonction
    plt.title('x^2 - {0}'.format(i))
plt.tight_layout() # optimise la position  des titres/labels
```

- `plt.tight_layout()` presque indispensable pour éviter les chevauchements.

subplot



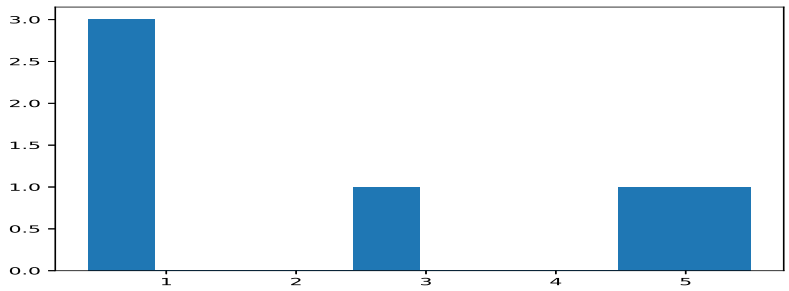
Histogrammes

- *plt.hist* permet de représenter des histogrammes
- Par défaut, *plt.hist(x)* affiche l'histogramme du tableau x avec 10 intervalles (appelés *bins*) de même longueur

$$h = \frac{max(x) - min(x)}{10}$$

.

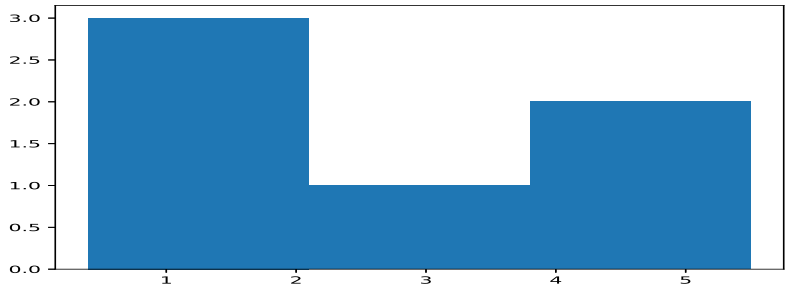
```
x = [0.5, 0.6, 2.5, 0.4, 4.5, 5.5]
plt.hist(x)
```



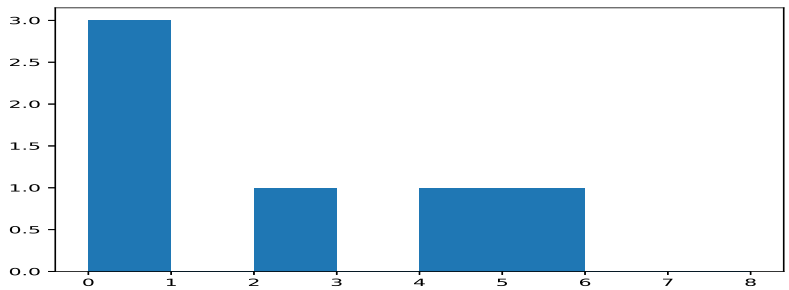
Histogrammes

- On peut utiliser *plt.hist* avec deux arguments, *plt.hist(x, y)*
- Si y est un entier, l'intervalle $[min(x), max(x)]$ est découpé en y intervalles (bins) de taille identique.
- Si y est une liste numérique ou un tableau numpy 1D, les bins utilisés sont ceux définis par y.

```
plt.figure()
nbins = 3
plt.hist(x, nbins)
```



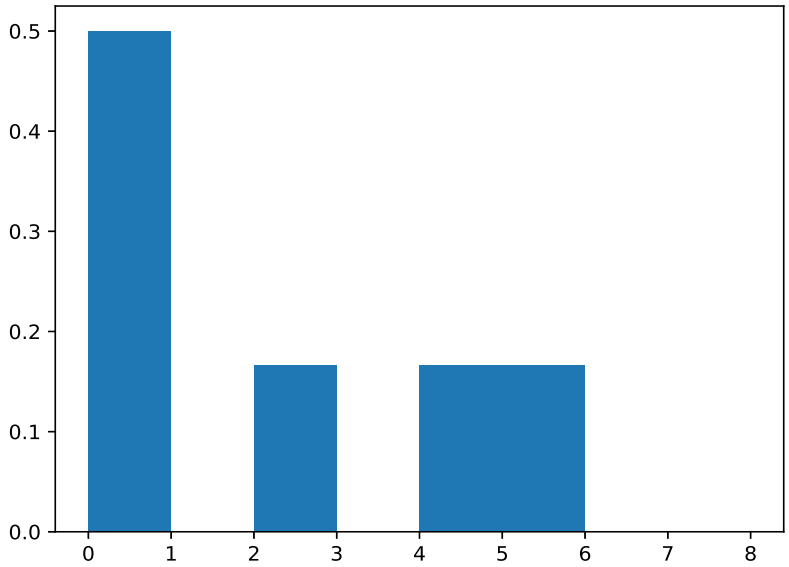

```
plt.figure()
y = np.arange(0, 9)
plt.hist(x, y)
```



Histogrammes

- On peut aussi utiliser *density=True* dans *plt.hist*.
  L'aire de l'histogramme est alors égale à 1 (densité empirique de la distribution).

```
x = [0.5, 0.6, 2.5, 0.4, 4.5, 5.5]
y = np.arange(0, 9)
plt.hist(x, y, density=True)
```

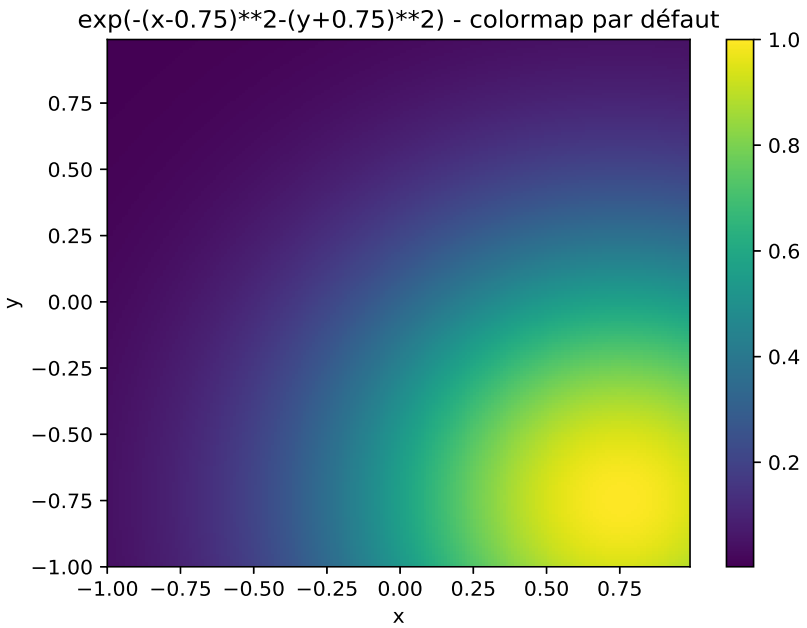


pcolor (ou pcolormesh)

pcolor ou (pcolormesh) permet des représentations en 2D
(on veut représenter $f(x, y)$ pour $f : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$)

```
n = 200 # nb de cellules dans 1 direction
M = np.zeros((n,n))
x = np.linspace(-1,1,n)
y = np.linspace(-1,1,n)
for i in np.arange(n):
    for j in np.arange(n):
        M[i,j] = np.exp(-(x[i]-0.75)**2-
                 (y[j]+0.75)**2)

# transposee!
plt.pcolormesh(x, y, M.T)
plt.colorbar() # echelle graduee
plt.xlabel('x'); plt.ylabel('y')
plt.title('exp(-(x-0.75)**2-'+'(y+0.75)
                **2) - colormap par defaut')
```

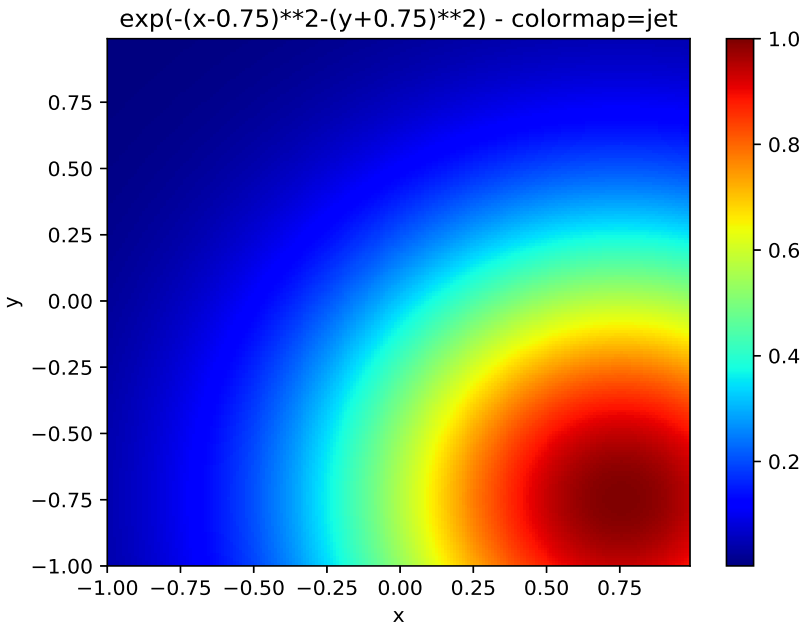


pcolor (ou pcolormesh) - colormap

Possibilité de modifier la carte des couleurs (colormap)

```
x = np.linspace(-1,1,n)
y = np.linspace(-1,1,n)
for i in np.arange(n):
    for j in np.arange(n):
        M[i,j] = np.exp(-(x[i]-0.75)**2-
                 (y[j]+0.75)**2)

# transposee!
plt.pcolormesh(x, y, M.T,cmap='jet')
plt.colorbar() # echelle graduee
plt.xlabel('x'); plt.ylabel('y')
plt.title('exp(-(x-0.75)**2-'+'(y+0.75)
                **2) - colormap=jet')
```

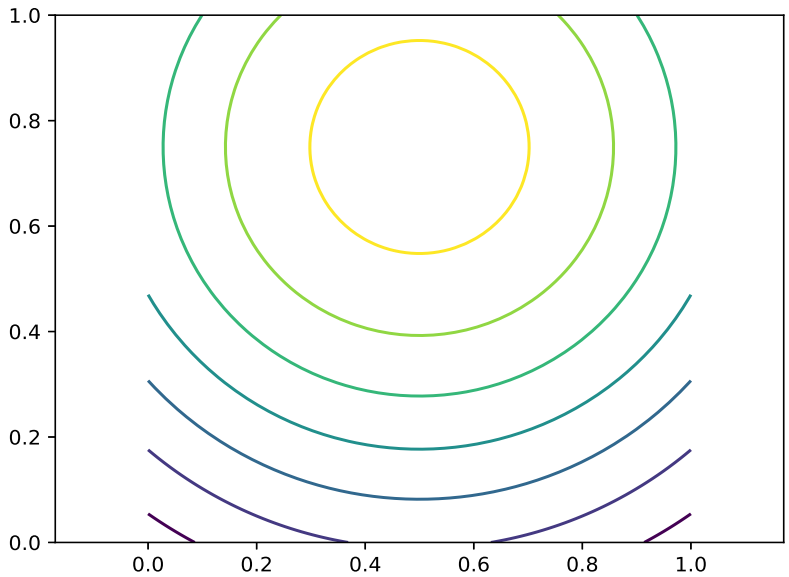


contour et contourf

- Affichage des lignes de niveau
- plt.contourf remplit avec des couleurs

```
x= np.linspace(0, 1, 100); y=x
z = np.zeros((100, 100))
for i in np.arange(100):
      for j in np.arange(100):
          z[i,j] =
          np.exp(-(x[i]-0.5)**2 -
                 (y[j]-0.75)**2)

plt.contour(x, y, z.T)
plt.axis('equal')
```



Et bien d'autres ...

- *quiver*: représentation de champs vectoriels avec des flèches
- *tripcolor*: équivalent de *pcolormesh* pour un maillage triangulaire
- Représentation de surfaces en 3D avec mplot3d
- ...

Exporter une figure

- *savefig* permet d'exporter une figure
- Le format de sortie est celui de l'extension du fichier en argument

```
f= lambda x: x**2 + 1
x = np.linspace(0, 10, 100)
plt.plot(x, f(x))
plt.savefig('myfig.png') # dans le répertoire
    courant
plt.savefig('dos1/myfig.png') #dans le sous-
    dossier dos1
plt.savefig('../dossier/myfig.eps') # .. remonte
    au dossier parent, qui contient le dossier
    dos2
plt.savefig('tmp/myfig.pdf')
plt.savefig('tmp/myfig.jpg')
```