Calcul Scientifique

October 8, 2014

0.0.1 Journée à la découverte de Python

1 Python pour le Calcul Scientique

Dev@LR - Rémy Mozul

2 Sommaire

- Introduction
- Plus de python
- Numpy
- Scipy
- Matplotlib
- Code compilé

2.1 Introduction

- Ipython
- Règles de développement
- Documentations existantes

2.1.1 ipython

- Interactive python
- Surcouche à l'interpéteur python classique
- Complétion automatique
- Aide interactive
- Support des commandes shell (pour les Unixiens)
- Meilleure gestion de l'historique des commandes

```
In [1]: import math
        math?
        !ls -l #unix
        #! dir #dos
```

```
total 616
-rw-r--r--  1 mozul  staff  219706  8 oct 10:49 Calcul Scientifique.ipynb
-rw-r--r--@ 1 mozul  staff     856  8 sep 16:47 fixed_mathjax.tpl
-rw-r--r--@ 1 mozul  staff    2515  1 oct 10:35 latex-to-slide-patch.tar.gz
-rw-r--r--@ 1 mozul  staff   72229 22 sep 17:56 numpy_indexing.png
```

```
-rw-r--r--  1 mozul  staff      97  1 oct 10:41 run_slides.sh
-rw-r--r--@ 1 mozul  staff    5095  8 sep 16:47 slides.tpl
```

2.1.2 Règles de développement

- Python offre beaucoup de libertés
- Peut-être trop parfois !
- Important de se fixer des règles :
  - Conventions de nommage
  - Toujours documenter ses fonctions/classes (docstring)
  - Tests

Les conseils officiels : http://legacy.python.org/dev/peps/pep-0008/

```
In [2]: import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.1.3 Documentation

- Enormément de documentation en ligne : https://docs.python.org/3/
  - Tutoriaux
  - Documentation des modules/fonctions
  - Code sources des modules
- Recherche internet
  - De module existant
  - De résolution de problème

2.2 Plus de python

- Indexation des listes
- List comprehension et lambda functions
- Filter, map

2.2.1 Indexation des listes

- indexation commence à 0
- opérateur d'indexation *[debut:fin:pas]*
- *end* est non inclus dans la selection
- si l'index i est négatif, il est interprété comme $n+j$ (où n est la taille)
- si les entiers entre les *:* sont omis, les valeurs par défaut *0*, taille de la liste et 1 sont utilisées
- syntaxe utilisable avec le constructeur *range(debut,fint,pas)*

Exercice :

- Générer une liste allant de 0 à 9 (avec la fonction range et le constructeur de list)
  - Afficher le premier élément, le dernier, l'avant dernier
- A partir de celle-ci, créer une liste
  - allant de 0 à 4
  - allant de 4 à 9
  - contenant les nombres pairs
  - contenant les nombres impairs dans l'ordre décroissant jusqu'à 3
- Générer une liste de nombre pairs allant de 2 à 10

```
In [3]: #python 2
        #l = range(10)
        #l = list(xrange(10))

        #python 3
        l = list(range(10))
        print(l[0]); print(l[-1]); print(l[-2])
        a = l[:]; print(a)
        a = l[:5]; print(a)
        b = l[4:]; print(b)
        c = l[::2]; print(c)
        d = l[9:2:-2]; print(d)
        l2= list(range(10,1,-2)); print(l2)
```

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

2.2.2 List comprehension et lambda function

- Manière classique de générer une liste : avec une boucle *for* et la méthode *append*
- List comprehension : manière compacte de générer des listes en une ligne
  - Utilise une sequence ou un itérateur
  - Syntaxe : *[ ]* contenant une expression suivi de 0 ou plus *for* ou *if* mots-clef
- Lambda function : manière compacte de générer des petites fonctions anonyme en une ligne (utile quand une fonction est un paramètre d'une autre)

– Syntaxe : *lambda args : expression(args)*

Exercice :

- Créer, avec une boucle, une liste contenant $sin(x)$ pour des valeurs x parcourant 0 à 1 par pas de 0.1
- Créer la même liste en une seule ligne
- Créer une fonction f référencant une *lambda function* faisant $sin(x^2)$
- Créer une liste contenant $sin(x^2)$ pour des valeurs x parcourant 0 à 1 par pas de 0.1 en utilisant f

```
In [4]: import math
        s = []
        for x in range(11):
            s.append(math.sin(x*0.1))
        s2 = [math.sin(x*0.1) for x in range(11)]
        print(s == s2)

        f = lambda x : math.sin(x*x)
        s3 = [f(x*0.1) for x in range(11)]
```

```
True
```

2.2.3 filter et map

2 fonctions qui prennent en entrée une fonction et une liste. En python 3, la valeur de retour est un objet permettant de créer des liste. En python 2, la valeur de retour est un objet list.

- Filter : retourne la liste des éléments d'entrée dont l'évaluation par la fonction est vraie -> le résultat de la fonction est interprété comme un booléen
- Map : retourne l'évaluation par la fonction de chaque élément de la liste d'entrée

Note : En général, quand il existe des syntaxes équivalentes, la list comprehension est plus rapide que les filter/map/, qui sont plus rapide que les boucles

Exercice : Tester les 2 fonctions :

- récupérer une liste d'entier dont tout les éléments sont divisibles par 3
- récupérer une liste correspondant à $sin(x)$ où x est une autre liste.

```
In [5]: l1 = list(filter(lambda x: x%3==0, range(15)))
        print(l1)
        l2 = list(map(math.sin,l1))
        print(l2)
```

```
[0, 3, 6, 9, 12]
[0.0, 0.1411200080598672, -0.27941549819892586, 0.4121184852417566, -0.5365729180004349]
```

2.3 Numpy

- Introduction
- Tableau : type, attribut, mémoire
- Indexing/Slicing
- Vue : référence/copy
- Constructeurs, IO
- Manipulation

2.3.1 Introduction

- Package pour le traitement de tableau à n-dimensions
- Pensé pour le calcul scientifique
- Efficace

Restrictions :

- Taille totale du tableau fixe
- Type des éléments homogène
- Type statique

Site officiel (avec documentation et tutoriaux) : http://docs.scipy.org/doc/numpy/user/

2.3.2 Type

Les tableaux numpy sont optimisés pour le traitement de valeurs numériques :

- booléens (32/64)
- entiers non signés (32/64)
- entiers signés (32/64)
- réels (32/64)
- complexes (32/64 bits)

Lors de la création d'un tableau, le type des éléments peut être :

- choisi ou inféré par le constructeur
- spécifié

L'attribut de la classe *array* qui stocke le type est *dtype*.
Exercices :

- Utiliser le constructeur de la classe *array* pour créer
  - une vecteur d'entiers
  - un vecteur de réels.
- Vérifier le type du contenu du chacun des tableaux grâce à la l'attribut *dtype*.

```
In [6]: import numpy as np
        np?

In [7]: a = np.array([1,2,3])
        print(a)
        print(a.dtype)

[1 2 3]
int64

In [8]: b = np.array( [[1.2,2.3,3.4],[4.5,5.6,6.7]])
        print(b)
        print(b.dtype)

[[ 1.2  2.3  3.4]
 [ 4.5  5.6  6.7]]
float64
```

```
In [9]: c = np.array([1,2,3],dtype='float')
        print(c.dtype)
        print(c)
        a[0] = 0.6
        c[0] = 0.6
        print(a)
        print(c)

float64
[ 1.  2.  3.]
[0 2 3]
[ 0.6  2.   3. ]
```

2.3.3 Quelques attributs

- dtype : le type des élements du tableaux
- size : entier contenant le nombre d'éléments totaux stockés dans le tableau
- shape : tuple contenant le nombre d'éléments de chaque dimensions
- data : objet reprensentant la mémoire allouée référencée par le tableau

Exercice :

- Créer un tableau contenant neuf éléments.
- Vérifier sa taille et sa forme et ce qu'est l'attribut data.
- Essayer de modifier
  - le nombre d'éléments totaux
  - la forme pour en faire une matrice 3x3.

```
In [10]: d = np.array(range(9))
         print(d.size); print(d.shape); print(d.data)

9
(9,)
<memory at 0x1126632a0>

In [11]: d.size = 10


         ---------------------------------------------------------------------------
    AttributeError                            Traceback (most recent call last)

         <ipython-input-11-2048010f9b70> in <module>()
    ----> 1 d.size = 10


         AttributeError: attribute 'size' of 'numpy.ndarray' objects is not writable


In [12]: d.shape = [3,3]
         print(d)

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

2.3.4 Indexing/Slicing

- Accès similaires aux listes
- Syntaxe utilisable pour chaque indices (séparées par des virgules)
- Selection

Exercice : Jouer avec le tableau précédent pour sélectionner des sous parties du tableau.

```
In [13]: e = d[1,:]
         print(e)

[3 4 5]

In [14]: f = d[1:2,::2]
         print(f)

[[3 5]]

In [15]: g = d[d>4]
         print(g)

[5 6 7 8]
```

2.3.5 Référence/Copie

- Numpy, quand il le peut fait des *MemoryView* plutôt que des copies. Ce sont bien des *vue* de la même mémoire. Ainsi dans les exemples précédents, modifier le contenu d'une vue, modifie toutes les autres.
- Pour faire une copie il faut utiliser la fonction *copie* du module NumPy.
- La documentation précise si une fonction/méthode rend une *vue* ou une *copie*.

Exercice :

- Créer un tableau de taille 9.
- Faire une vue avec la taille 3x3.
- Faire une copie de la deuxième lignes.
- Lire la documentation de la fonction *reshape*

```
In [16]: a = np.array(range(9))
         b = a
         b.shape = [3,3]

In [17]: print(b);
         b[1,1] = 12
         print(a);

[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 0  1  2]
 [ 3 12  5]
 [ 6  7  8]]

In [18]: c = np.copy(b[:,1])
         print(c);

[ 1 12  7]
```

```
In [19]: c[2] = 14
         print(c);
         print(b)
```

```
[ 1 12 14]
[[ 0  1  2]
 [ 3 12  5]
 [ 6  7  8]]
```

```
In [20]: np.reshape?
```

```
In [21]: A = a
         B = np.reshape(A,[3,3]) # makes a view
         B[0,0] = 1
         print(A)
```

```
[[ 1  1  2]
 [ 3 12  5]
 [ 6  7  8]]
```

```
In [22]: C = B.T
         D = np.reshape(C,9) # makes a copy
         D[0] = 100
         print(C)
         print(D)
```

```
[[ 1  3  6]
 [ 1 12  7]
 [ 2  5  8]]
[100   3   6   1  12   7   2   5   8]
```

2.3.6 Constructeurs et IO

Une liste de constructeurs :

- Pour une forme donnée : *empty*, *zeros*, *ones* (et pas plus loin !)
- Pour des vecteurs : *arange*, *linspace*
- Pour des matrices : *eye*, *diag*
- Pour des grilles : *meshgrid*
- Pour des nombre aléatoires : le sous-module *random*

Pour les lectures/écritures :

- ASCII : *savetxt* et *loadtxt*
- Binaire : *save* et *load*

Exercices : En essayer quelque uns.

```
In [23]: print(np.empty([2,2]))
         print(np.zeros([2,2]))
         print(np.ones([2,2]))
```

```
[[  0.00000000e+000   0.00000000e+000]
 [  2.25309871e-314   6.36598737e-311]]
[[ 0.  0.]
 [ 0.  0.]]
[[ 1.  1.]
 [ 1.  1.]]
```

```
In [24]: print(np.arange(3))
         print(np.arange(1.,2.,0.1))
         print(np.linspace(1.,2.,10))
         print(np.random.random([2,2]))
```

```
[0 1 2]
[ 1.   1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9]
[ 1.          1.11111111  1.22222222  1.33333333  1.44444444  1.55555556
  1.66666667  1.77777778  1.88888889  2.        ]
[[ 0.08676842  0.91837174]
 [ 0.16724932  0.79237793]]
```

```
In [25]: print(np.eye(2))
         print(np.diag(b[0:2]))
```

```
[[ 1.  0.]
 [ 0.  1.]]
[ 1 12]
```

```
In [26]: plop = np.meshgrid(np.linspace(0.,1.,2),np.linspace(0.,1.,3))
         print(plop[0].shape); print(plop[1].shape)
```

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

```
In [27]: print(np.random.random([2,2]))
```

```
[[ 0.26869066  0.30983082]
 [ 0.75972114  0.81467183]]
```

2.3.7 Manipulation

Opérations éléments par éléments :

- Les opéartions arithmétiques classiques sont supportées : `python +,-,*,/` et `**` (pour la puissance)
- Quand c'est possible, il vaut mieux utiliser les opérateurs `python +=, -=, *=` et `/=`

Méthode et fonctions utiles :

- Méthodes : *min, max, argmin, argmax, sort, etc*
- Fonctions : *dot* et *tensordot* pour faire du produit matriciel/tensoriel, toutes les fonctions du module math existent dans NumPy pour travailler avec des tableaux NumPy
- Module : *linalg* pour faire de l'algèbre linéaire (résolution de système, etc)

Exercice :

- Evaluer un polynôme en une centaine de points.
- Faire de même avec la fonction *sinus*.
- Faire un produit matrice/matrice.

```
In [28]: #np.array.<tab>
         #np.linalg?
```

```
In [29]: x = np.linspace(-1.,1.,500)
         %timeit p = x**5-4*x**4+3*x**3-2*x+1
         %timeit s = np.sin(x)
```

```
10000 loops, best of 3: 71.2 µs per loop
100000 loops, best of 3: 5.19 µs per loop
```

```
In [30]: import math
         xl = list(x)
         %timeit  [l**5-4*l**4+3*l**3-2*l+1 for l in xl]
         %timeit map(math.sin,xl)
```

```
100 loops, best of 3: 3.99 ms per loop
1000000 loops, best of 3: 239 ns per loop
```

```
In [31]: A = np.array( [[1,1],
                        [0,1]] )
         B = np.array( [[2,0],
                        [3,4]] )
         print(A*B)         # elementwise product
         print(np.dot(A,B)) # matrix product
```

```
[[2 0]
 [0 4]]
[[5 4]
 [3 4]]
```

2.4 SciPy

Le module SciPy se fonde sur NumPy pour fournir des nombreux algorithmes utilisés en calcul scientifique. Il se décompose en plusieurs sous-module dont voici la liste :

scipy.cluster	Vector quantization / Kmeans
scipy.constants	Constantes mathématiques et physiques
scipy.fftpack	Transformée de Fourier
scipy.integrate	Routines d'intégration
scipy.interpolate	Interpolation
scipy.io	Entrée/Sortie de données
scipy.linalg	Routines d'algèbre linéaire
scipy.ndimage	Traitement d'image à n dimensions
scipy.odr	Orthogonal distance regression
scipy.optimize	Optimisation
scipy.signal	Traitement de signal
scipy.sparse	Matrices creuses
scipy.spatial	Algorithme et structure de données d'espace
scipy.special	Fonctions mathématiques spéciales
scipy.stats	Statistiques

Détailler tout les modules seraient fastidieux et ennuyeux.
Travail libre : tester une fonction d'un module particulièrement interessant.
Exercice proposé :

- Utiliser le module *integrate* pour calculer la somme du polynôme créé plutôt.
- Utiliser le module *interpolate* pour calculer les interpolations linéaire, quadratique et cubique de la fonction sinus sur quelques points entre $[0, 2\pi]$

```
In [32]: from scipy import integrate
         integrate.quad(lambda x: x**5-4*x**4+3*x**3-2*x+1, -1.,1.)
```

```
Out[32]: (0.3999999999999999, 2.101996032375398e-14)
```

```
In [33]: from scipy import interpolate
         x = np.linspace(0.,2*math.pi,5)
         s = np.sin(x)
         lin = interpolate.interp1d(x,s)
         qua = interpolate.interp1d(x,s,'quadratic')
         cub = interpolate.interp1d(x,s,'cubic')
```

2.5 Matplotlib

- Introduction
- Plot
- Quelques commandes
- Axes
- Text et Annotations
- Figure et Subplot

2.5.1 Introduction

- Utilise des tableaux NumPy (ou des listes) pour générer des graphiques.
- Très pratique pour des représentation de données 1 ou 2D.
- Pour de la representation 3D il vaut mieux passer par d'autres modules (vtk, mayavi, etc).

Documentation officielle : http://matplotlib.org/1.4.0/contents.html

2.5.2 Plot

Le module matplotlib travail sur des objets de manière tacite. Les commandes *plot* créerons de nouveaux objets au sein d'autres que l'utilisateur ne voit pas forcément.

Une fois les différents objets créés, la figure peut-être générée avec la commande *show*.

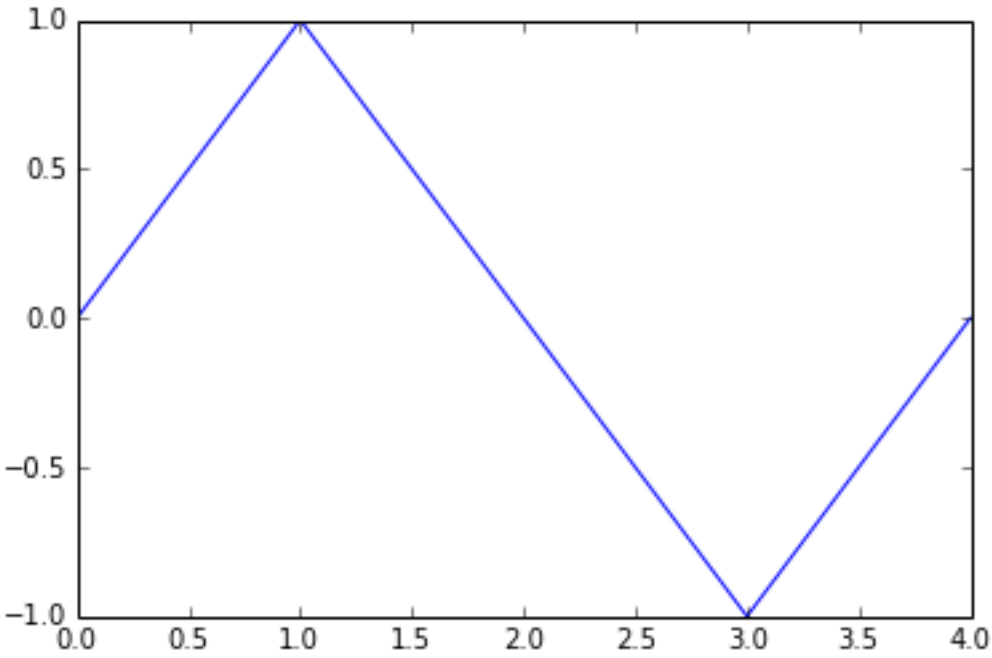
Exercice :

- Tracer la fonction $sin(x)$ entre $[0, 2\pi]$ avec les commandes *plot* et *show* du module *matplotlib.pyplot*. *plot* peut prendre uniquement le tableau de valeur en entrée
- Changer la valeur de l'axe des abscisses en ajoutant le tableaux contenant les points d'évaluation de la fonction.
- Changer le styple de l'affichage grâce à une chaine de caractère en troisième argument 'ro'.

```
In [34]: %matplotlib inline
         from  matplotlib import pyplot as plt
```

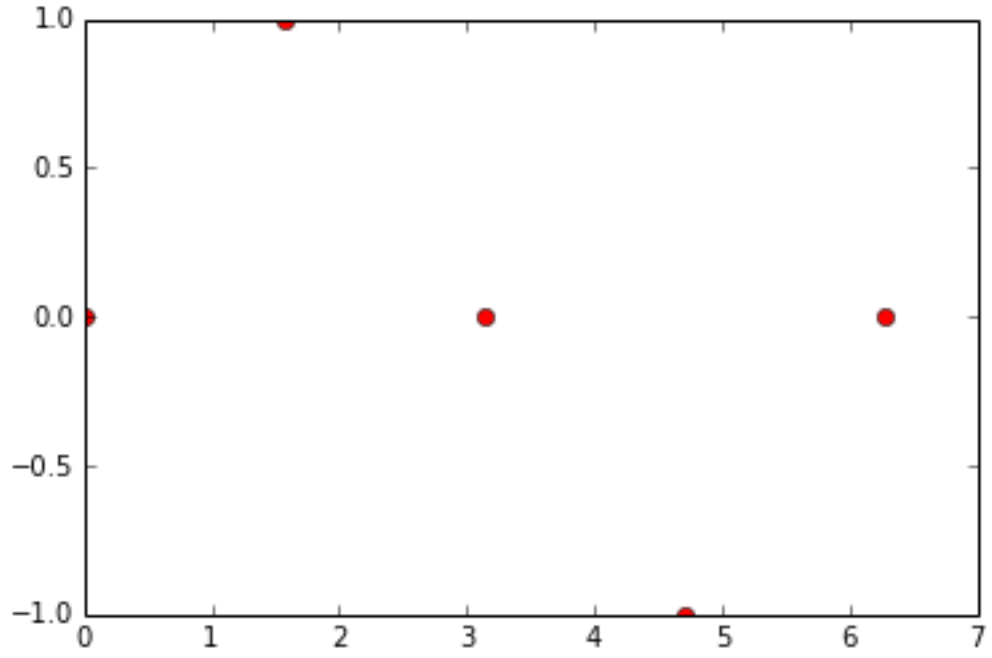
```
In [35]: plt.plot(s)
```

```
Out[35]: [<matplotlib.lines.Line2D at 0x11402e710>]
```




```
In [36]: plt.plot(x,s,'ro')
```

```
Out[36]: [<matplotlib.lines.Line2D at 0x11412b4a8>]
```



- Les couleurs peuvent être décrites par :

  - une lettre prédefine (b: bleu, g: vert, r: rouge, c: cyan, m: magenta y: jaune, k: noire, w: blanc)
  - une valeur entre 0 et 1 pour un niveau de gris
  - un tuple de 3 valeurs entre 0 et 1 pour un style RVB
  - un marqueur html en hexadécimal (#eeefff)

- Les styles de lignes (qui relient les points) :
  - '-' solide
  - '–' dashed
  - '-.' dash_dot
  - ':' dotted
  - '' ou ' ' ou None pour rien

- Les styles des marqueurs de point :
  - '.' point
  - ',' pixel
  - 'o' circle
  - 'v' triangle_down
  - '^' triangle_up
  - 's' square
  - '*' star
  - pour les autres http://matplotlib.org/api/markers_api.html#module-matplotlib.markers

2.5.3 Quelques commandes

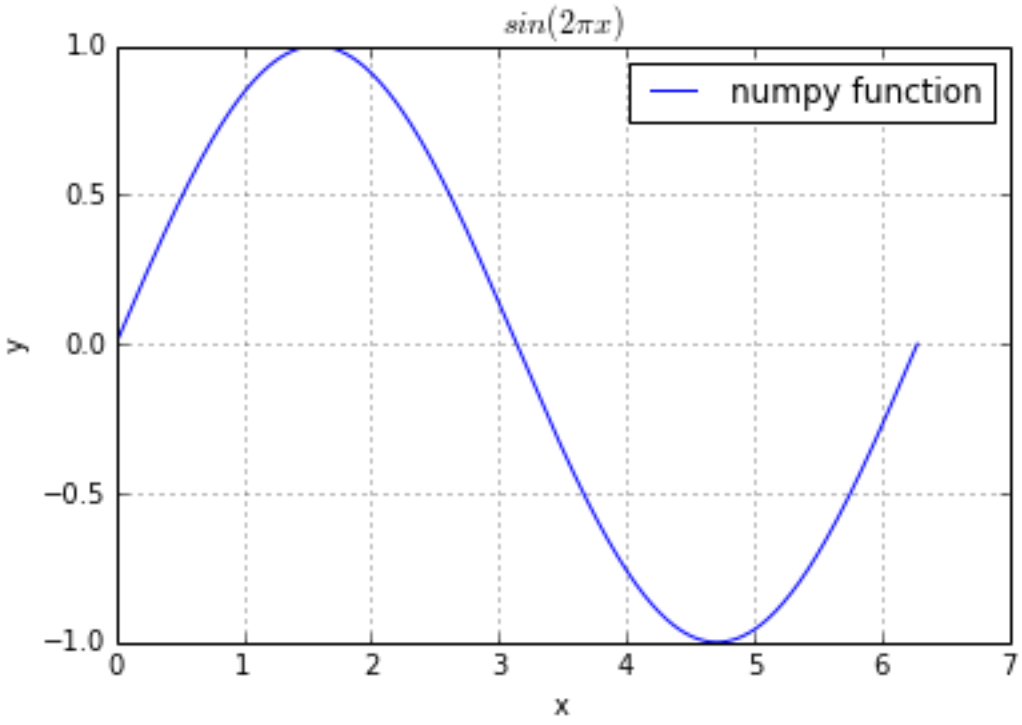
Quelques fonctions simples pour rendre le graphique plus agréable :

- *title* pour mettre un titre à la figure
- *xlabel* pour mettre un label sur l'axe des abscisses
- *ylabel* pour mettre un label sur l'axe des ordonnées

Ces commandes retournent un objet de type *text* qui peut être référencé pour en changer les propriétés a postorieri (taille de la police, orientation, http://matplotlib.org/api/text_api.html#matplotlib.text.Text)

- *grid* permet d'ajouter ou d'enlever la grille de fond
- *legend* afficher la légende (utilise le paramètre *label* d'un plot)
- *setp* changer une propriété d'un objet

```
In [37]: lx = np.linspace(0.,2*math.pi,1000)
         si = plt.plot(lx,np.sin(lx),label='numpy function')
         plt.xlabel('x')
         plt.ylabel('y')
         plt.title("$sin(2\pi x)$")
         plt.legend(loc='upper right')

         plt.grid(True)
```

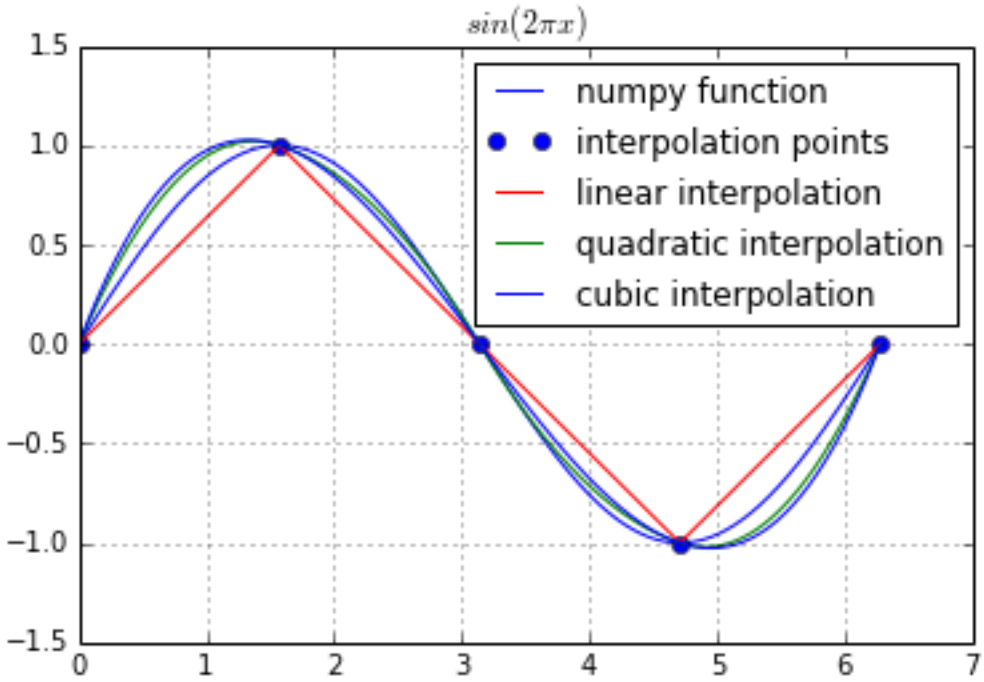


Pour afficher plusieurs courbes dans une même figure :

- Soit données les courbes les une derrières les autres dans un seul *plot* .
- Soit appeler plusieurs fois la commande *plot* avant l'appel à *show*

Exercice : Tracer sur une même figure la fonction sinus, les points d'interpolation uniquement avec des o, puis les différentes interpolations.

```
In [38]: ll = plt.plot(lx,np.sin(lx),'b-',x,s,'o',label='numpy function')
         plt.setp(ll[1],label='interpolation points')
         plt.plot(lx,lin(lx),'r',label='linear interpolation')
         plt.plot(lx,qua(lx),'g',label='quadratic interpolation')
         plt.plot(lx,cub(lx),'b',label='cubic interpolation')

         plt.title("$sin(2\pi x)$")
         plt.legend(loc='upper right')

         plt.grid(True)
```



2.5.4 Axes

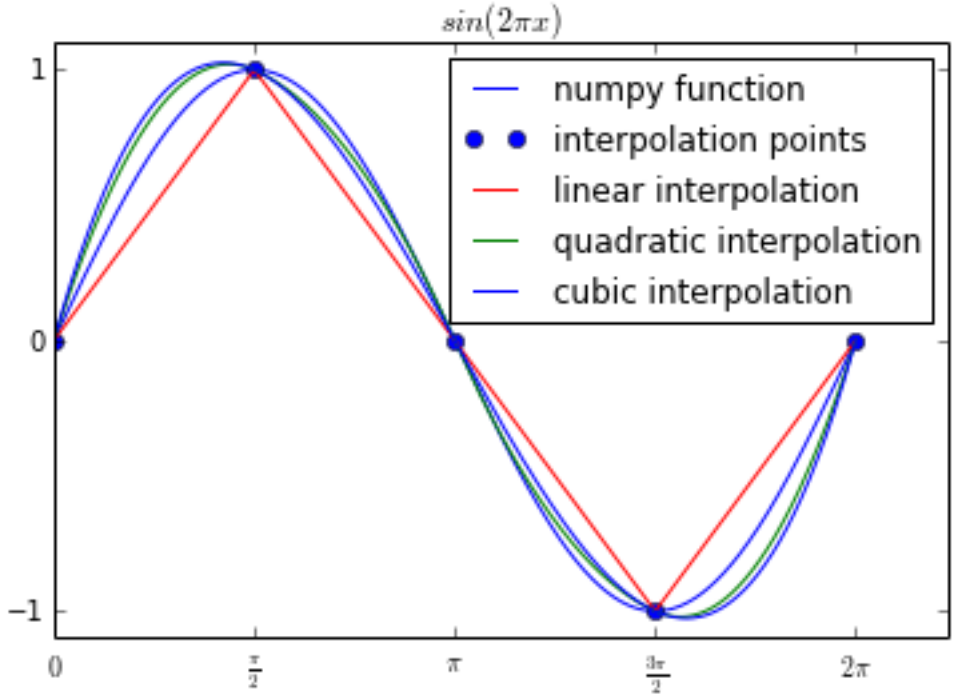
Quelques commandes agissant sur les axes :

- Ajustement de la taille : *xlim*, *ylim* ou *axis*
- Spécification des gradudations : *xticks*, *yticks*
  - Permet de choisir les gradudations
  - Permet aussi de choisir ce qui sera affiché sur les axes

Exercice : Réajuster les axes de la figure précédent pour afficher $[0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}, 2\pi]$.

```
In [39]: ll = plt.plot(lx,np.sin(lx),'b-',x,s,'o',label='numpy function')
         plt.setp(ll[1],label='interpolation points')

         plt.plot(lx,lin(lx),'r',label='linear interpolation')
         plt.plot(lx,qua(lx),'g',label='quadratic interpolation')
         plt.plot(lx,cub(lx),'b',label='cubic interpolation')

         plt.title("$sin(2\pi x)$")
         plt.legend(loc='upper right')

         plt.ylim([-1.1,1.1])
         plt.yticks([-1,0,1])
         plt.xticks([0,math.pi/2.,math.pi,3*math.pi/2.,2*math.pi],
                    [r"$0$",r"$\frac{\pi}{2}$",r"$\pi$",r"$\frac{3\pi}{2}$",r"$2\pi$"])

Out[39]: ([<matplotlib.axis.XTick at 0x1143e8f28>,
           <matplotlib.axis.XTick at 0x1143d0908>,
```

```
 <matplotlib.axis.XTick at 0x1143dec88>,
 <matplotlib.axis.XTick at 0x1143bf860>,
 <matplotlib.axis.XTick at 0x11446b2b0>],
<a list of 5 Text xticklabel objects>)
```



2.5.5 Text et Annotate

Functions permettant d'ajouter du texte sur une figure :

- *text* : n'importe où dans la figure
- *annotate* : du texte pointant sur un point

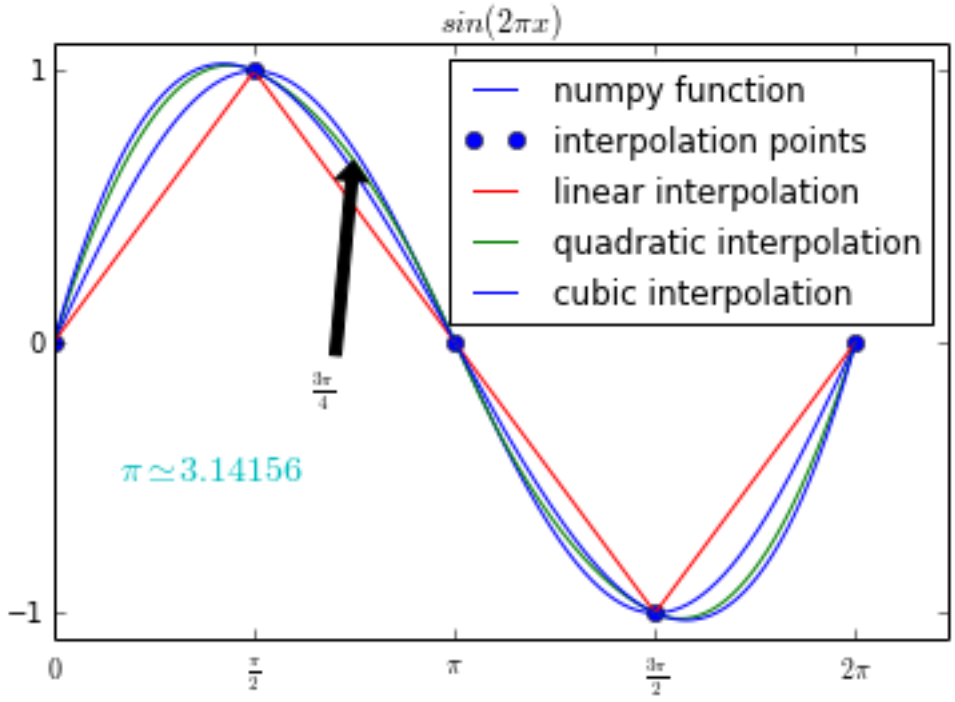
Chacune de ses fonctions retourne un objet, dont les propriétés peuvent modifiées grâce à *setp*. Propriétés interessantes :

- color
- backgroundcolor
- size
- style

Exercice : Ajouter un texte (par exemple la valeur approximative de π, ainsi qu'une annotation sur la valeur de *sinus* en $\frac{3\pi}{4}$

```
In [40]: ll = plt.plot(lx,np.sin(lx),'b-',x,s,'o',label='numpy function')
         plt.setp(ll[1],label='interpolation points')

         plt.plot(lx,lin(lx),'r',label='linear interpolation')
         plt.plot(lx,qua(lx),'g',label='quadratic interpolation')
```

```
plt.plot(lx,cub(lx),'b',label='cubic interpolation')

plt.title("$sin(2\pi x)$")
plt.legend(loc='upper right')

plt.ylim([-1.1,1.1])
plt.yticks([-1,0,1])
plt.xticks([0,math.pi/2.,math.pi,3*math.pi/2.,2*math.pi],
           [r"$0$",r"$\frac{\pi}{2}$",r"$\pi$",r"$\frac{3\pi}{2}$",r"$2\pi$"])

t = plt.text(0.5,-0.5,"$\pi \simeq 3.14156$")
plt.setp(t,color='c',fontsize=14)
t = plt.annotate(r'$\frac{3\pi}{4}$',xy=(3*math.pi/4,math.sqrt(2)/2.),
                 xytext=(2.,-0.2),arrowprops=dict(facecolor='black', shrink=0.05))
```



2.5.6 Figure et Subplot

Il est possible de générer plusieurs figures. Par défaut toutes les commandes présentées travaillent sur la figure courante. Il suffit d'appeler la commande *figure* (avec un entier en paramètre) pour créer une autre figure.

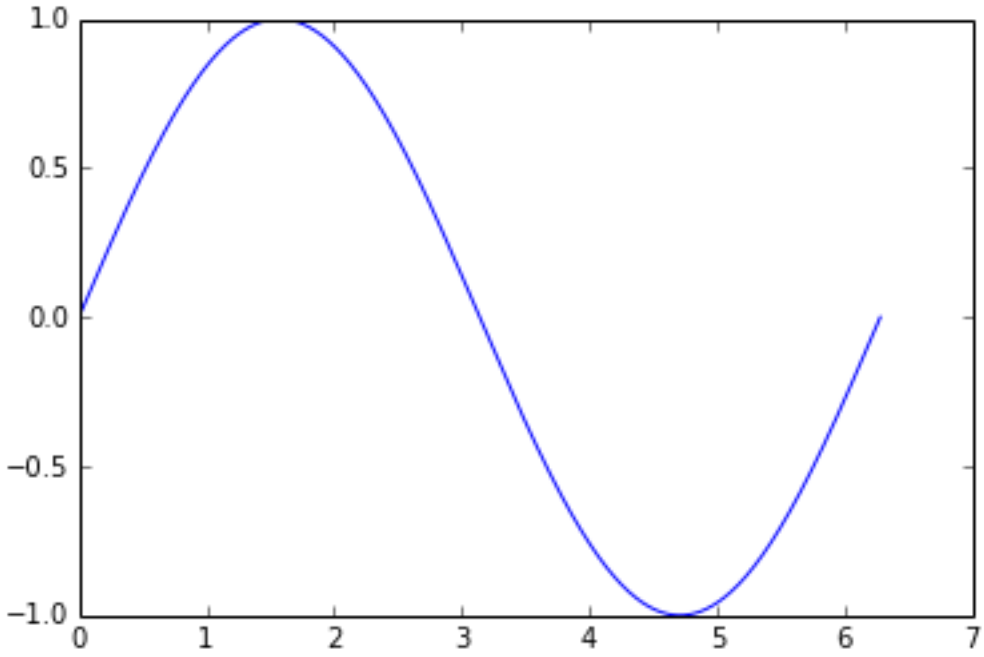
De même dans une figure, plusieurs courbes peuvent présentée sur des axes différents, avec la command *subplot*. Trois arguments sont nécessaires : le nombres de lignes, nombre de colonnes, l'indice du graphe.

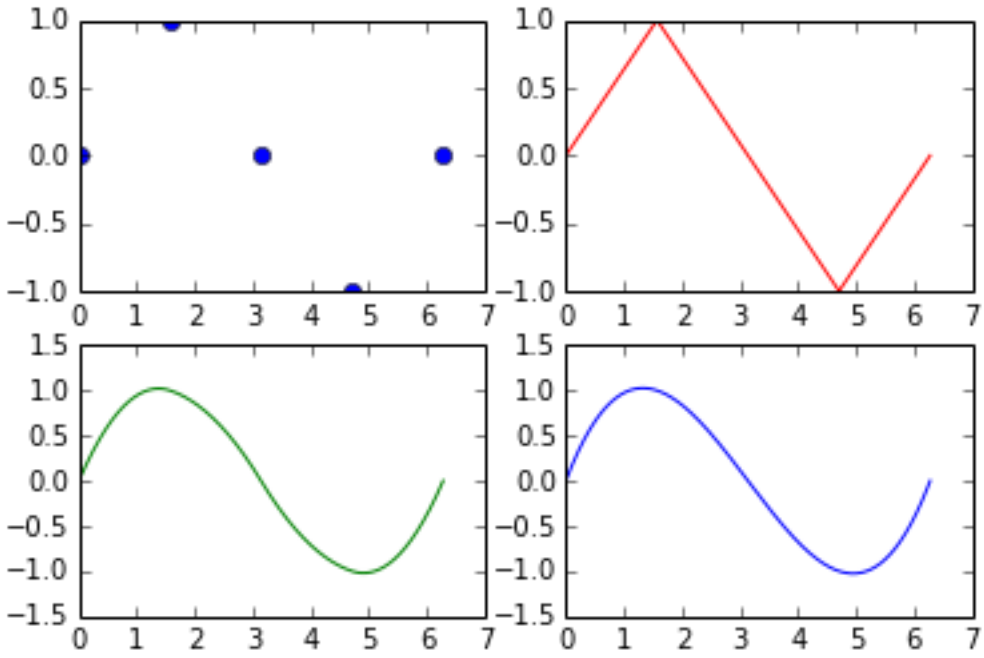
Exercice : Générer deux figures une contenant sinus et une contenant les points d'interpolation et les interpolations sur des graphes différents.

```
In [41]: plt.plot(lx,np.sin(lx),'b-',label='numpy function')
         plt.figure(2)
         plt.subplot(221)
```

```
plt.plot(x,s,'o',label='interpolation points')
plt.subplot(222)
plt.plot(lx,lin(lx),'r',label='linear interpolation')
plt.subplot(223)
plt.plot(lx,qua(lx),'g',label='quadratic interpolation')
plt.subplot(224)
plt.plot(lx,cub(lx),'b',label='cubic interpolation')
```

Out[41]: [<matplotlib.lines.Line2D at 0x1147024a8>]





2.6 Code compilé

- Python reste plus lent que du code compilé. Pour tendre vers cette efficacité il faut soit :
  - Faire en sorte que le code Python devienne compilable.
  - Appelé dans le python des fonctions d'une bibliothèque compilé.
- Le langage de prédilection est le C/C++
- Pour transformer automatiquement (plus ou moins) du Python en C :
  - Cython
  - Pythran
  - Julia
- Pour appeler une bibliothèque dans Python :
  - Cython pour du C
  - f2py pour du Fortran (ou rendre sa bibliothèque compatible en C avec le module iso_c_binding)