1. Bibliothèques en Python

Il existe plusieurs manières d'importer une bibliothèque. Imaginons que l'on ait créé le fichier `test.py` dont le code est le suivant:

```
print "Hello world"

def f(x):
    return x ** 2

def g(x):
    return x ** 0.5

variable = 5
```

On suppose que le fichier `test.py` se trouve dans le répertoire courant. Voici plusieurs manières de l'importer comme une bibliothèque.

Syntaxe	**Accès à `f`**	**Accès à `g`**	**Accès à `variable`**
`import test`	`test.f`	`test.g`	`test.variable`
`import test as t`	`t.f`	`t.g`	`t.variable`
`from test import *`	`f`	`g`	`variable`
`from test import g`	Impossible	`g`	Impossible
`from test import g as h`	Impossible	`h`	Impossible

Remarque importante: une fois une bibliothèque chargée, toute modification de son code-source est sans effet. Il faut utiliser `reload(test)` pour forcer le rechargement.

On pourra constater qu'à la différence de l'instruction `%run test.py`, la chaîne de caractères `"Hello world"` n'est affichée qu'au premier chargement de la bibliothèque. Il faudrait utiliser `reload(test)` pour ré-exécuter le contenu du fichier.

In [1]:

```
import test
print test.f
print test.g
print test.variable
```

```
Hello world
<function f at 0x103bfcc80>
<function g at 0x103bfccf8>
5
```

In [2]:

```
reload(test)
import test as t
print t.f
print t.g
print t.variable
```

```
Hello world
<function f at 0x103bfcc08>
<function g at 0x103bfcc80>
5
```

In [3]:

```
from test import *
print f
print g
print variable
```

```
<function f at 0x103bfcc08>
<function g at 0x103bfcc80>
5
```

In [4]:

```
# Remise à zéro de l'environnement (uniquement les noms de variables!)
%reset -f

from test import g
print g
print variable    # Erreur
```

```
<function g at 0x103bfcc80>

---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
 last)
<ipython-input-4-10141bc29347> in <module>()
      4 from test import g
      5 print g
----> 6 print variable    # Erreur

NameError: name 'variable' is not defined
```

In [5]:

```
# Remise à zéro de l'environnement (uniquement les noms de variables!)
%reset -f

from test import g as h
print h
print g    # Erreur
```

```
<function g at 0x103bfcc80>

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
 last)
<ipython-input-5-793b05415397> in <module>()
      4 from test import g as h
      5 print h
----> 6 print g    # Erreur

NameError: name 'g' is not defined
```

2. La bibliothèque `numpy`

La bibliothèque `numpy` introduit le type `ndarray` ainsi que de nombreuses fonctions de calcul flottant notamment. Traditionnellement, on l'importe:

- soit directement dans l'environnement courant (`from numpy import *`);
- soit sous un nom abrégé (`import numpy as np`).

La bibliothèque `numpy` est spécialisée dans la manipulation des tableaux (array). On l'utilisera essentiellement pour les vecteurs et les matrices

• Les tableaux `numpy` ne gèrent que les objets de même type

• La bibliothèque `numpy` propose un grand nombre de routines pour un accès rapide aux données (ex. recherche, extraction), pour les manipulations diverses (ex. tri), pour les calculs (ex. calcul statistique et scientifique, calcul élément par élément, calcul matriciel)

• Les tableaux `numpy` sont plus performants (rapidité, gestion de la volumétrie) que les itérables usuel de Python (listes, tuples...)

• Les tableaux `numpy` sont sous-jacents à de nombreux packages dédiés au calcul scientifique sous Python.

• Une matrice est un tableau (array) à 2 dimensions, un vecteur est un tableau (array) avec une seule ligne

• Indexation : En plus des méthodes d'accès aux éléments déjà vues pour les listes (plages d'éléments avec :) il supporte l'indexation par une liste (ou un tableau) d'entiers. On fait alors référence au tableau constitué des cases situées aux indices spécifiés. On peut aussi indexer par un tableau de bool de même taille. On fait alors référence aux emplacements où se trouve la valeur True.

• Ces indexations sont en outre accessibles en écriture, soit par une seule valeur (qui est alors copiée dans toutes les cases correspondantes), soit par un tableau de valeurs de même forme.

• À la différence des autres conteneurs, la plupart des opérateurs $(+, *, **, etc.)$ agissent élément par élément sur les tableaux.

• Il est possible de faire des tableaux de tableaux. Lorsqu'un tableau contient des tableaux de même type et de même longueur, il devient multi-dimensionnel et supporte alors une indexation multiple (un indice par axe, séparés par des virgules).

Pour plus d'infos sur cette bibliothèque : http://docs.scipy.org/doc/numpy/reference/index.html (http://docs.scipy.org/doc/numpy/reference/index.html)

2.1 Les vecteurs

In [6]:

```
# Création de vecteurs :
from numpy import *

T1 = array([5, 4, 3, 2, 1, 0])
print T1
print type(T1)
print len(T1)
```

```
[5 4 3 2 1 0]
<type 'numpy.ndarray'>
6
```

In [7]:

```
# Indexation, accès
T1 = array([5, 4, 3, 2, 3, 0])
print T1[2]
print T1[::2]
print T1[T1]
print T1>2
print T1[T1 > 2]
T1[T1 > 2] *= -1
print T1
```

```
3
[5 3 3]
[0 3 2 3 2 5]
[ True  True  True False  True False]
[5 4 3 3]
[-5 -4 -3  2 -3  0]
```

In [8]:

```
# Création de vecteurs de valeurs particuliers :
#suite arithmétique
a=arange(0,10)
b=arange(start=10,stop=0,step=-2)
print a
print b

#subdivision uniforme de [a,b] de pas h ATTENTION b n'est pas inclus !!
c=arange(start=0,stop=1,step=0.1)
d=arange(start=0.2,stop=1.2,step=0.3)
print c
print d
c=arange(0,1,0.1)
print c
```

```
[0 1 2 3 4 5 6 7 8 9]
[10  8  6  4  2]
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
[ 0.2  0.5  0.8  1.1]
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9]
```

In [9]:

```
# subdivision uniforme de [a,b] à n éléments
# ATTENTION b EST INCLUS CETTE FOIS !!!!
a=linspace(start=0,stop=10,num=5)
print a
```

```
[  0.    2.5   5.    7.5  10. ]
```

In [10]:

```
# vecteur de 1...
a=ones(5)
print a
b=full(5,3.2)
print b
```

```
[ 1.  1.  1.  1.  1.]
[ 3.2  3.2  3.2  3.2  3.2]
```

In [11]:

```
# Indexation
T1 = array([5, 4, 3, 2, 3, 0])
print T1
# de 1 à 3
print(T1[1:3])
# de 1 à 5 de 2 en 2 (pas négatif possible), 5 est exclu
print(T1[1:5:2])
# début jusqu'à 3
print(T1[:3])
# de 2 à la fin
print(T1[2:])
# dernier élément
print(T1[-1])
#3 derniers éléments
print(T1[-3])
# deux en deux
print T1[::2]
# indexation par un vecteur d'entiers
print T1[T1]
# indexation par un booléen
print T1[T1 > 2]
# On peut modifier les éléments :
T1[T1 > 2] *= -1
print T1
T1[2]=1
print T1
```

```
[5 4 3 2 3 0]
[4 3]
[4 2]
[5 4 3]
[3 2 3 0]
0
2
[5 3 3]
[0 3 2 3 2 5]
[5 4 3 3]
[-5 -4 -3  2 -3  0]
[-5 -4  1  2 -3  0]
```

In [12]:

```
# ATTENTION Duplication !!!
t1 = array([1, 2, 3, 4])
t2 = t1[1:3]    # Pas de duplication, toute modification de t2 aura un effet sur t1
print t1
print t2
t2[0] = 5
print t2
print t1
```

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

In [13]:

```
# Comment dupliquer comme il faut !
t1 = array([1, 2, 3, 4])
t2 = array(t1[1:3])    # Duplication
print t1
print t2
t2[0] = 5
print t2
print t1
```

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

2.2 Les matrices

Pour fabriquer des matrices avec `numpy`, on fait un tableau de tableaux

In [14]:

```
T2 = array([[0, 1],    # Il est possible de passer à la ligne avant de fermer des c
            [2, 3],    # ça permet de rendre le code plus lisible (ici une liste de
            [4, 5]])
print T2
print type(T2)
print len(T2)
print T2[0]
print T2[2, 1]
print T2[1:, :]
print T2[1:,:1]
# demander la taille de la matrice
print T2.shape
```

```
[[0 1]
 [2 3]
 [4 5]]
<type 'numpy.ndarray'>
3
[0 1]
5
[[2 3]
 [4 5]]
[[2]
 [4]]
(3, 2)
```

2.3 Opérations sur les matrices et les vecteurs

Voici une liste non exhaustive des champs et méthodes du type `ndarray`.

Champ ou méthode	Usage
`T`	Transposée (pour les tableaux à deux dimensions)
`astype(t)`	Renvoie une copie du tableau dont les cases ont été converties dans le type `t`

Champ ou méthode	Usage
`copy()`	Renvoie une copie du tableau
`dtype`	Type (commun) des éléments du tableau
`fill(value)`	Remplit toutes les cases avec une valeur
`flatten()`	Renvoie une copie du tableau réarrangé sur une seule dimension
`ndim`	Nombre de dimensions du tableau
`shape`	La liste des tailles du tableau selon chaque dimension
`size`	Le nombre total d'éléments du tableau
`tofile(filename)`	Enregistre le tableau dans un fichier

In [15]:

```
T2 = array([[0, 1, 2],
            [3, 4, 5],
            [6, 7,8]], float)
print T2.shape
print T2.ndim
print T2.dtype
print T2.T
```

```
(3, 3)
2
float64
[[ 0.  3.  6.]
 [ 1.  4.  7.]
 [ 2.  5.  8.]]
```

2.4 Fonctions introduites par numpy

La bibliothèque `numpy` introduit ou redéfinit un grand nombre de fonctions numériques. Celles-ci peuvent agir sur des valeurs numériques individuelles, ou sur des tableaux (ou listes) élément par élément. En voici quelques-unes.

Fonction	Usage
`numpy.abs` `conj` `real` `imag` `angle`	Module, conjugué, partie réelle, partie imaginaire, argument
`cos` `sin` `tan` `sinh` `cosh` `tanh`	
`arccos` `arcsin` `arctan` `arcsinh` `arccosh` `arctanh`	
`exp` `log`	
`numpy.round` `floor` `ceil`	Arrondi au plus proche, arrondi inférieur, arrondi supérieur

Elle introduit également de nombreuses fonctions spécifiques aux tableaux. En voici une petite liste.

Fonction	Usage

`sum` `prod`	Somme et produit des éléments
`numpy.min` `numpy.max` `mean` `median` `std`	Minimum, maximum, moyenne, médiane, écart-type
`cumprod` `cumsum`	Produit ou somme cumulés
`argmin` `argmax` `argsort`	Indice du minimum, du maximum, tableau d'indices permettant de trier le tableau
`convolve`	Convolution discrète

Mentionnons encore quelques constantes utiles: `pi`, `e`, `inf`, `nan`, etc.

Attention: importer `numpy` dans l'espace de base sans préfixe (`from numpy import *`) ne remplace pas les fonctions prédéfinies de Python (`min`, `max`, `abs`...). On devra donc les préfixer si on souhaite utiliser la version `numpy`. Cela n'a en général pas d'importance, sauf pour `min`, `max` et `round`.

In [16]:

```
import numpy
from numpy import *

t = array([[1, 2, 3, 4],[2,3,1,5]])
#print max(t)
print numpy.max(t)
print t.max()
```

```
5
5
```

Remarque: certaines des fonctions qui agissent sur les tableaux prennent un paramètre supplémentaire optionnel axis pour indiquer dans quelle dimension effectuer le calcul.

In [17]:

```
t = array([[1, 2], [3, 4]])
print t.max(axis = 0)
print t.max(axis = 1)
print t.max(axis = None)
```

```
[3 4]
[2 4]
4
```

La bibliothèque `numpy` fournit aussi des fonctions pour créer et gérer facilement des tableaux à plusieurs dimensions.

Fonction	**Usage**
`arange(start=0, stop, step=1)`	Similaire à `range`, mais renvoie un tableau et les arguments peuvent être de type flottant
`linspace(start, stop, n=50)`	Renvoie un tableau à `n` valeurs uniformément réparties entre `start` et `stop` (bornes incluses)
`array(c)`	Crée un tableau à partir du conteneur `c`. Le type des éléments peut aussi être choisi.

Fonction	Usage
`zeros(shape)` `ones(shape)`	Crée un tableau de la forme désirée, rempli de zéros ou de uns. Le type des éléments peut aussi être choisi.
`identity(n)`	Crée une matrice identité de taille $n \times n$
`diag(A)`	Diagonale d'une matrice (si `A` est bidimensionnel) ou matrice diagonale (si `A` est monodimensionnel)
`dot(A, B)`	Produit matriciel
`reshape(T, shape)`	Renvoie le tableau obtenu en réarrangeant le tableau selon une nouvelle forme
`concatenate(l)`	"Concatène" une liste de tableaux entre eux
`roll(a, shift)`	Effectue une permutation circulaire des éléments d'un tableau en décalant les éléments vers la droite (les derniers reviennent à gauche)
`meshgrid(range(m), range(n))`	Renvoie deux matrices $m \times n$, la première contenant les indices des lignes, la seconde ceux des colonnes. Utile pour créer une matrice à partir de son terme général
`fromfunction(f, shape)`	Crée un tableau à partir d'une fonction
`fromfile(filename, dtype)`	Création depuis un fichier. L'inconvénient par rapport à `tofile` est qu'il faut préciser le type des éléments (`dtype`) et la forme est perdue (le tableau obtenu n'aura qu'une seule dimension, comme s'il avait subi `flatten()`)

In [18]:

```
A = identity(2, int)
B = array([[1, 2], [3, 4]])
print A
print B
print A * B
print dot(A, B)
```

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

2.5 Fonctions introduites par `numpy.linalg`

Fonction	Usage
`dot(A, B)`	Produit matriciel
`matrix_power(M, n)`	Exposant matriciel
`norm(x)`	Norme (euclidienne par défaut) d'un vecteur
`det(A)` `tr(A)`	Déterminant et trace
`eig(A)` `eigvals(A)`	Vecteurs et valeurs propres

Fonction	Usage
`matrix_rank(A)`	Rang d'une matrice
`solve(A, b)`	Résolution du système linéaire $Ax = b$
`inv(A)`	Inversion matricielle
`cholesky(A)`	Décomposition de Cholesky
`qr(A)`	Décomposition QR

D'autres fonctions sont décrites sur http://docs.scipy.org/doc/numpy/reference/routines.linalg.html (http://docs.scipy.org/doc/numpy/reference/routines.linalg.html).

In [19]:

```
from numpy.linalg import *

A = array([[2, 3, 5, 7],
           [11, 13, 17, 19],
           [23, 29, 31, 37],
           [41, 43, 47, 53]])

print det(A), trace(A)
eigenvalues, eigenvectors = eig(A)
print eigenvalues
print eigenvectors
# On calcule inv(P).A.P
print numpy.round(dot(dot(inv(eigenvectors), A), eigenvectors), 10)
v = solve(A, [1, 2, 3, 4])
print dot(A, v)
print v
```

```
880.0 99
[ 104.33603014   -4.54479131    1.02294627   -1.8141851 ]
[[ 0.08851383  0.5100893   0.43683788  0.179836  ]
 [ 0.27543406  0.27501112 -0.43784015 -0.40940659]
 [ 0.5366094   0.25224171 -0.57398719  0.76267732]
 [ 0.79268641 -0.77495284  0.53666319 -0.46729923]]
[[ 104.33603014   -0.           -0.           -0.        ]
 [   0.          -4.54479131    0.           -0.        ]
 [  -0.          -0.            1.02294627    0.        ]
 [  -0.          -0.            0.           -1.8141851 ]]
[ 1.  2.  3.  4.]
[-0.03636364 -0.17272727  0.09545455  0.15909091]
```

3. La bibliothèque `Matplotlib.pyplot`

La bibliothèque `matplotlib` (et sa sous-bibliothèque pyplot) sert essentiellement à afficher des graphismes. Son utilisation ressemble beaucoup à celle de Matlab. Traditionnellement, on l'importe:

- soit directement dans l'environnement courant (`from matplotlib.pyplot import *`);
- soit sous un nom abrégé (`import matplotlib.pyplot as plt`).

Dans ce qui suit, on suppose qu'on utilise le backend `inline`, propre aux notebooks. Si on utilise un autre backend, il pourra être nécessaire d'ajouter `show()` à la fin des commandes pour faire apparaitre la fenêtre qui contient le dessin.

Voici une courte liste de commandes possibles.

Commande	**Effet**
plot(y)	Affiche les points dont les ordonnées sont dans `y`
plot(x, y)	Affiche les points de coordonnées (x_k, y_k)
matshow(a)	Affiche une matrice
imshow(a)	Idem, mais avec plus d'options (et la possibilité de faire des sous-figures)
hist(y, bins=10)	Affiche un histogramme de la répartition des valeurs des y_k. On peut choisir le nombre de barres, voire leurs bornes
figure()	Commence une nouvelle figure (si on ne l'a pas fait, cette fonction est appelée implicitement dès qu'on commence à afficher quelque chose)
figure(figsize=(a,b))	Idem, mais permet d'imposer une taille (en pouces) à la figure
subplot(lines, columns, index)	Indique qu'on va commencer une sous-figure
suptitle("titre")	Afficher un titre au-dessus de la figure. Attention aux caractères accentués (précéder la chaîne du préfixe `u`). Il est possible de mettre du LaTeX entre dollars `$`
legend()	Afficher les légendes (s'il y en a)
axis([xmin, xmax, ymin, ymax])	Définit explicitement la taille des axes. Par défaut, ceux-ci s'adaptent automatiquement à la taille du contenu de la figure
axis("scaled")	Indique que le repère doit être orthonormé (pas de distorsion à l'affichage). À utiliser **après** `plot(...)` car son utilisation désactive l'ajustement automatique des axes
axis("off")	Ne pas afficher les axes
loglog()	Passer en échelle log-log

La fonction `plot` (la plus utilisée) peut prendre une grande variété de paramètres supplémentaires pour contrôler l'aspect des courbes tracées. Par défaut, les points sont reliés par des segments. Voici quelques paramètres optionnels possibles:

- `linestyle=s`, avec `s` qui vaut par exemple
  - `"solid"` ou `"-"` (lignes pleines),
  - `"dashed"` ou `"--"` (tirets),
  - `"dotted"` ou `":"` (pointillés),
  - `""` (pas de lignes);
- `color=c`, avec `c` qui vaut par exemple `"red"`, `"green"`, `"blue"` etc.
- `marker=m`, avec `m` qui vaut par exemple
  - `""` (pas de marqueur);
  - `","` (petits points),
  - `"."` (gros points),
  - `"+"` ou `"x"` (croix),
  - `"*"` (étoiles),
  - `"o"` (cercles),
  - `"s"` (carrés);
- `markersize=s` pour contrôler la taille des marqueurs;

- `legend=l` où `l` est une chaîne de caractères pour donner un titre à la courbe. Celui-ci n'apparaitra que si l'on utilise `legend()` par la suite. Mêmes remarques que pour `suptitle`.

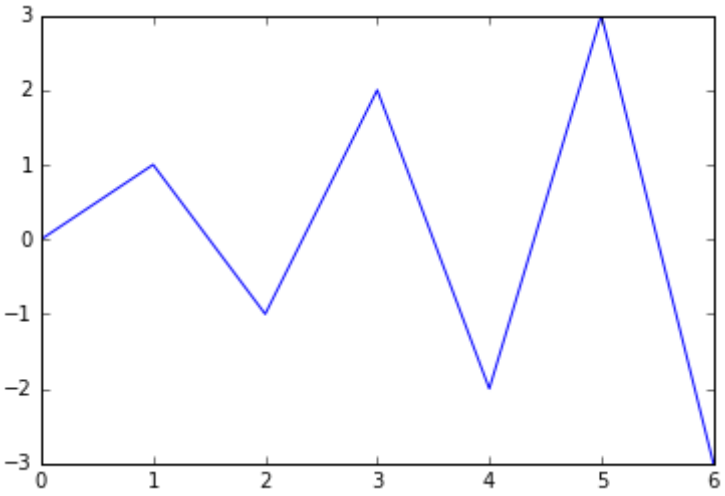
On peut aussi utiliser l'écriture `plot(x, y, "style")` où `style` est donné sous forme abrégée, par exemple:

- `"y-x"` pour des croix reliées par des lignes jaunes;
- `"g:*"` pour des étoiles reliées par des pointillés verts;
- `"bo"` pour des cercles bleus sans lignes;
- `"c-"` pour des lignes cyans sans marqueur.

Une vaste liste de fonctions supplémentaires et d'exemples illustrés peut se trouver à la page http://matplotlib.org/index.html (http://matplotlib.org/index.html)

In [21]:

```
import matplotlib.pyplot as plt

plt.plot([0,1,-1,2,-2,3,-3])
plt.show()    # Il peut être nécessaire d'appeler cette fonction pour faire apparai
```



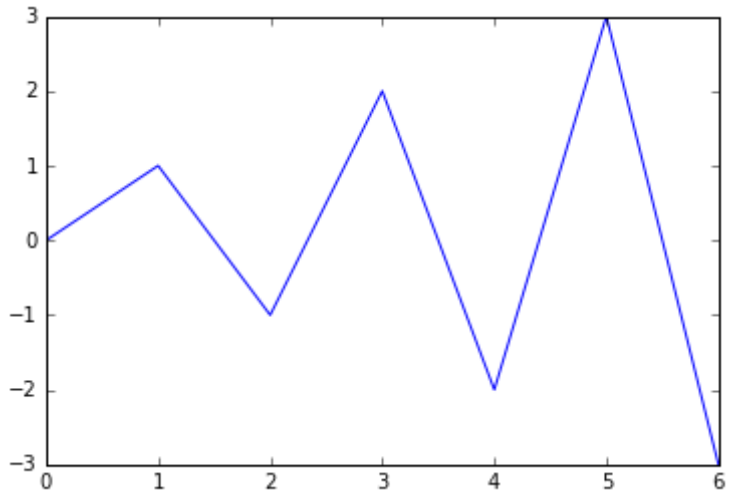
In [22]:

```
%pylab inline
# permet d'importer numpy et matplotlib.pyplot

plot([0,1,-1,2,-2,3,-3])
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[22]:

```
[<matplotlib.lines.Line2D at 0x10dfe3150>]
```

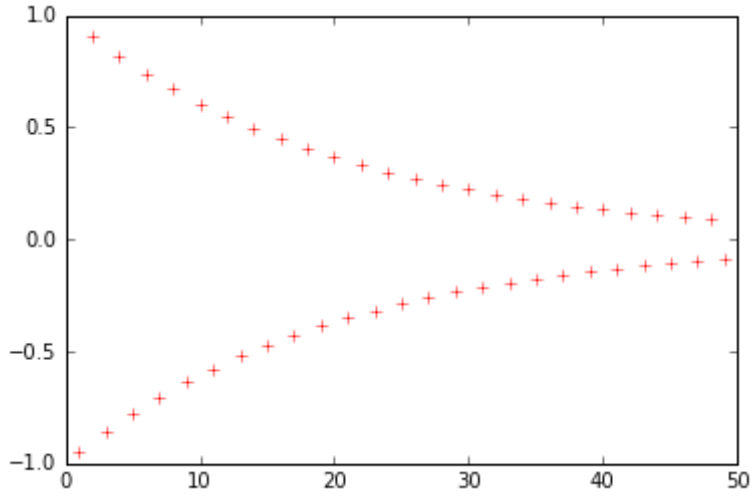


In [23]:

```
# Représentation d'une suite

n = arange(50)
u = (-1) ** n * exp(-n / 20.)
plot(u, "r+")
```

Out[23]:

```
[<matplotlib.lines.Line2D at 0x10df72e50>]
```

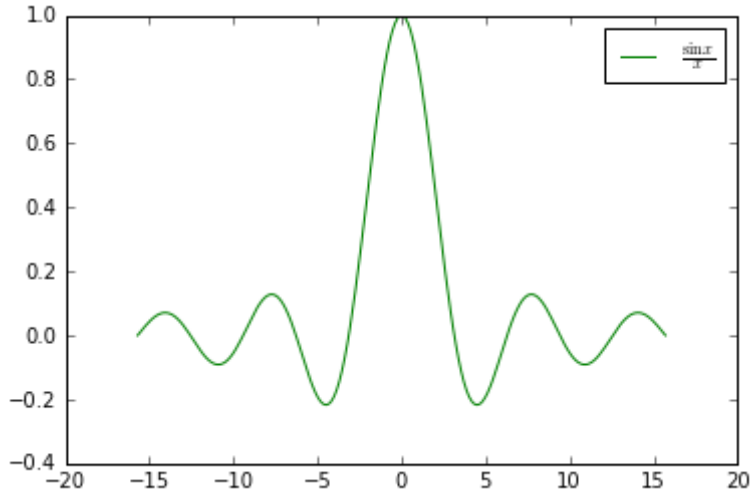


In [24]:

```
# Représentation d'un graphe de fonction

x = linspace(-5 * pi, 5 * pi, 200)
y = sin(x) / x
plot(x, y, "g", label='$\\frac{\\sin x}{x}$')    # Ici il faut échapper le caractèr
legend()
```

Out[24]:

```
<matplotlib.legend.Legend at 0x10dece350>
```

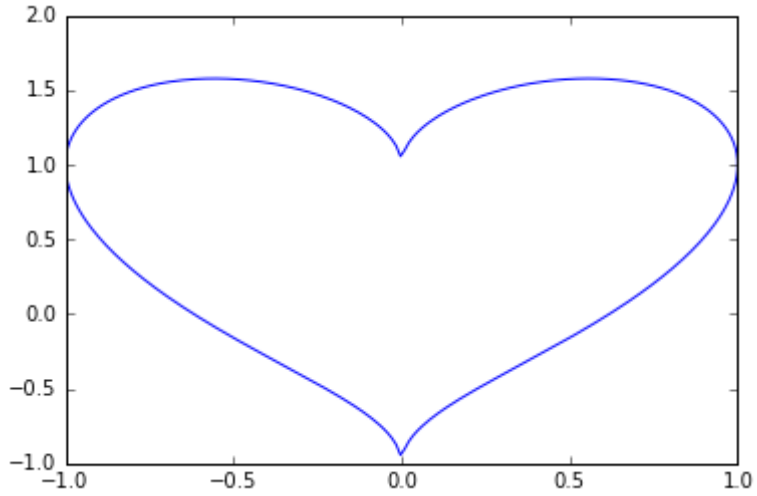


In [25]:

```
# Représentation d'une courbe paramétrée

t = linspace(0, 2 * pi, 500)
x = cos(t)
y = sin(t) + sqrt(abs(cos(t)))
plot(x, y)
```

Out[25]:

```
[<matplotlib.lines.Line2D at 0x10e81ad90>]
```

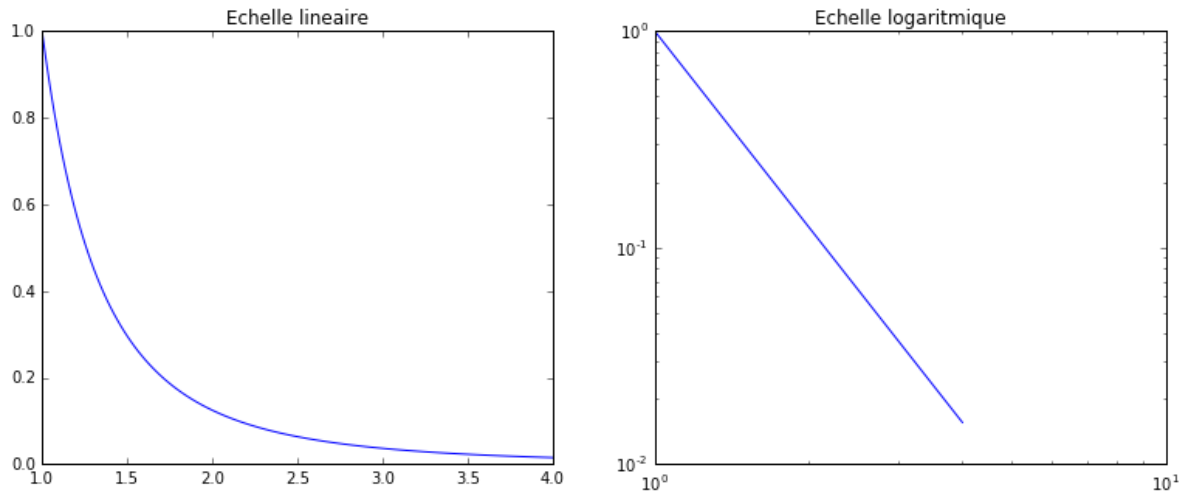


In [27]:

```
# Echelle logarithmique

figure(figsize=(13,5))

t = linspace(0, 2, 100)
x = 2 ** t
y = 1 / x ** 3

subplot(1, 2, 1)
title('Echelle lineaire')    # Pour les chaîne avec caractères accentués, mettre un
plot(x, y)

subplot(1, 2, 2)
title('Echelle logaritmique')
loglog()
plot(x, y)
```

Out[27]:

```
[<matplotlib.lines.Line2D at 0x10eb59a10>]
```



In [ ]: