In [1]:

```
from IPython.core.display import HTML
css_file = './custom.css'
HTML(open(css_file, "r").read())
```

Out[1]:

In [2]:

```
import sys #only needed to determine Python version number
import matplotlib #only needed to determine Matplotlib version

print('Python version ' + sys.version)
print('Matplotlib version ' + matplotlib.__version__ )

%reset -f
```

```
Python version 3.6.2 |Anaconda custom (64-bit)|
(default, Jul 20 2017, 13:51:32)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
Matplotlib version 2.1.0
```

M62_CM1 Rappels sur le langage Python.

In [3]:

```
# Enable inline plotting
%matplotlib inline
%autosave 300
```

```
Autosaving every 300 seconds
```

Table of Contents

1 Notions de base de Python

1.1 Indentation

Le corps d'un bloc de code (boucles, sous-routines, etc.) est défini par son indentation: l'indentation est une partie intégrante de la syntaxe de Python.

1.2 Commentaires

Le symbole dièse `#` indique le début d'un commentaire: tous les caractères entre `#` et la fin de la ligne sont ignorés par l'interpréteur.

1.3 Variables et affectation

Dans la plupart des langages informatiques, le nom d'une variable représente une valeur d'un type donné stockée dans un emplacement de mémoire fixe. La valeur peut être modifiée, mais pas le type. Ce n'est pas le cas en Python, où les variables sont typées dynamiquement.

In [4]:

```
b = 2 # b is an integer
print(b)
```

```
2
```

In [5]:

```
b = b*2.0 # b is a float
print(b)
```

```
4.0
```

L'affectation `b = 2` crée une association entre le nom b et le nombre entier 2. La déclaration `b*2.0` évalue l'expression et associe le résultat à b; l'association d'origine avec l'entier 2 est détruite. Maintenant b se réfère à la valeur en virgule flottante 4.0. Il faut bien prendre garde au fait que l'instruction d'affectation (=) n'a pas la même signification que le symbole d'égalité (=) en mathématiques (ceci explique pourquoi l'affectation de 3 à x, qu'en Python s'écrit `x = 3` , en algorithmique se note souvent $x \leftarrow 3$).

On peut aussi effectuer des affectations parallèles:

In [6]:

```
a, b = 128, 256
print(a)
print(b)

a,b = b,a
print(a)
print(b)
```

```
128
256
256
128
```

Attention: Python est sensible à la casse. Ainsi, les noms `n` et `N` représentent différents objets. Les noms de variables peuvent être non seulement des lettres, mais aussi des mots; ils peuvent contenir des chiffres (à condition toutefois de ne pas commencer par un chiffre), ainsi que certains caractères spéciaux comme le tiret bas `_` (appelé *underscore* en anglais).

1.4 Chaîne de caractères (Strings)

Une *chaîne de caractères* est une séquence de caractères entre guillemets (simples ou doubles). Les chaînes de caractères sont concaténées avec l'opérateur plus `+` , tandis que l'opérateur `:` est utilisé pour extraire une portion de la chaîne. Voici un exemple:

In [7]:

```
string1 = 'Press return to exit'
string2 = 'the program'
print(string1 + ' ' + string2) # Concatenation
print(string1[0:12])# Slicing
```

```
Press return to exit the program
Press return
```

Une chaîne de caractères est un objet **immuable**, i.e. ses caractères ne peuvent pas être modifiés par une affectation, et sa longueur est fixe. Si on essaye de modifier un caractère d'une chaîne de caractères, Python renvoie une erreur comme dans l'exemple suivant:

In [8]:

```
s = 'Press return to exit'
#s[0] = 'p' # Décommenter pour voir l'exception
```

1.5 Listes

Une liste est une suite d'objets, rangés dans un certain ordre. Chaque objet est séparé par une virgule et la suite est encadrée par des crochets. Une liste n'est pas forcement homogène: elle peut contenir des objets de types différents les uns des autres. La première manipulation que l'on a besoin d'effectuer sur une liste, c'est d'en extraire et/ou modifier un élément: la syntaxe est `ListName[index]` . Voici un exemple:

In [9]:

```
fraise = [12, 10, 18, 7, 15, 3] # Create a list
print(fraise)
fraise[2]
fraise[1] = 11
print(fraise)
```

```
[12, 10, 18, 7, 15, 3]
[12, 11, 18, 7, 15, 3]
```

En Python, les éléments d'une liste sont **indexés à partir de** 0.

In [10]:

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

Out[10]:

```
(12, 11, 18, 7, 15, 3)
```

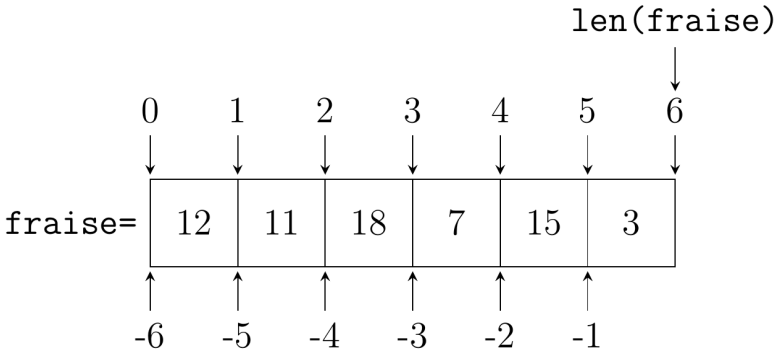
Si on tente d'extraire un élément avec un index dépassant la taille de la liste, Python renvoi un message d'erreur:

In [11]:

```
#fraise[6] # Décommenter pour voir l'exception
```

On peut extraire une sous-liste en déclarant l'indice de début (inclus) et l'indice de fin (exclu), séparés par deux-points: `ListName[i:j]` , ou encore une sous-liste en déclarant l'indice de début (inclus), l'indice de fin (exclu) et le pas, séparés par des deux-points: `ListName[i:j:k]` . Cette opération est connue sous le nom de *slicing* (en anglais).

Un dessin et quelques exemples permettrons de bien comprendre cette opération fort utile:



In [12]:

```
fraise[2:4]
```

Out[12]:

```
[18, 7]
```

In [13]:

```
fraise[2:]
```

Out[13]:

```
[18, 7, 15, 3]
```

In [14]:

```
fraise[:2]
```

Out[14]:

```
[12, 11]
```

In [15]:

```
fraise[:]
```

Out[15]:

```
[12, 11, 18, 7, 15, 3]
```

In [16]:

```
fraise[2:5]
```

Out[16]:

```
[18, 7, 15]
```

In [17]:

```
fraise[2:6]
```

Out[17]:

```
[18, 7, 15, 3]
```

In [18]:

```
fraise[2:7]
```

Out[18]:

```
[18, 7, 15, 3]
```

In [19]:

```
fraise[2:6:2]
```

Out[19]:

```
[18, 15]
```

In [20]:

```
fraise[-2:-4]
```

Out[20]:

```
[]
```

In [21]:

```
fraise[-4:-2]
```

Out[21]:

```
[18, 7]
```

In [22]:

```
fraise[-1]
```

Out[22]:

```
3
```

À noter que lorsqu'on utilise des tranches, les dépassements d'indices sont licites.

Voici quelques opérations et méthodes très courantes associées aux listes:

- `a.append(x)` ajoute l'élément `x` en fin de la liste `a`
- `a.extend(L)` ajoute les éléments de la liste `L` en fin de la liste `a`, équivaut à `a+L`
- `a.insert(i,x)` ajoute l'élément `x` en position `i` de la liste `a`, équivaut à `a[i:i]=x`
- `a.remove(x)` supprime la première occurrence de l'élément `x` dans la liste `a`
- `a.pop([i])` supprime l'élément d'indice `i` dans la liste `a` et le renvoi
- `a.index(x)` renvoie l'indice de la première occurrence de l'élément `x` dans la liste `a`

- `a.count(x)` renvoie le nombre d'occurrence de l'élément `x` dans la liste `a`
- `a.sort()` modifie la liste `a` en la triant
- `a.reverse()` modifie la liste `a` en inversant les éléments
- `len(a)` renvoie le nombre d'éléments de la liste `a`
- `x in a` renvoi `True` si la liste `a` contient l'élément `x`, `False` sinon
- `x not` in a renvoi `True` si la liste `a` ne contient pas l'élément `x`, `False` sinon
- `max(a)` renvoi le plus grand élément de la liste `a`
- `min(a)` renvoi le plus petit élément de la liste `a`

In [23]:

```
a = [2, 37, 20, 83, -79, 21] # Create a list
print(a)
```

```
[2, 37, 20, 83, -79, 21]
```

In [24]:

```
a.append(100) # Append 100 to list
print(a)
```

```
[2, 37, 20, 83, -79, 21, 100]
```

In [25]:

```
L = [17, 34, 21]
a.extend(L)
print(a)
```

```
[2, 37, 20, 83, -79, 21, 100, 17, 34, 21]
```

In [26]:

```
a.count(21)
```

Out[26]:

```
2
```

In [27]:

```
a.remove(21)
```

In [28]:

```
print(a)
```

```
[2, 37, 20, 83, -79, 100, 17, 34, 21]
```

In [29]:

```
a.count(21)
```

Out[29]:

```
1
```

In [30]:

```
a.pop(4)
```

Out[30]:

```
-79
```

In [31]:

```
print(a)
```

```
[2, 37, 20, 83, 100, 17, 34, 21]
```

In [32]:

```
a.index(100)
```

Out[32]:

```
4
```

In [33]:

```
a.reverse()
```

In [34]:

```
print(a)
```

```
[21, 34, 17, 100, 83, 20, 37, 2]
```

In [35]:

```
a.sort()
```

In [36]:

```
print(a)
```

```
[2, 17, 20, 21, 34, 37, 83, 100]
```

In [37]:

```
len(a) # Determine length of list
```

Out[37]:

```
8
```

In [38]:

```
a.insert(2,7) # Insert 7 in position 2
```

In [39]:

```
print(a)
```

```
[2, 17, 7, 20, 21, 34, 37, 83, 100]
```

In [40]:

```
a[0] = 21 # Modify selected element
```

In [41]:

```
print(a)
```

```
[21, 17, 7, 20, 21, 34, 37, 83, 100]
```

In [42]:

```
a[2:4] = [-2,-5,-1978] # Modify selected elements
```

In [43]:

```
print(a)
```

```
[21, 17, -2, -5, -1978, 21, 34, 37, 83, 100]
```

ATTENTION: si `a` est une liste, la commande `b=a` ne crée pas un nouvel objet `b` mais simplement une référence (pointeur) vers `a` . Ainsi, tout changement effectué sur `b` sera répercuté sur `a` aussi! Pour créer une copie `c` de la liste `a` qui soit vraiment indépendante on utilisera la commande `deepcopy` du module `copy` comme dans les exemples suivants:

In [44]:

```
import copy
a = [1.0, 2.0, 3.0]
b = a # 'b' is an alias of 'a'
b[0] = 5.0 # Change 'b'
print(a) # The change is reflected in 'a'
print(b)
```

```
[5.0, 2.0, 3.0]
[5.0, 2.0, 3.0]
```

In [45]:

```
a = [1.0, 2.0, 3.0]
c = copy.deepcopy(a) # 'c' is an independent copy of 'a'
c[0] = 5.0 # Change 'c'
print(a) # 'a' is not affected by the change
print(c)
```

```
[1.0, 2.0, 3.0]
[5.0, 2.0, 3.0]
```

Qu'est-ce qui se passe lorsque on copie une liste `a` avec la commande `b=a` ? En effet, une liste fonctionne comme un carnet d'adresses qui contient les emplacements en mémoire des différents éléments de la liste. Lorsque on écrit `b=a` on dit que `b` contient les mêmes adresses que `a` (on dit que les deux listes *pointent* vers le même objet). Ainsi, lorsqu'on modifie la valeur de l'objet, la modification sera visible depuis les deux alias.

1.6 Matrices (sans **NumPy**)

Les matrices peuvent être représentées comme des listes imbriquées: chaque ligne est un élément d'une liste. Par exemple, le code

In [46]:

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

définit `a` comme la matrice 3×3

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}.$$

La commande `len` (comme *length*) renvoie la longueur d'une liste. On obtient donc le nombre de ligne de la matrice avec `len(a)` et son nombre de colonnes avec `len(a[0])` :

In [47]:

```
print(a)
```

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

In [48]:

```
print(a[1]) # Print second row (element 1)
```

```
[4, 5, 6]
```

In [49]:

```
print(a[1][2]) # Print third element of second row
```

```
6
```

In [50]:

```
print(len(a))
```

```
3
```

In [51]:

```
print(len(a[0]))
```

```
3
```

Dans Python les indices commences à zéro, ainsi `a[0]` indique la première ligne, `a[1]` la deuxième etc.

$$\mathbb{A} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots \\ a_{10} & a_{11} & a_{12} & \dots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

1.7 Dictionnaires

Un dictionnaire est une sorte de liste mais au lieu d'utiliser des index, on utilise des clés, c'est à dire des valeurs autres que numériques.

Pour initialiser un dictionnaire, on utile la syntaxe suivante:

In [52]:

```
a={}
```

Pour ajouter des valeurs à un dictionnaire il faut indiquer une clé ainsi qu'une valeur:

In [53]:

```
a["nom"] = "engel"
a["prenom"] = "olivier"
print(a)
```

```
{'nom': 'engel', 'prenom': 'olivier'}
```

La méthode `get` permet de récupérer une valeur dans un dictionnaire et, si la clé est introuvable, de donner une valeur à retourner par défaut:

In [54]:

```
data={}
data = {"name": "Olivier", "age": 30}
print(data.get("name"))
print(data.get("adresse", "Adresse inconnue"))
```

```
Olivier
Adresse inconnue
```

Pour vérifier la présence d'une clé on utilise `in`

In [55]:

```
"nom" in a
```

Out[55]:

```
True
```

In [56]:

```
"age" in a
```

Out[56]:

```
False
```

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes:

In [57]:

```
del a["nom"]
print(a)
```

```
{'prenom': 'olivier'}
```

- Pour récupérer les clés on utilise la méthode `keys`
- Pour récupérer les valeurs on utilise la méthode `values`
- Pour récupérer les clés et les valeurs en même temps, on utilise la méthode `items` qui retourne un tuple.

In [58]:

```
fiche = {"nom":"engel","prenom":"olivier"}

for cle in fiche.keys():
    print(cle)

for valeur in fiche.values():
    print(valeur)

for cle,valeur in fiche.items():
    print(cle, valeur)
```

```
nom
prenom
engel
olivier
nom engel
prenom olivier
```

On peut utiliser des tuples comme clé comme lors de l'utilisation de coordonnées:

In [59]:

```
b = {}
b[(3,2)]=12
b[(4,5)]=13
b
```

Out[59]:

```
{(3, 2): 12, (4, 5): 13}
```

Comme pour les listes, pour créer une copie indépendante utiliser la méthode `copy` :

In [60]:

```
d = {"k1":"olivier", "k2":"engel"}
e = d.copy()
print(d)
print(e)
d["k1"] = "XXX"
print(d)
print(e)
```

```
{'k1': 'olivier', 'k2': 'engel'}
{'k1': 'olivier', 'k2': 'engel'}
{'k1': 'XXX', 'k2': 'engel'}
{'k1': 'olivier', 'k2': 'engel'}
```

1.8 Fonction `range`

La fonction `range` crée un itérateur. Au lieu de créer et garder en mémoire une liste d'entiers, cette fonction génère les entiers au fur et à mesure des besoins:

- `range(n)` renvoi un itérateur parcourant $0, 1, 2, \dots, n-1$;
- `range(n,m)` renvoi un itérateur parcourant $n, n+1, n+2, \dots, m-1$;
- `range(n,m,p)` renvoi un itérateur parcourant $n, n+p, n+2p, \dots, m-1$

In [61]:

```
A = range(0,10)
print(A)
```

```
range(0, 10)
```

Pour les afficher on crée une `list` :

In [62]:

```
A = list(A)
print(A)
```

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

In [63]:

```
print(list(range(0)))
```

```
[]
```

In [64]:

```
print(list(range(1)))
```

```
[0]
```

In [65]:

```
print(list(range(3,7)))
```

```
[3, 4, 5, 6]
```

In [66]:

```
print(list(range(0,20,5)))
```

```
[0, 5, 10, 15]
```

In [67]:

```
print(list(range(0,20,-5)))
```

```
[]
```

In [68]:

```
print(list(range(0,-20,-5)))
```

```
[0, -5, -10, -15]
```

In [69]:

```
print(list(range(20,0,-5)))
```

```
[20, 15, 10, 5]
```

1.9 Fonction `print`

Pour afficher à l'écran des objets on utilise la fonction `print(object1, object2, ...)` qui convertis `object1, object2` en chaînes de caractères et les affiche sur la même ligne séparés par des espace.

In [70]:

```
a = 12345,6789
b = [2, 4, 6, 8]
print(a,b)
```

```
(12345, 6789) [2, 4, 6, 8]
```

Le retour à la ligne peut être forcé par le caractère `\n` , la tabulation par le caractère `\t` .

In [71]:

```
print("a=", a, "\tb=", b)
print("a=", a, "\nb=", b)
```

```
a= (12345, 6789)        b= [2, 4, 6, 8]
a= (12345, 6789)
b= [2, 4, 6, 8]
```

Pour mettre en colonne des nombres on pourra utiliser l'opérateur `%` : la commande `print('%format1, %format2,...' %(n1,n2,...)` affiche les nombres `n1,n2,...` selon les règles `%format1, %format2,...` . Typiquement on utilise

- `wd` pour un entier
- `w.df` pour un nombre en notation *floating point*
- `w.de` pour un nombre en notation scientifique

où `w` est la largeur du champ total et `d` le nombre de chiffres après la virgule.

In [72]:

```
a = 1234.56789
n = 9876
print('%7.2f' %a)
```

```
1234.57
```

In [73]:

```
print('n = %6d' %n)
```

```
n =   9876
```

In [74]:

```
print('n = %06d' %n)
```

```
n = 009876
```

In [75]:

```
print('%12.3f %6d' %(a,n))
```

```
    1234.568   9876
```

In [76]:

```
print('%12.4e %6d' %(a,n))
```

```
  1.2346e+03   9876
```

1.10 Opérations arithmétiques

- `+` Addition
- `-` Soustraction
- `*` Multiplication
- `/` Division
- `**` Exponentiation
- `//` Quotient de la division euclidienne
- `%` Reste de la division euclidienne

In [77]:

```
a = 100
b = 17
c = a-b
a,b,c
```

Out[77]:

```
(100, 17, 83)
```

In [78]:

```
a = 2
c = b+a
a,b,c
```

Out[78]:

```
(2, 17, 19)
```

In [79]:

```
a = 3
b = 4
c = a
a = b
b = c
a, b, c
```

Out[79]:

```
(4, 3, 3)
```

Certains de ces opérations sont aussi définies pour les chaînes de caractères et les listes comme dans l'exemple suivant:

In [80]:

```
s = 'Hello '
t = 'to you'
a = [1, 2, 3]
print(3*s) # Repetition
```

```
Hello Hello Hello
```

In [81]:

```
print(3*a) # Repetition
```

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

In [82]:

```
print(a + [4, 5]) # Append elements
```

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

In [83]:

```
print(s + t) # Concatenation
```

```
Hello to you
```

In [84]:

```
#print(3 + s) # Décommenter pour voir l'exception
```

Il existe aussi les opérateurs augmentés:

On écrit	Équivaut à
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a **= b`	`a = a ** b`
`a %= b`	`a = a % b`

1.11 Opérateurs de comparaison et connecteurs logiques

Les opérateurs de comparaison renvoient `True` si la condition est vérifiée, `False` sinon. Ces opérateurs sont

On écrit	Ça signifie
<	$<$
>	$>$
<=	$\leq$
>=	$\geq$

On écrit	Ça signifie
`==`	$=$
`!=`	$\neq$
`in`	$\in$

Attention: bien distinguer l'instruction d'affectation `=` du symbole de comparaison `==` .

Pour combiner des conditions complexes (par exemple $x > -2$ et $x^2 < 5$), on peut combiner des variables booléennes en utilisant les connecteurs logiques:

On écrit	Ça signifie
`and`	et
`or`	ou
`not`	non

Deux nombres de type différents (entier, à virgule flottante, etc.) sont convertis en un type commun avant de faire la comparaison. Dans tous les autres cas, deux objets de type différents sont considérés non égaux.

In [85]:

```
a = 2 # Integer
b = 1.99 # Floating
c = '2' # String
print('a>b?',a>b)
print('a==c?',a==c)
print('(a>b) and (a==c)?',(a>b) and (a==c))
print('(a>b) or (a==c)?',(a>b) or (a==c))
```

```
a>b? True
a==c? False
(a>b) and (a==c)? False
(a>b) or (a==c)? True
```

2 Fonctions [...]

Supposons de vouloir calculer les images de certains nombres par une fonction polynomiale donnée. Si la fonction en question est un peu longue à saisir, par exemple
$f: x \mapsto 2x^7 - x^6 + 5x^5 - x^4 + 9x^3 + 7x^2 + 8x - 1$ il est rapidement fastidieux de la saisir à chaque fois que l'on souhaite calculer l'image d'un nombre par cette fonction.

3 Structure conditionnelle [...]

Supposons vouloir définir la fonction valeur absolue:

$$|x| = \begin{cases} x & \text{si } x \geq 0, \\ -x & \text{sinon.} \end{cases}$$

On a besoin d'une instruction qui opère une disjonction de cas. En Python il s'agit de l'instruction de choix introduite par le mot-clé `if` . La syntaxe est la suivante:

```
if condition_1:
    instruction_1.1
    instruction_1.2
elif condition_2:
    instruction_2.1
    instruction_2.2
...
else:
    instruction_n.1
    instruction_n.2
```

où `condition_1` , `condition_2` ... représentent des ensembles d'instructions dont la valeur est `True` ou `False` (on les obtient en général en utilisant les opérateurs de comparaison). La première condition `condition_i` ayant la valeur `True` entraîne l'exécution des instructions `instruction_i.1` et `instruction_i.2` . Si toutes les conditions sont fausses, les instructions `instruction_n.1` et `instruction_n.2` sont exécutées.

Bien noter le rôle essentiel de l'**indentation** qui permet de délimiter chaque bloc d'instructions et la présence des **deux points** après la condition du choix (mot clé `if`) et après le mot clé `else` .

4 Boucles [...]

Les structure de répétition se classent en deux catégories: les *répétitions conditionnelles* pour lesquelles le bloc d'instructions est à répéter autant de fois qu'une condition est vérifiée, et les *répétitions inconditionnelles* pour lesquelles le bloc d'instructions est à répéter un nombre donné de fois.

5 Modules

Un module est une collection de fonctions. Il y a différents types de modules: ceux qui sont inclus dans la version de Python comme `random` ou `math` , ceux que l'on peut rajouter comme `numpy` ou `matplotlib` et ceux que l'on peut faire soi-même (il s'agit dans les cas simples d'un fichier Python contenant un ensemble de fonctions).

Pour importer un module, on peut utiliser la commande `import ModuleName` . Il est alors possible d'obtenir une aide sur le module avec la commande `help(ModuleName)` . La liste des fonctions définies dans un module peut être affichée par la commande `dir(ModuleName)` . Les fonctions s'utilisent sous la forme `ModuleName.FunctionName(parameters)` .

Il est également possible d'importer le contenu du module sous la forme `from ModuleName import *` et alors les fonctions peuvent être utilisées directement par `FunctionName(parameters)` .

Python offre par défaut une bibliothèque de plus de deux cents modules qui évite d'avoir à réinventer la roue dès que l'on souhaite écrire un programme. Ces modules couvrent des domaines très divers: mathématiques (fonctions mathématiques usuelles, calculs sur les réels, sur les complexes, combinatoire\dots), administration système, programmation réseau, manipulation de fichiers, etc. Ici on en présente seulement quelques-uns, à savoir ce dont on se servira dans les projets.

5.1 Le module `math`

Dans Python seulement quelque fonction mathématique est prédéfinie:

- `abs(a)` Valeur absolue de a
- `max(suite)` Plus grande valeur de la suite
- `min(suite)` Plus petite valeur de la suite
- `round(a,n)` Arrondi a à n décimales près
- `pow(a,n)` Exponentiation, renvoi a^n
- `sum(L)` Somme des éléments de la suite
- `divmod(a,b)` Renvoie quotient et reste de la division de a par b
- `cmp(a,b)` Renvoie $\begin{cases} -1 & \text{si } a < b, \\ 0 & \text{si } a = b, \\ 1 & \text{si } a > b. \end{cases}$

  Toutes les autres fonctions mathématiques sont définies dans le module `math` . Comme mentionné précédemment, on dispose de plusieurs syntaxes pour importer un module:

In [125]:

```
import math
print(math.pi)
print(math.sin(math.pi))
print(math.log(1.0))
```

```
3.141592653589793
1.2246467991473532e-16
0.0
```

In [126]:

```
from math import *
print(pi)
print(sin(pi))
print(log(1.0))
```

```
3.141592653589793
1.2246467991473532e-16
0.0
```

Voici la liste des fonctions définies dans le module `math` :

In [127]:

```
import math
dir(math)
```

Out[127]:

```
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
```

```
 'tau',
 'trunc']
```

Notons que le module définit les deux constantes π (`pi`) et e (`e`).

5.2 Le module `random`

Ce module propose diverses fonctions permettant de générer des nombres (pseudo-)aléatoires qui suivent différentes distributions mathématiques. Il apparait assez difficile d'écrire un algorithme qui soit réellement non-déterministe (c'est-à-dire qui produise un résultat totalement imprévisible). Il existe cependant des techniques mathématiques permettant de simuler plus ou moins bien l'effet du hasard. Voici quelques fonctions fournies par ce module:

- `random.randrange(p,n,h)` choisit un éléments aléatoirement dans la liste `range(p,n,h)`
- `random.randint(a,b)` choisit un *entier* aléatoirement dans l'intervalle $[a;b]$
- `random.choice(seq)` choisit un éléments aléatoirement dans la liste `seq`
- `random.random()` renvoie un *décimal* aléatoire dans $[0;1[$
- `random.uniform(a,b)` choisit un *décimal* aléatoire dans $[a;b]$

In [128]:

```
import random
random.randrange(50,100,5)
```

Out[128]:

75

In [129]:

```
random.randint(50,100)
```

Out[129]:

68

In [130]:

```
random.choice([1,7,10,11,12,25])
```

Out[130]:

7

In [131]:

```
random.random()
```

Out[131]:

```
0.8731470716087093
```

In [132]:

```
random.uniform(10,20)
```

Out[132]:

```
18.420470773729523
```

6 Dessiner des courbes avec le module `matplotlib`

Le tracé de courbes scientifiques peut se faire à l'aide du module `matplotlib` .
`Matplotlib` est un package complet; `pylab` et `pyplot` sont des modules de `matplotlib` qui sont installés avec `matplotlib` .
`Pylab` associe les fonctions de `pyplot` (pour les tracés) avec les fonctionnalité du module `numpy` pour obtenir un environnement très proche de celui de MATLAB.

On peut écrire et exécuter des script Python (qui utilisent Matplotlib) en ligne à l'adresse https://trinket.io/python/a8645625fd (https://trinket.io/python/a8645625fd)
Pour l'utiliser, on peut importer le module `pylab` .

La référence complète de matplotlib est lisible à l'adresse: http://matplotlib.sourceforge.net/matplotlib.pylab.html (http://matplotlib.sourceforge.net/matplotlib.pylab.html) Il est en particulier recommandé de regarder les *screenshots* (captures d'écrans), qui sont donnés avec le code utilisé pour les générer.

Vous pouvez consulter aussi

- http://apprendre-python.com/page-creer-graphiques-scientifiques-python-apprendre (http://apprendre-python.com/page-creer-graphiques-scientifiques-python-apprendre)
- https://www.courspython.com/introduction-courbes.html (https://www.courspython.com/introduction-courbes.html)

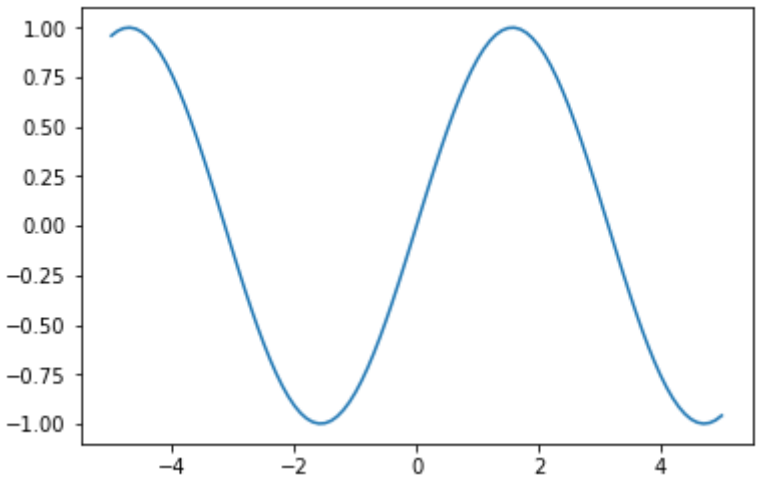
6.1 1D

Pour tracer le graphe d'une fonction $f\colon [a,b] \to \mathbb{R}$, Python a besoin d'une grille de points x_i où évaluer la fonction, ensuite il relie entre eux les points $(x_i, f(x_i))$ par des segments. Plus les points sont nombreux, plus le graphe est proche du graphe de la fonction f. Pour générer les points x_i on peut utiliser

- l'instruction `linspace(a,b,n+1)` qui construit la liste de $n+1$ éléments
$$[a, a+h, a+2h, \dots, b = a+nh] \quad \text{avec } h = \frac{b-a}{n}$$
- ou l'instruction `arange(a,b,h)` qui construit la liste de $n = E(\frac{b-a}{h}) + 1$ éléments
$$[a, a+h, a+2h, \dots, a+nh]$$
Voici un exemple avec une sinusoïde:

In [133]:

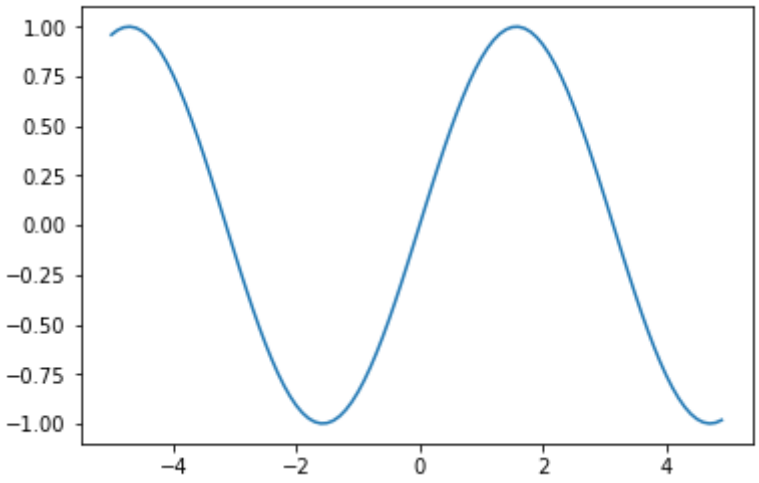
```
%matplotlib inline
from matplotlib.pylab import *
x = linspace(-5,5,101) # x = [-5,-4.9,-4.8,...,5] with 101 ele
y = sin(x) # operation is broadcasted to all elements of the a
plot(x,y);
#show()
```



ou encore

In [134]:

```
from matplotlib.pylab import *
x = arange(-5,5,0.1) # x = [-5,-4.9,-4.8,...,5] with 101 eleme
y = sin(x) # operation is broadcasted to all elements of the a
plot(x,y);
# show()
```



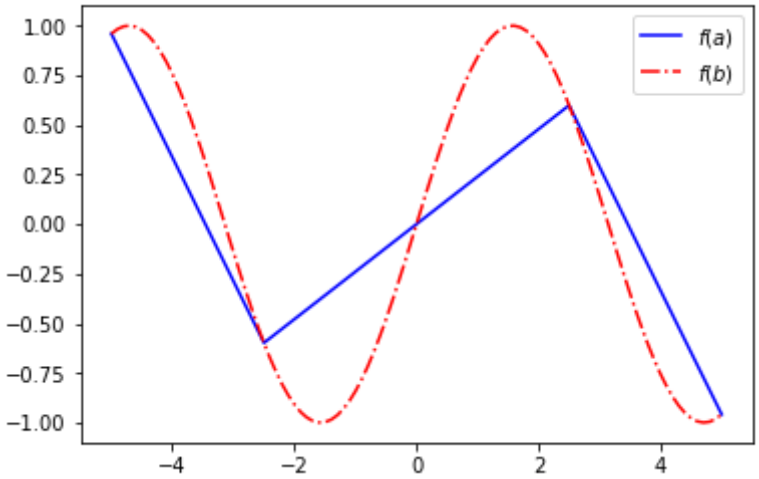
On obtient une courbe sur laquelle on peut zoomer, modifier les marge et sauvegarder dans différents formats.

6.1.1 Plusieurs courbes sur le même repère

On peut même tracer plusieurs courbes sur la même figure. Par exemple, dans la figure suivante, on a tracé la même fonction: la courbe bleu correspond à la grille la plus grossière, la courbe rouge correspond à la grille la plus fine:

In [135]:

```
from matplotlib.pylab import *
a = linspace(-5,5,5)
fa = sin(a) # matplotlib importe numpy qui redefinie les fonct
plot(a,fa,'b-',label="$f(a)$");
b = linspace(-5,5,101)
fb = sin(b)
plot(b,fb,'r-.',label="$f(b)$" );
legend();
```

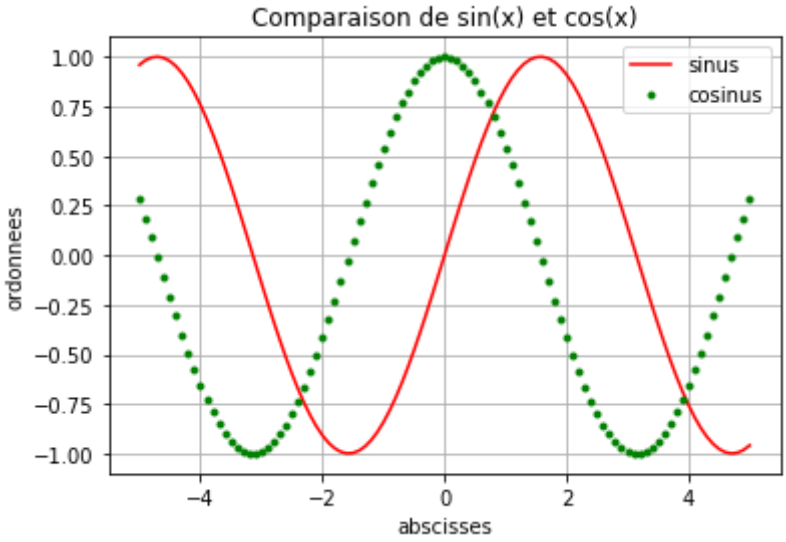


Pour **tracer plusieurs courbes sur le même repère**, on peut les mettre les unes à la suite des autres en spécifiant la couleur et le type de trait, changer les étiquettes des axes, donner un titre, ajouter une grille, une légende etc.

Par exemple, dans le code ci-dessous `"r-"` indique que la première courbe est à tracer en rouge (red) avec un trait continu, et `"g."` que la deuxième est à tracer en vert (green) avec des points.

In [136]:

```
from matplotlib.pylab import *
x = linspace(-5,5,101)
y1 = sin(x)
y2 = cos(x)
plot(x,y1,"r-",x,y2,"g.")
legend(['sinus','cosinus'])
xlabel('abscisses')
ylabel('ordonnees')
title('Comparaison de sin(x) et cos(x)')
grid(True)
```



Quelques options de `pylab` :

	Line style		Color		Symbols
-	solid line	b	blue	v	triangle down symbols
--	dashed line	r	red	>	triangle right symbols
:	dotted line	m	magenta	+	plus symbols
-.	dash-dot line	k	black	D	diamond symbols
.	points	g	green	,	pixels
o	filled circles	c	cyan	<	triangle left symbols
^	filled triangles up	y	yellow	s	square symbols
		w	white	x	cross symbols
				*	star symbols

On peut déplacer la légende en spécifiant l'une des valeurs suivantes:

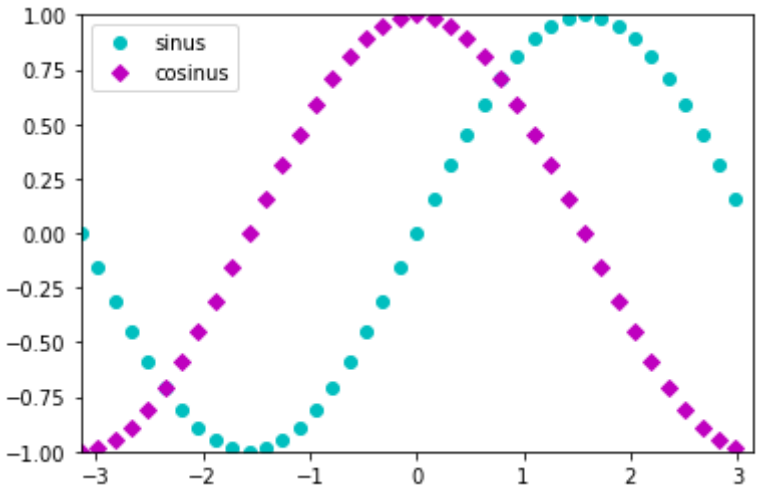
`best`, `upper right`, `upper left`, `lower right`, `lower left`, `center right`, `center left`, `lower center`, `upper center`, `center` :

In [137]:

```
from matplotlib.pylab import *
x = arange(-pi,pi,0.05*pi)
plot(x,sin(x),'co',x,cos(x),'mD')
legend(['sinus','cosinus'],loc='upper left')
axis([-pi, pi, -1, 1]) # axis([xmin, xmax, ymin, ymax])
```

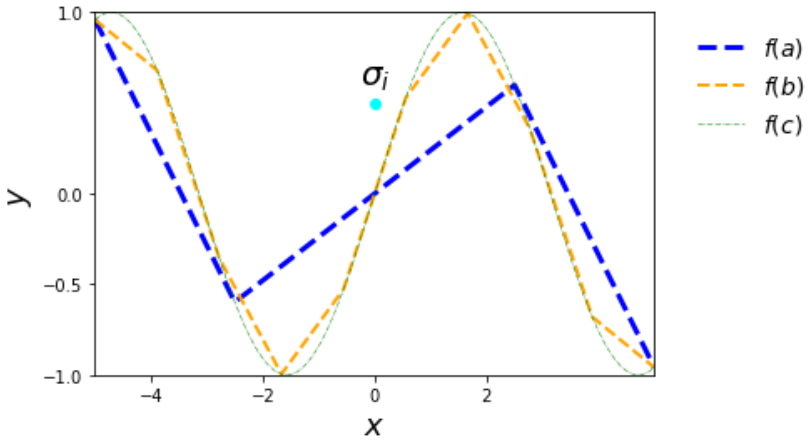
Out[137]:

```
[-3.141592653589793, 3.141592653589793, -1, 1]
```



In [138]:

```
from matplotlib.pylab import *

a = linspace(-5,5,5)
fa = sin(a)
plot(a,fa,ls='--', lw=3, color="blue", label=r"$f(a)$")
b = linspace(-5,5,10)
fb = sin(b)
plot(b,fb,ls='--', lw=2.0, color="orange", label=r"$f(b)$")
c = linspace(-5,5,101)
fc = sin(c)
plot(c,fc,ls='-.', lw=0.5, color="green", label=r"$f(c)$")

plot([0], [0.5], lw=0.5, marker='o', color="cyan")
text(0, 0.6,r"$\sigma_i$", horizontalalignment='center', fonts

xlim(-5, 5)
ylim(-1, 1)
yticks([-1, -0.5, 0, 1])
xticks([-4, -2, 0, 2])

xlabel("$x$", fontsize=18)
ylabel("$y$", fontsize=18)

legend(bbox_to_anchor=(1.04,1),loc='upper left', ncol=1, fonts
```



6.1.2 Plusieurs "fenêtres" graphiques

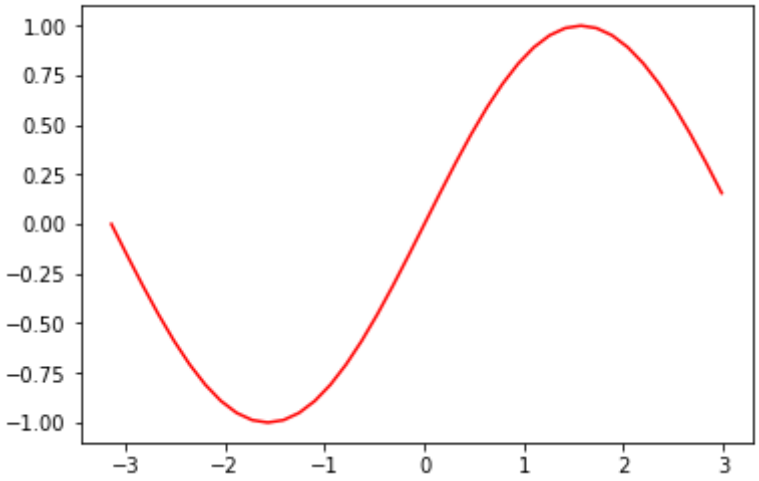
On génère deux fenêtres contenant chacune un graphe:

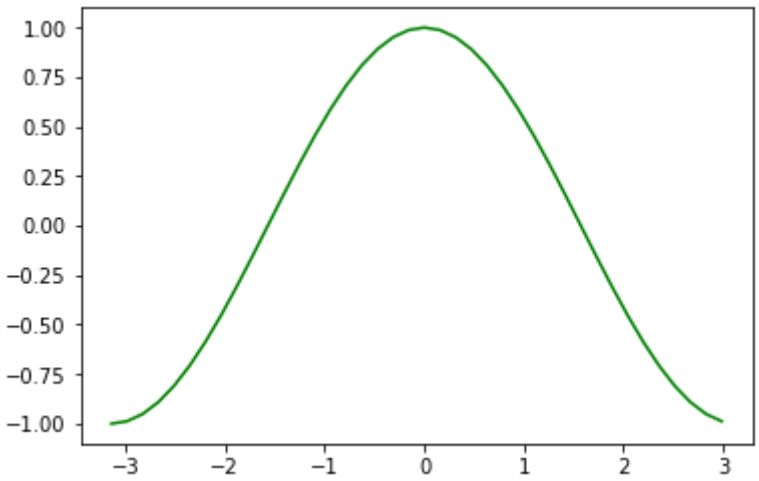
In [139]:

```
from matplotlib.pylab import *
x = arange(-pi,pi,0.05*pi)
figure(1)
plot(x, sin(x), 'r')
figure(2)
plot(x, cos(x), 'g')
```

Out[139]:

```
[<matplotlib.lines.Line2D at 0x7fa60844f240>]
```





6.1.3 Plusieurs repères dans la même fenêtre

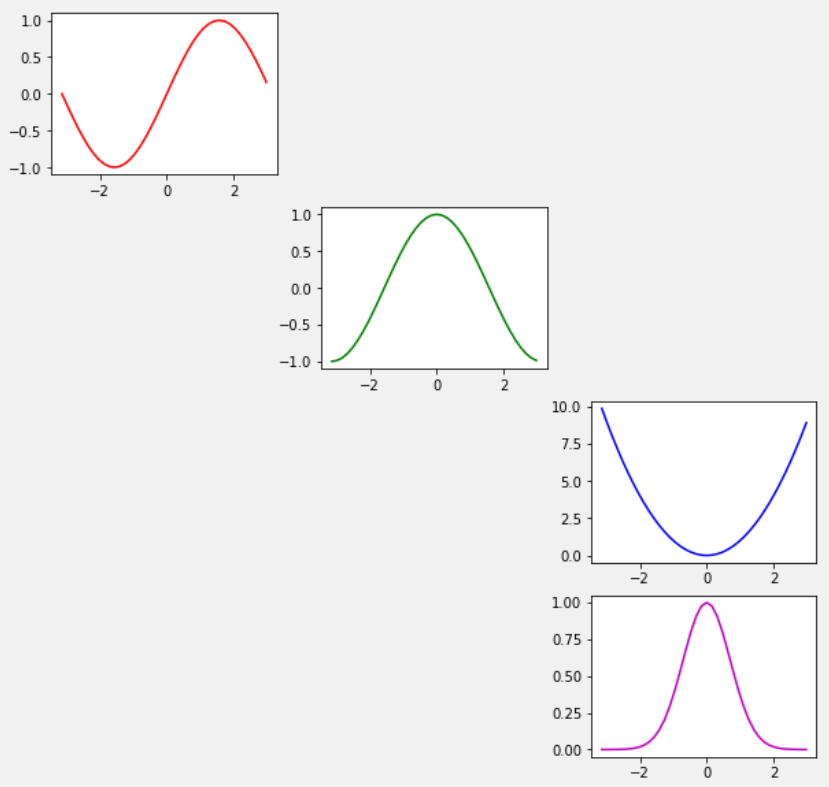
La fonction `subplot(x,y,z)` subdivise la figure sous forme d'une matrice `(x,y)` et chaque case est numérotée, `z` étant le numéro de la case où afficher le graphe. La numérotation se fait de gauche à droite, puis de haut en bas, en commençant par 1.

In [140]:

```
from matplotlib.pylab import *
x = arange(-pi,pi,0.05*pi)

figure(figsize=(10, 10), facecolor="#f1f1f1")
# axes coordinates as fractions of the canvas width and height
#left, bottom, width, height = 0.1, 0.1, 0.8, 0.8
#axes((left, bottom, width, height), facecolor="#e1e1e1")

subplot(4,3,1)
plot(x, sin(x), 'r')
subplot(4,3,5)
plot(x, cos(x), 'g')
subplot(4,3,9)
plot(x, x*x, 'b')
subplot(4,3,12)
plot(x, exp(-x*x), 'm')
```

Out[140]:

```
[<matplotlib.lines.Line2D at 0x7fa608367588>]
```

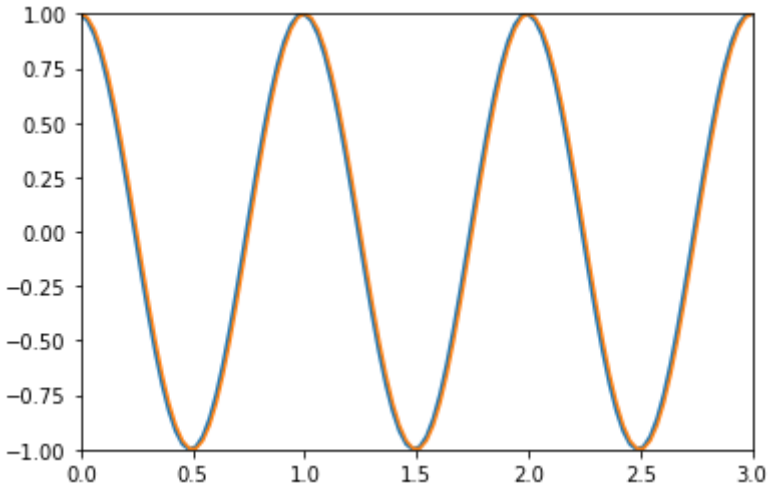


6.1.4 Animations

Pour **afficher une animation** on utilisera le module `animation` de `matplotlib` . Voici un exemple avec deux méthodes différentes : la première utilise la commande magique `%matplotlib inline` , la deuxième la commande magique `%matplotlib notebook` qui permet d'intéragir avec l'animation.

In [141]:

```
%reset -f
%matplotlib inline

from matplotlib.pylab import *
import matplotlib.animation as animation
from IPython.display import display, clear_output

fig = figure()
axis([0,3,-1,1])
line, = plot([],[],lw=2)

x = linspace(0,3,100)
animate = lambda i: line.set_data(x, cos(2*pi*(x-0.01*i)))

# CI
plot(x,cos(2*pi*x))

# Marche en temps
for i in range(len(x)):
    animate(i)
    clear_output(wait=True)
    display(fig)
clear_output(wait=True)
```



In [142]:

```
%reset -f
%matplotlib notebook
%matplotlib notebook
from matplotlib.pylab import *
from matplotlib import animation

x = linspace(0,3,100)

# First set up the figure, the axis, and the plot element we w
fig = figure() # initialise la figure
line, = plot([],[],lw=2)
axis([0,3,-1,1])

# Define the initialization function, which plots the backgrou
# init = lambda : plot([],[])
init = lambda : plot(x,cos(2*pi*x))

# Define the animation function, which is called for each new
animate = lambda i: line.set_data(x,cos(2*pi*(x-0.01*i)))

# call the animator. blit=True means only re-draw the parts th
ani = animation.FuncAnimation(fig, animate, init_func=init, fr

# Eventually
# from IPython.display import HTML, Image
# ani.save('animation.gif', writer='imagemagick', fps=60)
# Image(url='animation.gif')
```

```
Traceback (most recent call last):
  File "/home/minnolina/anaconda3/lib/python3.6/s
ite-packages/matplotlib/cbook/__init__.py", line
389, in process
    proxy(*args, **kwargs)
  File "/home/minnolina/anaconda3/lib/python3.6/s
ite-packages/matplotlib/cbook/__init__.py", line
227, in __call__
    return mtd(*args, **kwargs)
  File "/home/minnolina/anaconda3/lib/python3.6/s
ite-packages/matplotlib/animation.py", line 1560,
in _stop
    self.event_source.remove_callback(self._loop_
delay)
AttributeError: 'NoneType' object has no attribut
e 'remove_callback'
```

6.2 2D

La représentation graphique de l'évolution d'une fonction f de deux variables x et y n'est pas une tâche facile, surtout si le graphique en question est destiné à être imprimé. Dans ce type de cas, un graphe faisant apparaître les lignes de niveaux de la fonction en question peut être une solution intéressante et lisible. Commençons donc par considérer l'exemple simple d'une fonction de la forme:

$$f(x, y) = \frac{\sin(x^2 + y^2)}{x^2 + y^2}$$

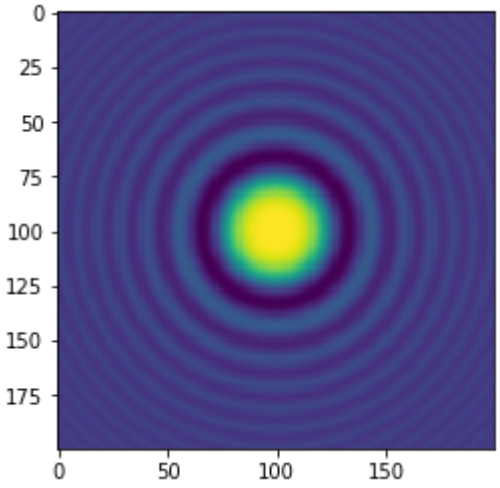
Le tracé de cette fonction en courbes de niveaux va nécessiter la création d'un maillage bidimensionnel permettant de stocker l'intervalle de chacune des variables. La fonction destinée à cela s'appelle `meshgrid` (incluse dans le module `NumPy`). On construit donc le maillage en question sur le rectangle $[-5; 5] \times [-5; 5]$. La fonction `meshgrid` fait appel dans ce cas à deux fonctions `linspace` pour chacune des variables. `z` est ici un objet `array` qui contient les valeurs de la fonction f sur chaque nœud du maillage.

In [143]:

```
%matplotlib inline
from matplotlib.pylab import *
x=linspace(-2*pi,2*pi,200)
y=linspace(-2*pi,2*pi,200)
xx,yy = meshgrid(x,y)
z = sin(xx**2 + yy**2) / (xx**2 + yy**2)
```

In [144]:

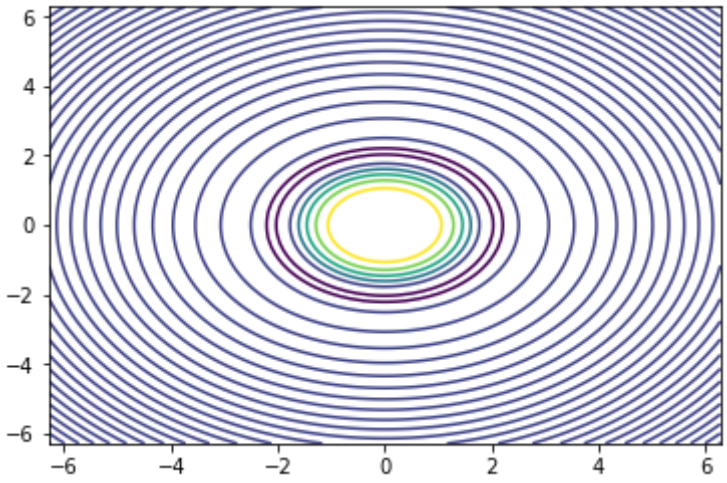
```
imshow(z);
```



Pour tracer l'évolution de f en lignes de niveaux on utilisera les fonctions `contour` et `contourf` avec comme arguments les variables `x` et `y`, les valeurs de `z` correspondantes (et eventuellement le nombre de lignes de niveaux choisit).

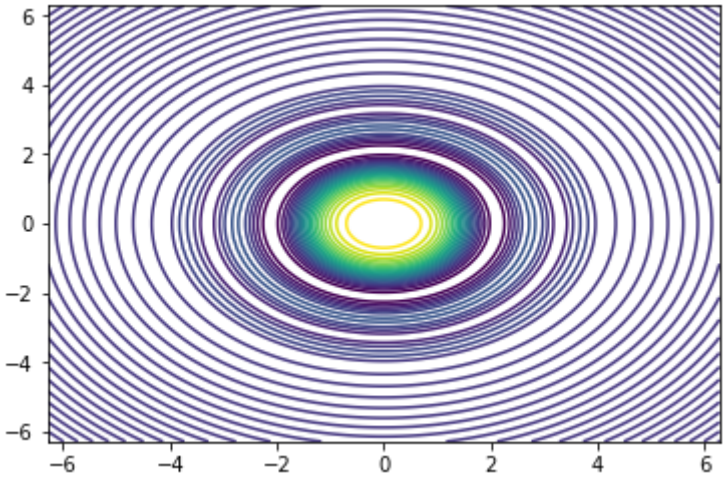
In [145]:

```
h = contour(x,y,z)
```



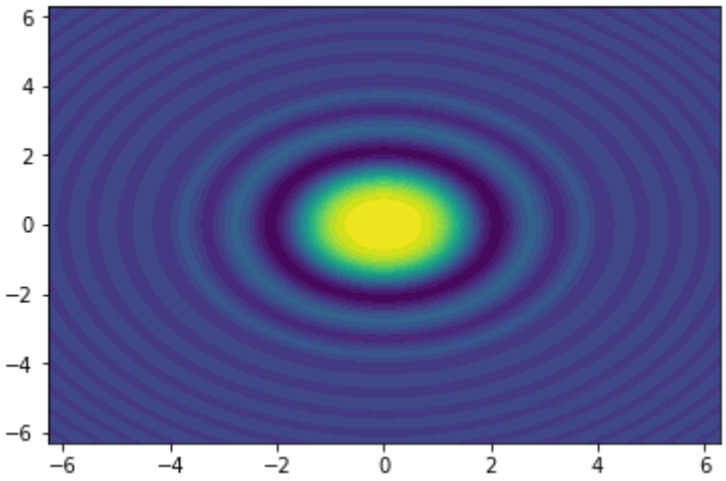
In [146]:

```
h = contour(x,y,z,20)
```



In [147]:

```
h = contourf(x,y,z,20)
```



Par défaut, ces courbes de niveaux sont colorées en fonction de leur *altitude* z correspondante.

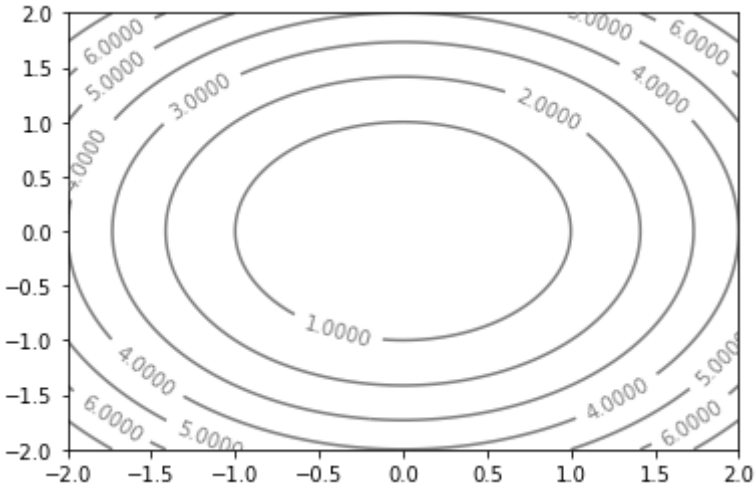
S'il n'est pas possible d'utiliser de la couleur, on peut imposer une couleur uniforme pour le graphe et utiliser des labels sur les lignes de niveaux afin de repérer leurs altitudes. La fonction en question s'appelle `clabel` et prend comme principal argument la variable graphe précédente. La première option `inline=1` impose d'écrire les valeurs sur les lignes de niveaux, les deux options suivantes gérant la taille de la police utilisée et le format d'écriture des valeurs.

In [148]:

```
x=linspace(-2,2,101)
y=linspace(-2,2,101)
xx,yy = meshgrid(x,y)
z = xx**2 + yy**2
graphe4 = contour(x,y,z,8,colors='grey')
clabel(graphe4,inline=1,fontsize=10,fmt='%1.4f')
```

Out[148]:

```
<a list of 15 text.Text objects>
```



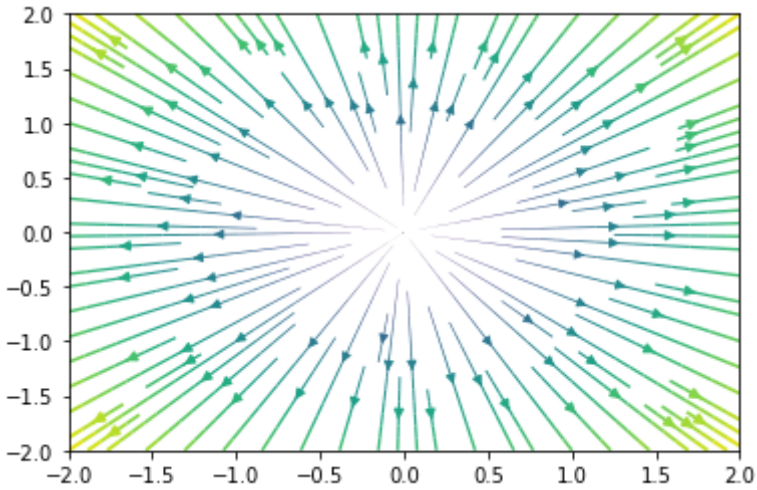
Pour visualiser un champ de vecteur on pourra utiliser `streamplot` :

In [149]:

```
x=linspace(-2,2,101)
y=linspace(-2,2,101)
xx,yy = meshgrid(x,y)
z = xx**2 + yy**2
dx = 2*xx
dy = 2*yy
speed = sqrt(dx*dx+dy*dy)
streamplot(xx, yy, dx, dy, density=1.2, color=speed, linewidth
```

Out[149]:

```
<matplotlib.streamplot.StreamplotSet at 0x7fa6086
1e908>
```



6.3 Représentations des erreurs: échelles logarithmique et semi-logarithmique

Quand on étudie les propriétés de convergence d'une méthode numérique, on trace souvent des graphes représentant

- l'erreur E en fonction de h, le pas de discrétisation (par exemple pour une formule de quadrature ou le calcul approchée de la solution d'une EDO);
- l'erreur E en fonction de k, le pas d'itération (par exemple pour les méthodes de recherche des zéros d'une fonction).

Pour ces graphes on a recours à des représentations en échelle logarithmique ou semi-logarithmique.

- Utiliser une **échelle logarithmique** signifie représenter $\log_{10}(h)$ sur l'axe des abscisses et $\log_{10}(E)$ sur l'axe des ordonnées. Le but de cette représentation est clair: si $E = Ch^p$ alors $\log_{10}(E) = \log_{10}(C) + p\log_{10}(h)$. En échelle logarithmique, p représente donc la pente de la ligne droite $\log_{10}(E)$. Ainsi, quand on veut comparer deux méthodes, celle présentant la pente la plus forte est celle qui a l'ordre le plus élevé (la pente est $p = 1$ pour les

méthodes d'ordre un, $p = 2$ pour les méthodes d'ordre deux, et ainsi de suite).

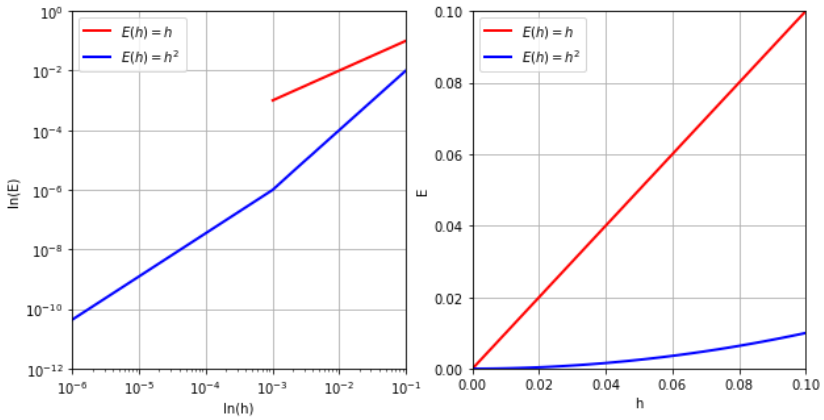
Il est très simple d'obtenir avec Python des graphes en échelle logarithmique: il suffit de taper `loglog` au lieu de `plot` .

Par exemple, ci-dessous à gauche on trace des droites représentant le comportement de l'erreur de deux méthodes différentes. La ligne rouge correspond à une méthode d'ordre un, la ligne bleu à une méthode d'ordre deux. Sur la figure à droite on trace les mêmes données qu'à gauche mais avec la commande `plot` , c'est-à-dire en échelle linéaire pour les axes x et y. Il est évident que la représentation linéaire n'est pas la mieux adaptée à ces données puisque la courbe $E(h) = h^2$ se confond dans ce cas avec l'axe des x quand $x \in [10^{-6}; 10^{-2}]$, bien que l'ordonnée correspondante varie entre $x \in [10^{-12}; 10^{-4}]$, c'est-à-dire sur 8 ordres de grandeur.

En générale, on utilise cette approche pour estimer l'ordre de convergence d'une méthode numérique. Or, dans ces cas, on n'a pas exactement une droite. On peut alors calculer la droite de meilleure approximation au sens des moindres carrés et évaluer la pente de cette droite.

In [150]:

```
import random
x  = linspace(1.e-6, 1.e-1,100, endpoint=True)
y1 = [xi+random.uniform(-1.e-5,1.e-5) for xi in x]
y2 = [(xi+random.uniform(-1.e-5,1.e-5))**2 for xi in x]

figure(1, figsize=(10, 5))

# log x and y axis
subplot(121)
loglog(x,y1, linewidth=2, color='red',label='$E(h)=h$')
loglog(x,y2, linewidth=2,color='blue',label='$E(h)=h^2$')
axis([1.e-6, 1.e-1, 1.e-12, 1.])
xlabel('ln(h)')
ylabel('ln(E)')
legend(loc='best')
grid(True)

# linear x and y axis
subplot(122)
plot(x,y1, linewidth=2, color='red',label='$E(h)=h$')
plot(x,y2, linewidth=2,color='blue',label='$E(h)=h^2$')
axis([1.e-6, 1.e-1, 1.e-12, 0.1])
xlabel('h')
ylabel('E')
legend(loc='best')
grid(True)

show()

print("Pente estimee =",polyfit(log(x),log(y1),1)[0])
print("Pente estimee =",polyfit(log(x),log(y2),1)[0])
```




```
Pente estimee = nan
Pente estimee = 1.7963806583505648

/home/minnolina/anaconda3/lib/python3.6/site-pack
ages/ipykernel_launcher.py:30: RuntimeWarning: in
valid value encountered in log
/home/minnolina/anaconda3/lib/python3.6/site-pack
ages/ipykernel_launcher.py:30: RankWarning: Polyf
it may be poorly conditioned
```

- Plutôt que l'échelle log-log, nous utiliserons parfois une **échelle semi-logarithmique**, c'est-à-dire logarithmique sur l'axe des y et linéaire

sur l'axe des x. Cette représentation est par exemple préférable quand on trace l'erreur E d'une méthode itérative en fonction des itérations k ou plus généralement quand les ordonnées s'étendent sur un intervalle beaucoup plus grand que les abscisses. Si $E(k) = C^{k^n} E(0)$ avec $C \in]0; 1[$ alors $\log_{10}(E(k)) = \log_{10}(E(0)) + k^n \log_{10}(C)$, c'est-à-dire une droite si $n = 1$, une parabole si $n = 2$ etc. La commande Python pour utiliser l'échelle semi-logarithmique est `semilogy` .

Par exemple, ci-dessous on trace à gauche des courbes représentant le comportement de l'erreur de trois méthodes différentes. La ligne rouge correspond à une méthode d'ordre un, la parabole bleu à une méthode d'ordre deux. Sur la figure à droite on trace les mêmes données qu'à gauche mais avec la commande `plot` , c'est-à-dire en échelle linéaire pour les axes x et y. Il est évident que la représentation linéaire n'est pas la mieux adaptée à ces données.

In [151]:

```
x  = linspace(0, 100,100, endpoint=True)
y1 = 10**(-(x))
y2 = 10**(-(x**2))

figure(1, figsize=(10, 5))

# linear x and semilog y axis
subplot(121)
semilogy(x,y1, linewidth=2, color='red',label='$E(k)=10^{-k}$'
semilogy(x,y2, linewidth=2,color='blue',label='$E(k)=10^{-k^2}
axis([0, 50, 1.e-10, 1.])
xlabel('k')
ylabel('log(E)')
legend(loc='best')
grid(True)

# linear x and y axis
subplot(122)
plot(x,y1, linewidth=2, color='red',label='$E(k)=10^{-k}$')
plot(x,y2, linewidth=2,color='blue',label='$E(k)=10^{-k^2}$')
axis([0, 50, 1.e-10, 1.])
xlabel('k')
ylabel('E')
legend(loc='best')
grid(True)
```