Université de Limoges

FACULTÉ
DES SCIENCES
ET TECHNIQUES

Prépa Python

P-F. Bonnefoi

Version du 21 juin 2013

Agenda

Contenu de l'intervention

- ☐ Installation et découverte de l'environnement de travail et d'évaluation proposé ;
- ☐ Différences et les singularités de l'utilisation de Python (représentation des données, mécanisme d'exceptions, orienté objet etc.) ;
- ☐ Notions de base de données et utilisation pratique ;
- ☐ Programmation d'algorithmes pour l'analyse numérique (conception du programme, utilisation de graphe, rédaction d'un rapport final pour l'évaluation) ;
- ☐ Programmation d'algorithmes pour la manipulation de structure de données (manipulation de chaîne de caractères, de tableau, de fonction, d'exploration de graphe, pile/file etc.)

La programmation

- ⋆ 9h Accueil ;
- ⋆ 10h30 Pause café à l'IREM ;
- ⋆ 12h30 Repas au RU ;
- ⋆ 13h45 Reprise ;
- ⋆ 15h30 Pause café à l'IREM ;
- ⋆ 17h00 Fini... *Déjà ?*

Quelques remarques sur Python

Programmation « traditionnelle » :

- ☐ traitement d'entier de taille limitée ;
- ☐ traitement difficile de chaînes de caractères ;
- ☐ utilisation de fonction ;
- ☐ utilisation de bibliothèque ;
- ☐ utilisation de pile pour les appels ;
- ☐ allocation et gestion mémoire ;
- ☐ pointeurs ;
- ☐ typage fort ;
- ☐ compilation ;
- ☐ vitesse d'exécution.

La réponse de Python :

- ☐ gestion simplifiée de chaîne de caractères (y compris avec des caractères internationaux) ;
- ☐ conversion des représentations ;
- ☐ gestion automatique de la mémoire (allocation, réservation, libération) ;
- ☐ utilisation d'un mode dictionnaire et langage interprété ;
- ☐ utilisation des mécanismes d'exceptions ;
- ☐ les modules et les espaces de noms ;
- ☐ programmation « orientée objet » ;
- ☐ notation propre ⇒ conception propre : l'écriture indentée ;
- ☐ vitesse de conception.

Installation de Python, iPython

Sous Windows

1. installer Google Chrome ou Firefox 6 ;
   Il est possible d'installer une version « portable » de Chrome sur une clé USB.
   `http://portableapps.com/apps/internet/google_chrome_portable`

2. installer anaconda :
   Récupérer l'archive sur `http://continuum.io/downloads`
   Sous la ligne de commande Windows (Win+R, `cmd.exe`) :

```
conda update conda
conda update ipython
```

3. Installer SQLite3 :

```
http://www.sqlite.org/download.html
```

   Copier le programme `sqlite3.exe` dans `C:\WINDOWS`

anaconda est également disponible pour Mac OSX. **Faciliter l'accès dans l'interface graphique**
Créer un raccourci dans Windows :

```
C:\Python26\python.exe "C:\python26\scripts\ipython" notebook -pylab inline
```

La démarche pour l'évaluation des éléves

La notion de carnet ou « notebook »

- ⋆ un lieu d'expérimentation et de développement interactif ;
- ⋆ une conception linéaire d'un algorithme ;
- ⋆ un support à la rédaction d'un rapport ;
- ⋆ un **outil pour l'évaluation !**

Exemple de rapport

Résolution d'équation du second degré

$a * x^2 + b * x + c = 0$

In [7]:
```
a = 1
b = 1
c = -4

d = b**2-4*a*c # discriminant

if d < 0:
    print "Cette équation n'a pas de solution réelle"
elif d == 0:
    x = (-b+sqrt(d))/(2*a)
    print "Cette équation a une seule solution ", x
else:
    x1 = (-b+sqrt(d))/(2*a)
    x2 = (-b-sqrt(d))/(2*a)
    print "Cette équation a deux solutions ", x1, " and", x2
```

```
Cette équation a deux solutions  1.56155281281  and -2.56155281281
```

IPython

Les options sur la ligne de commande

- ▷ `ipython notebook` : permet d'utiliser son navigateur Web pour éditer et exécuter des commandes Python ;
- ▷ `ipython notebook --pylab` : permet de charger au lancement pylab permettant de disposer des fonctionnalités de NumPy, Matplotlib et du module « `math` » ;

Quelques commandes utiles à utiliser dans iPython

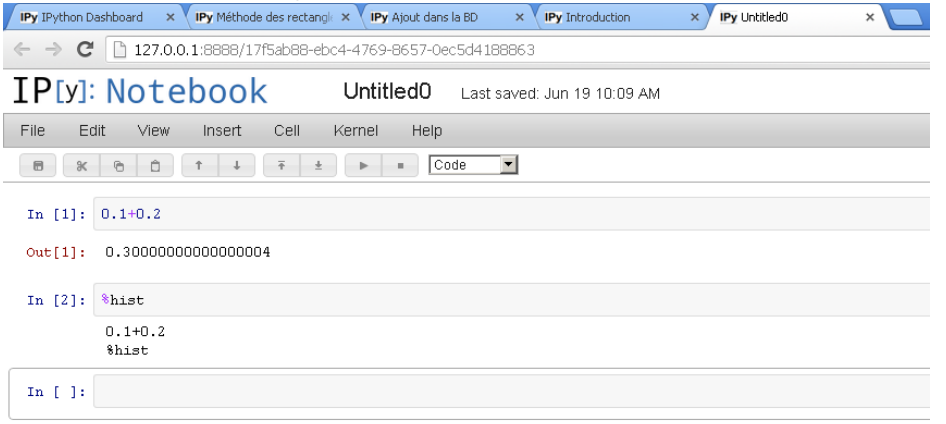
- ▷ `notebook` :
- ▷ `Tab-completion` : permet de compléter la saisie en tapant sur la touche « tabulation » après avoir saisie quelques caractères débutant la saisie. Lorsque plusieurs choix sont possibles, un menu est proposé.
- ▷ `?` : permet d'obtenir la documentation sur une commande :

```
In [9]: len?
Type:       builtin_function_or_method
String Form:<built-in function len>
Namespace:  Python builtin
Docstring:
len(object) -> integer

Return the number of items of a sequence or mapping.
```

- ▷ `%who` & `whos` : affiche les variables, les fonctions définies, les modules chargées. `%who` affiche la liste et `%whos` affiche les détails ;
- ▷ `%pylab` : permet de disposer des modules « NumPy », « Matplotlib » et « math » ;
- ▷ `%hist` : affiche l'historique des commandes saisies.

iPython Notebook

Dans ce mode d'utilisation, on utilise le navigateur Web :



La page Web est décomposée en « cellule », qui peut contenir :

— du code Python telle qu'il serait entré dans l'interprète iPython :
  ◇ `Majuscule-Entrée` : exécute le contenu de la cellule et affiche le résultat ;
  ◇ `Ctrl-m h` : affiche la liste des commandes ;
  ◇ `Ctrl-m l` : permet de faire apparaître les numéros de ligne pour faciliter la correction des erreurs ;

— du texte formaté en « Markdown » qui permet de disposer de lien web, de gras, italique et d'un mode d'entrée d'équation au format LaTeX.

▷ les commandes de iPython vues précédemment fonctionnent ;

▷ `%pylab inline` : similaire à la commande précédente avec en plus l'intégration des graphes (utile lorsqu'elle est combinée avec l'option notebook).

Les maths dans Python : le module « `math` »

`Fonction`	Description
`ceil (x)`	retourne arrondie à la valeur supérieure à l'entier plus grand ou égal à x
`floor (x)`	retourne la valeur arrondie à l'entier plus petit ou égal à x
`exp (x)`	retourne e^x
`log (x, base)`	retourne la valeur du logarithme de x suivant la base indiquée. Si la base n'est pas indiquée, retourne la valeur du logarithme népérien
`log10 (x)`	retourne $log_{10}(x)$
`pow (x, y)`	retourne x^y
`sqrt (x)`	retourne $\sqrt{x}$
`degrees (x)`	réalise la conversion de radians vers degrés
`radians(x)`	réalise la conversion de degrés vers radians
`sin (x)`	retourne le sinus de x exprimé en radians
`cos (x)`	retourne le cosinus de x exprimé en radians
`tan (x)`	retourne la tangente de x exprimé en radians
`asin (x)`	retourne l'arcsinus de x en radians
`acos (x)`	retourne l'arcosinus de x en radians
`atan (x)`	retourne l'arctangente de x en radians
`hypot (x, y)`	retourne la valeur de l'hypoténuse $\sqrt{x^2 + y^2}$ ou la distance du point (x, y) par rapport à l'origine
`pi`	la constante π
`e`	la constante e

Représentation binaire des nombres

L'octet

La valeur 70 représentée en binaire $\Longrightarrow$

0	1	0	0	0	1	1	0
128	64	32	16	8	4	2	1

Cet octet peut ensuite être associé avec un caractère au travers de la table « ASCII » ou du codage UTF-8.

Les entiers relatifs

Suivant la taille de la représentation machine :
- on associe le bit de « poids fort », c-à-d associé à la puissance de 2 la plus grande, au signe ;
- on choisit un codage facilitant l'arithmétique binaire ;
- on peut représenter des valeurs comprises entre -2^{nbre_bits-1} et $2^{nbre\_bits-1}-1$

Exemple sur une représentation suivant une taille de 8bits, $nbre\_bits = 8$:

0	1	0	0	0	1	1	0	= 70
0	1	1	1	1	1	1	1	= 127
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	= -1
1	1	1	1	1	1	1	0	= -2
-128	64	32	16	8	4	2	1	

- ⋆ on ne code qu'une seule fois la valeur 0 : *il n'y a pas* $+0$ *et* -0
- ⋆ les opérations arithmétiques sont simplifiées :

$3-4 = 3+(-128+124) = -128+127 = -1$

-128	64	32	16	8	4	2	1	
0	0	0	0	0	0	1	1	3
1	1	1	1	1	1	0	0	-4
1	1	1	1	1	1	1	1	-1

La conversion d'un entier négatif en sa représentation correspond à inverser les bits de sa valeur absolue et d'ajouter 1 au résultat ou à représenter $2^{nbre\_bits} - \mid x \mid$. Cette opération s'appelle « complément à 2^n ».

Représentation binaire des nombres

Réels

Un réel est manipulé par la machine sous une forme approchée appelée « nombre à virgule flottante » ou « *float* » constitué de :

▷ un signe ;
▷ une mantisse, c-à-d un nombre fixe de bits significatifs ;
▷ un exposant suivant une base 2, 10 ou 16 (hexadécimal) ;

Exemple :

⋆ en base 10 : $13,625$ ou $13625 * 10^{-3}$, la mantisse est 13625 ;
⋆ en base 2 :

Puissance de 2	3	2	1	0		-1	-2	-3
valeur	8	4	2	1	,	0,5	0,25	0,125
bit	1	1	0	1	,	1	0	1

$$\begin{aligned} 1101,101_2 &= 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 + 1*2{-}1 + 0*2{-}2 + 1*2{-}3 \\ &= 1*8 + 1*4 + 0*2 + 1*1 + 1*0,5 + 0*0,25 + 1*0,125 \\ &= 8 + 4 + 0 + 1 + 0,5 + 0 + 0,125 \\ &= 13,625_{10} \end{aligned}$$

Ce qui donne $1101101 * 2^{-3}$, la mantisse est 1101101.

Représentation binaire des nombres : la norme IEEE 754

- ☐ on associe un bit pour exprimer si le nombre est positif ou négatif : le bit de « poids fort » situé à gauche ;
- ☐ on fixe le nombre de bits alloués à la représentation de l'exposant :
  - ◇ on simplifie le codage de cet exposant, en exprimant différemment son signe :
    - ⋆ on décide de ne pas pas exprimer le signe par l'utilisation d'un bit, mais en le déduisant par une opération de « décalage » : $exposant\_décalé = exposant + décalage$
      Ainsi, pour un exposant variant de +100 à -100, si on choisit un décalage de 100*, la valeur de l'exposant décalé ira de* 0 *à* 200 .
    - ⋆ on calcule l'espace de valeurs possibles suivant le nombre fixe de bits associés à l'exposant ;
      Exemple pour un exposant sur 8 bits : $2^{nbre\_bits\_exposant} = 256$
    - ⋆ on choisit une valeur particulière de cet espace, $2^{nbre\_bits\_exposant} - 1$, pour exprimer :
      - ▷ la valeur infini : ∞ avec une mantisse nulle ;
      - ▷ une valeur d'erreur : NaN, « not a number » avec une mantisse non nulle ;
    - ⋆ on utilise la valeur zéro de l'exposant pour exprimer :
      - ▷ le nombre 0 : exposant et mantisse nuls ;
      - ▷ des nombres « dénormalisés » : exposant nul mais mantisse non nulle ;
    - ⋆ on divise cet espace de valeurs en deux pour exprimer au choix, un exposant positif ou négatif ;
      Exemple pour un exposant sur 8 bits : $2^{nbre\_bits\_exposant-1} = 128$
    - ⋆ on détermine la valeur du décalage compte tenu de la taille de l'espace de valeurs restant :
      $décalage = 2^{nbre\_bits\_exposant-1} - 1$
      Exemple pour un exposant sur 8 bits : $décalage = 2^7 - 1 = 127$
      l'exposant peut varier alors de :
      - ▷ $+127$, ce qui donne $exposant\_décalé = 127 + 127 = 254$ la valeur maximale possible ;
      - ▷ à -126, ce qui donne $exposant\_décalé = -126 + 127 = 1$ la valeur minimale possible ;

Représentation binaire des nombres : la norme IEEE 754

La norme IEEE754 facilite le traitement par un ordinateur

Les comparaisons entre deux nombres à virgule flottante est facilité : elle est similaire, dans la plupart des cas, à celle réalisée sur des entiers.

Représentation des valeurs et stockage de la mantisse

Les nombres représentés suivant cette norme doivent obligatoire avoir un bit à 1 au début de leur représentation : $0,00101_2 = 1,01_2 * 2^{-3}$ Ce qui permet :

- de « supprimer » du stockage ce premier bit à 1 (il devient implicite) ;
- d'augmenter de 1 bit la capacité de représentation.

Calcul de la capacité de représentation numérique

La valeur est calculée suivant la formule suivante :

$$(-1)^{bit\_signe} * mantisse * 2^{exposant-décalage}$$

On distingue deux « précisions » différentes pour un codage sur 32 bits ou 64 bits :

- simple précision :
  - valeur la plus proche de zéro : $\pm 2^{-126} \cong \pm 1,175494351 * 10^{-38}$
  - valeur la plus éloignée de zéro : $\pm(2 - 2^{23}) * 2^{127} \cong 3,4028235 * 10^{38}$
- double précision :
  - valeur la plus proche de zéro : $\pm 2^{?1022} \cong \pm 2,2250738585072020 * 10^{-308}$
  - valeur la plus éloignée de zéro : $\pm(2^{-1024} - 2^{971}) \cong \pm 1,7976931348623157 * 10^{308}$

	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^{bit\_signe} * mantisse * 2^{exposant-127}$	24 bits	7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^{bit\_signe} * mantisse * 2^{exposant-1023}$	53 bits	16

Représentation binaire des nombres : la norme IEEE 754

Les nombres « dénormalisés »

Les nombres en représensation « dénomarlisés » :
- ○ permettent d'exprimer des valeurs plus petites que celles exprimables dans la représentation normalisée ;
- ○ ne disposent pas de « bit implicite » à 1 en début de la représentation contrairement à la version normalisée ;
- ○ ne possèdent pas autant de chiffres significatifs que la représentation normalisé ;
- ○ permettent une perte graduelle de précision quand les résultats de calculs sont trop proches de zéro pour être représentés par la version normalisée.

L'exposant de ces nombres est à zéro ce qui donne un exposant de -126 en simple précision et de -1022 en double précision.

Ce qui donne :
- ▷ en simple précision, plus petit nombre différent de zéro et plus grand nombre négatif différent de zéro : $\pm 2^{-1074} \cong \pm 2^{-149} \cong \pm 1,4012985 * 10^{-45}$;
- ▷ en double précision, plus petit nombre différent de zéro et plus grand nombre négatif différent de zéro : $\pm 2^{-1074} \cong \pm 4,9406564584124654 * 10^{324}$;

Les valeurs ∞ et `NaN`

$x \div 0 = \pm\infty$
$0 \div 0 = NaN$
$x \div \infty = \pm 0$
$\infty \div \infty = NaN$
$(+\infty) + (+\infty) = +\infty$
$(+\infty) - (+\infty) = NaN$

La capacité des valeurs numériques

Les valeurs entières ne sont pas limitées en dimension.
Calcul de la clé du numéro de sécurité sociale sur 13 chiffres :

```
In [17]: numero_secu=1700187001001

In [18]: 97-(numero_secu%97)
Out[18]: 73
```

Les nombres à virgule flottante sont limités suivant la « taille des mots » processeur :

- ▷ pour 32 bits la limite est $\pm 3.4 * 10^{38}$;
- ▷ pour 64 bits la limite est $\pm 1.8 * 10^{308}$:

```
In [12]: 9*1.8E307
Out[12]: 1.62e+308

In [13]: 10*1.8E307
Out[13]: inf
```

Calcul numérique et approximation

Les règles d'associativité et de commutativité

Soit l'équation : $\frac{(x-x+y)}{y} = \frac{((x-x)+y)}{y} = \frac{(x-(x+y))}{y}$

Calcul sous Python :

▷ Pour des valeurs x et y proches :

```
>> x=10000000.0
>> y=9999999.0
>> print ((x-x)+y)/y
1.0
>> print (x+(y-x))/y
1.0
```

▷ Pour des valeurs x et y éloignées :

```
>> x=1000000000000.0
>> y=0.00000000001
>> print ((x-x)+y)/y
1.0
>> print (x+(y-x))/y
0.0
```

Exemple de calcul

Erreur de calcul :

```
In [106]:
v=-1.0
for i in range(0,201):
    print acos(v)
    v = v + 0.1
---------------------------------------
ValueError       Traceback (most recent call last)
<ipython-input-106-e13af1a75e60> in <module>()
      1 v=-1.0
      2 for i in range(0,201):
--> 3     print acos(v)
      4     v = v + 0.1

ValueError: math domain error

3.14159265359
2.69056584179
...

0.643501108793
0.451026811796
2.10734242554e-08
```

Une autre version :

```
v = -100
for i in range(0,201) :
    print acos(v/100.0)
    v = v + 1
```

Pourquoi ?

```
In [3]: format(0.01,'.30f')
Out[3]: '0.010000000000000000208166817117'

In [4]: sum([0.01]*100)
Out[4]: 1.0000000000000007

In [5]: round(1.0000000000000007, 13)
Out[5]: 1.0
```

```
In [9]: float.hex(0.1)
Out[9]: '0x1.999999999999ap-4'
```

Explications :

★ « `0x1.999999999999ap-4` » est un nombre flottant normalisé en double précision exprimé en hexadécimal ;

★ « `0x` » : indique la notation hexadécimale ;

★ « `1.` » : correspond au bit implicite ;

★ « `99 99 99 99 99 99 a` » : la mantisse sur 52 bits ;

★ « `p-4` » : correspond à l'exposant 2^{-4}

En binaire, cela donne :

$1.1001100110011001100110011001100110011001100110011001101 * 2^{-4}$

```
In [3]: from decimal import Decimal
In [4]: Decimal(0.1)
Out[4]: Decimal('0.1000000000000000055511151231...')
```

Les listes de valeurs et Python

La génération de listes

Avec l'instruction « `range` » :

```
1 liste = range(1,10)
2 print liste
```

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

En utilisant l'opérateur de répétition « * » sur une liste :

```
1 liste_automatique = [ 0.0 ]*10
2 print liste_automatique
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Pour parcourir les éléments d'une liste :

```
1 for element in liste:
2     print element,":",
```

```
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 :
```

Pour créer une nouvelle liste à partir d'une autre à l'aide d'un parcours et de la méthode « `append` » :

```
1 liste_originale = [ 3, 5, 7, 8]
2 nouvelle_liste = []
3 for element in liste_originale :
4     nouvelle_liste.append(element*2)
5 print nouvelle_liste
```

```
[6, 10, 14, 16]
```

Pour créer une nouvelle liste à partir d'une autre :

```
1 liste_originale = [ 3, 5, 7, 8]
2 nouvelle_liste = [element * 2 for element in liste_originale]
3 print nouvelle_liste
```

```
[6, 10, 14, 16]
```

Les listes de valeurs et le module « NumPy »

La génération de liste de valeurs dans un intervalle donné

- ▷ en donnant le pas ;

```
In [9]: vals_x = arange(1,2,0.1)

In [10]: vals_x
Out[10]: array([ 1. ,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])

In [12]: len(vals_x)
Out[12]: 10
```

En Python, sans NumPy :

```
In [1]: vals_x = [1.0 + i*0.1 for i in range(0,10)]

In [2]: vals_x
Out[2]: [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7000000000000002, 1.8, 1.9]
```

- ▷ en donnant le nombre de valeurs désirées dans l'intervalle :

```
In [4]: vals_x = linspace(1,2,10)

In [5]: vals_x
Out[5]:
array([ 1.        ,  1.11111111,  1.22222222,  1.33333333,  1.44444444,
        1.55555556,  1.66666667,  1.77777778,  1.88888889,  2.        ])

In [6]: len(vals_x)
Out[6]: 10
```

Un exemple de résolution analytique et graphique

L'équation du second degré

```
1  # coding=utf-8
2  from pylab import *
3
4  a = 1
5  b = 1
6  c = -4
7
8  d = b**2-4*a*c # discriminant
9
10 if d < 0:
11     print "Cette équation n'a pas de solution réelle"
12 elif d == 0:
13     x = (-b+sqrt(d))/(2*a)
14     print "Cette équation a une seule solution ", x
15 else:
16     x1 = (-b+sqrt(d))/(2*a)
17     x2 = (-b-sqrt(d))/(2*a)
18     print "Cette équation a deux solutions ", x1, " et", x2
19 vals_x = linspace(-5,5,10)
20 vals_y = [x**2+x for x in vals_x]
21
22 plot(vals_x,vals_y)
23 plot(vals_x,[4]*len(vals_x))
24 xticks(vals_x)
25 grid(True)
26 show()
```

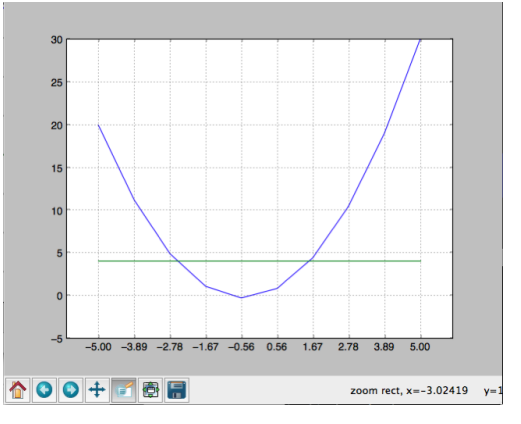
Résolution d'équation du second degré

$a * x^2 + b * x + c = 0$

In [7]:

```
a = 1
b = 1
c = -4

d = b**2-4*a*c # discriminant

if d < 0:
    print "Cette équation n'a pas de solution réelle"
elif d == 0:
    x = (-b+sqrt(d))/(2*a)
    print "Cette équation a une seule solution ", x
else:
    x1 = (-b+sqrt(d))/(2*a)
    x2 = (-b-sqrt(d))/(2*a)
    print "Cette équation a deux solutions ", x1, " and", x2
```

```
Cette équation a deux solutions  1.56155281281  and -2.56155281281
```

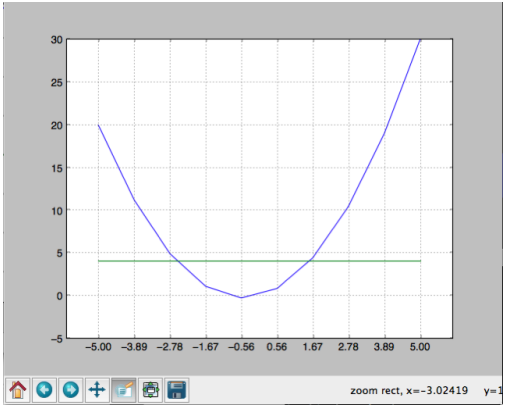


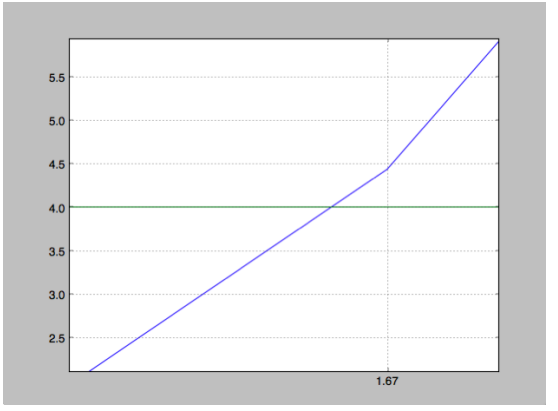
Un exemple de résolution analytique et graphique

Exécution du programme dans iPython

```
pef@darkstar:~/Python_snippets/Math$ ipython -pylab
Python 2.7.5 (default, Jun 11 2013, 12:38:41)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.2 - An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: MacOSX].
For more information, type 'help(pylab)'.

In [1]: %run quadratic.py
Cette équation a deux solutions  1.56155281281  and -2.56155281281
```

Résolution graphique





Un exemple de résolution analytique et graphique

Le traitements des nombres complexe avec le module « `cmath` »

```
1  # coding=utf-8
2  from pylab import *
3  import cmath
4  a = 1
5  b = 1
6  c = 4

8  d = b**2-4*a*c # discriminant

10 if d < 0:
11     print "Cette équation n'a pas de solution réelle"
12     x1 = (-b - cmath.sqrt(d))/(2*a)
13     x2 = (-b + cmath.sqrt(d))/(2*a)
14     print "Cette équation a deux solutions complexes ", x1, " et", x2
15 elif d == 0:
16     x = (-b+sqrt(d))/(2*a)
17     print "Cette équation a une seule solution ", x
18 else:
19     x1 = (-b+sqrt(d))/(2*a)
20     x2 = (-b-sqrt(d))/(2*a)
21     print "Cette équation a deux solutions ", x1, " and", x2
```

Ce qui donne pour $a = 1, b = 1, c = 4$:

```
Cette équation n'a pas de solution réelle
Cette équation a deux solutions complexes  (-0.5-1.9364916731j)  et (-0.5+1.9364916731j)
```

Traitement de chaîne et dictionnaire

Comversion notation romaine -> décimale

chiffre romain	M	D	C	L	X	V	I
valeur	1000	500	100	50	10	5	1

Pour stocker les valeurs un dictionnaire :

```
romain_decimal = { 'M':1000, 'D':500, 'C':100, 'L':50, 'X':10, 'V':5, 'I':1 }
print romain_decimal['C']
100
```

Une proposition de programme :

```
1  romain_decimal = { 'M':1000, 'D':500, 'C':100, 'L':50, 'X':10, 'V':5, 'I':1 }
2  chaine = "MCCMLXXIV"
3  valeur_decimale = 0
4  chiffre_romain_precedent = chaine[0]
5  accumulateur = romain_decimal[chiffre_romain_precedent]
6  for chiffre_romain_courant in chaine[1:]:  # on part de la seconde valeur
7      if romain_decimal[chiffre_romain_courant] < romain_decimal[chiffre_romain_precedent] :
8          valeur_decimale = valeur_decimale + accumulateur
9          accumulateur = romain_decimal[chiffre_romain_courant]
10         chiffre_romain_precedent = chiffre_romain_courant
11         continue # on passe à l'occurence suivante de la boucle
12     if romain_decimal[chiffre_romain_courant] == romain_decimal[chiffre_romain_precedent] :
13         accumulateur = accumulateur + romain_decimal[chiffre_romain_courant]
14         continue  # on passe à l'occurence suivante de la boucle
15     # romain_decimal[chiffre_romain_courant] > romain_decimal[chiffre_romain_precedent]
16     valeur_decimale = valeur_decimale - accumulateur
17     accumulateur = romain_decimal[chiffre_romain_courant]
18     chiffre_romain_precedent = chiffre_romain_courant
19 valeur_decimale = valeur_decimale + accumulateur
20 print "La valeur traduite de ", chaine, " est ", valeur_decimale
```

Les erreurs les plus courantes

Liées à l'édition du source

- ▷ le « : » oublié avant la définition d'un bloc par une indentation ;
- ▷ l'utilisation de caractères espaces et/ou de tabulation pour indenter le source : il faut que ce soit fait toujours de la même manière ;
- ▷ utilisation de caractères accentués (ajout de l'« encoding » correct) ;

Liées à Python

- ▷ oubli d'ajout du module pour en utiliser les fonctions :

```
from pylab import *
import cmath
```

- ▷ définir une variable dans une conditionnelle et s'en servir ensuite même si elle n'a pas été définie :

```
1  if a == 0 :
2      b = 1
3  print b
```

Ici, si `a==0` *la variable* `b` *sera définie et affichable à la ligne 3, sinon il ne sera pas défini et créera une exception à la ligne 3.*

$\Longrightarrow$ définir chaque variable au début du source en leur affectant une valeur d'initialisation.

Complexité d'un algorithme

- ▷ Le temps d'exécution d'un programme dépend en général de l'ordinateur utilisé.
  Pour simplifier, nous mesurerons le temps d'exécution d'un programme **en nombre d'instructions élémentaires** exécutées.
- ▷ Ce temps d'exécution dépend évidemment des données traitées par le programme.
  Pour simplifier, nous nous interesserons au **temps maximum d'exécution** en fonction de la taille des données.
- ▷ On peut toujours mesurer cette taille des données par un entier n comme par exemple le nombre d'éléments d'un vecteur.
- ▷ On simplifie généralement la description du temps maximum d'exécution d'un programme en ne retenant que l'**ordre de grandeur** correspondant.

Ceci nous permet de définir la *complexité d'un programme* comme étant l'*ordre de grandeur de son temps maximum d'exécution (mesuré en nombre d'instructions élémentaires exécutées) en fonction de la taille des données (mesurées par un entier n).*

Ordre	de grandeur	Taille	n des	données
du temps max	d'exécution	10	100	1000
	n	10	100	1000
	$n * log(n)$	33	660	10^4
Complexité	n^2	100	10^4	10^6
du	n^3	1000	10^6	10^9
programme	2^n	1024	10^{10}	10^{330}
	n^n	10^{10}	10^{200}	10^{3000}

Notation de Landau

Pour exprimer qu'une fonction $f(n)$ est de l'ordre de grandeur d'une fonction $g(n)$, on utilise la notation de Landau qui consiste à écrire $f(n) = O(g(n))$ lorsqu'il existe une constante C et un nombre n_0 tels que $\forall n \geq n_0, \mid f(n) \mid < C \mid g(n) \mid$.

Dire, par exemple, qu'un programme a la complexité en $O(1)$ revient à dire que son temps d'exécution maximum est indépendant de la taille de ses données.