Présentation générale du langage de programmation Python

Didier Villers
didier.villers@umons.ac.be

4 février 2015

Sommaire

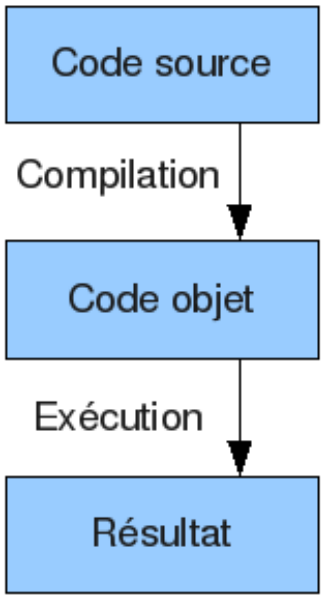
Rôle des langages de programmation

- Décrire des instructions dans un langage compréhensible par un être humain, mais transformable en d'autres instructions compréhensibles par l'ordinateur (langage machine)
- Automatiser le traitement de l'information ;
- Effectuer des calculs, des simulations ;
- Traiter l'information en temps réel ;
- Fournir un interface à l'utilisateur ;

Évolution des langages

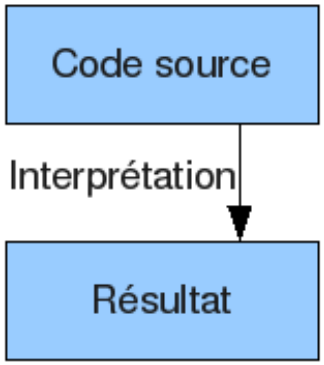
- L'assembleur (à partir des années 50's)
  - Mnémoniques directement équivalentes aux instructions machines, donc dépendant du processeur utilisé
  - Instructions de bas niveaux (appel d'une variable en mémoire, opération arithmétique entre 2 opérandes,...)
- Fortran, Cobol, Pascal, C, Basic,... (années 60s et 70s)
  - Indépendants de l'ordinateur utilisé
  - Plus proche d'un langage courant, description procédurale
- Les langages à objets (années 80s et 90s)
  - Définition de briques logicielles indépendantes et autonomes, réutilisables aisément
  - Java, C++, Python, perl, Ruby . . .sont les plus connus, + Smalltalk, OCaml, Julia, Scala, . . .
- Des langages très spécialisés, par exemple. . .
  - Pour pages web (PHP)
  - Pour bases de données (SQL)

Compilation et compilateur



- Etape de traduction du code source en langage machine
- Liaison éventuelle du code avec des bibliothèques existantes de code compilé
- Exécution ultérieure du code machine (sur un ordinateur ne disposant pas du compilateur par exemple)
- Le compilateur peut optimiser le code (passes multiples)

Interprétation et interpréteur



- Traduction dynamique du code source et exécution immédiate en répétant sans cesse :
  1. lecture et analyse d'une instruction
  2. exécution de l'instruction (si elle est valide)
- Le code est souvent moins optimisé, donc plus lent
- Il est nécessaire de disposer de l'interpréteur sur l'ordinateur
- On peut créer dynamiquement du code à interpréter pendant l'exécution
- On peut éviter la phase lente de compilation

Langages intermédiaires à Bytecodes

En fait, c'est souvent un peu plus compliqué...

- Le programme est tout de même traité en entier
- Il est traduit dans un pseudo-code indépendant de l'ordinateur
- On conserve les avantages suivants :
  - Facilité de développement (cycle écriture-exécution rapide, utilisation de "briques" logicielles)
  - Portabilité (un même programme pour des ordinateurs et systèmes d'exploitation différents)

$\rightarrow$ Python ! (en version 2 et/ou 3)

Avantages généraux

- langage de haut niveau (orienté objet)
- permet d'écrire des petits programmes ou suites d'instructions (scripts)
- sous licence libre, gratuit
- utilisable pour la programmation occasionnelle par des non-informaticiens
- nombreuses librairies existantes (modules)
- moderne et efficace pour les informaticiens
- excellente lisibilité intrinsèque du code
- bien documenté (aide et manuels en ligne, livres, forums, exemples...)

Avantages techniques

- mode interactif (console, Idle, IPython,...)
- non déclaratif
- typage de haut niveau, dynamique et fort
- ramasse-miette intégré
- interfaçable avec d'autres langages (à partir de et vers)
- version de base "piles comprises"
  - module mathématique
  - accès aux fichiers et répertoires (+ formats de données standards)
  - compression, archivage, gestion de bases de données
  - fonctions génériques du système d'exploitation
  - réseau et communication, protocoles internet (+email, html)
  - multimedia (son, image)
  - interface graphique (Tkinter)
  - outils de documentation et gestion d'erreurs (débogage)
  - modules spécifiques Windows, Mac, Linux,. . .
  - . . .

Avantages à l'apprentissage

- Initiation possible via un navigateur, sans installation !
- Installation aisée
  1. de la version de base
  2. d'une "distribution" étendue (avec des modules complémentaires)
- éditeur inclus (Idle) ou autre (SciTe,. . .)
- mode interactif pour les premiers essais + version interactive spécifique : IPython & IPython Notebook
- principes de base identiques à de nombreux langages
- on n'est pas obligé d'utiliser toute la puissance du langage
- cycle d'écriture/essais très rapide

Avantages pour le scientifique, le chimiste

- possible de débuter en quelques jours
- alternative à des logiciels spécialisés (Matlab, Scilab,...)
- bon pour les calculs scientifiques, le graphisme, les simulations
- modules spécialisés
  - représentations graphiques 2D (Matplotlib)
  - représentations graphiques 3D (Mayavi, Vpython, VTK,)
  - calculs scientifiques (numpy, scipy,...)
  - traitement d'images (PIL)
  - chimie (pymol, mmtk, chimera,...)

Débuter et installer

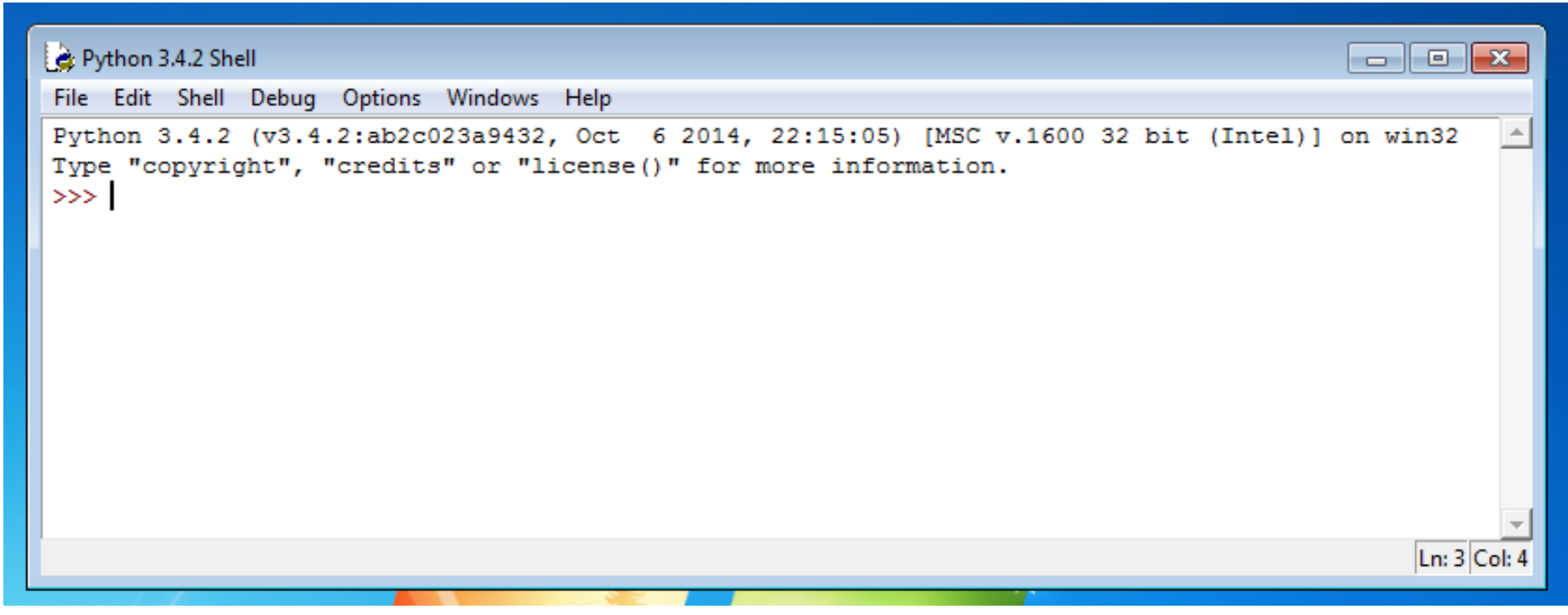
- Sans installer, via des sites web
- Versions de base de Python en versions 2 et 3 disponibles sur python.org !
- Versions plus complètes
- Exerciseurs en ligne pour l'apprentissage des bases

Références :

`http://dvillers.umons.ac.be/wiki/floss:python`

Python Shell

Idle permet d'obtenir l'interpréteur de commande de Python



Essayez les commandes copyrigth, credits, license(), quit, help, help(),...

Utilisation du mode interactif

Mode interactif :

On peut utiliser l'environnement interactif de Python comme une calculatrice !

par exemple :

```
>>> (8.314*300/24E-3)/101325
1.025660049346163
>>>
```

Notion de variable

Variables :

On peut attribuer des noms de variables, et pas seulement pour des nombres...
Ce sera plus facile, plus clair d'utiliser des noms significatifs !

```
>>> R=8.314
>>> L=0.001
>>> V=24*L
>>> n=1
>>> zero=273.16
>>> T=20+zero
>>> P=n*R*T/V
>>> atm=101325
>>> print P,P/atm   #(en Python 2 et print(P,P/atm) en Python 3)
101555.51000000001 1.0022749568221072
>>>
```

Un peu de calcul

On peut effectuer quelques calculs sur des entiers :

```
>>> 1236*5698
7042728
>>> 12569+6233
18802
>>> 12+69+532+65-9
669
>>> 12356*458955
5670847980L
>>> 123*456
56088
>>> 123**456  ?? A ESSAYER ??
```

On peut travailler avec des très très très grands nombres...

Divisons des nombres !

Effectuons des divisions d'entiers :

```
>>> 72/9
8
>>> 7/3
2
>>>
```

Diviser 2 entiers fournit un entier. Pensez qu'on ne peut placer que 2 périodes de 3 jours dans une semaine...
Mais que donne -7/3 ? ?

```
>>> -7/3
-3
>>>
```

Les nombres en virgule flottante !

```
>>> 7./3
2.3333333333333335
>>> 7/3.
2.3333333333333335
>>> 7./3.
2.3333333333333335
>>>
```

7. et 7 sont donc de deux “types” différents : “integer” et “float”

```
>>> type(7)
<type 'int'>
>>> type(7.)
<type 'float'>
>>>
```

De nombreuses autres possibilités avec les nombres...

```
>>> Navogadro=6.02214199E23
>>> kboltzmann=1.3806505E-23
>>> print Navogadro*kboltzmann
8.31447334956
>>> 2**0.5
1.414213562373095 1
>>> (5+2j)*(3-7j)
(29-29j)
>>> (1.+1./1E6)**1E6
2.7182804690957272
>>>
```

- Les expressions numériques s'évaluent en respectant les régles habituelles de priorités : parenthèses, exponentiation, multiplication, division, addition, soustraction ("PEMDAS")
- On peut aussi travailler facilement avec des tableaux contenant des milliers de données !

Un peu de logique : le type booléen !

```
>>> 12<16
True
>>> 12<11
False
>>> 12==12,12==13
(True, False)
>>> 12<16 or 12<11
True
>>> 12<16 and 12<11
False
>>> type(12<11)
<type 'bool'>
>>>
```

Les tests, comparaisons et leurs combinaisons logiques sont utiles pour réaliser des opérations de manière conditionnelle. (pour la logique booléenne : cf. wikipedia)

Les chaînes de caractères

```
>>> a='bonjour'
>>> b="bonjour"
>>> c='Bonjour'
>>> print a==b,a==c
True False
>>> d="pâté123#"
>>> print d
pâté123#
>>> é=d
SyntaxError: invalid syntax
>>> long="""un
deux
...
dix"""
>>> print long
un
deux
...
dix
>>>
```

- appelées aussi “string”
- mots, phrases, ou texte long
- délimitées par ' (apostrophe) ou " (guillemet)
- la casse est significative
- caractères accentués, spéciaux et chiffres permis
- CONSEIL : éviter les accents dans les noms des variables
- peuvent comprendre des retours à la ligne (Enter) si délimitées par """

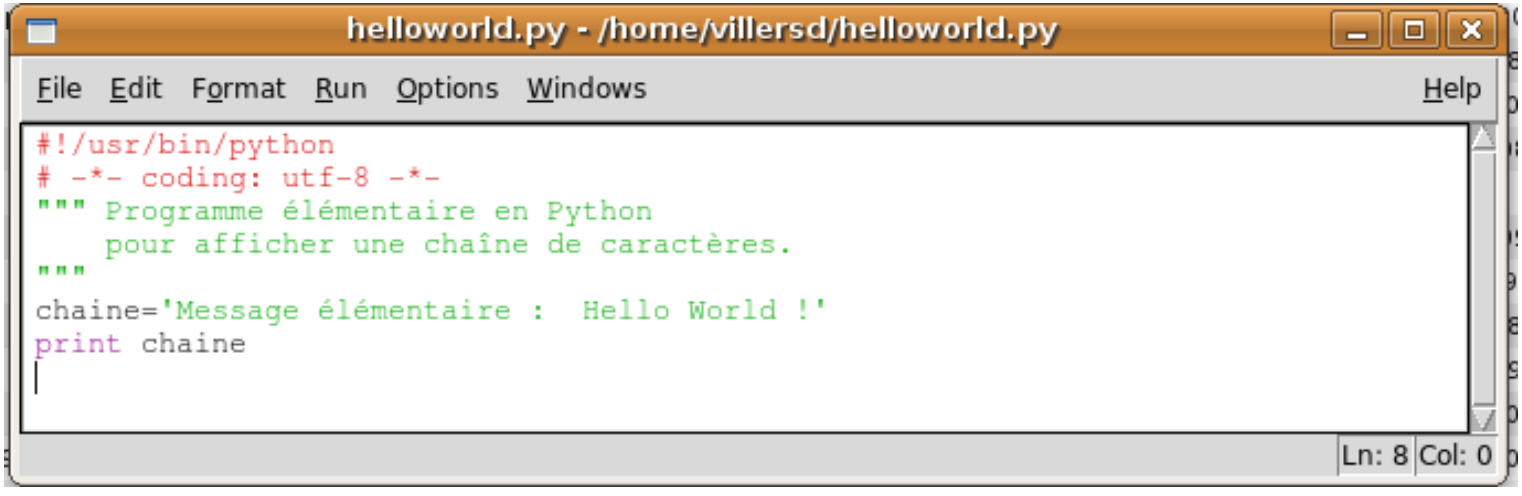
Opérations sur les chaînes

```
>>> s='Mons, le 15 septembre 2009'
>>> s[8:]
' 15 septembre 2009'
>>> s.find('le')
6
>>> s.split()
['Mons,', 'le', '15', 'septembre', '2009']
>>> s.upper()
'MONS, LE 15 SEPTEMBRE 2009'
>>> s.replace(' ','_')
'Mons,_le_15_septembre_2009'
>>>
```

- spécifier une sous-chaîne
- rechercher une sous-chaîne
- découper une chaîne
- changer la casse
- remplacer

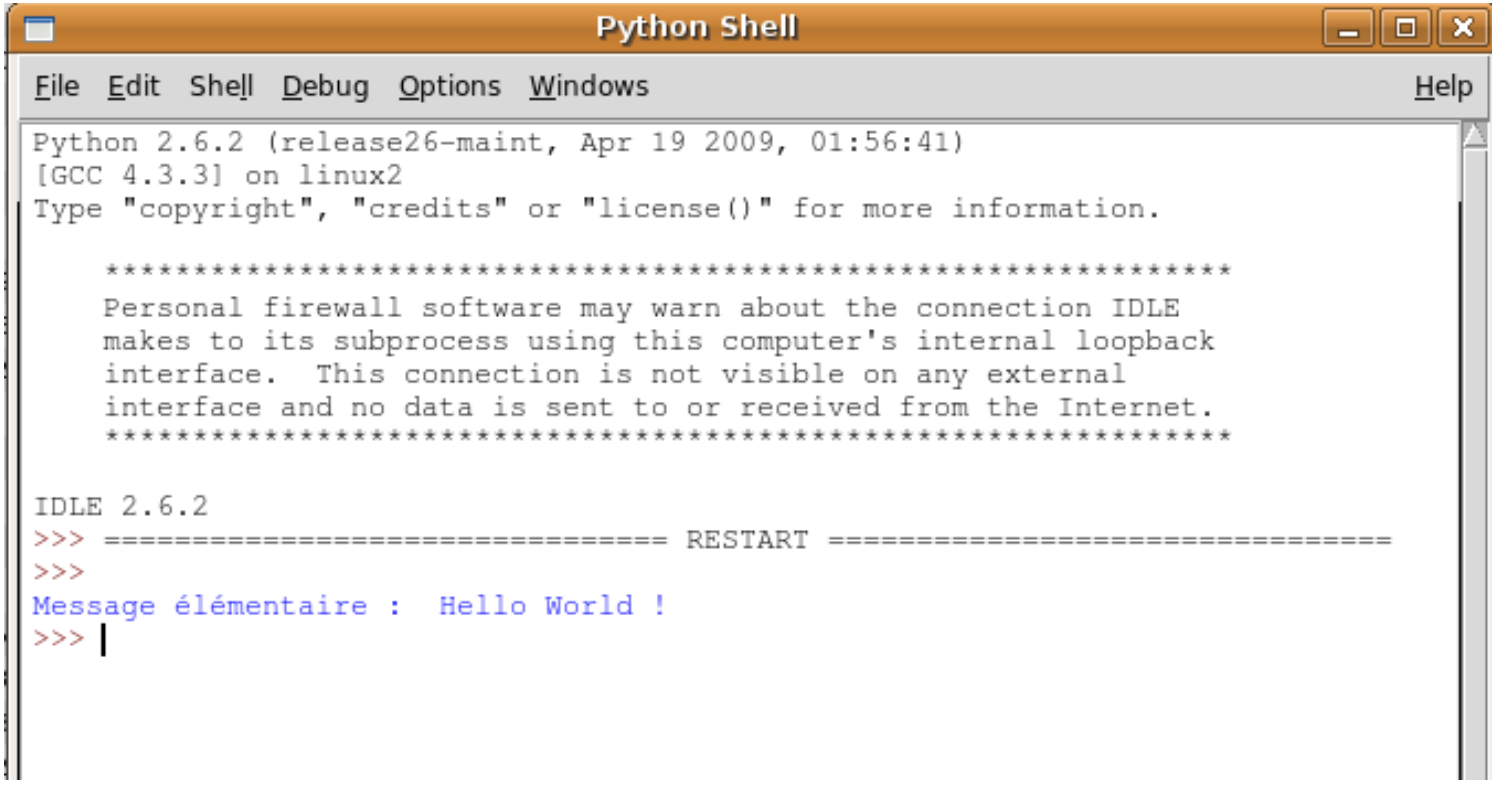
Créer et sauvegarder son premier programme

Fenêtre d'édition de programme Python de Idle :



Exécuter son premier programme

Menu Run - Run module, ou raccourci clavier F5



Types de haut niveau

Au delà des types de base (integer, float, string,...) on peut avoir des types sophistiqués (containers)

- listes
- dictionnaires
- tuples
- sets
- ...

Listes

- collections/séquences ordonnées d'objets (types de base ou autres), introduits entre crochets et séparés par des virgules
- peuvent être homogènes ou hétérogènes (types identiques ou mélangés)
- on peut les compléter ou enlever des éléments dynamiquement
- indicées (numérotées) à partir de 0
- utilisables comme tableaux multidimensionnels
- nombreuses manipulations possibles (opérations, méthodes) : accéder, concaténer, trier, compléter, rechercher, réduire,...

Listes (bis)

```
>>> l1=[31, 16, 'mot','rouge', 1+3j, [2, 'bleu',3.14]]
>>> l1
[31, 16, 'mot', 'rouge', (1+3j), [2, 'bleu', 3.14]]
>>> l2=[121, 'vert', 'tomate']
>>> l3=l1+l2
>>> l3
[31, 16, 'mot', 'rouge', (1+3j), [2, 'bleu', 3.14], 121, 'vert', 'toma
>>> len(l3)
9
>>> l3[4]
(1+3j)
>>> l3[5]
[2, 'bleu', 3.14]
>>> l3[5][1]
'bleu'
>>> l3[2:6]
['mot', 'rouge', (1+3j), [2, 'bleu', 3.14]]
```

Listes (ter)

```
>>> l3[-1]
'tomate'
>>> l3.pop()
'tomate'
>>> l3
[31, 16, 'mot', 'rouge', (1+3j), [2, 'bleu', 3.14], 121, 'vert']
>>> l3.pop()
'vert'
>>> l3.pop()
121
>>> l4=l3.pop()
>>> l4
[2, 'bleu', 3.14]
>>> l3.append(19.3)
>>> l3
[31, 16, 'mot', 'rouge', (1+3j), 19.3]
>>>
```

Dictionnaires

- collections/ensembles *non-ordonnées* de paires de clés et valeurs
- Chaque clé doit être unique (ne peut pas être présente 2 fois dans un dictionnaire) et identifie la valeur correspondante (les clés sont souvent des nombres ou des chaînes)
- Les valeurs peuvent être un objet de n'importe quel type (de base ou autres)
- Clés et valeurs sont séparées par le caractère " :"
- Les paires clés :valeurs sont séparées par des virgules et le tout encadré par une paire d'accolades { }forme le dictionnaire.

Dictionnaires (bis)

```
>>> densite={'eau':1.,'fer':7.8,'cuivre':8.9,'or':18.9,'platine':21.4}
>>> densite.keys()
['cuivre', 'fer', 'eau', 'or', 'platine']
>>> densite
{'cuivre': 8.9, 'fer': 7.8, 'eau': 1.0, 'or': 18.9, 'platine': 21.4}
>>> densite['or']
18.9
>>> densite.has_key('glace')
False
>>> sorted([densite.values(),densite.keys()])
[[8.9, 7.8, 1.0, 18.9, 21.4], ['cuivre', 'fer', 'eau', 'or', 'platine'
>>> import operator
>>> sorted(densite.iteritems(), reverse=True,
                                  key=operator.itemgetter(1))
[('platine', 21.4), ('or', 18.9), ('cuivre', 8.9),
                                  ('fer', 7.8), ('eau', 1.0)]
```

Tuples

- Les tuples sont comme les listes, MAIS :
- entourés de parenthèses au lieu de crochets
- les éléments sont non-modifiables après la création
- pas de méthodes sur les tuples (rechercher, enlever un élément,...
- les tuples sont plus rapides d'accès que les listes
- ils peuvent être utilisés comme clés de dictionnaires
- il est possible de convertir un tuple en liste et vice-versa

Structure conditionnelle

Instruction d'exécution conditionnelle if...elif...else (si...sinon-si...autrement)

- Commence par *if expression :*
- Si l'expression est vraie, le bloc d'instructions qui suit est exécuté
- Si c'est faux, *elif expression :* permet d'enchaîner une seconde condition
- Si aucune condition n'est vérifiée, *else :* permet de déterminer les instructions à effectuer

Exemple if

```
a=input('Donnez une note ? ')
if a >= 12:
    print("Dispense pour 5 ans")
elif a < 10:
    print("Epreuve à représenter")
else:
    print("Report de session")
```

Structures de répétition while et for

While

- Commence par *while expression :*
- Si l'expression est vraie, le bloc d'instructions qui suit est exécuté
- L'expression est à nouveau évaluée
- Lorsque l'expression est (devient) fausse, le bloc n'est plus exécuté

for

- Commence par *for element in sequence :*
- Le bloc d'instructions qui suit est exécuté autant de fois qu'il y a d'éléments dans la séquence
- Else : permet d'exécuter un autre bloc après avoir considéré tous les éléments

Exemples while et for

```
print('Structure while !')
c=0
while c< 4:
    print(c)
    c=c+1
print('valeur finale = ',c)

print('Structure for')
a=range(11)
print(a)
for n in a:
    print(n*7)
```

Indentations des structures

L'indentation est intégrée à Python

- Les retraits permettent de reconnaître et exécuter des structures dans des structures
- Efficace, léger et très favorable à une écriture compacte et lisible des programmes
- Tabulations (en fait remplacées par 4 espaces !)
- Importance de la configuration correcte de l'éditeur
- Si l'indentation n'est pas respectée précisément : erreur

Exemples d'indentation

```
print('Table de multiplication')
a=range(11)
for i in a:
    for j in a:
        print(i*j,)
    print(' sont multiples de ',i)
```

Fonctions en Python

- Permettent d'étendre le langage
- Résolvent un problème délimité
- Font appel elles-mêmes à d'autres fonctions
- Dépendent de variables (arguments, paramètres)
- Appelables autant de fois que souhaité, avec des arguments quelconques
- Renvoient (ou non) un résultat utilisable dans une expression
- Utilisent des noms de variables à portée locale
- Définies par le programmeur, ou existantes dans des "librairies modules" supplémentaires
- Spécifiées ou définies avant l'utilisation

Exemple de fonction

```
def fractions(nummol):
    sum=0.
    for num in nummol:
        sum+=num
    fract=[]
    for num in nummol:
        fract.append(num/sum)
    return fract

n=input('Donnez les nombres de moles des constituants (entre crochets
print(fractions(n))
```

Utilisation de librairies de fonctions standard

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(0)
1.0
>>> math.__dict__
{'pow': <built-in function pow>, 'fsum': <built-in function fsum>, 'co

A essayer :  math.__doc__ et math.__name__
```

On peut modifier les noms des fonctions et la façon de les stipuler (espaces de noms) par la directive import

Modules, objets, classes, librairies

Python est un langage très moderne, il comprends donc des structures très avancées

- Classes (programmation objet), regroupant variables, données et fonctions
- Module : ensemble de code repris dans un seul fichier
- Paquet ou Librairie : ensemble de modules avec une arborescence en répertoires

La programmation avancée en Python comprend notamment aussi

- la gestion des erreurs
- des procédures de tests
- la génération de documentation sous différentes formes

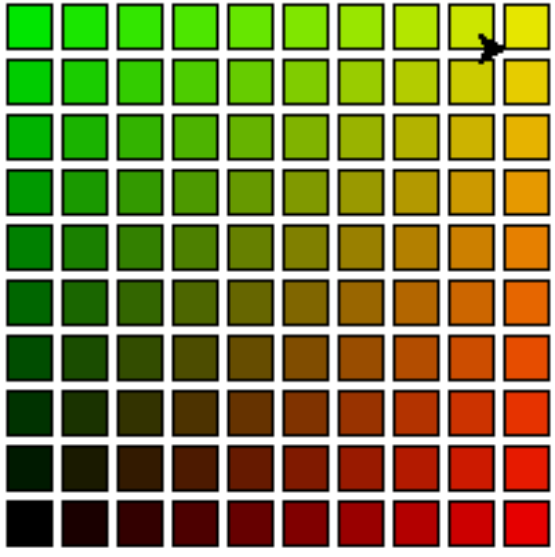
Notion d'algorithme

Algorithme : description des opérations à effectuer pour résoudre un problème

- Indépendant des ordinateurs
- Logique et systématique
- Langage courant structuré
- Transposable pour différents langages de programmation
- Détermine le temps d'exécution et la mémoire nécessaire en terme de proportionnalité à la taille du problème

Exemple : la multiplication matricielle nécessite de l'ordre de N^3 opérations (si N est la taille des matrices)
Référence : http ://fr.wikipedia.org/wiki/Algorithmique

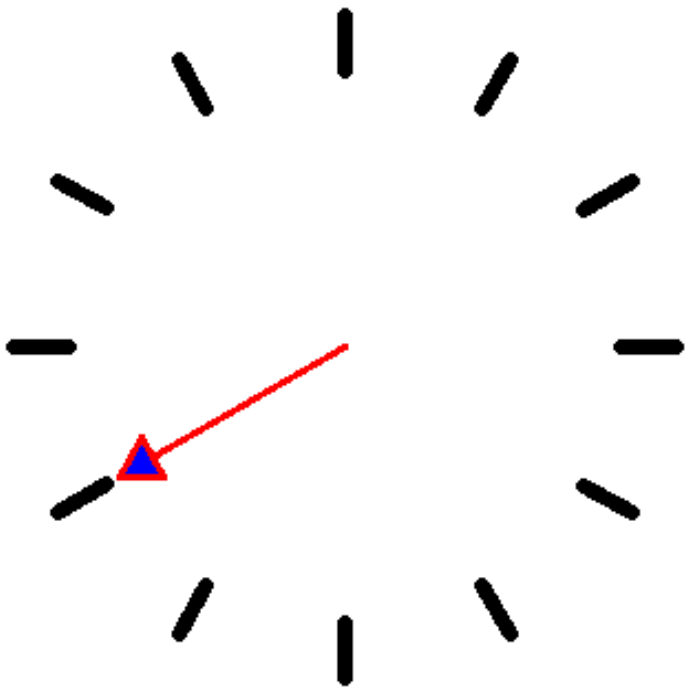
Simple, avec le module standard Turtle



```
i=0
while i < 10:
    j=0
    while j <10:
        up()
        goto(x+i*20,y+j*20)
        down()
        fill(1)
        n=0
        while n <4 :
            forward(16)
            left(90)
            n=n+1
        color([i*0.1,j*0.1,0])
        fill(0)
        color(0,0,0)
        j=j+1
    i=i+1
```

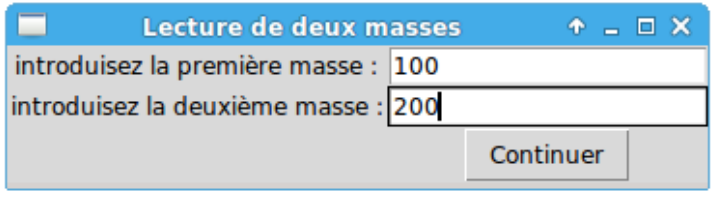
Module XTurtle (Python 2)

cf. http ://code.google.com/p/xturtle/downloads/list/



Simple, avec l'interface graphique Tkinter

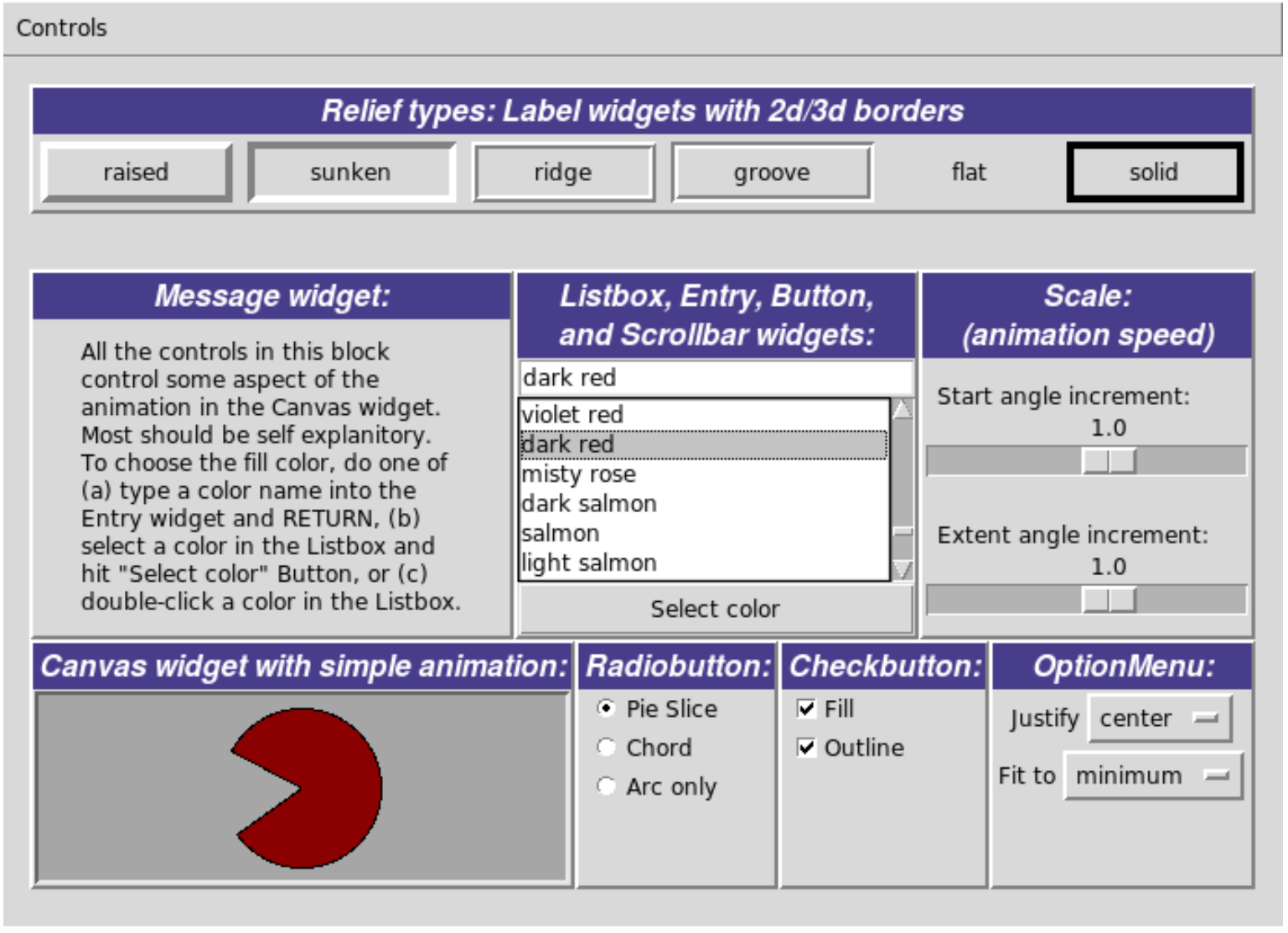
```
>>>
Masses lues :  100.0  et  200.0
>>>
```




```
from tkinter import *
fen01 = Tk()
fen01.title("Lecture de deux masses")
chaine1 = Label (fen01, text = "introduise
chaine2 = Label  (fen01, text = "introduis
chaine1.grid(row =0)
chaine2.grid(row =1)
entr1= Entry(fen01)
entr2= Entry(fen01)
entr1.grid(row =0, column =1)
entr2.grid(row =1, column =1)
bou1=Button(fen01,text='Continuer',command=
bou1.grid(row=2,column=1)
fen01.mainloop()
m1 = float(entr1.get())
m2 = float(entr2.get())
fen01.destroy()
print('Masses lues : ', m1,' et ',m2)
```

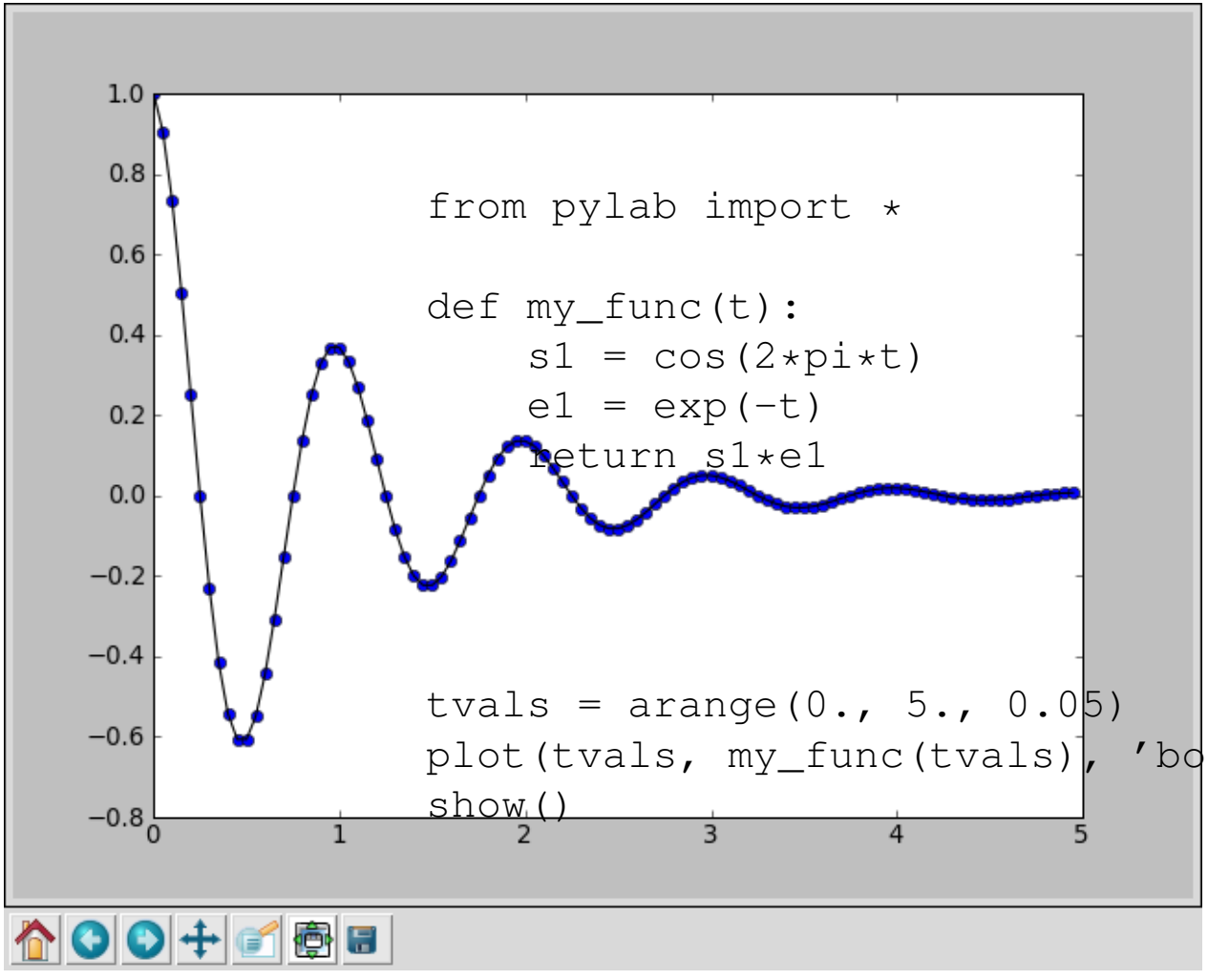
Tkinter : animation

Tkinter : tkDemo, demonstration of Tk widgets

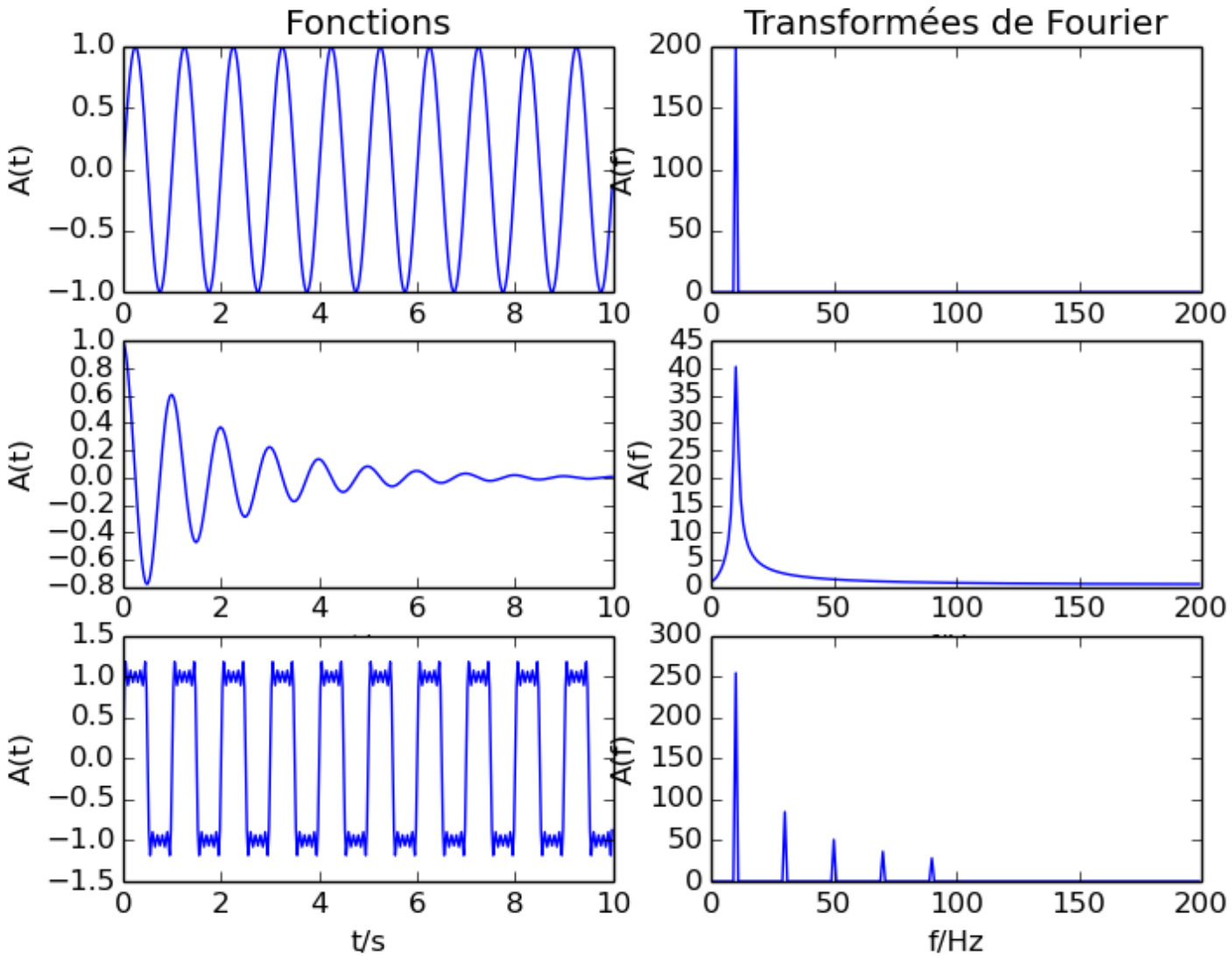


Graphiques simples avec matplotlib

Réf. : http ://matplotlib.sourceforge.net/

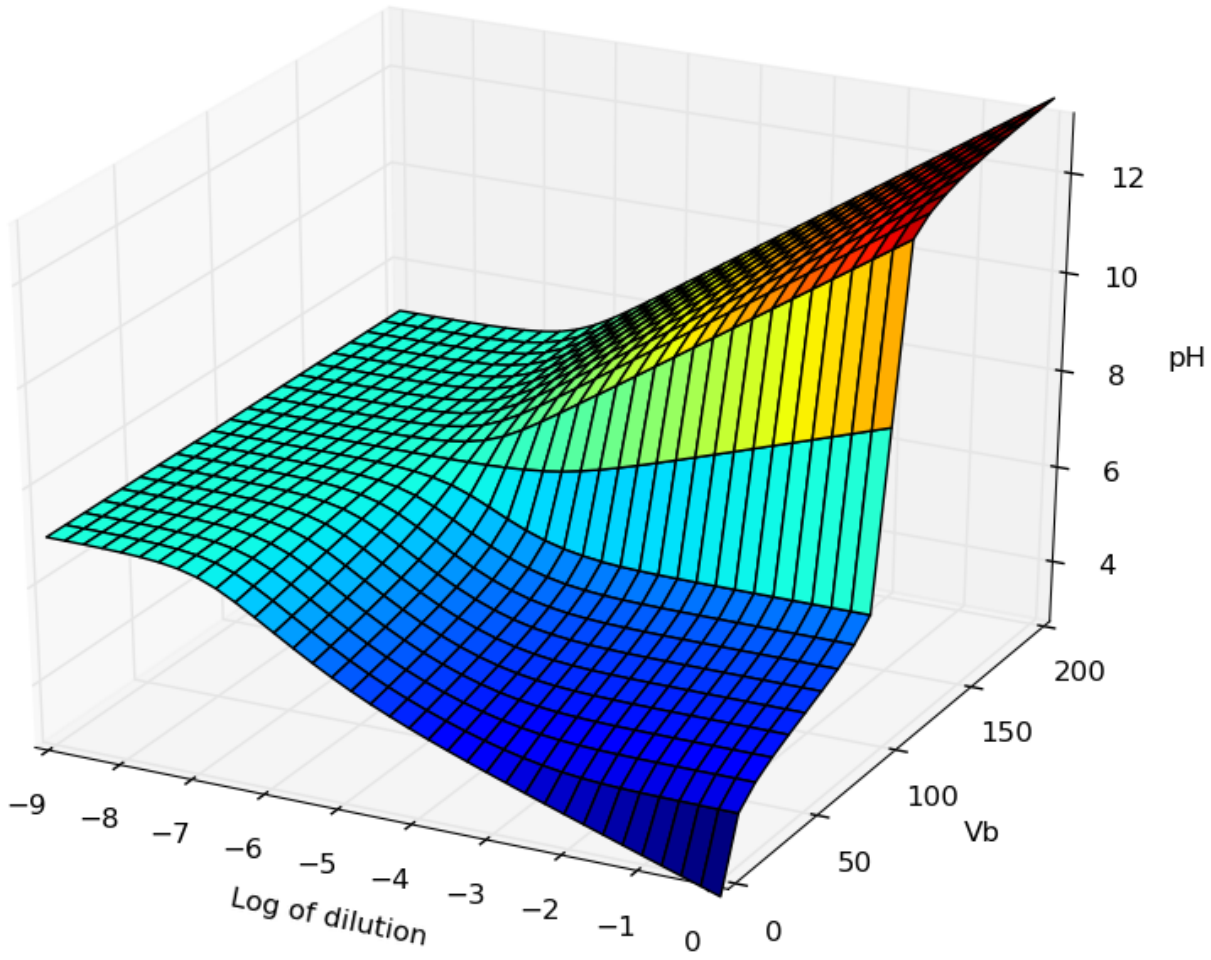


Calculs numériques (FFT) et graphiques plus élaborés



Exemples d'applications

pH d'un acide en fonction d'un ajout de base et d'une dilution



Calculs sur des molécules

```
methanol - Formula =  CH4O
mol wt =  32.04186 -  Numb atoms =  6 -  Numb bonds =  5
12.0107 6 C x= 0.956 y= -0.086 z= -0.056
15.9994 8 O x= 0.488 y= -1.374 z= 0.299
1.00794 1 H x= 0.587 y= 0.64 z= 0.672
1.00794 1 H x= 0.584 y= 0.177 z= -1.05
1.00794 1 H x= 2.049 y= -0.08 z= -0.052
1.00794 1 H x= 0.831 y= -1.996 z= -0.365
partial charges =  (0.28, -0.68, 0.0, 0.0, 0.0, 0.4)
total charge =  0
```

Ces données ont été générées à partir de la chaîne smile 'CO' du méthanol en utilisant des librairies Python existantes et les programmes de chimie OpenBabel

Développer des capacités à programmer

L'apprentissage des rudiments de la programmation vous permettra :

- d'utiliser des petits programmes existants en les modifiant légèrement (niveau élémentaire)
- d'écrire un programme pour solutionner un problème scientifique, en utilisant du code et des librairies existants (niveau normal)
- d'élaborer un programme original pour solutionner un problème scientifique (niveau supérieur)
- Utiliser des techniques de programmation avancées pour solutionner un problème original (niveau excellent)

Quelque soit le niveau de la programmation, les programmes devront respecter les règles d'écriture communément admises.

Apprendre par la pratique

La pratique est la clé de l'apprentissage de tout langage ! Pour atteindre les objectifs, vous procéderez par étape :

- Reproduire quelques programmes très simples pour se familiariser avec le cycle édition-exécution
- Apprendre les bases en suivant le canevas proposé, un manuel/tutoriel, et en effectuant des exercices
- Manipuler les outils d'aide, documenter et commenter
- Apprendre à rechercher et corriger les erreurs
- Rechercher des exemples simples d'applications (scientifiques, mathématiques,...)
- Programmer des problèmes inédits, simples
- Utiliser des sources de codes et documentations diverses : livres, forums, sites web
- Se donner un projet à réaliser, d'envergure adaptée à ses capacités, et le réaliser

Aide en ligne, sites, manuels, fichiers, forums,...

- Aide sur en ligne à partir de Idle, sous windows (touche F1)
- sur python.org
- sur les sites officiels de librairies utilisées
- ...

Références

- Des documents du cours, des exemples et des applications, des réalisations d'étudiants d'années antérieures, des suggestions de travaux sont sur la page : http ://dvillers.umons.ac.be/wiki/teaching :progapp-chim :start
- Des références générales sur Python sont regroupées à la page : http ://dvillers.umons.ac.be/wiki/floss :python
- Se référer également à la page Moodle du cours