Programmer en Python

Généralités

- Qu'est-ce qu'un langage de programmation ?
- Compilation ou interprétation, ou… ?

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 équivalentes aux instructions machines, donc fonction 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
  - Proche d'un langage courant, description procédurale
- Les langages à objets (années 80s et 90s)
  - Briques logicielles indépendantes et autonomes
  - Réutilisations aisées, sans devoir les approfondir
  - Java, C++, Python, perl, Ruby. . .sont les plus connus
- Des langages spécialisés (PHP, SQL,…)

Compilation et compilateur

```
box1=>operation: Code source
→ Compilation
box2=>operation: Code objet
→ Exécution
box3=>operation: Résultat
box1->box2->box3
```

- 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

```
box1=>operation: Code source
```

- Traduction dynamique du code source et

```
  → Interprétation
  box2=>operation: Résultat
  box1->box2
```

exécution immédiate en répétant sans cesse :
    - lecture et analyse d'une instruction
    - 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

Python / Langages à Bytecodes

```
  box1=>operation: Code source
Python (.py)
  → Compilation
  box2=>operation: Python Bytecode
(.pyc)
  → Interprétation
  box3=>operation: Résultat
  box1->box2->box3
```

- Pour Python (et d'autres langages), c'est un peu plus compliqué…
- Le programme est compilé vers un pseudo-code indépendant de l'ordinateur
- Le Bytecode est interprété par la suite
- Avantages :
    - Facilité de développement (cycle écriture-exécution rapide, "briques" logicielles)
    - Portabilité (même programme pour des ordinateurs et OS différents)

Premier aperçu de Python

- Avantages généraux
- Avantages techniques
- Avantages pour l'apprentissage
- Avantages pour le scientifique, le chimiste
- Les premiers pas avec Python

Avantages généraux

- langage de haut niveau (orienté objet)
- permet d'écrire des petits programmes ou suites d'instructions (scripts)
- licence libre (et 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
- 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 pour l'apprentissage

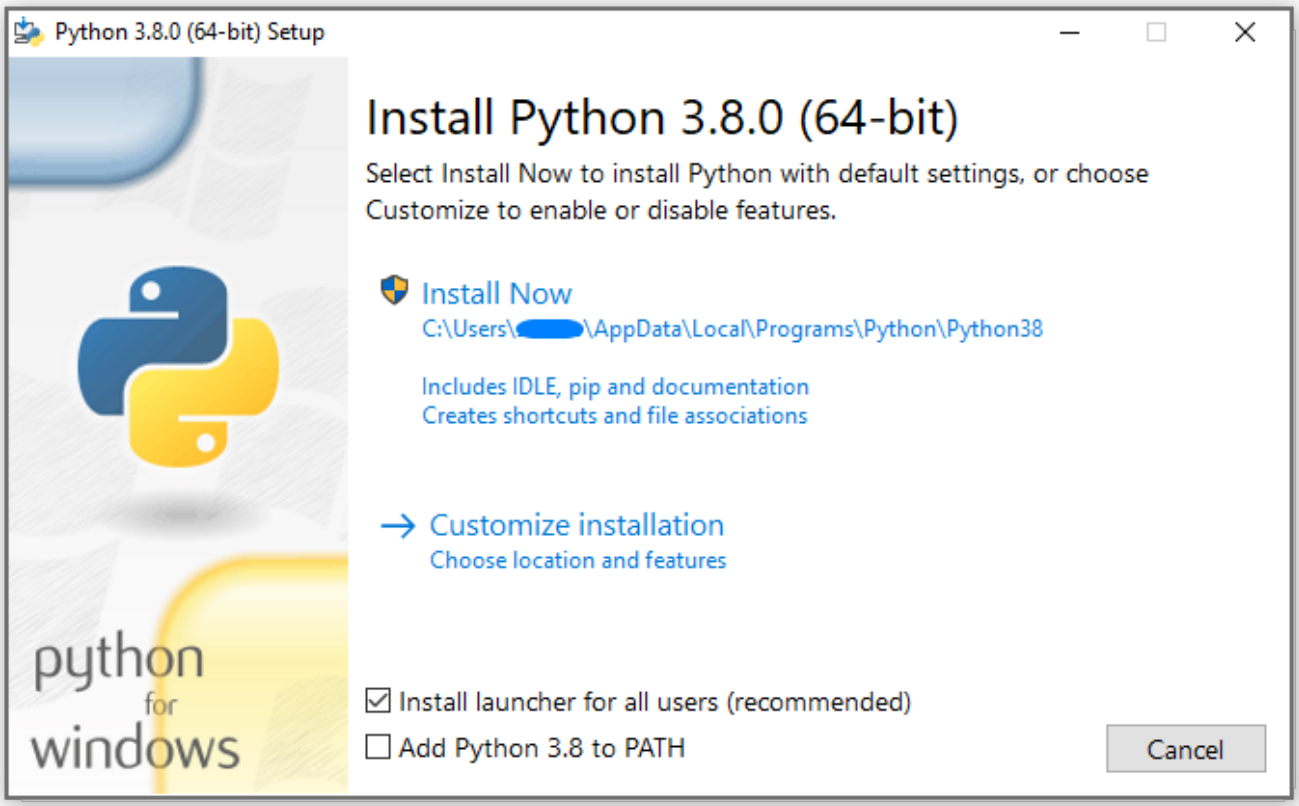
- Installation aisée
    - de la version de base
    - de "distributions" étendues (avec des modules complémentaires)
- éditeur inclus (Idle) ou autre (SciTe, Pycharm, Eric,...)
- mode interactif pour les premiers essais
- principes de base identiques à de nombreux langages
- on n'est pas obligé d'utiliser 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,...)
    - biochimie (biopython)

Les premiers pas avec Python 3

- Sans installation : https://repl.it/languages/python3
- Python Setup and usage



Ídle3 : interface d'exécution et d'édition

Idle3 : interpréteur Python en console, avec exécution directe

```
>>> (8.314*300/24E-3)/101325
1.0256600049346163
>>>
```

- Commandes : copyright, credits, license(), quit, help, help(),...

Notion de variable

On peut attribuer des noms de variables, pas seulement pour des nombres...

```
>>> R=8.314
>>> L=0.001
>>> V=24*L
>>> n=1
>>> zero=273.15
>>> T=20+zero
>>> P=n*R*T/V
>>> atm=101325
>>> print(P,P/atm)
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
5670847980
>>> 123*456
56088
>>> 123**456 ?? A ESSAYER ??
```

- On peut travailler avec des très très très grands nombres…

Division et division entière

```
>>> a =7/3
2.3333333333333335
>>> b = 7//3
2
>>>
```

- En python, chaque “objet” possède son type et un identifiant :
  - type(a)
  - id(a)
  - type(b)
  - id(b)

De nombreuses autres possibilités avec les nombres…

```
>>> Navogadro=6.02214199E23
>>> kboltzmann=1.3806505E-23
>>> print(Navogadro*kboltzmann)
8.31447334956
>>> 2**0.5
1.4142135623730951
>>> (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)
<class 'bool'>
```

- Les tests, comparaisons et leurs combinaisons logiques sont utiles pour réaliser des opérations de manière conditionnelle. Pour la logique booléenne : cf. Algèbre de Boole

Les chaînes de caractères

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

Les chaînes de caractères

```
>>> a=’bonjour’
>>> b="bonjour"
>>> c=’Bonjour’
>>> print(a==b,a==c)
True False
>>> d="pâté123#"
>>> print d
pâté123#
>>> é=d
>>> long="""un
deux
...
dix"""
>>> print(long)
un
deux
...
dix
>>>
```

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'
>>>
```

Créer son premier programme

- Utiliser Idle3 comme éditeur (ou tout autre éditeur)
- Sauvegarder
- Exécuter
- Fermer
- Rouvrir Idle3 et le programme
- Exécuter

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Programme élémentaire en Python
pour afficher une chaîne de caractères
"""
chaine = 'Message : Hello World !'
print(chaine)
```

Un peu plus loin dans Python

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)

```
>>> a1 = [31, 16, 'mot','rouge', 1+3j, [2, 'bleu', 3.14]]
>>> a1
[31, 16, 'mot', 'rouge', (1+3j), [2, 'bleu', 3.14]]
>>> a2 = [121,'vert', 'tomate']
>>> a3=a1+a2
>>> a3
[31, 16, 'mot', 'rouge', (1+3j), [2, 'bleu', 3.14], 121, 'vert', 'tomate']
>>> len(a3)
9
>>> a3[4]
(1+3j)
>>> a3[5]
[2, 'bleu', 3.14]
>>> a3[5][1]
'bleu'
>>> a3[2:6]
['mot', 'rouge', (1+3j), [2, 'bleu', 3.14]]
```

Listes (ter)

```
>>> a3[-1]
'tomate'
>>> a3.pop()
'tomate'
>>> a3
[31, 16, 'mot', 'rouge', (1+3j), [2, 'bleu', 3.14], 121, 'vert']
>>> a3.pop()
'vert'
>>> a3.pop()
121
>>> a4=a3.pop()
>>> a4
[2, 'bleu', 3.14]
>>> a3.append(19.3)
>>> a3
[31, 16, 'mot', 'rouge', (1+3j), 19.3]
```

Dictionnaires

- collections/ensembles non-ordonnées de paires de clés et valeurs

- Chaque clé doit être unique (n'apparaître qu'une fois) 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)

```
>>> d = {'e1': 8, 'e2': 9, 'e4': 11, 'e3': 3, 'e5' : 1}
>>> d.keys()
dict_keys(['e5', 'e3', 'e1', 'e4', 'e2'])
>>> d
{'e5': 1, 'e3': 3, 'e1': 8, 'e4': 11, 'e2': 9}
>>> d['e4']
11
>>> 'e7' in d
False
>>> import operator
>>> print(sorted(d.items(), key=operator.itemgetter(1)))
[('e5', 1), ('e3', 3), ('e1', 8), ('e2', 9), ('e4', 11)]
>>>
```

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

Ensemble (set)

- collection non ordonnée d'éléments non répétés (uniques)
- L'utilisation des ensembles se fait par analogie avec les propriétés et opérations de la théorie mathématique des ensembles : appartenance, cardinalité (nombre d'éléments), union, intersection, différence, ...

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

Structure conditionnelle (exemple)

```
a = int(input('Donnez une note ? '))
if a >= 18:
    print("Excellent")
elif a >= 16:
    print("Très bien")
elif a >= 14:
    print("Bien")
elif a >= 12:
    print("Satisfaisant")
elif a >= 10:
    print("Réussi")
else:
    print("À représenter")
```

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
- **break** permet de quitter prématurément la structure de répétition

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
- **break** permet de quitter prématurément la structure de répétition

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 (utilisation d'espaces)
- 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 f1(x):
    return x**2.

def f3(w):
    print("a ",a)
    return a * w**3.

def f2(x):
    a = 1.111111
    print("a ",a)
    return x**1.5 *f3(x)

a = 2.
u = 9.
print("f1 ",f1(u))
print("f2 ",f2(u))
```

```
print("f3 ",f3(u))
```

Exemple de fonction (2)

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

li=input("Donnez les nombres de moles des constituants (séparés par des virgules)")
print(li,type(li))
n = [float(c) for c in li.split(',')]
print(n,type(n))
print(fractions(n))
```

Utilisation de librairies de fonctions standard

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.cos(0)
1.0
>>> math.__dict__
{'tanh': <built-in function tanh>, 'asin': <built-in function asin>, ...
...
>>> math.__doc__
'This module is always available.  It provides access to the\nmathematical functions defined by the C standard.'
>>> math.__name__
'math'
```

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 → 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 aussi :

- la gestion des erreurs
- des procédures de tests
- la génération de documentation sous différentes formes
- la possibilité d'utiliser des versions spécifiques de librairies
- ...

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

Pour terminer

Exemples d'applications

Simple, avec le module standard Turtle

```
from turtle import *

reset()
i=0
while i<120:
    forward(i)
    left(90)
    i=i+2
input('Hit Enter to continue')
```

Coloré, avec le module standard Turtle

```
from turtle import *

reset()
x=-100
y=-100
i=0
while i < 10:
    j=0
    while j <10:
```

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

input('Hit Enter to continue')
```

Simple, avec l'interface graphique Tkinter

```
from tkinter import *

fen01 = Tk()
fen01.title("Lecture de deux masses")
chaine1 = Label (fen01, text = "introduisez la première masse :")
chaine2 = Label  (fen01, text = "introduisez la deuxième masse :")
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=fen01.quit)
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

anima_auto_rebond.py

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
```

```
# Petit exercice utilisant la librairie graphique Tkinter

from tkinter import *

# définition des gestionnaires
# d'événements :

def move():
    "déplacement de la balle"
    global x1, y1, vx, vy, dt, flag
    x1, y1 = x1 +vx*dt, y1 + vy*dt
    if x1 < 0 or x1 > 220:
        vx=-vx
    if y1 < 0 or y1 > 220:
        vy = -vy
    can1.coords(oval1,x1,y1,x1+30,y1+30)
    if flag >0:
        fen1.after(2,move)     # boucler après 50 millisecondes

def stop_it():
    "arret de l'animation"
    global flag
    flag =0

def start_it():
    "démarrage de l'animation"
    global flag
    if flag ==0: # pour éviter que le bouton ne puisse lancer plusieurs
boucles
       flag =1
       move()

#========== Programme principal =============

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 40, 115       # coordonnées initiales
vx, vy = 10, 5     # vitesse du déplacement
dt=0.1       # pas temporel
flag =0               # commutateur

# Création du widget principal ("parent") :
fen1 = Tk()
fen1.title("Exercice d'animation avec Tkinter")
# création des widgets "enfants" :
can1 = Canvas(fen1,bg='dark grey',height=250, width=250)
can1.pack(side=LEFT, padx =5, pady =5)
oval1 = can1.create_oval(x1, y1, x1+30, y1+30, width=2, fill='red')
bou1 = Button(fen1,text='Quitter', width =8, command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1, text='Démarrer', width =8, command=start_it)
```

```
bou2.pack()
bou3 = Button(fen1, text='Arrêter', width =8, command=stop_it)
bou3.pack()
# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()
```

tkinter : tkDemo, demonstration of Tk widgets

Fix Me! : ? tkdemo était proposé pour la branche 2 de Python. Rechercher pour la version python 3 ?

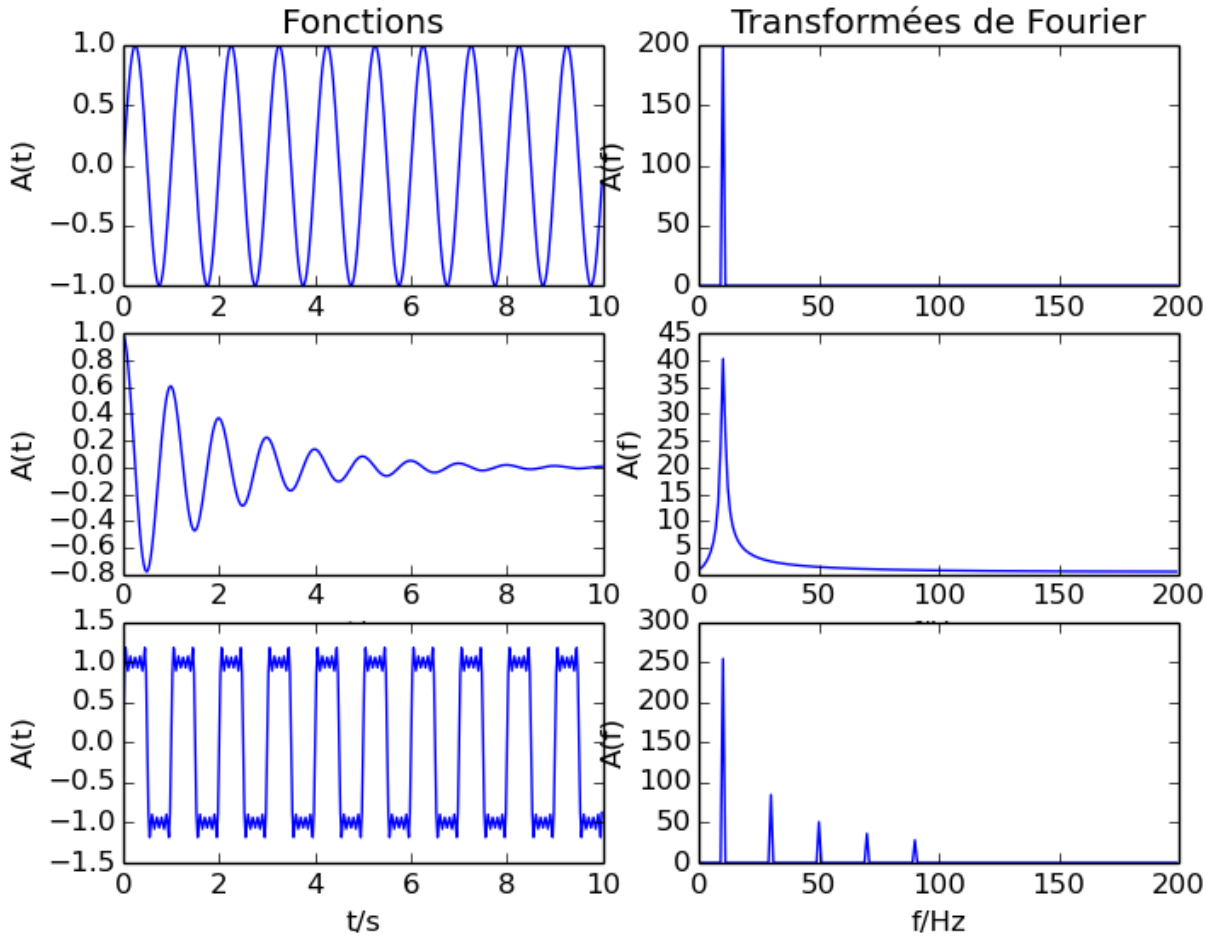
Voir par exemple ce lien

Graphiques simples avec matplotlib

```
# cosinusoïde amortie
from pylab import *

def my_func(t):
    s1 = cos(2*pi*t)
    e1 = exp(-t)
    return s1*e1

tvals = arange(0., 5., 0.05)
#plot(tvals, my_func(tvals))
#show()

plot(tvals, my_func(tvals), 'bo', tvals, my_func(tvals), 'k')
show()
```

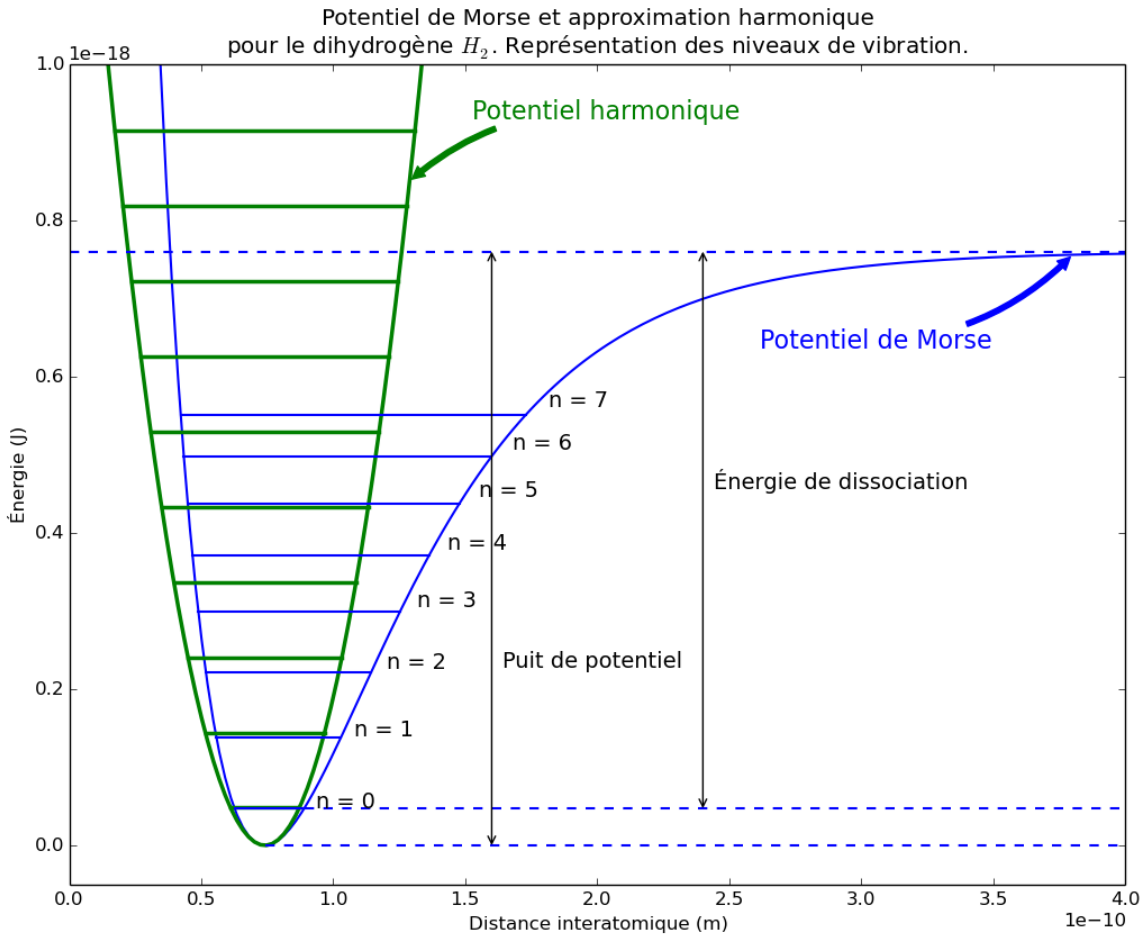
Calculs numériques (FFT) et graphiques plus élaborés

transformees_de_fourier



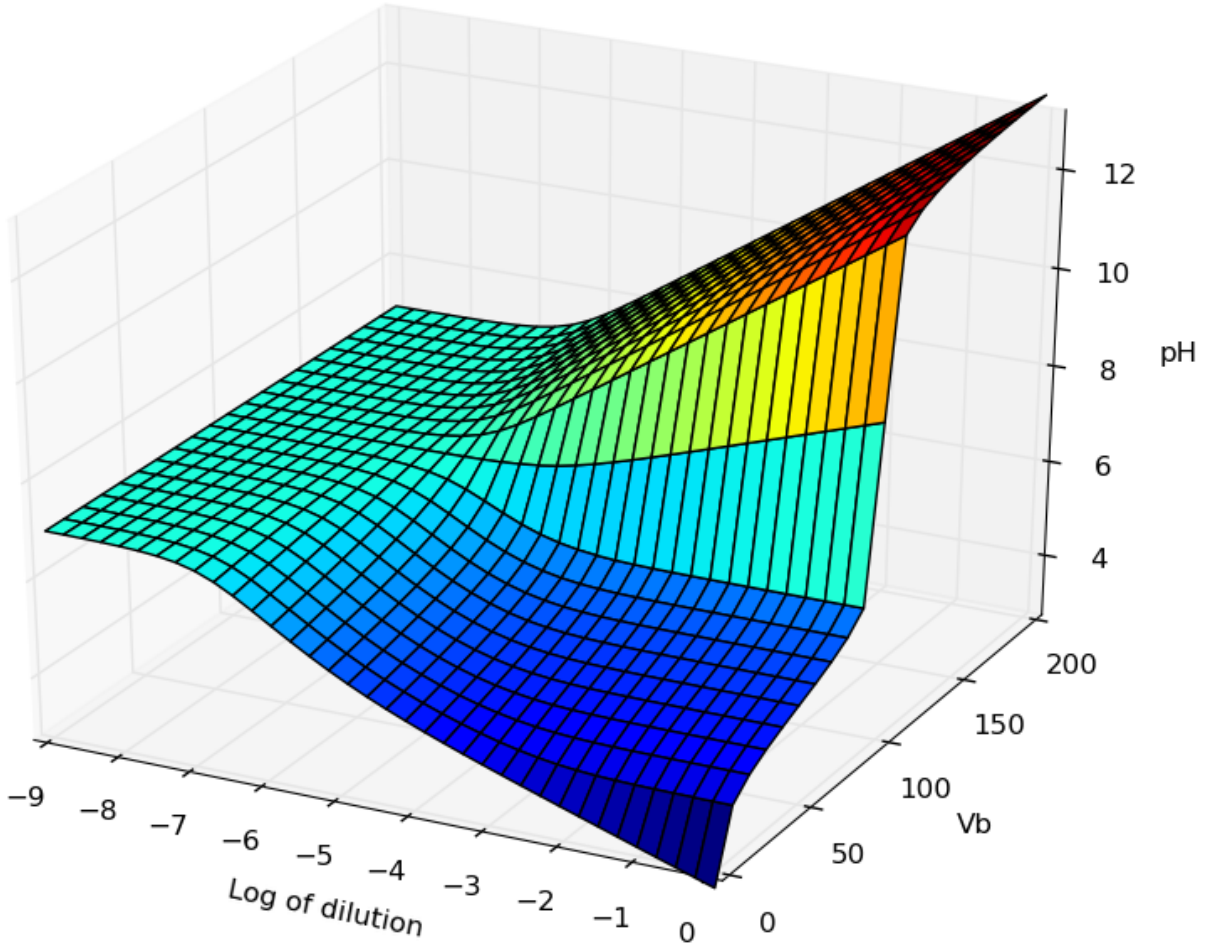
Graphiques et illustrations plus complexes

potentiel_morse



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

ph-3d



Calculs sur des molécules

```
methanol - Formula = CH4O
mol wt = 32.04186 - Numb atoms = 6 - Numb bonds =
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

Objectifs du cours

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 : progappchim
- Des références générales sur Python sont regroupées à la page : python