INFORMATIQUE

J. Mellac

Fevrier 2013

Table des matières

1 Notions de base. **3**
1.1 Architecture d'un ordinateur. 3
1.2 Représentation des nombres en mémoire. 6
1.2.1 Les nombres entiers 6
1.2.2 Les nombres réels 8
1.2.3 Exercices 11
1.2.4 Historique 12

2 Algorithme et programmation. **18**
2.1 Données et variables. 18
2.2 Type d'une variable. 21
2.2.1 Type int 21
2.2.2 Type bool. 22
2.2.3 Nombres flottants. Type float. 22
2.2.4 Type complex. 23
2.2.5 Type str. 23
2.2.6 Type list 26
2.2.7 Type tuple 28
2.3 Instructions composées et script. 29
2.3.1 Instructions de répétition. 29
2.3.2 Instructions de choix. 33
2.4 Fonctions. 36
2.4.1 Fonctions prédéfinies. 36
2.4.2 Fonctions originales. 37
2.5 Structures de données. 43
2.5.1 Compléments : chaines de caractères, listes. 43
2.5.2 Fichiers 46
2.5.2.1 Gestion des fichiers 46
2.5.3 Dictionnaires 49
2.5.4 Ensembles 51
2.5.4.1 Exercices 52

3 Etude de quelques algorithmes.
Notion de complexité algorithmique **57**
3.1 Complexité algorithmique 57
3.2 Exemple : Algorithme d'Euclide 59
3.3 Etude de quelques algorithmes 61
3.3.1 Algorithme de Horner 61
3.3.2 Recherche dans un tableau 62
3.3.2.1 Cas général 62
3.3.2.2 Recherche dichotomique dans un tableau trié 63
3.3.2.3 Recherche d'un mot dans un texte 65

3.3.2.4 Recherche du maximum dans un tableau de nombres . . . 66
3.3.2.5 Moyenne et variance . . . 66
3.4 Résolution d'équations par dichotomie . . . 67
3.5 Calcul approché d'intégrales . . . 71
3.5.1 Méthode des rectangles . . . 72
3.5.2 Méthode des trapèzes . . . 73
3.5.2.1 Exercices . . . 76

4 Calcul numérique 79
4.1 Pivot de Gauss . . . 79
4.1.1 Généralités . . . 79
4.1.2 Exemple . . . 80
4.1.3 Etude générale et problèmes . . . 81
4.1.4 Description de l'algorithme . . . 83
4.1.4.1 Mise en oeuvre de l'algorithme . . . 83
4.2 Résolution d'équation par la méthode de newton . . . 88
4.2.1 Méthode de la sécante . . . 90
4.2.2 Utilisation de Numpy et ou de Scipy . . . 92
4.3 Résolution numérique d'équations différentielles . . . 92
4.3.1 Méthode d'Euler . . . 93

5 Bases de données 99
5.1 Généralités . . . 99
5.2 Algèbre relationnelle . . . 102
5.2.1 Opérateurs ensemblistes . . . 102
5.2.2 Projection . . . 104
5.2.3 Selection ou restriction . . . 104
5.2.4 Jointure . . . 105
5.2.5 Division . . . 105
5.2.6 Renommage . . . 106
5.3 Langage de requête SQL . . . 106
5.3.1 Création d'une table . . . 107
5.3.2 Requêtes . . . 107
5.3.3 Exemples . . . 108
5.3.3.1 Sélection . . . 108
5.3.3.2 Projection . . . 108
5.3.3.3 Union, intersection, différence . . . 109
5.3.3.4 Fonctions d'agrégation . . . 110
5.3.3.5 Jointure . . . 111
5.3.3.6 Clauses GROUP BY ET HAVING . . . 111

6 Numpy : Vecteurs, matrices et tableaux 113
6.1 Notions de base. . . . 113
6.2 Opérations sur les tableaux ou les matrices . . . 117
6.3 Matrices particulières . . . 120

7 Modules scipy et matplotlib 123
7.1 scipy . . . 123
7.2 Tracé de courbes avec matplotlib . . . 124

Chapitre 1

Notions de base.

1.1 Architecture d'un ordinateur.

Carte mère.
C'est un élément important d'un micro-ordinateur, cette carte supporte en particulier le microprocesseur et la mémoire RAM

Microprocesseur : C'est un circuit électronique exécutant les instructions envoyées par un programme. Il est cadencé par une horloge permettant de synchroniser tous les circuits électroniques. La fréquence de l'horloge est appelée fréquence du microprocesseur. Un microprocesseur utilise des registres ou mémoires internes très rapides dans lesquels sont stockées les données ou les instructions qu'il doit traiter.
Taille des registres :

- 8 bits ou 1 octet.
- 16 bits ou 2 octets.
- 32 bits ou 4 octets.
- 64 bits ou 8 octets

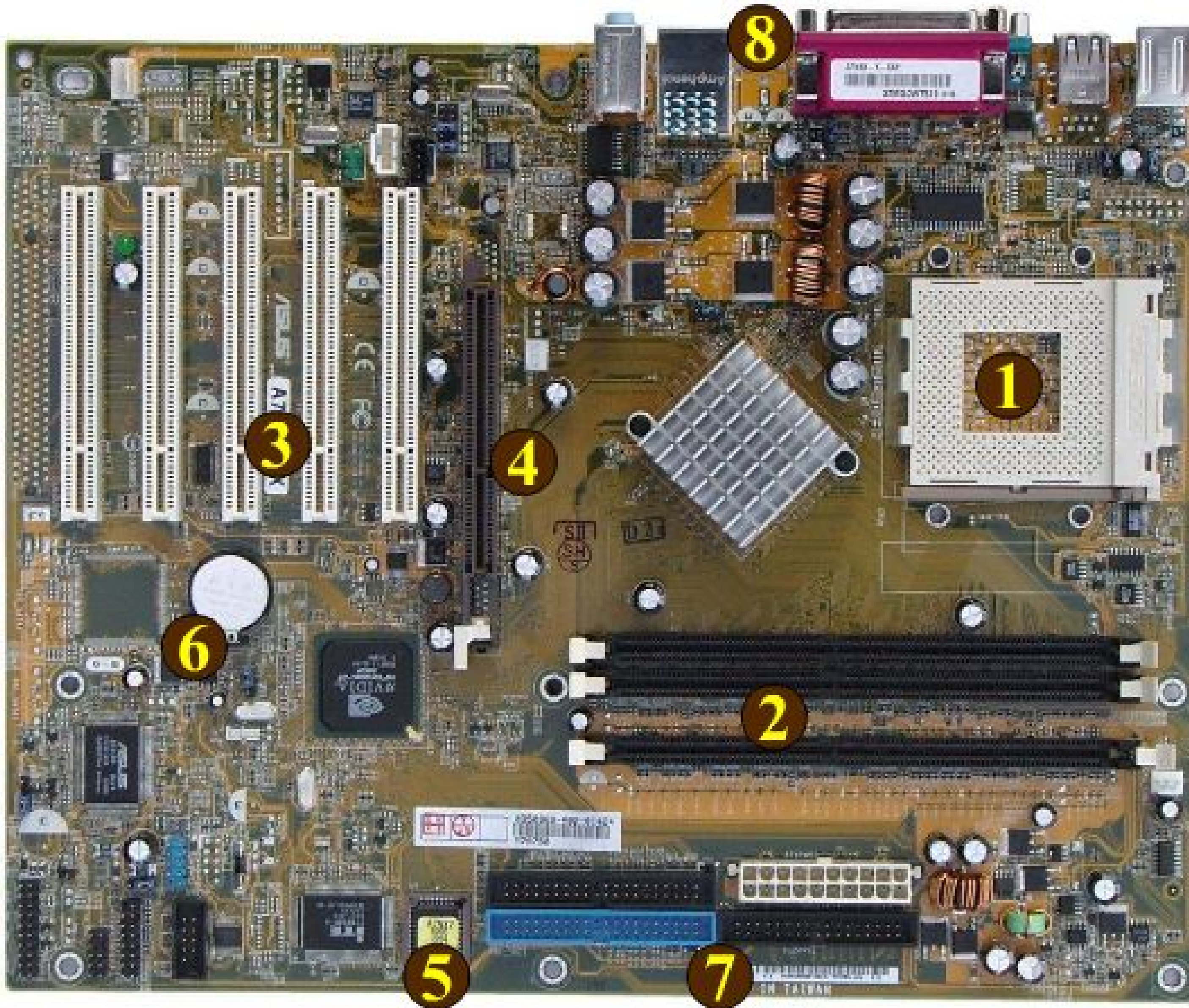
Les micro ordinateurs actuels contiennent des registres de 32 bits ou de 64 bits.

Mémoire RAM : Mémoire volatile dans laquelle le microprocesseur stocke les données dont il a besoin. Sa taille est mesurée en mégaoctets ou en gigaoctets.

- 1 Ko correspond à $2^{10} = 1024$ Octets
- 1 Mo correspond à $2^{20} = 1048576$ Octets
- 1 Go correspond à $2^{30} = 1073741824$ Octets
- 1 To correspond à $2^{40} = 1099511627776$ Octets

Le microprocesseur peut lire ou écrire dans la mémoire, les données étant effacées lorsqu'on éteint l'ordinateur.

Cartes supplémentaires. La carte mère est munie de branchements ou slots permettant l'installation de cartes adaptateurs ou cartes filles. Parmi ces cartes on trouve les cartes sons, cartes réseau, cartes graphiques ...



- 1) Ce gros carré blanc est le socket, c'est-à-dire l'emplacement destiné à accueillir le microprocesseur .
- 2) Emplacements (slots) destinés à accueillir les barrettes de RAM (Random Access Memory).
- 3) Ces grandes barres blanches sont destinées à accueillir divers types de cartes (carte son, modem 56k, carte PCI/Firewire, etc. . .) : ce sont des ports PCI (Peripheral Component Interconnect)
- 4) Port AGP (Accelerated Graphic Port).
- 5) Cette petite puce contient ce qu'on appelle le BIOS (Basic Input/Output System) ; le BIOS est un petit programme qui permet de vérifier que tous les composants nécessaires au démarrage de l'ordinateur sont bien présents ; un ordinateur ne peut démarrer sans BIOS : en effet, c'est le premier programme qui s'exécute lorsque vous allumez votre PC.
- 6) Cette pile plate sert à alimenter le BIOS ; c'est grâce à elle que votre ordinateur retient l'heure même lorsque vous l'éteignez ou le débranchez.
- 7) Le plus gros connecteur sert à l'alimentation de la carte mère. Les deux à gauche sont les ports IDE (primaire et secondaire) : ils permettent de connecter des disques durs et des lecteurs/graveurs de CD/DVD. Le dernier sert à connecter le lecteur disquette.
- 8) On retrouve sur le côté de la carte mère les ports externes du PC.

Unités de stockage

Disques durs. Elément de l'ordinateur disposant d'une grande capacité de stockage des données. Lecteurs de disquettes. Lecteur de CDROM ou de DVDROM. Clés usb. Lecteurs de bandes.

Périphériques d'entrée sortie.

Il s'agit du clavier, de la souris, de l'imprimante, du scanner etc... Le moniteur est lui un périphérique de sortie, sauf les moniteurs à écran tactile, qui interagissent également en entrée.

Démarrage de l'ordinateur

Le démarrage de l'ordinateur est piloté par un logiciel appelé BIOS (Basic Input Output System) permettant d'initialiser les composants de l'ordinateur. Le BIOS est stocké dans une ROM (Read Only Memory) ou mémoire morte ou dans une EEPROM qui est une mémoire morte réinscriptible on dit aussi mémoire flash, elle est analogue a celle que l'on trouve sur une clé usb, elle permet une mise à jour, (flashage) du BIOS.
Depuis l'année 2011 le BIOS est progressivement remplacé par un UEFI : Unified Extensible Firmware Interface (UEFI, "interface micrologicielle extensible unifiée") qui permet un démarrrage rapide de l'ordinateur.

Une fois cette étape franchie le système d'exploitation prend le contrôle de l'ordinateur et effectue la gestion de la mémoire, des entrées sorties, des périphériques. Les systèmes d'exploitations les plus courants sont :
Les unix utilisés surtout sur les gros systèmes et les serveurs web, mais que l'on rencontre maintenant sur les micro ordinateurs :
Solaris (Sun), AIX (IBM) Linux, BSD, MacosX (Apple).
Windows dont la deniére version eest la version 8.

Le système d'exploitation, permet également l'utilisation de logiciels applicatifs qui interagissent avec ce même systeme.

Il gère également les fichiers de tout type à l'aide d'une structure arborescente au sein de laquelle on trouve des répertoires ou dossiers contenant les fichiers. L'accès aux répertoires et aux fichiers est réalisé soit par des logiciels du même type que les navigateurs internet : Explorateur windows ou Nautilus sous Linux, ou directement en ligne de commande sur laquelle les chemins sont du type :

/home/untel/nomfichier
ou

```
\Mesdocuments\fichier$
```

L'accès au fichier est controlè par des droits d'accès. Les permissions peuvent être accordées :
En lecture.
En écriture.
En exécution pour un fichier exécutable.

Les (des) droits peuvent être accordés soit au propriétaire du fichier ou au propriétaire et au groupe d'utilisateurs dont fait partie le propriétaire ou à tous.

Voici une exemple pour un système Unix, les droits sont obtenus par la commande :

```
ls -l nom_de_fichier.
```

```
-rw- r-- --- nom_propri nom_grou
```

-rw- signifie que le fichier est un fichier “normal” (d pour un répertoire) rw- signifie que le propriétaire a les droits de lecture (r) et d'écriture (w) mais pas d'éxécution (-), ce qui est probablement dû au fait que ce fichier n'est pas exécutable. Le groupe a le droit de lecture uniquement et les autres aucun droit. Ces données sont suivies du nom du propriétaire et du nom du groupe.
Ces droits peuvent être modifiés par le propriétaire du fichier ou par l'administrateur du système.
Sous Windows ces indications peuvent être obtenues par un clic droit sur le fichier puis en cliquant sur l'onglet sécurité.

1.2 Représentation des nombres en mémoire.

1.2.1 Les nombres entiers

Les entiers naturels peuvent être écrits dans différentes bases. base 2 ou numérotation binaire, c'est celle qui est utilisée en informatique, base 8 ou octal, base 10 ou décimal qui est la repésentation que l'on utilise, base 16 ou hexdécimal. Cette dernière numérotation nécessite l'utilisation des lettres A, B, C, D, E et F pour repésenter les chiffres (en décimal) 10, 11, 12, 13, 14, et 15 utilisés dans cette base.

Exemple :
le nombre écrit en hexadécimal (base 16) $47C8_{16}$ représente $4*16^3+7*16^2+12*16+8=18376$ en décimal.
Entiers relatifs en binaire :

Les entiers sont codés sur n bits, mais seuls les n-1 premiers représentent le nombre lui même. Le bit le plus à gauche, dit bit de poids fort, représente le signe de l'entier.
Nombres positifs : le bit de poids fort est 0.
Nombres négatifs : Le bit de poids fort est 1.

Il existe différentes techniques pour représenter les nombres négatifs :

Représentation par signe et par valeur absolue.
Le bit de poids fort est à 1
Les autres bits contiennent la valeur absolue du nombre.

Exemple sur un octet :

la valeur d'un entier codé sur un octet peut varier de -127 à 128, (-2^7+1 à 2^7) sur deux octets de -32767 à 32768 (de $-2^{15}+1$ à 2^{15}).

Decimal	Binaire							
12	0	0	0	0	1	1	0	0
-12	1	0	0	0	1	1	0	0

L'inconvénient majeur de cette méthode est la difficulté de réaliser des additions entre entiers de signe contraire. La solution est d'utiliser le complément à 2, qui s'obtient en ajoutant 1 au complément à 1.

Représentation par le complément à 1.
Le complément à 1 consiste à inverser chaque bit.

L'entier positif est représenté sous sa forme naturelle.
L'entier négatif est représenté par le complément à 1.

Exemple sur un octet :

Decimal	Binaire							
12	0	0	0	0	1	1	0	0
-12	1	1	1	1	0	0	1	1

Représentation par le complément à 2.
Le complément à 2, appelé complément vrai, consite à ajouter 1 en binaire au complément à 1.

L'entier positif est représenté sous sa forme naturelle.
L'entier négatif est représenté par le complément à 2.

Exemple sur un octet :

Decimal	Binaire							
12	0	0	0	0	1	1	0	0
C1	1	1	1	1	0	0	1	1
+1								
C2	1	1	1	1	0	1	0	0

On peut remarquer que la somme du nombre et du complément à 2 est égale à 2^n, ce qui donne une somme nulle en ne tenant pas compte des dépassements, c'est à dire de la retenue dans la somme 1+1 des bits de poids forts. En calculant sur n bits le résultat contiendra donc n zéros.

En codant de cette manière sur n bits on peut représenter les entiers relatifs de -2^{n-1} à $2^{n-1}-1$. Si l'entier x est positif ou nul il est représenté en binaire par$x$, s'il est strictement négatif par $x+2^n$.

Exemple :
Calculons $46_d - 24_d = 00101110_2 - 00011000_2$.
Le complément à 1 de 00011000 est :11100111, le complément à 2 est donc 11101000. La somme donne :

Decimal	Binaire							
46	0	0	1	0	1	1	1	0
-24	1	1	1	0	1	0		0
=	0	0	0	1	0	1	1	0

Or $00010110_2 = 2+4+16 = 22_d$.

Une méthode pratique pour obtenir un complément à 2 est de garder tous les chiffres binaires, à patir de la droite, jusqu'au premier 1 puis d'inverser tous les autres.
Exemple :
Le complément à 2 de 01001101 est 10110101
Si un entier est codé sur n bits, tous les bits supplémentaires qui apparaitraient dans un calculs sont ignorés, ce qui amène une erreur de calcul. On dit qu'il y a dépassement de capacité.
Le langagage Python, qui sera présenté ultérieurement, code les entiers sur 32 bits (quatre octets) ou sur 64 bits (huit octects) suivant le type de processeur de l'ordinateur.

Algorithme de passage en base k d'un entier donné en décimal :

Soit $a_i \cdots a_1 a_0$ l'écriture de n en base k, elle est obtenue par l'algorithme suivant :
Données : n, k
$i \leftarrow 0$
tant que n $\neq 0$ faire
a_i reste de la division euclidienne de n par k
n quotient de la division euclidienne de n par k
$i \leftarrow i+1$

Résultat : la représentation $a_i \cdots a_1 a_0$.

Exemple :

Ecrire 76 en base 8.
76=8*9+4 9=8*1+1 1=8*0+1.

76=8*(8*1+1)+4=8*8*1+8*1+4. d'où $76_d = 114_8$. les restes des divisions sont pris en commençant par le dernier vers le premier.

Réciproquement un nombre représenté par $a_i \cdots a_1 a_0$ en base k a pour valeur $a_i * k^i + a_{i-1} * k^{i-1} + \cdots a_1 * k + a_0$ en décimal.

1.2.2 Les nombres réels

Les nombres réels sont en fait représentés par des approximations décimale.

Représentation en virgule flottante.

Cette représentation est basée sur la décomposition sous forme de produit en :

Un facteur représentant les chiffres : la mantisse.
Un facteur représentant la puissance de 2 (binaire) de la partie fixe : l'exposant.

Une représentation en virgule flottante n'est pas unique :
$1,01101 = 101101\ 2^{-5}$. Toutefois cette représentation suit une normalisation.

Forme normalisée :

La représentation standard IEEE 754 en simple précision utilise le bit de poids fort pour le signe, suivi par un exposant sur 8 bits et 23 bits pour la mantisse : La mantisse commencant par 1,... ce premier 1 est omis dans le codage.
Ce codage utilise quatre octets
Décalage de l'exposant
L'exposant peut être positif ou négatif. Cependant, la représentation habituelle des nombres signés (complément à 2) rendrait la comparaison entre les nombres flottants un peu plus difficile. Pour régler ce problème, l'exposant est décalé, afin de le stocker sous forme d'un nombre non signé. Ce décalage est de $2^{e-1} - 1$ (e représente le nombre de bits de l'exposant) ; il s'agit donc d'une valeur constante une fois que le nombre de bits e est fixé.

L'interprétation d'un nombre (autre qu'infini) est donc :

valeur=signe x mantisse $.2^{(exposant - decalage)}$
avec
signe = +1 ou -1 décalage = $2^{e-1} - 1$

1	8	23
Signe	Exposant	Mantisse

Une représentation en simple précision de la forme : $d_1\, d_2 \,..d_{32}$ représente donc le nombre décimal :

$(-1)^{d_1}\ 2^{(d_2 d_3 \cdots d_9)_2}\ 2^{-127}\ (1, d_{10} \cdots d_{32})_2$.

Exemple :

1) Les 32 bits suivants (simple précision) :

1100 0001 1110 0000 0000 0000 0000 0000 représentent

$(-1)^1\, 2^{(10000011)_2}\, 2^{-127}\, (1,11)_2$.

$= -2^{131}.\, 2^{-127}.\, (1 + 2^{-1} + 2^{-2}) = -16.\, 1,75 = -28,0.$

2) Codons le nombre décimal -118,625 en utilisant le mécanisme IEEE 754.

Premièrement, nous avons besoin du signe, de l'exposant et de la partie fractionnaire. C'est un nombre négatif, le signe est donc "1".

Puis nous écrivons le nombre (sans le signe) en binaire. Nous obtenons 1110110,101 (avec divisions par deux successives pour la partie décimale : $0,625 = \frac{1}{2} + \frac{1}{8}$).

Ensuite, nous décalons la virgule vers la gauche, de façon à ne laisser qu'un 1 sur sa gauche : 1110110,101 (bin) = 1,110110101. 2^6. C'est un nombre flottant normalisé : la mantisse est la partie à droite de la virgule, remplie de 0 vers la droite pour obtenir 23 bits. Cela donne 110 1101 0100 0000 0000 0000 (on omet le 1 avant la virgule, qui est implicite).

L'exposant est égal à 6, et nous devons le convertir en binaire et le décaler. Pour le format 32-bit IEEE 754, le décalage est $2^{8-1} - 1 = 127$. Donc 6 + 127 = 133 (dec) = 1000 0101 (bin).
On a donc -118,625 (dec) = 1100 0010 1110 1101 0100 0000 0000 0000 (bin).

Double précision :

En représentant les réels suivant cette norme en double précision on utilise huit octets.

Une représentation en double précision de la forme : $d_1\, d_2\, ..d_{64}$ représente donc le nombre décimal :

$(-1)^{d_1}\, 2^{(d_2 d_3 \cdots d_{12})_2}\, 2^{-1023}\, (1, d_{13} \cdots d_{64})_2$.

On a e=11 d'où la multiplication par par 2^{-1023} car $2^{e-1} - 1 = 1023$.

1	11	52
Signe	Exposant	Mantisse

Exemple :

1100 0100 0110 1001 0011 1100 0011 1000 0000 0000 0000 0000 0000 0000 0000 0000

Le signe est représenté par 1.
L'exposant est représenté par 10001000110.
La mantisse est représentée par
1001001111000011100000000000000000000000000000000000.

Le signe du nombre est -. Le nombre 10001000110 est égal à 1094 et l'exposant du nombre est donc
n = 1094 - 1023 = 71.

Sa mantisse est
m = 1, 1001001111000011100000000000000000000000000000000000

$=1 + 1/2 + 1/2^4 + 1/2^7 + 1/2^8 + 1/2^9 + 1/2^{10} + 1/2^{15} + 1/2^{16} + 1/2^{17} = -\frac{206727}{131072}$.

Erreur de débordement arithmétique de réel.

Un débordement arithmétique de réel survient quand un réel théorique à représenter dépasse le plus grand nombre réel représentable en machine. L'erreur générée est fatale et provoque l'arrêt immédiat du programme.

Exemple : division par 0.

Erreur d'arrondi.

Une erreur d'arrondi survient quand un réel théorique à représenter tombe strictement entre deux nombres réels voisins représentables en machine.

Ces erreurs sont plus sournoises. Elles ne provoquent pas d'erreur d'exécution (ni erreur fatale ni avertissement) mais peuvent conduire, directement ou par propagation, à des résultats erronés.

Exemples.

1) Le nombre $(1,4)_{10}$ est représenté en double précision par :

0 011 1111 1111 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110 0110

Or cette représentation n'est qu'une approximation car le motif 0110 qui apparaît dans la mantisse devrait se répéter à l'infini. L'erreur d'arrondi est donc ici une erreur intrinsèque à la représentation des nombres réels en machine.

La valeur approchée obtenue est : 1, 4000000000000000222044604925031 en double précision.

2) Soient x et y deux réels de valeurs respectives 10^{12} et 10^{-11}.

Expression évaluée	Valeur théorique	valeur effective
$((x - x) + y)/y$	1.0	1.0
$((x + (y - x))/y$	1.0	0.0

Dans la deuxième, l'expression $(y - x)$ esst arrondie à $-x$ lors du calcul, ce qui donne un résultat erroné.

Quelques conseils peuvent être utiles en pratique :

— il ne faut jamais comparer l'égalité stricte de deux réels mais toujours les comparer à ϵ près (par exemple, $|x - y| < 10^{-6}$).

— il faut éviter d'additionner des nombres réels d'ordres de grandeur très différents.

— il faut éviter de soustraire des résultats entachés d'erreurs et ayant des valeurs voisines.

En simple précision :

Plus grand nombre positif pris en compte : 0 11111110 11111111111111111111111 : $3.40282347.\ 10^{38}$.

Plus petit nombre positif pris en compte : 0 00000001 00000000000000000000000 : $1.17549435.\ 10^{-38}$.

En double précision :

Plus petit nombre positif pris en compte : $2.2250738585072020.\ 10^{-308}$.

Plus grand nombre positif pris en compte : $1.797693134862316.\ 10^{308}$.

1.2.3 Exercices

Exercice 1

1) Donner la valeur décimale des entiers suivants, la base dans laquelle ces entiers sont codés étant précisée.

(a) 1011011 et 101010 en binaire
(b) A1BE et C4F3 en hexadécimal (base 16)
(c) 77210 et 31337 en octal (base 8).

2. Coder l'entier 2 397 successivement en base 2, 8 et 16. Comment passe t-on de la base deux à la base 8 puis à la base 16 ?

3. Donner la valeur décimale du nombre 10101, dans le cas où il est codé en base 2, 8 ou 16.

4. Soit un ordinateur dont les mots mémoire sont composés de 32 bits. Cet ordinateur dispose de 4 Mo de mémoire. Un entier étant codé sur un mot, combien de mots cet ordinateur peut-il mémoriser simultanément ?

5. Coder en binaire pur (sans bit de signe) sur un octet les entiers 105 et 21 puis effectuer l'addition binaire des entiers ainsi codés. Vérifier que le résultat sur un octet est correct. Même question avec les entiers 184 et 72.

6. Coder en binaire les entiers 79 et 52 puis effectuer la multiplication binaire des entiers ainsi codés, utiliser le nombre d'octets nécessaires pour réaliser le produit.

Exercice 2

Représentation des entiers relatifs.

1. Donner les intervalles de codage des entiers relatifs sur 8 bits et sur 16 bits. Dans la suite de l'exercice, on travaille sur 8 bits, sauf mention du contraire.
2. Coder les entiers $(+97)_d$ et $(-34)_d$ en complément à 2. Utiliser ce résultat pour calculer en binaire 97 - 34. Vérifier le résultat.
3. Décoder $(00110101)_2$ et $(10110101)_2$.
4. Effectuer les additions en binaire sur huit bits puis sur seize bits :
(a) 0110 1011 + 1011 1101
(b) 0110 1011 + 1111 0000
(c) 1001 0110 + 1011 1011.

Exercice 3 Codage en IEEE 754.

Coder les réels suivants (représentés en base 10) en simple précision :
40
-0.078125
13,625
-87,375

Exercice 4

1) Représenter les nombres hexadécimaux suivant en base deux, en regroupant les chiffres binaires par paquets de quatre.
$41FE8000_{16}$
$3EA80000_{16}$
$C5E00000_{16}$

00380000_{16}

2) Déterminer le nombre décimal repésentés par ces trente deux chiffres binaires dans la norme IEEE 754, dans chaque cas.

Exercice 5

Conversion en virgule flottante IEEE 754 (32 bits)
Quelle est la valeur décimale des représentations binaires suivantes :
a. 1 10000010 11110110000000000000000
b. 0 10000001 11100000000000000000000
c. 1 10000100 00011100000000000000000
d. 0 10000010 11000000000000000000000

Exercice 6

Voici 3 réels représentés dans le format IEEE-754 simple précision (notation hexadécimale) ; Donnez leurs valeurs décimales respectives.
4258 0000
BF30 0000
40B0 0000

1.2.4 Historique

J'inclus une partie du polycopié de Bob Cordeau, auteur d'un très bon cours sur Python.

1.3.2 Bref historique des langages

- Années 50 (approches expérimentales) : FORTRAN, LISP, COBOL, ALGOL…
- Années 60 (langages universels) : PL/1, Simula, Smalltalk, Basic…
- Années 70 (génie logiciel) : C, PASCAL, ADA, MODULA-2…
- Années 80 (programmation objet) : C++, LabView, Eiffel, Perl, VisualBasic…
- Années 90 (langages interprétés objet) : Java, tcl/Tk, Ruby, Python…
- Années 2000 (langages commerciaux propriétaires) : C#, VB.NET…

Des centaines de langages ont été créés, mais l'industrie n'en utilise qu'une minorité.

1.4 Production des programmes

1.4.1 Deux techniques de production des programmes

La compilation est la traduction du source en langage objet. Elle comprend au moins quatre phases (trois phases d'analyse — lexicale, syntaxique et sémantique — et une phase de production de code objet). Pour générer le langage machine il faut encore une phase particulière : l'édition de liens. La compilation est contraignante mais offre au final une grande vitesse d'exécution.

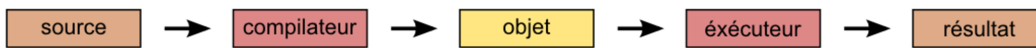


FIGURE 1.1 – Chaîne de compilation

Dans la technique de l'interprétation chaque ligne du source analysé est traduite au fur et à mesure en instructions directement exécutées. Aucun programme objet n'est généré. Cette technique est très souple mais les codes générés sont peu performants : l'interpréteur doit être utilisé à chaque exécution…

FIGURE 1.2 – Technique de l'interprétation

1.4.2 Technique de production de Python

- Technique mixte : l'interprétation du bytecode compilé. Bon compromis entre la facilité de développement et la rapidité d'exécution ;
- le *bytecode* (forme intermédiaire) est portable sur tout ordinateur muni de la machine virtuelle Python.

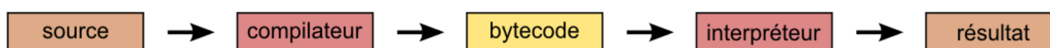


FIGURE 1.3 – Interprétation du bytecode compilé

Pour **exécuter un programme**, Python charge le fichier source `.py` (ou `.pyw`) en mémoire vive, en fait l'analyse (lexicale, syntaxique et sémantique), produit le bytecode et enfin l'exécute.

Afin de ne pas refaire inutilement toute la phase d'analyse et de production, Python sauvegarde le bytecode produit (dans un fichier `.pyo` ou `.pyc`) et recharge simplement le fichier bytecode s'il est plus récent que le fichier source dont il est issu.

En pratique, il n'est pas nécessaire de compiler explicitement un module, Python gère ce mécanisme de façon transparente.

1.4.3 La construction des programmes

Le génie logiciel étudie les méthodes de construction des programmes. Plusieurs modèles sont envisageables, entre autres :

- la méthodologie procédurale. On emploie l'analyse descendante (division des problèmes) et remontante (réutilisation d'un maximum de sous algorithmes). On s'efforce ainsi de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions ;
- la méthodologie objet. Centrée sur les données, elle est considérée plus stable dans le temps et meilleure dans sa conception. On conçoit des fabriques (*classes*) qui servent à produire des composants (*objets*) qui contiennent des données (*attributs*) et des actions (*méthodes*). Les classes dérivent (*héritage* et *polymorphisme*) de classes de base dans une construction hiérarchique.

Python offre les *deux* techniques.

1.5 Algorithme et programme

1.5.1 Définitions

Définition

Algorithme : ensemble des étapes permettant d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.

Un algorithme se termine en un temps fini.

Définition

Programme : un programme est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur.

Il est souvent écrit en plusieurs parties dont une qui *pilote* les autres : le programme principal.

1.5.2 La présentation des programmes

Un programme *source* est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement *commenté*.

La signification de parties non triviales (et uniquement celles-ci) doit être expliquée par un commentaire. En Python, un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne :

```
#---------------------
# Voici un commentaire
#---------------------

9 + 2 # En voici un autre
```

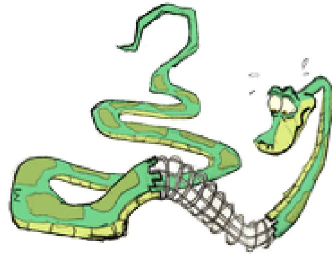
1.6 Les implémentations de Python

- **CPython** : *Classic Python*, codé en C, implémentation [1] portable sur différents systèmes
- **Jython** : ciblé pour la JVM (utilise le bytecode de JAVA)
- **IronPython** : *Python.NET*, écrit en C#, utilise le MSIL (*MicroSoft Intermediate Language*)
- **Stackless Python** : élimine l'utilisation de la pile du langage C (permet de récurser tant que l'on veut)
- **Pypy** : projet de recherche européen d'un interpréteur Python écrit en Python

1. Une implémentation signifie une « mise en œuvre ».

Chapitre 2

La calculatrice Python



Comme tout langage, Python permet de manipuler des données grâce à un *vocabulaire* de mots réservés et grâce à des *types de données* – approximation des ensembles de définition utilisés en mathématique.

Ce chapitre présente les règles de construction des identificateurs, les types de données simples (les conteneurs seront examinés au chapitre 4) ainsi que les types chaîne de caractères (Unicode et binaires).

Enfin, *last but not least*, ce chapitre s'étend sur les notions non triviales de variables, de références d'objet et d'affectation.

2.1 Les modes d'exécution

2.1.1 Les deux modes d'exécution d'un code Python

- Soit on enregistre un ensemble d'instructions Python dans un fichier grâce à un éditeur (on parle alors d'un *script Python*) que l'on exécute par une commande ou par une touche du menu de l'éditeur ;
- soit on utilise l'interpréteur Python embarqué qui exécute la *boucle d'évaluation* (☞ Fig. 2.1).

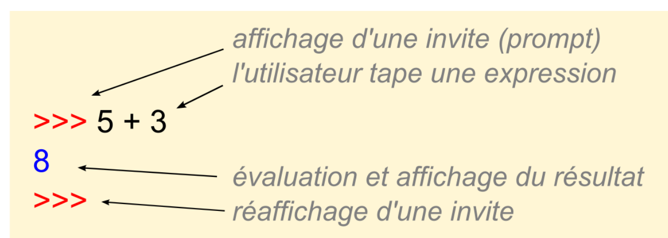


Figure 2.1 – La boucle d'évaluation de l'interpréteur Python

2.2 Identificateurs et mots clés

2.2.1 Identificateurs

Comme tout langage, Python utilise des *identificateurs* pour nommer ses objets.

Définition

Un identificateur Python est une suite non vide de caractères, de longueur quelconque, formée d'**un** *caractère de début* et de **zéro ou plusieurs** *caractères de continuation.*

Sachant que :
- un *caractère de début* peut être n'importe quelle lettre Unicode (*cf.* annexe C, p. 93), ainsi que le caractère souligné (_) ;
- un *caractère de continuation* est un caractère de début, un chiffre ou un point.

Attention

Les identificateurs sont sensibles à la casse et ne doivent pas être un mot réservé de Python 3.

2.2.2 Style de nommage

Il est important d'utiliser une politique cohérente de nommage des identificateurs. Voici le style utilisé dans ce document [1] :
- `UPPERCASE` ou `UPPER_CASE` pour les constantes ;
- `TitleCase` pour les classes ;
- `UneExceptionError` pour les exceptions ;
- `camelCase` pour les fonctions et les méthodes ;
- `unmodule_m` pour les modules ;
- `lowercase` ou `lower_case` pour tous les autres identificateurs.

Exemples :

```
NB_ITEMS = 12        # UPPER_CASE
class MaClasse: pass # TitleCase
def maFonction(): pass # camelCase
mon_id = 5           # lower_case
```

Pour ne pas prêter à confusion, éviter d'utiliser les caractères `l` (minuscule), `O` et `I` (majuscules) seuls.
Enfin, on évitera d'utiliser les notations suivantes :

```
_xxx    # usage interne
__xxx   # attribut de classe
__xxx__ # nom spécial réservé
```

2.2.3 Les mots réservés de Python 3

La version 3.3.0 de Python compte 33 mots clés :

`and`	`del`	`from`	`None`	`True`
`as`	`elif`	`global`	`nonlocal`	`try`
`assert`	`else`	`if`	`not`	`while`
`break`	`except`	`import`	`or`	`with`
`class`	`False`	`in`	`pass`	`yield`
`continue`	`finally`	`is`	`raise`	
`def`	`for`	`lambda`	`return`	

2.3 Notion d'expression

Définition

Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.
Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux représentant directement des valeurs, d'identificateurs et d'opérateurs.

Exemples de deux expressions simples et d'une expression complexe :

```
>>> id1 = 15.3
>>>
>>> id2 = maFonction(id1)
>>>
>>> if id2 > 0:
...     id3 = math.sqrt(id2)
... else:
...     id4 = id1 - 5.5*id2
```

1. Voir les détails dans la PEP 8 : « Style Guide for Python », Guido van Rossum et Barry Warsaw

2.4 Les types de données entiers

Python 3 offre deux types entiers standard : `int` et `bool`.

2.4.1 Le type `int`

Le type `int` n'est limité en taille que par la mémoire de la machine [1].

Les entiers littéraux sont décimaux par défaut, mais on peut aussi utiliser les bases suivantes (*cf.* annexe D, p. 95) :

```
>>> 2013       # décimal
2013
>>> 0b11111011101 # binaire
2013
>>> 0o3735     # octal
2013
>>> 0x7dd      # hexadecimal
2013
```

Opérations arithmétiques

Les principales opérations :

```
>>> 20 + 3
23
>>> 20 - 3
17
>>> 20 * 3
60
>>> 20 ** 3
8000
>>> 20 / 3
6.666666666666667
>>> 20 // 3
6 (division entière)
>>> 20 % 3 # modulo
2
>>> abs(3 - 20) # valeur absolue
17
```

Bien remarquer le rôle des deux opérateurs de division :

/ : produit une division flottante, même entre deux entiers ;

// : produit une division entière.

Bases usuelles

Un entier écrit en base 10 (par exemple `179`) peut se représenter en binaire, octal et hexadécimal en utilisant les syntaxes suivantes :

```
>>> 0b10110011 # binaire
179
>>> bin(179)
'0b10110011'
>>> 0o263  # octal
179
>>> oct(179)
'0o263'
>>> 0xB3   # hexadécimal
179
>>> hex(179)
'0xb3'
```

1. Dans la plupart des autres langages les entiers sont codés sur un nombre fixe de bits et ont un domaine de définition limité auquel il convient de faire attention.

Chapitre 2

Algorithme et programmation.

2.1 Données et variables.

Algorithme.

C'est une procédure de calcul bien définie, qui prend en entrée une valeur ou un ensemble de valeurs et qui donne en sortie une valeur ou un ensemble de valeurs. C'est donc une séquence d'étapes de calculs permettant d'obtenir un résultat en fonction des données d'entrées.

Exemple : Algorithme d'Euclide qui prend en entrée deux entiers et qui donne en sortie le pgcd de ces entiers.

Un algorithme peut être décrit dans le langage naturel ou sous forme d'un pseudo code indépendant de tout langage informatique.

L'étape suivante consiste à traduire cet algorithme sous forme de programme informatique, dépendant du langage utilisé. Le langage Python a été retenu.

Le langage Python présente la particularité de pouvoir être utilisé soit en mode interactif, directement depuis le clavier, soit en écrivant des programmes (ou scripts) qui seront sauvegardés sur disque et pourront être exécutés. Ceci présente l'avantage de pouvoir vérifier ligne par ligne des instructions qui pourront être ensuite intégrées dans un script.

Exemple :
En mode interactif, l'interpréteur présente un prompt ou signal d'invite constitué de trois caractère >>>. après écriture de l'instruction la touche "Entrée" permet d'obtenir le résultat.

```
>>> 5+7
12
```

Un programme d'ordinateur manipule des données qui peuvent être très diverses, mais dans la mémoire elle se ramènent à une suite de nombres binaires. Pour pouvoir accéder aux données on fait usage de variables de différents types.

Définitions.

Variable :Une variable est un identifiant associé à une valeur.
Informatiquement, c'est une référence d'objet située à une adresse mémoire.
Un nom de variable peut contenir un ou des chiffres mais doit commencer par une lettre. Les lettres accentuées ainsi que les caractères spéciaux sont interdits à l'exception du signe "souligné".

Affectation : On affecte une variable par une valeur en utilisant le signe = (qui n'a rien à voir avec l'égalité en math !). Dans une affectation, le membre de gauche reçoit le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche.
On utilise également le terme assigner une variable.

Exemples : a=2, i=i+1
La dernière affectation montre que la valeur d'une variable peut évoluer au cours du temps et elle n'a rien à voir avec l'égalité en math .

Affectation multiple.

Avec Python on peut affecter plusieurs variables simultanément :

```
>>> x=y=8
>>> x
8
>>> y
8
>>> # Affectations a l'aide d'un seul operateur:
>>> a,b=11,12
>>> a
11
>>> b
12
>>> #Echange de valeur sans variable tampon:
>>> b,a=a,b
>>> b
11
>>> a
12
>>>
```

Les identifiants ou noms utilisés pour les variables (mais également plus tard pour les fonctions) distinguent les majuscules des minuscules et ne doivent pas être choisis parmi les mots clés du langage.

```
 La version 3 de Python compte 33 mots cles :

and    del    from     None     True
as    elif    global   nonlocal try
assert else   if       not      while
break  except import   or       with
class  False     in    pass     yield
continue finally is    raise
def      for
```

Expression : Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.
Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux, d'identifiants d'opérateurs et d'instructions, qui sont les éléments de base du langage.
Une instruction informatique désigne une étape dans un programme informatique. Une instruction dicte à l'ordinateur l'action nécessaire qu'il doit effectuer avant de passer à léinstruction suivante. Un programme informatique est constitué d'une suite d'instructions. Exemple : instruction d'affectation.

Exemples :

a=2
b=sqrt(3)

sont deux expressions simples.

```
 if (c>=0) :
   d=sqrt(c) \\
 else:
   d=sqrt(-c) \\

est une expression compos\'ee. \\
```

Afficher la valeur d'une variable :

```
 >>> pi=3.14
>>> v=pi*pi
>>> pi
3.14
>>> v
9.8596
>>> print(pi)
3.14
>>> print(v)
9.8596
>>>
```

L'exemple ci-dessus fait intervenir deux variables, pi et v. Pour les afficher on peut entrer au clavier le nom de la variable, suivi de la touche “entrée” ou entrer print(nom de la variable) suivi de la touche “entrée”. toutefois la deuxième méthode sera nécessaire dans le cas d'une utilisation au sein d'un script.

2.2 Type d'une variable.

Les variables sont classées par type. Certains langage de programmation (C, Pascal) nécessite de déclarer le type d'une variable avant de pouvoir l'utiliser. Ce n'est pas le cas de Python qui attribue automatiquement un type à une variable au moment de son affectation.

2.2.1 Type int

Le type int représente les entiers relatifs. La taille de ces entiers n'est limitée que par la mémoire de la machine, contrairement à d'autres langages, comme mentionné précédemment.

Exemple :

Voici la factorielle de 100, qui contient 156 chiffres.

```
  >>> j
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639
76156518286253697920827223758251185210916864000000000000000000000000
>>> k=str(j)
>>> len(k)
156
>>> type(j)
<class 'int'>
```

Les entiers sont représentés en décimal par défaut, mais les bases deux, huit ou seize peuvent être utilisées.

```
>>> 2009 # decimal
2009
>>> 0b11111011001 # binaire
2009
>>> 0o3731 # octal
2009
>>> 0x7d9 # hexadecimal
2009
```

Opération arithmétiques sur les entiers.

- Addition : +
- Soustraction : -
- Multiplication : *
- Elevation à une puissance : ** exemple a**b représente a puissance b
- qutient dans la division euclidienne : // Il est à noter que la version 2 de Python utilisait l'instruction /
- reste dans la division euclidienne :
  `%`
- Valeur approchée d'un quotient : /
- Valeur absolue : abs.

Remarque :La divison euclidienne a un sens pour les entiers relatifs comme pour les entiers naturels. $(a,b) \in \mathbb{Z}\times\mathbb{N}^*$, $\exists!(q,r) \in \mathbb{Z}\times\mathbb{N}$ tel que $0 \leq r < b$. Le reste r est donc toujours positif ou nul.

```
20+ 3 # 23
20 - 3 # 17
20 * 3 # 60
20 ** 3 # 8000
```

```
20 / 3 # 6.666666666666667
20 // 3 # 6 (division entiere)
20 % 3 # 2 (modulo)
abs(3 - 20) # valeur absolue
Bien remarquer le role des deux operateurs de division :
/ : produit une division flottante ;
// : produit une division entiere.
```

2.2.2 Type bool.

Le type booléen peut prendre deux valeurs : 'FALSE' (faux) ou 'TRUE' (vrai).

```
>>> 2>8
False
>>> 6<12
True
```

Ce type est utilsé pour tester la véracité d'une condition avec les opérateurs de comparaison :

== (identique à),
!= (différent de),
> < (Strictement plus grand ou strictement plus petit),
>= <= (supérieur ou égal ou inférieur ou égal).
On trouve aussi les opérateurs logiques : and, not, or

```
>>> (4==4) or (8>45)
True
>>> (4==4) and (8>45)
```

la proposition (a and b) est vraie si et seulement si les deux le sont.
la proposition (a or b) est vraie si et seulement si une des l'est. not a prend la valeur opposée de a.

2.2.3 Nombres flottants. Type float.

Le type float permet de représenter les réels suivant la norme IEEE 754 présentée précédemment.

Les nombres de ce type supportent les mêmes opérations que les entiers à part les opérations de division entière.

Dans le cas d'une opération entre entier et flottant, le nombre entier est converti en flottant et le résultat est un flottant.

Il existe un autre type représentant les entiers, noté Decimal mais qui fait partie d'un module, ou bibliothèque, ce type est représenté par une fonction. Il s'git d'un ensemble de fonctions ou de procédures qui ne sont pas chargées en mémoire au démarrage de Python. Ce type Decimal permet des calculs plus précis sur les réels :

```
>>> 1.1+2.2
3.3000000000000003
>>> 0.1+0.1-0.2
0.0
>>> 0.1+0.1+0.1-0.3
5.551115123125783e-17
>>> from decimal import Decimal
```

```
>>> Decimal('1.1')+Decimal('2.2')
Decimal('3.3')
>>> Decimal('0.1')+Decimal('0.1')+Decimal('0.1')-Decimal('0.3')
Decimal('0.0')
```

Les trois premiers calculs sont faits sur des nombres flottants (de type float), on constate qu'il s'agit de valeurs approchées pour le premier et le troisème calcul. L'introduction de la fonction Decimal de la bibliothèque decimal permet d'obtenir un résultat exact.

2.2.4 Type complex.

Les complexes sont écrits en notation cartésienne formée de deux flottants. La partie imaginaire est suffixée par j (le i des maths).

```
print(1j) # 1j

print((2+3j) + (4-7j)) # (6-4j)

print((9+5j).real) # 9.0

print((9+5j).imag) # 5.0

print((abs(3+4j))) # 5.0 : module
```

Un module mathématique leur est réservé : cmath.

```
 >>> from cmath import *

>>> phase(1+1j)

0.7853981633974483

>>> polar(1+1j)

(1.4142135623730951, 0.7853981633974483)
```

l'instruction *phase* donne un argument en radians du nombre complexe, l'instruction *polar* donne le module et un argument de ce même nombre.

2.2.5 Type str.

Ce type représente une suite de caractères ou chaine de caractères codée dans la norme utf-8, dite norme unicode, dans la version 3 de Python. Cette norme est conçue pour coder l'ensemble des caractères internationaux. Trois syntaxes de chaines sont disponibles :

```
>>> phrase1="aimez vous les oeufs"

>>> phrase2=' "oui" repondit il'
```

```
>>> phrase3='avec de la mayonnaise'

>>> print(phrase1,phrase2,phrase3)

aimez vous les oeufs  "oui" repondit il avec de la mayonnaise
```

Une chaine peut être indifféremment encadrée de guillemets ou d'apostrophes. Toutefois si elle contient des apostrophes elle doit être entourée de guillemets et si elle contient des guillemets elle doit être entourée de d'apostrophes.

On peut écrire une chaine sur plusieurs lignes en utilisant antislash suivi de n.

Si l'on veut conserver une chaine de caractères inchangée, dans sa forme comme dans ses caractères spéciaux on l'encadre par un triple guillemets ou un triple apostrophes.

```
>>> phrase4="ligne1 \n ligne2"
>>> print(phrase4)

ligne1
 ligne2

phrase5="""Voici une phrase
                       biscornue et etrange et ce caractere \ et "
                             ainsi que cette lettre @ """

 print(phrase5)
Voici une phrase
                       biscornue et etrange et ce caractere \ et "
                             ainsi que cette lettre @
```

Opérations sur les chaines.

Concaténation : assembler plusieurs chaines entre elles. Ceci s'obtient par l'opérateur +

```
  phrase6="j'aime"
>>> phrase7="les homards"

>>> a=phrase6+phrase7
>>> print(a)
j'aimeles homards
```

Noter l'absence d'espace entre ces deux chaines de caractères.

Répétition : on répète une chaine p par p*n où n est le nombre de répétitions souhaitées.

```
>>> p="perroquet "
>>> p*4
'perroquet perroquet perroquet perroquet '
```

L'espace est pris en compte dans la définition de p.

On peut agir sur une chaine en utilisant des fonctions (notion procédurale) ou des méthodes (notion objet).

```
>>> p2="fonction"
>>> #Application de la fonction longueur
>>> len(p2)
8

>>> #Methode: passer en majuscules
>>> p4=p2.upper()
>>> p4
'FONCTION'
```

une fonction s'utilise comme en maths : f(objet).

Une méthode s'utilise en ajoutant à l'objet .methode() sous la forme objet. methode().

On peut accéder aux caractères d'une chaine : ch[i] représente le i+1 ième caractère de la chaine ch. La numérotation commence à 0. ch[0] est le premier caractère de la chaine ch.

```
>>> p5="chaine de caracteres"
>>> p5[0]
'c'
>>> p5[5]
'e'
>>> len(p5)
20
>>> p5[19]
's'
>>> p5[6]
' '

>>> p5[4]=v
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    p5[4]=v
NameError: name 'v' is not defined
```

Le septième caractère (p5[6]) est un espace, le vingtième et dernier caractère est p5[19].

Il est impossible de modifier une chaine de caractère en modifiant un ou des caractères comme le montre la tentative de modification p5[4]=v qui fait apparitre un message d'erreur.
Ceci est dû au fait que c'est un objet non mutable ou immutable, notion définie ci dessous.

Sous chaines.
Une sous chaine est un ensemble de caractères consécutifs à l'intérieur d'une chaine. Un espace est considéré comme un caractère.
Exemple :

s='je cherche mon stylo'
L'instruction s[i :j] permet d'obtenir les caractères compris entre la position i et la position j-1. La position i correspond au i+1 ième caractère car les caractères de la chaine sont numérotés à partir de zèro.

s[1 :5] donne
'e ch'

Cette chaine contient les caractère de 1 à 4.

s[3 :12] donne 'cherche m'
Cette chaine contient les caractères de 3 à 11.

L'opérateur **in** permet de v'érifier la présence d'un caractère au sein d'une chaine.

'e' in s
True

'u' in s
False

Il est possible de transformer un nombre en chaine de caractère :

str(1.6)
'1.6'

Ou de réaliser la transformation inverse :

int('2')
2
float('2.8')
2.8

2.2.6 Type list

Sous Python, une liste est une collection d'objets séparés par des virgules, de types quelconques, l'ensemble étant encadré par des crochets.
L'opération l+m appliquée aux listes l et m crée une nouvelle liste concaténant les éléments de l et de m.
L'opération l*n crée une nouvelle liste dans laquelle l est répétée n fois.
len(l) donne le nombre d'éléments de la liste.
L'instruction l[i] permet d'accéder au i+1 ième élément de la liste. Contrairement aux chaines de caractère ceci permet également de modifier la valeur de l[i]. Une liste est un objet mutable. On peut supprimer cet élément par la fonction del(). La méthode append permet d'ajouter un élément en fin de liste.

```
>>> l=[1,2,3,1.2,5,'chaine']
>>> m=[6,7]
>>> l+m
[1, 2, 3, 1.2, 5, 'chaine', 6, 7]
>>> m*4
[6, 7, 6, 7, 6, 7, 6, 7]
>>> l.append(15)
>>> l
[1, 2, 3, 1.2, 5, 'chaine', 15]
>>> del(l[4])
>>> l
[1, 2, 3, 1.2, 'chaine', 15]
>>> len(l)
6
```

Remarque : Dans le langage Python le passage d'une variable à un programme se fait par référence.
Une référence est une adresse qui indique où la donnée se trouve en mémoire. quand vous écrivez $a = 2$ Vous

n'assignez pas la valeur 2 à la variable "a", vous assignez une référence vers la valeur, donc une d'adresse qui indique où elle se trouve en mémoire.
Et quand vous faites : print a
Python va chercher dans "a" la référence, et retrouver l'élément 2 en suivant "l'adresse".
C'est important car Quand vous faites ceci :

a = b

Vous ne copiez pas l'élément. Vous copiez la référence. Du coup, on ne prend pas deux fois la place en mémoire, et la copie est très rapide.

L'assignation par référence n'a vraiment d'importance que dans le cas où un objet est mutable. En Python, il existe en effet deux types d'objets : les mutables (listes, dictionnaires, sets, objets custo, etc) et les nons mutables (strings, int, floats, tuples, etc).
Les mutables sont ceux qu'on peut modifier après leur création. Les non mutables sont ceux qu'on ne peut pas modifier après création.

Cela peut créer quelques ambiguites comme le montre l'exemple suivant faisant intervenir des listes.

```
 l=[1,2,3]
>>> m=l
>>> l
[1, 2, 3]
>>> m
[1, 2, 3]
>>> l.append(4)
>>> #Ce qui precede signifie que l'on ajoute 4 a la liste l
>>> l
[1, 2, 3, 4]
>>> m
[1, 2, 3, 4]
>>> # cet element a egalement ete rajout\'e a m
```

L'exemple ci dessus montre que toute modification de l est appliquée à m. m est un alias de l, elles sont référencées à la même adresse mémoire, il s'agit du même objet. Ceci est dû au fait qu'une liste est un objet mutable.

Ce n'est plus le cas pour une chaine de caractères, qui est un objet non mutable :

```
 ch="chaine1"

>>> ch2=ch

>>> ch

'chaine1'

>>> ch2

'chaine1'

>>> ch="modification"
```

```
>>> ch

'modification'

>>> ch2

'chaine1'
```

Dans le deuxième cas, la modification de la valeur de ch crée une nouvelle variable, c'est à dire que l'adresse mémoire est modifiée et la précédente est supprimée.

Dans le cas des listes une modification de la valeur de la liste ne modifie pas son référencement dans la mémoire.

Il est possible de convertir une chaine de caractères en liste par l'instruction **list** :

list(s)
['j', 'e', ' ', 'c', 'h', 'e', 'r', 'c', 'h', 'e', ' ', 'm', 'o', 'n', ' ', 's', 't', 'y', 'l', 'o']
Il est à noter que le caractère "espace" figure dans la liste obtenue.
L'instruction **in** est utilisable pour une liste.

2.2.7 Type tuple

un **tuple** ou **n-uplet** est constitué par des éléments encadrés par des parenthèses et séparés par des virgules. Ces éléments peuvent être de type différents.

('plis',2,[1,2,3])
('plis', 2, [1, 2, 3])

Un tuple peut être représenté par une variable et l'accés à ses éléments se fait comme pour une chaine de caractères ou pour une liste.
L'instruction **in** est utilisable également pour un tuple.

t=('plis',2,[1,2,3])
t[0]
'plis'
2 in t
True

Un tuple peut être converti en liste :

list(t)
['plis', 2, [1, 2, 3]]

L'opération inverse est possible :

l=[1,2,3,4]
tuple(l)
(1, 2, 3, 4)
ch='Bonjour'
tuple(ch)

('B', 'o', 'n', 'j', 'o', 'u', 'r')

2.3 Instructions composées et script.

Un script ou programme Python est une suite d'instructions exécutées séquentiellement, qui seront regroupées dans un fichier dont le nom se termine par .py : nomfichier.py. Ce fichier pourra être sauvegardé sur disque dur, cle usb ou tout autre support. Voici un exemple, représentant un calcul de pgcd d'entiers par l'algorithme d'Euclide. Le script est appelé euclide.py.

euclide.py

```
a, b=input("entrez deux entiers separes par une virgule ")
a1, b1=int(a),int(b)
while (b1!=0):
   r=a1 % b1
   a1=b1
   b1=r
print("le pgcd est ",a1)
```

On peut remarquer l'indentation des instructions qui suivent l'instruction while. Il faut remarquer que dans un même bloc d'exécution le niveau d'indentation doit être le même.

```
 while (b1!=0):
   r=a1 % b1
   a1=b1
   b1=r
```

conduirait à un message d'erreur.

Nous commenterons ce script ultérieurement au fur et à mesure de l'introduction de nouvelles instructions.

2.3.1 Instructions de répétition.

Boucle while.

L'instruction **while** ci dessus amorce une instruction composée. Le double point à la suite de la condition booléenne ($b1! = 0$) ($b1$ différent de zéro) introduit le bloc d'instructions à répéter, lequel doit se trouver en retrait (ou indenté). toutes les instructions de ce bloc doivent avoir exactement le même niveau d'indentation.
A chaque entrée dans la boucle la condition $b1! = 0$ est évaluée et la boucle est parcourue si la condition est réalisée.

Si la condition est fausse, la boucle est ignorée et la dernière instruction, **print("le pgcd est ",a1)**, est exécutée.
On peut remarquer que la boucle while peut ne jamais être parcourue si la condition est fausse dès le départ.
Le bloc d'exécution qui suit l'instruction **while** fait les opérations suivantes :
Calcul du reste dans la division de a1 par b1 puis affectatation de ce résultat à la variable r.
affectation de la valeur de b1 dans la variable a1.
affectation de la valeur de r dans la variable b1, qui est donc strictement décroissante à chaque étape.
sortie de la boucle lorsque r et donc b1 atteint la valeur 0.
C'est la décroissance stricte de $b1$ qui assure que la boucle se termine.

Voici un deuxième script qui calcule les dix premiers termes de la suite de Fibonacci définie par $u_0 = u_1 = 1,\ u_{n+2} = u_{n+1} + u_n,\ n \in \mathbb{N}$.

fibo.py

```
a,b,c=1,1,1
while (c<11):
   print(b,end=" ")
   a,b,c=b,a+b,c+1
```

l'expression **end=" "** permet d'afficher les termes de la suite sur une même ligne, séparés par un espace. Sans cette instruction les termes seraient écrits les uns en dessous des autres.

Dans ce deuxième script, c'est la variable **11 -c** qui est strictement décroissante et qui assure que la boucle se termine.

Écrire un programme utilisant une boucle while

1 On identifie la condition de la boucle ; il est souvent plus commode de chercher une condition de sortie de la boucle, puis de calculer sa négation ou tout simplement d'utiliser l'opérateur not.
2 On écrit le corps de la boucle, en s'assurant que celui-ci modifiera la valeur de la condition à certaines itérations.
3 On prévoit une initialisation des variables en amont de la boucle.
4 Il est parfois nécessaire de faire un dernier traitement à la suite de la boucle ; dans tous les cas, on n'oubliera pas de revenir au niveau d'indentation du while.

Démontrer qu'une boucle se termine effectivement

On identifie un variant, autrement dit une expression (c'est souvent le simple contenu d'une variable)
• qui est un entier positif tout au long de la boucle,
• et qui diminue strictement après chaque itération.
On peut alors en conclure que la boucle termine.

Invariant de boucle

Lorsque l'on a écrit un programme, il reste à vérifier qu'il est correct, c'est-à-dire qu'il calcule bien ce qu'on attend ; on peut tester quelques cas significatifs, mais il est beaucoup plus satisfaisant de démontrer qu'il est correct dans tous les cas. Une des manières les plus efficaces de le faire est d'établir un invariant de boucle, c'est-à-dire une propriété qui est vérifiée tout au long de l'exécution d'une boucle. Ce genre de raisonnement est à rapprocher du raisonnement par récurrence : en ne s'intéressant qu'aux valeurs initiales des variables et à leur évolution au cours d'une seule itération, on peut en déduire des propriétés valides quel que soit le nombre d'itérations.

En résumé :

On utilise une invariant de boucle, c'est-à-dire une propriété
• qui est vérifiée avant d'entrer dans la boucle,
• qui si elle est vérifiée avant une itération est vérifiée après celle-ci,
• qui lorsqu'elle est vérifiée en sortie de boucle permet d'en déduire que le programme est correct.

Exemple : Algorithme itératif de calcul de la factorielle.

```
n=input('entrez un entier naturel ')
n=int(n)
i=1
j=1
while (i <= n) :
    j=j*i
    i=i+1
print('la factorielle de ',n,' est ',j)
```

Un invariant de boucle est $j = (i-1)!$ on peut vérifier la boucle à l'aide de cet invariant :

Au démarrage du script, $j = 1 = (i-1)! = 0!$
Lorsqu'on entre dans la boucle avec la valeur i on a $j = (i-1)*(i-2)*\cdots*1 = (i-1)!$
A la sortie de la boucle on a $j = i*j = i!$
Lorsque $i = n+1$ la boucle n'est plus parcourue, on a $j = n! = (i-1)!$. On purrait vérifier que la boucle s'arrête en remarquant que la variable **n-i** est strictement décroissante.

Un invariant de boucle possible pour l'algorithme d'euclide présenté précedemment serait :
les diviseurs communs de a_1 et b_1 sont ceux de a et b
C'est vrai avant la première entrée dans la boucle, a chaque étape, entrée et sortie de la boucle et en fin de boucle.
Or le pgcd est le plus grand de ces éléments, ce qui vérifie l'algorithme.
La boucle s'arrête car $a1$ est strictement décroissant.

Boucle for.

La boucle **for** permet de parcourir tout itérable (liste, chaine de caractère,tuple) élément par élément.

```
>>> for lettre in "trucmuche":
            print(lettre+"*",end="")



t*r*u*c*m*u*c*h*e*
>>> l=[1,2,3,4,5,6]
>>> for e in l:
        print(e,end="  ")



1  2  3  4  5  6
```

Vous remarquerez le même principe d'indentation.

Une boucle **for** est utile quand on connait par avance le nombre d'itératons nécessaires, c'est une boucle inconditionnelle. dans le cas de l'algorithme d'euclide ce nombre d'itérations est inconnu et la boucle **while** est indispensable, ce ne serait pas le cas dans le calcul de la factorielle.

Reprenons ce calcul en utilisant une l'itérable **range** . Il peut être utilisé de différentes manières :
range(n) fait intervenir les entiers de 0 à $n-1$.
range(m,n) fait intervenir les entiers de m à $n-1$ avec $m < n$.
range(m,n,h) fait intervenir les entiers de m à $n-1$ par pas de h, c'est à dire

$m, m+h, m+2h, \cdots, m+kh$. Dans ce dernier cas m peut être strictement supérieur à n si $h < 0$.

L'instruction **list(range(n))** construit la liste des entiers de 0 à $n-1$, même type de construction dans les autres cas d'utilisation de **range**.

list(range(8))
[0, 1, 2, 3, 4, 5, 6, 7]
list(range(6,12))
[6, 7, 8, 9, 10, 11]
list(range(12,1,-1))
[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2]

Utilisation avec la boucle **for**

factorielle2.py

n=int(input('entrez un entier naturel n '))
p=1
for i in range(n) :
p=p*(i+1)
print('la factorielle de ',n,' est : ',p)

entrez un entier naturel n 6
la factorielle de 6 est : 720

Un invariant de boucle serait ici :
p=i ! C'est vrai au début de la boucle car $0! = 1$.
Si $p = i!$ à l'entrée de la boucle, à la sortie $p = i! \times (i+1) = (i+1)!$
La dernière valeur de la boucle étant $n-1$ on a $p = (n-1)! \times n = n!$.
Ce script calcule donc bien la factorielle de l'entier naturel n.

Sortir d'une boucle.

Interrompre une boucle : **break**. Lorsqu'une boucle **for** ou **while** contient break, le programme continue à la suite des instruction de cette boucle.

Court circuiter une boucle : **continue**. Passe immédiatement à l'itération suivante de la boucle **for** ou **while**.

```
for i in range(1,11):
      if i==8:
          break
      print(i,end=" ")
print("\nBoucle interrompue pour i=",i)

1 2 3 4 5 6 7
Boucle interrompue pour i= 8

for i in range(1,11):
       if i==8:
          continue
      print(i,end=" ")
print("\nLa Boucle  a saute la valeur 8")
```

```
1 2 3 4 5 6 7 9 10
La Boucle  a saute la valeur 8
```

Remarquer les différents niveaux d'indentation, ainsi que l'utilisation de l'instruction **if**.

2.3.2 Instructions de choix.

syntaxe : **if (condition) :**

ou

if (condition) :

elif condition :

else :
Les instructions **elif** et **else** sont optionnelles et il peut y avoir plusieurs **elif** successifs.

Exemple. Résolution d'une équation du second degré avec $a \neq 0$.

```
 a,b,c=input('entrez a b et c avec a non nul ')
a,b,c=int(a),int(b),int(c)
delta=b**2-4*a*c

if delta>0:
    d=sqrt(delta)
    print((-b-d)/(2*a),(-b+d)/(2*a))
elif delta<0:

    print('pas de racines reelles')
else:

    print('x='-b/(2*a))
```

Etant donné trois nombres a, b, c que fait le script suivant :

```
 if a>b:
       if a>c:
          m=a
       else:
         m=c
else:
    if b>c:
       m=b
    else:
       m=c
```

<u>Instructions d'entrées sorties</u>

Dans un langage de programmation, on appelle entrées-sorties les constructions qui permettent d'interrompre le flot d'exécution du programme pour communiquer avec l'utilisateur, donnant ainsi un aspect interactif au programme.

Pour entrer une donnée on utilise l'instruction **input**, rencontrée dans **euclide.py** qui a pour effet d'interrompre le programme et d'attendre l'introduction d'une donnée.
De même, il existe une instruction un peu à part : l'instruction print, qui permet d'afficher à l'écran la ou les expressions qui lui sont données en argument. Elle ne manipule donc pas l'état au sens où elle ne modifie pas les contenus des variables, mais seulement l'aspect de l'écran. On s'en sert souvent pour afficher des chaînes de caractères :

print("Bonjour")
Bonjour

mais elle permet d'afficher des expressions plus complexes :

b=4
print('somme= ',b+6)
somme= 10

a=input('entrez un entier ')
entrez un entier 45
a
'45'

Il faut remarquer que le type de **a** est une chaine de caractère. Ce sera toujours le cas avec l'instruction **input**, dans le cas présent il serait donc nécessaire de convertir **a** en entier par l'instruction **a=int(a)**. Il est à remarquer que dans la version 2 de Python l'instruction est **raw_input** et non **input**.

Documenter un programme

Dès que l'on écrit un programme de plus d'une dizaine de lignes, il devient indispensable de le rendre parfaitement lisible, pour permettre à un autre programmeur de le comprendre, ou pour pouvoir soi-même reprendre son programme plus tard. Deux éléments clés de lisibilité dont il faut prendre l'habitude très tôt sont :
- choisir des noms de variables parlants ;
- ajouter des commentaires dans les programmes, autrement dit des lignes écrites en langue naturelle que la machine ne cherche pas à interpréter comme des instructions et qui expliquent le rôle des différentes parties du programme.
Pour préciser qu'une ligne est un commentaire, on la fait précéder d'un symbole particulier. En Python, il s'agit d'un dièse #. Si l'on veut écrire un commentaire sur plusieurs lignes, il faut faire précéder chacune d'entre elles de ce symbole ou entourer le commentaire de trois guillemets doubles """. Ces commentaires doivent donner des informations supplémentaires sur le sens du programme. Inutile de commenter en disant par exemple « ceci est une boucle », mais expliquer plutôt le rôle de cette boucle.

Equation réduite d'une droite :

L'équation cartésienne d'une droite dans le plan est de la forme : $ax + by + c = 0$, où a et b ne sont pas simultanément nuls, par exemple $2x - 3y + 6 = 0$. Si b n'est pas nul la droite a un coefficient directeur qui est $-\frac{a}{b}$, sinon l'équation réduite de la droite est $x = -\frac{c}{a}$, voici un script qui reprend ces éléments.

```
#Equation cartesienne d'une droite.
if b==0:
```

```
    #Droite parallele a l'axe des ordonnees
    if a!=0:
        #Sinon ce n'est pas une droite
        print("x= ",-c/a)
else:
    #Cas general
    cofdir=-a/b
    ordori=-c/b
    print("y= ",cofdir,"x + ",ordori)
```

Mettre un programme au point en le testant

Pour vérifier si un programme ne produit pas d'erreur au cours de son exécution et s'il effectue réellement la tâche que l'on attend de lui, une première méthode consiste à exécuter plusieurs fois ce programme, en lui fournissant des entrées, appelées tests, qui permettent de détecter les erreurs éventuelles. Pour qu'elles jouent leur rôle, il faut choisir ces entrées de sorte que :
- on sache quelle doit être la sortie correcte du programme avec chacune de ces entrées ;
- chaque cas distinct d'exécution du programme soit parcouru avec au moins un choix d'entrées ;
- les cas limites soient essayés : par exemple les nombres nuls ou négatifs, la liste vide ...

2.4 Fonctions.

2.4.1 Fonctions prédéfinies.

Ce sont les fonctions directement utilisables, par exemples **print** ou **input** que l'on a déjà rencontrées, ou des fonctions qui nécessitent l'importation d'un module de fonctions. Ces fonctions ne sont pas directement accessibles dans le système pour éviter de surcharger inutilement la mémoire. Il est possible d'importer tout le module, en fait on importe seulement les noms ou références des éléments du module, qui contient parfois plusieurs centaines d'éléments,ou simplement les fonctions utilisées dans le calcul. Les exemples qui suivent montrent les différentes utilisations possibles de ces modules.

```
import math #importation de l'ensemble du module
>>> math.sqrt(25)
5.0
>>>a= math.sin(3.14/4)

>>> a
0.706825181105366
```

Cette deuxieme methode importe egalement l'ensemble du module mais l'utilisation des fonctions est simplifiee. Il est inutile de faire preceder le nom de la fonction du nom du module.

```
from math import *
>>> sqrt(25)
5.0
>>> log(exp(5))
5.0
>>> sin(asin(0.5))
0.5
```

Dans la derniere methode on n'importe que les fonctions utilisees.

```
 from math import sqrt,sin,exp,asin
>>> sqrt(25)
5.0
>>> asin(sin(0.8))
0.8
>>> exp(1)
2.718281828459045

>>> cos(0.5)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'cos' is not defined
>>>
 Dans le dernier calcul le message d'erreur provient du fait que la fonction n'a pas ete importee.
```

On rencontrera d'autres modules utiles. **numpy** qui permet des manipulations de tableaux, **matplotlib** qui permet des tracés de courbes, **scipy** qui apporte des fonctions pour le calcul scientifique avancé et **sympy** qui permet de faire du calcul formel à partir de Python.

La fonction **input** a été introduite auparavant. Les variables définies par cette fonction sont considérées comme des chaines de caractères.

```
from math import sqrt
>>> nombre=input('entrez un nombre')
entrez un nombre 36
>>> nombre=int(nombre)
>>> sqrt(nombre)
6.0

from math import sqrt
>>> nombre=input('entrez un nombre ')
entrez un nombre 36
>>> sqrt(nombre)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: a float is required
>>>
```

Dans le deuxième exemple le nombre est considéré comme une chaine de caractères, ce qui explique le message d'erreur.

2.4.2 Fonctions originales.

Ce sont les fonctions que vous définirez vous même et qui viendront s'ajouter aux instructions existantes du langage.

Syntaxe :

```
def nomfonction(parametres):
"Documentation de la fonction" (Optionnel mais recommande).

Bloc d'instructions

return resultat # peut etre inexistant
```

Exemples.

```
def cube(x):
...     "Calcule le cube d'un nombre"
...     return x**3
...
>>> cube(2)
8
>>> cube(3)
27

def volume_sphere(r):
...     "Calcule le volume de la sphere de rayon r"
...     return 4*3.14*cube(r)/3
...
>>> volume_sphere(2)
33.49333333333333
```

Ces deux exemples montrent qu'une fonction (ici cube) peut être utilisée au sein d'une autre fonction si elle a été définie avant cette fonction (ici volume sphere). Lorsque la fonction ne contient pas l'instruction **return** on parle parfois de procédure.

Exemple.

```
def table(base):
...     "Table de multiplication par base"
...     n=1
...     while n<11:
...        print(n*base, end=' ')
...        n=n+1
...
>>> table(4)
4 8 12 16 20 24 28 32 36 40 >>>
```

Dans les exemples précédents il n'y a qu'un paramètre, on dit aussi argument, c'est ce que l'on appelle en général une variable en Mathématiques mais il peut y en avoir plusieurs ou aucun. Le nom attribué à ce(s) paramètre(s) ne doit pas contenir de lettre accentuée. Le nom de la variable passée comme paramètre n'a rien à voir avec le nom du paramètre correspondant dans la fonction.

Exemple de fonction utilisant plusieurs paramètres.

```
def carres(x,y):
...     "Somme de carres"
...     return x**2+y**2
...
>>> carres(3,4)
25
```

Exemple de fonction utilisant un nombre inconnu de paramètres.

```
def moyenne(*args):
    "Calcul de la moyenne d'un nombre quelconque d'entiers"
    s=0
    for n in args:
       s=s+n
    return s/n

print(moyenne(1,2,3,4,5,6))
3.5
```

Variables locales, variables globales.

Les variables utilisées à l'intérieur du corps d'une fonction (y compris les arguments d'appel) ne sont accessibles qu'à la fonction elle même. On dit que ce sont des variables locales. C'est le cas de **base** et de **n** dans la procédure table ou des arguments **x et y** dans la fonction carres.

Exemple.

```
 p,q=15,38
>>> def affiche():
...     p=20
...     print(p,q)
...
>>> affiche()
20 38
>>> p,q
(15, 38)
```

Cet exemple montre que la variable p ne prend la valeur 20 qu'à lintérieur de la procédure **affiche**, lorsqu'on appelle p en dehors de cette procédure, sa valeur est celle qui lui a été affectée précédemment. On dit que les variables p et q définies dans cet exmple, en dehors de **affiche**, sont des variables globales. Le nom p utilisé dans **affiche** et celui défini au dessus sont deux objets distincts, . La procédure **affiche** ne modifie pas p, c'est une valeur attribuée localement qui est affichée.

Néanmoins si on veut qu'une fonction modifie définitivement la valeur d'une variable, pas seulement au sein de cette fonction il suffit d'ajouter l'instruction global dans le corps de la fonction.

Exemple.

```
p,q=15,20
def affiche2():
...     global p
...     p=20
...     print(p,q)
...
>>> affiche2()
20 20
>>> p,q
(20, 20)
```

Cette fois la modification de p dans la fonction se retrouve en dehors de cette fonction.

Les paramètres d'appel ou arguments d'une fonction se comportent comme des variables locales :

Exemple

a=4

def f(x) : x=x**2 return x
f(a)
16

a
4

Les règles de passage des paramètres sont les suivantes :

— Le passage des paramètres se fait par référence (ce n'est qu'un cas particulier d'affectation après tout). Les paramètres des fonctions sont des références aux objets référencés par l'appelant, et par conséquent :
  les objets peuvent être partagés entre l'appelant et la fonction appelée.
  Si les références pointent des objets modifiables [mutables], une modification dans l'appelé affecte l'appelant.
— L'affectation à des noms de paramètres n'affecte pas l'appelant ; les noms des paramètres sont locaux à la fonction.

Exemple

```
def func(x, y):
    x    = 27              # la valeur locale de x est changée
    y[0] = 'foofoo'        # impact sur la référence partagée


x = 1
y = [2, 3, 5, 7]

func(x, y)                 # x est non mutable, mais y l'est

print( x, y)

# 1 ['foofoo', 3, 5, 7]
```

La valeur de x n'est pas modifiée mais celle de y l'est.

Il arrive que l'utilisation d'une fonction soit limitée à la définition d'une autre fonction, on peut alors la définir comme fonction locale, elle ne pourra pas être utilisée en dehors du corps de la fonction enveloppante.

Exemple
Cette fonction détermine le maximum de trois nombres en commençant par déterminer le maximum des deux premiers :

```
def max3(x, y, z):
   def max2(u, v):
       if u > v:
          return u
       else:
          return v \\
  return max2(x, max2(y, z))
```

L'utilisation de variables globales au sein du corps d'une fonction est à manipuler avec précaution et sauf execption il est préférable de ne faire appel qu'aux variables locales.

Une fonction peut s'appeler elle même, on parle alors de récursivité.
Exemple : Factorielle.

Version récursive.

```
def fact(n):
 "Calcul de factorielle"
if n==1:
return 1
else:
return n*fact(n-1)
```

Version itérative.

```
def fac(n):
 "Calcul de factorielle"
v=1
for i in range(1,n+1):

   v=v*i
return v
```

Un argument d'appel d'une fonction peut être une autre fonction.

Exemple 1

Calcul de $\sum_{i=0}^{n} f(i)$ pour une fonction quelconque.

```
def somme_fonction(f, n):
    s = 0
   for i in range(n+1):
       s = s + f(i)
   return s
```

Appliquons ceci à la fonction carré, c'est à dire au calcul de

$$\sum_{i=0}^{n} i^2$$

```
 def carre(x):
   return x*x
```

```
puis on la passe en argument à la fonction somme_fonction :
```

```
somme_fonction(carre, 10)
385
```

Exemple 2 : Résolution approchée d'équation par dichomomie.

```
def f(x):
...     return x**3+x+1

def dicho(f,a,b,e):
    while b-a>e:
        c=(b+a)/2
        if f(a)*f(c)<0:
           b=c
        else:
           a=c
    print(a," est une solution a ",e," pres")

dicho(f,-1,0,10e-2)
-0.6875  est une solution a  0.1  pres
```

Ce denier algorithme de calcul approché d'équation par dichomomie sera étudié ultérieurement.

Remarque :

Il est possible d'utiliser une fonction sans la définir explicitement en utilisant la fonction lambda de Python : $lambda\ x : x**2$ définit la fonction carré.
On pourrait écrire :
somme_fonction(lambda x :x**2,10) en utlisant cette fonction. Les fonctions lambda ne peuvent pas contenir de commandes et elles ne peuvent contenir plus d'une expression.

Une fonction peut renvoyer une fonction comme résultat.

```
def expo(f):
    def h(x):
        return exp(f(x))
    return h
```

Cette fonction définit la fonction $x \mapsto exp(f(x))$ où exp représente la fonction exponentielle. Ainsi si $f = \ln$ cette fonction est la fonction identité $x \mapsto x$.

a=expo(lambda x :log(x))
a(6)
6.0

On a fait ici utilisation de la fonction lamba, mais il est possible d'utiliser directement la fonction log.

b=expo(log)
b(6)
6.0

2.5 Structures de données.

Parmi les structures de données complexes nous avons déjà rencontré les chaines et les listes.

2.5.1 Compléments : chaines de caractères, listes.

Nous avons déjaà vu que l'on peut accéder au i+1 ème élément d'une chaine de caractère par l'insstruction **print(ch[i])** pour une chaine de caractères de nom **ch**. L'istruction **print(ch[-i])** permet d'afficher le i ème caractère à partir de la droite.

```
 ch="animal"

for i in range(len(ch)):
        print(ch[-i],end="")
...
alamin>>>
```

L'instruction **len(ch)** représentant la longueur de **ch** c'est à dire le nombre de lettres contenues dans la chaine de caractères.

Ceci s'applique également aux listes :

```
l=list(range(10))

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

 for i in range(1,len(l)+1):
    print(l[-i],end=" ")
...
9 8 7 6 5 4 3 2 1 0
```

Dans ces deux exemples intervient la fonction **range**, l'instruction **range(n)**, où **n** est un entier représente l'ensemble des entiers compris entre **0 et n-1** , l'instruction **list(range(n))** construit la liste d'entiers compris entre **0 et n-1**.
Cette fonction **range** peut être utilisée avec un deux ou trois paramètres entiers, positifs ou négatifs.
la syntaxe est la suivante : **range(n,m,i)** qui représentent les entiers relatifs compris entre **n,m-1 par pas de i**

```
for i in range(3,16,2):
       print(i,end=" ")
...
3 5 7 9 11 13 15

l=list(range(3,16,2))
>>> print(l)
[3, 5, 7, 9, 11, 13, 15]

m=list(range(2,8))
>>> m
[2, 3, 4, 5, 6, 7]
```

Le dernier exemple montre **range(n,m)** représente l'ensemble des entiers entre **n et m-1**. Voyons un dernier exemple dans lequel des entiers sont négatifs.

```
l2=list(range(1,-12,-3))
>>> l2
[1, -2, -5, -8, -11]
```

Extraction d'une sous chaine ou d'une sous liste.

pour une chaine **ch** on peut obtenir un sous ensemble une des l'instruction **ch[n :m]** ou **ch[:m]** ou **ch[n :]**. Dans le premier cas on obtient la sous chaine contenu entre les caractères de rang **n et m-1**, dans le second entre les caractères **0 et m-1** et dans le troisème entre les caractères **n et le dernier caractère de la chaine**.

```
print(ch[2:8])
ticons

>>> print(ch[:15])
anticonstitutio

>>> print(ch[15:])
nnellement
```

La même manipulation peut être réalisée sur les listes.
L'objet obtenu est une nouvelle liste, ou chaine de caractère suivant le type de départ.

Il est possible de reconstituer une chaine de caractère ou une liste par ch[:] ou l[:], ce qui est intéressant dans le cas d'une liste car ceci crée un nouvel objet différent de la liste l.

l=[1,2,3,4]
m=l[:]
id(l)
140316402454040
id(m)
140316402417536
v=l
id(v)
140316402454040

L'instruction **id** indique l'adresse mémoire d'un objet, ceci montre que l'affectation v=l ne crée pas un nouvel objet, simplement un deuxième nom (un alias) pointant vers le même objet alors que m est un nouvel objet, une liste, contenant les mêmes éléments que l.

L'instruction **in** a déja ete rencontrée, utilisée dans des boucles **for**. Elle peut être utilsée seule.

```
l=list(range(0,9,2))
>>> n=3
>>> if n in l:
...    print("n est pair")
... else:
...    print("n est impair")
...
n est impair
```

Quelques méthodes applicables aux chaines de caractères.

ch.split() transforme une chaine de caractères en un liste constituée des mots contenus dans ch. **join(list(liste))** réalise l'opération inverse en transformant une liste en chaine de caractères.

```
ch="ceci est une chaine de caractere"
>>> l=ch.split()
>>> l
['ceci', 'est', 'une', 'chaine', 'de', 'caractere']

>>> print("*".join(l))
ceci*est*une*chaine*de*caractere

print(" ".join(l))
ceci est une chaine de caractere
```

La syntaxe dans le deuxième cas est **"séparateur".join(liste)**.

find(sch) cherche la position de la sous chaine sch dans la chaine.
count(sch) compte le nombre de sous chaine sch dans la chaine.

```
ch="ce texte tire sur papier glace est anticonstitutionnel"
>>> sch="ti"
>>> print(ch.find(sch))
9

>>> print(ch.count(sch))
4
```

Le début de la première sous chaine est au 10 ème caractère (9+1) et le nombre de sous chaine est 4.
L'utilisation d'**enumerate** permet de simplifier la manipulation des listes. En écrivant :

for i,x in enumerate(l) :
print(i,x,end=' ')

on obtient à la fois l'indice et l'élément correspondant.

ch="ceci est une chaine de caractères"
q=ch.split()
q
['ceci', 'est', 'une', 'chaine', 'de', 'caractères']

for i,x in enumerate(q) :
    print(i,x,end=" ")

0 ceci 1 est 2 une 3 chaine 4 de 5 caractères

2.5.2 Fichiers

Dans ce paragraphe nous allons étudier la gestion de fichiers texte par python. Notion de chemin d'accès, lecture et écriture de données numériques ou de type chaîne de caractères depuis ou vers un fichier.
On peut envisager l'utilisation de fichiers en tant que supports de données ou de résultats avant divers traitements, par exemple graphiques.

Pour utiliser un fichier il faut connaitre sa position sur le disque dur, plus précisément, le répertoire qui le contient.

Pour manipuler un fichier "monfichier", contenu dans le répertoire /home/perso on utilisera les instructions :
from os import chdir
chdir("/home/perso")

Cette syntaxe est la même sous Unix ou sous Windows. chdir étant l'abréviation de change directory. Le module os (Operating system) permettant de dialoguer avec le système d'exploitation à l'aide de fonctions ou de methodes.
Une autre fonction utile de ce module (c'est en fait une méthode) est la fonction getcwd() qui indique la position actuelle :

from os import chdir, getcwd
getcwd()
'/home/ubuntu'

2.5.2.1 Gestion des fichiers

Ouverture et fermeture des fichiers
Principaux modes d'ouverture des fichiers textes :
f1 = open("monFichier1", "r", encoding='utf-8') # "r" mode lecture
f2 = open("monFichier2", "w", encoding='utf-8') # "w" mode écriture
f3 = open("monFichier3", "a", encoding='utf-8') # "a" mode ajout
Python utilise les fichiers en mode texte par défaut (noté t). Pour les fichiers binaires, il faut préciser le mode b.
Le paramètre optionnel encoding assure les conversions entre les types byte , stocké dans le fichier sur le disque, et le type str, manipulé lors des lectures et écritures. L'utilisation de Python 3, utilise par défaut le codage utf-8, il est inutile de préciser "encoding='utf-8"' il faut par contre le préciser en cas d'utilisation d'un autre codage.
Les encodages les plus fréquents sont 'utf-8' (c'est l'encodage à privilégier en Python 3), 'latin1', 'ascii'. . .
Tant que le fichier n'est pas fermé, son contenu n'est pas garanti sur le disque.
Une seule méthode fermeture :
f1.close()

Le mode "w" suppose le fichier vide et efface le fichier dans le cas contraire, le mode "a" entre le nouveau texte à la suite de ce qui existe déjà.

<u>Écriture séquentielle</u>

Le fichier sur disque est considéré comme une séquence de caractères qui sont ajoutés à la suite, au fur et à mesure que l'on écrit dans le fichier.

f=open("fichier.txt","a")
s="j'écris dans le fichier \n"
f.write(s) # écrit la chaîne s dans f
l=['a','b','c','d']
f.writelines(l) # écrit les chaînes de la liste l dans f
f.close

La méthode write() réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier à accès séquentiel). Chaque nouvel appel de write() continue l'écriture à la suite de ce qui est déjà enregistré.
La méthode close() referme le fichier. Celui-ci est désormais disponible pour tout usage.
L'instruction \n sigifie "passage à la ligne"

Lecture séquentielle d'un fichier

Nous allons maintenant rouvrir le fichier, mais cette fois ≪ en lecture ≫, de manière à pouvoir y relire les informations que nous avons enregistrées dans l'étape précédente :

ou=open("fichier.txt","r")
t=ou.read()
print(t)
j'écris dans le fichier
abcd

La méthode **read()** peut être utilisée avec ou sans paramètres. Dans le cas où elle est utilisée sans paramètre, la totalité du fichier ezt transférée.
L'instruction **read(n)** lit les n caractères situés après la position actuelle dans le texte.
Dans les deux cas le type obtenu est le type chaine de caractères ou **str**.

La méthode **readline()** permet de lire ligne par ligne, tandis que la méthode **readlines()** lit tout le fichier (à partir de la position actuelle) et affiche les lignes sous forme de listes et non plus de chaine de caracteres.

```
nf=open("fich.txt","w")
nf.write('Ceci est la ligne 1\n')
nf.write('Ceci est la ligne 2\n')
nf.write('Ceci est la ligne 3\n')
nf.write('Ceci est la ligne 4\n')
nf.close()
```

```
ouvenf=open("fich.txt","r")
t=ouvenf.read(8)
print(t)
t=ouvenf.read(12)
print(t)
t=ouvenf.readline()
print(t)
t=ouvenf.readlines()
print(t)
ouvenf.close()
```

```
Le resultat est:

Ceci est
 la ligne 1

Ceci est la ligne 2

['Ceci est la ligne 3\n', 'Ceci est la ligne 4\n']
```

Remarques :

La liste apparaît ci-dessus en format brut, avec des apostrophes pour délimiter les chaînes, et les caractères spéciaux sous forme de codes numériques. Vous pourrez bien évidemment parcourir cette liste (à l'aide d'une boucle while, par exemple) pour en extraire les chaînes individuelles.

La méthode readlines() permet donc de lire l'intégralité d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros (Puisqu'il est copié intégralement dans une variable, c'est-à-dire dans la mémoire vive de l'ordinateur, il faut que la taille de celle-ci soit suffisante). Si vous devez traiter de gros fichiers, utilisez plutôt la méthode readline() dans une boucle.

Notez bien que readline() est une méthode qui renvoie une chaîne de caractères, alors que la méthode readlines() renvoie une liste. A la fin du fichier, readline() renvoie une chaîne vide, tandis que readlines() renvoie une liste vide.

Voici un exemple d'utilisation de fichiers, tiré de l'ouvrage de Gérard Swinnen :

Le script qui suit vous montre comment créer une fonction destinée à effectuer un certain traitement sur un fichier texte. En l'occurrence, il s'agit ici de recopier un fichier texte en omettant toutes les lignes qui commencent par un caractère '#' :

```
def filtre(source,destination):
"recopier un fichier en éliminant les lignes de remarques"
fs = open(source, 'r')
fd = open(destination, 'w')
while 1:
txt = fs.readline()
if txt =='':
break
if txt[0] != '#':
fd.write(txt)
fs.close()
fd.close()
return

Pour appeler cette fonction, vous devez utiliser deux arguments : le nom du fichier original, et le
nom du fichier destiné à recevoir la copie filtrée. Exemple :
filtre('test.txt', 'test_f.txt')
```

La boucle **while** ne contient aucune condition (1='TRUE'), elle bouclerait indéfiniment s'il n'y avait l'instruction **break** qui arrête la boucle à la fin du fichier. lorsque **txt=''** c'est à dire lorsque la méthode **readline()**

donne le caractère vide.

2.5.3 Dictionnaires

. Les types composites abordés jusqu'à présent (chaînes, listes et tuples) étaient tous des séquences, c'est-à-dire des suites ordonnées d'éléments.
Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.
Les dictionnaires constituent un autre type composite. Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables ou mutables comme elles), mais ce ne sont pas des séquences. Les éléments enregistrés ne seront pas disposés dans un ordre immuable.
En revanche, nous pourrons accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une clé, laquelle pourra être alphabétique, numérique, ou même d'un type composite sous certaines conditions. Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type. Ce peuvent être des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, mais aussi des fonctions.
Création d'un dictionnaire :

L'instruction **mondict={}** crée un dictionnaire vide.

```
>>> mondict = {}
>>> mondict["pseudo"] = "Prolixe"
>>> mondict["mot de passe"] = "*"
>>> mon_dictionnaire
{'mot de passe': '*', 'pseudo': 'Prolixe'}
>>>
```

"pseudo" et **"mot de passe"** sont les clés du dictionnaire, **"prolixe"** et **"*"** sont les valeurs correspondantes.

Un dictionnaire est constitué de couples (clé, valeur), la clé jouant ici le rôle de l'index pour une liste.

Voici un exemple de dictionnaire pour lequel la clé est le prénom et la valeur l'age :

```
dico={}
dico['marcel']=18
dico['gertrude']=17
dico['eric']=19
dico['david']=18

dico
{'gertrude': 17, 'marcel': 18, 'david': 18, 'eric': 19}
```

Il est possible de rajouter autant de couples (clés, valeurs) que l'on veut à un dictionnaire.

dico["quentin"]=20
dico["serge"]=17

dico
'quentin' : 20, 'marcel' : 18, 'david' : 18, 'eric' : 19, 'gertrude' : 17, 'serge' : 17
La fonction **len** indique le nombre de couples d'un dictionnaire.

len(dico)
6

Il est possible de retirer un couple d'un dictionnaire par **del nomdico[cle]**
del dico["gertrude"]
dico
'quentin' : 20, 'marcel' : 18, 'david' : 18, 'eric' : 19, 'serge' : 17

La méthode **keys()** donne la liste des clés du dictionnaire, la méthode **values()** donne la liste des valeurs du dictionnaire.

dico.keys()
dict_keys(['quentin', 'marcel', 'david', 'eric', 'serge'])
dico.values()
dict_values([20, 18, 18, 19, 17])

type(dico)
class 'dict'

Le type dictionnaire est **dict**.

La méthode has_key() permet de savoir si un dictionnaire comprend une clé déterminée. On fournit la clé en argument, et la méthode renvoie une valeur 'vraie' ou 'fausse' (en fait, 1 ou 0), suivant que la clé est présente ou pas.

La méthode **items()** extrait du dictionnaire une liste équivalente de tuples :

print(dico.items())
dict_items([('quentin', 20), ('david', 18), ('gertrude', 17), ('eric', 19), ('marcel', 18)])

Parcours d'un dictionnaire

A l'aide des clés :

for cle in dico :
print(cle)

quentin
david
gertrude
eric
marcel

Valeurs seules :

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

20
18
17
19
18

Clés et valeurs :

for clef, valeur in dico.items() :
print(clef,valeur)

quentin 20
david 18
gertrude 17
eric 19
marcel 18

Autre méthode :

for clef in dico :
print(clef,dico[clef])

quentin 20
david 18
gertrude 17
eric 19
marcel 18

Les clés peuvent être des objets de différent types, par exemple des tuples.

Dictionnaire décrivant une partie d'un échiquier :

echiquier = {}
echiquier['a', 1] = "tour blanche" # En bas à gauche de l'échiquier
echiquier['b', 1] = "cavalier blanc" # À droite de la tour
echiquier['c', 1] = "fou blanc" # À droite du cavalier
echiquier['d', 1] = "reine blanche" # À droite du fou
... Première ligne des blancs
echiquier['a', 2] = "pion blanc" # Devant la tour
echiquier['b', 2] = "pion blanc" # Devant le cavalier, à droite du pion
... Seconde ligne des blancs

Les tuples sont sous entendus, il est inutile de les placer entre parenthèses, mais cette notation serait évidemment correcte.

On peut remarquer que l'ordre d'introduction des éléments dans un dictionnaire n'est pas conservé par la suite, un dictionnaire n'est pas un objet ordonné, contrairement aux chaines de caracteres, aux listes et aux tuples.

2.5.4 Ensembles

Un ensemble est un objet contenant des élément distincts, non ordonné.

X=set('abcdefae')
print(X)
'e', 'd', 'f', 'a', 'c', 'b'

Les éléments a et e ne sont pas répétés et l'ordre des éléments est quelconque et peut varier pour deux affichages

successifs de X.

Opérations

Y=set('afghie')
print(X & Y) # intersection de X et Y
{'e', 'f', 'a'}
print(X|Y) # reunion de X et Y
'e', 'd', 'g', 'f', 'a', 'c', 'b', 'i', 'h'
print(X-Y)
{'d', 'c', 'b'}
print(Y-X)
{'g', 'i', 'h'}

2.5.4.1 Exercices

Exercice 1

Ecrire un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant à l'aide d'une astéristique ceux qui sont multiples de 3. (Utiliser l'opérateur %).

Exercice 2

On considère les listes :
$t_1 = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$ et $t_2 = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$ données, construire la liste $t = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 6, 17, 18, 19, 20]$ à partir des deux premières listes en utilisant la méthode **append**.

Exercice 3

Ecrire un programme qui détermine le plus grand élément d'une liste de nombres.

Exercice 4

Etant donnée une liste d'entiers naturels l, ecrire un programme donnant deux listes l1 et l2 contenant respectivement les entiers pairs et les entiers impairs de l.

Exercice 5

Ecrire un script qui transforme une liste l1 en une liste symétrique l2, sans modifier l1.
Exemple $l1 = [1, 2, 3, 4]$ est transformée en $l2 = [4, 3, 2, 1]$.

Exercice 6

1) Ecrire un script qui détermine (réponse 'true' ou 'false') si une chaine de caractères contient ou non le caractère 'd'.
2) Ecrire un script qui détermine le nombre d'occurences du caractère 'd' dans une chaine de caractères.

Exercice 7

Ecrire un script qui recopie une chaine de caractères dans une variable, en l'inversant, par exemple 'toto' devient 'otot'.

Exercice 8

Ecrire un script qui calcule la somme des éléments d'un tuple contenant uniquement des nombres.

Exercice 9 : Division euclidienne

On considère l'algorithme suivant :
a,b sont des entiers naturels
B = b
R = a
Q = 0
Tant que $R \geq B$
R = R -B
Q = Q + 1
Ecrire Q et R

Ecrire le script coorespondant à cet algorithme. Vérifier que les valeurs Q et R que donne cet algorithme sont le quotient et le reste dans la division euclidienne de a par b, c'est à dire que :
$a = bQ + R$ avec $R < b$.

Montrer que a = B * Q + R est un invariant de boucle et que R-B est strictement décroissante.

Exercice 10

Ecrire un programme permettant de calculer le plus petit commun multiple de deux entiers naturels m et n. On pourra réaliser une boucle dans laquelle on calcule les multiples successifs de m et on sort de la boucle lorsque ce multiple est divisible par n. Déterminer un invariant de boucle. Vérifier que la boucle s'arrête.

Exercice 11

Etant donné un réel x. Ecrire un programme déterminant l'entier naturel n tel que
$10^n \leq |x| < 10^{n+1}$, sans utiliser de logarithme.

Exercice 12

Écrire une fonction qui prend en arguments deux entiers x et y et qui renvoie -1 si $x < y$, 0 si $x = y$ et 1 si $x > y$.

Exercice 13

Écrire une fonction qui prend en arguments deux entiers $n \geq 0$ et $d > 0$ et qui renvoie un couple formé du quotient et du reste de la division euclidienne de n par d. Le quotient et le reste seront calculés par soustractions successives comme dans l'exercice 9.

Exercice 14

Ecrire une fonction qui indique si un nombre est premier ou non en donnant un résutat booléen.

Exercice 15

Ecrire une fonction qui retourne la valeur du pgcd et du ppcm de deux entiers, sachant que le produit du pgcd et du ppcm est égal au produit des deux nombres.

Exercice 16

Ecrire une fonction qui prend en argument une liste et qui vérifie si c'est un palindrome. La réponse sera un booléen.

$[1, 2, 3, 4, 3, 2, 1]$ est un palindrome, $[1, 2, 3, 4, 5, 6]$ n'est pas un palindrome.

Exercice 17

Ecrire un programme calculant la somme des éléments d'une liste contenant des entiers.

Exercice 18

Ecrire un programme calculant la moyenne et la variance des éléments d'une listes contenant des nombres flottants.
Soit $l = [x_1, x_2, \cdots x_n]$, la moyenne et la variance sont définies par :

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad Var(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i)^2 - \overline{x}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2.$$

Exercice 19

Ecrire une fonction prenant en entrée x et une liste l contenant des entiers et renvoyant la première position de x dans l si x est présent et None sinon.

Exercice 20

Pour se familiariser avec la boucle **while** établir sur une feuille (faire un tableau) les valeurs prises par a, b, c à chaque étape de la boucle, puis exécuter le script :

```
a,b,c=1,1,1
while (c<11):
   print(b,end=" ")
   a,b,c=b,a+b,c+1
```

Les valeurs successives prises par a et b sont les valeurs prises par la suite de Fibonacci.

Exercice 21

la somme

$$\sum_{k=1}^{n} \frac{(-1)^{k+1}}{k}$$

converge (tend vers) $\ln 2$ quand n tend vers l'infini.
Initialisaton : n=1, écrire s.

A l'aide d'une boucle **while** déterminer l'entier minimal n tel que $|\ln(2) - s| \leq 10^{-4}$. reprendre le calcul pour 10^{-5}, donner à chaque fois la valeur de s obtenue :
la boucle commencera par **while(abs(log(2)-s)>10e-4) :**

Exercice 22

Calculer la somme :

$$S = \sum_{i+j=1000} (i^2 + j)(i + j^2) = \sum_{i=1}^{1000} \left(\sum_{j=1}^{1000-i} (i^2 + j)(i + j^2) \right).$$

en utilisant deux boucles **for** imbriquées.

Exercice 23

Calculer la somme :

$$\sum_{1 \leq i \leq j \leq 5000} (i - j)^3.$$

à l'aide d'une double boucle **for**

Exercice 24

Conjecture : suite de syracuse.
En partant d'un entier n on lui applique les transformations suivantes : Si n est pair on le divise par deux, s'il est impair on le remplace par 3n+1. La conjecture, jamais prouvée, prévoit qu'au bout dun nombre fini d'étape on obtient 1.

Ecrire un script permettant de calculer le nombre d'étapes nécessaire pour atteindre 1. La valeur de n, pouvant être modifiée, sera initialisé au début du script. Utiliser
if condition :

else : Donner la valeur maximale atteinte au cours des étapes.

Donner le résultat pour n=1000, pour n=100 000.

Exercice 25

Parmi toutes les valeurs de départ dans $\{n \in \mathbb{N} | 1 \leq n \leq 10^5\}$ quelle est le nombre d'étapes maximum, pour quelle valeur de n est elle obtenue.

Exercice 26

Ecrire un programme inversant les chiffres d'un nombre n. Exemple : 123456 $\rightarrow$ 654321. Utiliser restes et quotients dans la division par dix. Le résultat étant un nombre entier.

Exercice 27

1) Ecrire un programme calculant la somme des chiffres d'un entier écrit en décimal.
2) Ecrire un programme calculant la somme des diviseurs propres d'un entier n, c'est à dire un compris mais n exclu.
3) Un nombre entier est parfait s'il est égal à la somme de ces diviseurs. Ecrire un programme permettant de tester ('TRUE' ou 'FALSE') si un nombre est parfait.
4) Ecrire un programme donnant la liste des nombres parfaits compris entre 1 et T.

Exercice 28

Ecrire un programme testant si un nombre donné est premier ou non. Il suffit de s'arrêter à $E(\sqrt{n}) + 1$ et de parcourir cet ensemble d'entiers de deux en deux à partir de trois, sans oublier de tester l'entier 2. La réponse sera False ou True.

Chapitre 3

Etude de quelques algorithmes. Notion de complexité algorithmique

3.1 Complexité algorithmique

Ces définitions s'inspirent du polycopié de Stéphane Gonnord.

Un algorithme répond à un problème. Il est composé d'un ensemble d'étapes simples nécessaires à la résolution, dont le nombre varie en fonction du nombre d'éléments à traiter. D'autre part, plusieurs algorithmes peuvent répondre à un même problème.
Pour savoir quelle méthode est plus efficace il faut les comparer. Pour cela, on utilise une mesure que l'on appelle la complexité qui représente le nombre d'étapes qui seront nécessaires pour résoudre le problème pour une entrée de taille donnée. La théorie de la complexité s'attache à connaître la difficulté (ou la complexité) d'une réponse par algorithme à un problème, dit algorithmique, posé de façon mathématique. Pour la définir, il faut présenter les concepts de problèmes algorithmiques, de réponses algorithmiques aux problèmes, et la complexité des problèmes algorithmiques.

On s'intéresse au coût exact quand c'est possible, mais également au coût moyen (que se passe-t-il si on moyenne sur toutes les exécutions du programme sur des données de taille n), au cas le plus favorable, ou bien au cas le pire. On dit que la complexité de l'algorithme est O(f(n)) où f est d'habitude une combinaison de polynômes, logarithmes ou exponentielles. Ceci reprend la notation mathématique classique et signifie que le nombre d'opérations effectuées est borné par cf(n), où c est une constante, lorsque n tend vers l'infini.

Différents algorithmes ont des coûts différents en termes de
— temps d'exécution (nombre d'opérations effectuées par l'algorithme),
— taille mémoire (taille nécessaire pour stocker les différentes structures de données pour l'exécution).
Ces deux concepts sont appelés la complexité en temps et en espace de l'algorithme.

La complexité algorithmique est un concept fondamental pour tout informaticien, elle permet de déterminer si un algorithme et meilleur
qu'un autre algortihme et s'il est optimal ou s'il ne doit pas être utilisé.

Prenons un exemple simple. Si l'on cherche à afficher la liste des diviseurs d'un nombre entier n, on peut écrire l'algorithme naïf

```
def diviseurs(n):
   for i in range(1,n+1):
     if n % i == 0:
       print(i)
```

qui effectue exactement n calculs de restes de divisions euclidiennes, n comparaisons et un nombre d'affichages inférieur ou égal à n. Mais on peut aussi utiliser le fait que si n = pxq avec $p \geq \sqrt{n}$ alors q est un diviseur de n inférieur ou égal à p. Il suffit donc de chercher chaque diviseur q inférieur ou égal à $\sqrt{n}$ et de calculer p = n/q pour obtenir tous les diviseurs. Dans le cas où n est un carré parfait on prend soin de ne pas afficher sa racine carrée deux fois :

```
from math import sqrt
def diviseurs(n):
  for i in range(1,int(sqrt(n))+1):
    if n % i == 0:
     print(i)
     if n//i != i:
       print(n//i)
```

Ce deuxième algorithme ne fait plus que $\sqrt{n}$ itérations, qui effectuent chacune un calcul de reste, une ou deux comparaisons, et zéro, un ou deux affichages. Au total, il coûte donc un calcul de racine carrée, $\sqrt{n}$ calculs de reste, entre 0 et $\sqrt{n}$ et $2\sqrt{n}$ comparaisons et entre 0 et $2\sqrt{n}$ affichages.

Dans le premier cas, l'algorithme a donc une complexité en $O(n)$, Dans le , deuxième cas l'algorithme a une complexité en $O(\sqrt{n})$.

<u>Détermination du coût d'un algorithme</u>

Pour déterminer le coût d'un algorithme, nous nous fonderons en général sur le modèle de complexité suivant :

— Une affectation, une comparaison ou l'évaluation d'une opération arithmétique ayant en général un faible temps d'exécution, nous le considérerons comme l'unité de base dans laquelle on mesure le coût d'un algorithme.
— Le coût des instructions p et q à la suite est la somme des coûts de l'instruction p et de l'instruction q.
— Le coût d'un test if b : p else : q est inférieur ou égal au maximum des coûts des instructions p et q, plus une unité qui correspond au temps d'évaluation de l'expression b.
— Le coût d'une boucle for i in range(m) : p est m fois le coût de l'instruction p si ce coût ne dépend pas de la valeur de i. Quand le coût du corps de la boucle dépend de la valeur du compteur i, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de i.
— Le cas des boucles while est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison et ainsi majorer le coût de l'exécution de la boucle.

Le cas particulier des boucles imbriquées illustre bien le principe de calcul du coût de la boucle for. Ainsi, si les deux boucles sont répétées respectivement m et m' fois, alors le corps de la boucle interne est exécuté mxm' fois en tout. Il est en effet répété à cause de cette boucle interne, mais aussi parce qu'elle-même est répétée dans son intégralité.

<u>Ordre de grandeur</u>

Temps d'exécution que l'on rencontre en pratique pour un problème de taille $n = 10^6$sur un ordinateur personnel.

	cout	temps pour $n = 10^6$	Remarques
O(1)	temps constant	1 ns	Le temps d'exécution ne dépend pas des données traitées, la plupart des données ne sont même pas lues.
O(log n)	logarithmique	10 ns	En pratique, cela correspond à une exécution quasi instantanée.
O(n)	linéaire	1 ms	Le problème de la gestion de la mémoire se posera avant celui de l'efficacité en temps.
O(n^2)	quadratique	1/4 h	Cette complexité reste acceptable pour des données de taille moyenne ($n < 10^6$) mais pas au delà.
O(n^k)	polynomiale	30 ans si k = 3	Ici n^k est le terme de plus haut degré d'un polynôme en n, il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(2^n)$	exponentielle	plus de 10^{300000} milliards d'années	Un algorithme d'une telle complexité est impraticable sauf pour de très petites données. Comme pour la complexité logarithmique, la base de l'exponentielle ne change fondamentalement rien à l'inefficacité de l'algorithme.

Ce tableau ne mentionne que la complexité en temps et non la complexité en taille mémoire.

On appelle complexité en espace d'un algorithme la place nécessaire en mémoire pour faire fonctionner cet algorithme. Elle s'exprime également sous la forme d'un O(f(n)) où n est la taille du problème.
Évaluer la complexité en espace d'un algorithme ne pose la plupart du temps pas de difficulté, il suffit de faire le total des tailles en mémoire des différentes variables utilisées. La seule vraie exception à la règle est le cas des fonctions récursives, qui cachent souvent une complexité en espace élevée.

3.2 Exemple : Algorithme d'Euclide

Etudions la complexité en temps de l'algorithme d'Euclide, permettant le calcul du pgcd de deux entiers.
Soient $a \in \mathbb{N}$ et $b \in \mathbb{N}^*$.On suppose que la détermination du pgcd nécessite n divisions euclidiennes successives :
On pose $r_n = a$, $r_{n-1} = b$.

$$r_n = r_{n-1} * q_1 + r_{n-2} \quad (3.1)$$
$$r_{n-1} = r_{n-2} * q_2 + r_{n-3} \quad (3.2)$$
$$\vdots \quad (3.3)$$
$$r_i = r_{i-1} * q_{n+1-i} + r_{i-2} \quad (3.4)$$
$$\vdots \quad (3.5)$$
$$r_2 = r_1 * q_n + r_0 \quad (3.6)$$
$$r_1 = r_0 * q_{n+1} \quad (3.7)$$

A chaque étape on a $0 < r_{i-1} < r_i$ $1 \leq i \leq n-1$.

On a pgcd(a,b)=r_0.

Notons $T_{a,b}$ le nombre de divisions nécessaires pour obtenir le pgcd, ici ce nombre est n, nous allons donner un majorant de cet entier.
On peut noter que si $a < b$, on a $T_{a,b} = 1 + T_{b,a}$ car on a une première division $a = b*0+a$ où a et b sont échangés.

L'étude de $T_{a,b}$ nécessite l'utilisation de la suite de Fibonacci, qui est une suite récurrente linéaire d'ordre 2, définie par :

$$F_0 = F_1 = 1,\ F_{n+2} = F_{n+1} + F_n,\ n \in \mathbb{N}$$

qui est associée à l'équation caractéristique :

$$r^2 - r - 1 = 0.$$

Dans la suite on suppose $0 < b < a$.

Proposition 1

Si $T_{a,b} = n \geq 1$ on a $a = r_n \geq F_{n+1}$ et $b = r_{n-1} \geq F_n$.

Démonstration

On remarque que $q_{n+1} \geq 2$, donc $r_1 \geq 2 = F_2$ et $r_0 \geq 1 = F_1$.

Par récurrence, $r_i = r_{i-1} * q_{n+1-i} + r_{i-2} \geq F_i + F_{i-1} = F_{i+1}$ et ceci entraine $a = r_n \geq F_{n+1}$ et $b = r_{n-1} \geq F_n$.

Proposition 2

On a $T_{F_{n+2},F_{n+1}} = n$.

Démonstration

$F_{n+2} = F_{n+1} + F_n$, d'où $T_{F_{n+2},F_{n+1}} = 1 + T_{F_{n+1},F_n}$, c'est une suite arithmétique de raison 1.

$T_{F_{n+2},F_{n+1}} = n - 1 + T_{F_3,F_2}$ et $F_3 = 2F_2$, $T_{F_3,F_2} = 1$.

Théorème de Lame

Sous les hypothèses précédentes, $0 < b < a$, le nombre $T_{a,b}$ est majoré par cinq fois le nombres de chiffres de b écrit en base dix.

Démonstration

On montre par récurrence que $F_{n+1} \geq \phi^n$ où $\phi = \dfrac{1+\sqrt{5}}{2}$ est la racine positive de l'équation caractéristique associée à la suite (F_n).

Le réel ϕ vérifie : $\phi^{n+1} = \phi^n + \phi^{n-1}$. On a $F_1 = 1 \geq \phi^0 = 1$ et $F_2 = 2 \geq \phi^1 = \phi \sim 1,62$.

On a alors $F_{n+2} = F_{n+1} + F_n \geq \phi^n + \phi^{n-1} = \phi^{n+1}$, ce qui termine la récurrence.

On sait que $b \geq F_n$ donc $b \geq \phi^{n-1}$ ce qui entraine $\ln b \geq (n-1)\ln\phi$. En notant k le nombre de chiffres de b écrit en base dix, on a

$10^k > b$ donc $\ln b < k \ln 10$ et $k \ln 10 > \ln b \geq (n-1)\ln\phi$. On a alors $n - 1 < k \frac{\ln 10}{\ln \phi}$.

On vérifie que $\dfrac{\ln 10}{\ln \phi} \leq 5$.

Remarques

1) En utilisant la contraposée de la proposition 1 on a : $b \leq F_{n+1} \Longrightarrow T_{a,b} \leq n$.

2) Les nombres de Fibonacci consécutifs représentent l'entrée pour le pire des cas pour l'algorithme d'Euclide. Ils correspondent à la complexité maximale.

3.3 Etude de quelques algorithmes

3.3.1 Algorithme de Horner

On considère un polynôme $A(X) = \displaystyle\sum_{k=0}^{n} a_k X^k$. On veut évaluer la valeur de ce polynôme pour une valeur réelle x.

Première méthode
On écrit une boucle réalisant le calcul souhaité. Le polynôme est représenté par un tableau ou liste en Python.

Exemple : [1,4,2] représente le polynôme $1 + 4X + 2X^2$.

En utilisant l'instruction **enumerate** etudiée précedemment ceci peut s'écrire :

```
def evaluer(a, x):
  s = 0
  for i, ai in enumerate(a):
      s = s + ai * x**i
  return s
```

Remarque la ligne $s = s + ai * x * *i$ peut s'écrire également $s+ = ai * x * *i$.
Le calcul de x^i nécessite $i - 1$ multiplications, le calcul de $a_i * x^i$ une autre multiplication soit au total $1 + 2 + \cdots n = \frac{n(n+1)}{2}$ mutiplications et n additions (au plus car certains coefficients peuvent être nuls). ceci donne une complexité quadratique.

Deuxième méthode

Méthode de Horner.
Elle consiste à réécrire la somme ci-dessus de la manière suivante :

$A(X) = a_0 + X(a_1 + X(a_2 + \cdots + X(a_{n-1} + Xa_n)\cdots))$.

Il n'y a plus de calculs de puissances, les seuls calculs sont des multiplications et des additions.

Il y a n additions et n multiplications, on passe d'une complexité quadratique à une complexité linéaire, bien meilleure.

Pour évaluer ce polynôme en un réel x, la première étape est le calcul de $a_{n-1} + xa_n$ puis on stocke cette valeur dans une variable s, la deuxième étape consiste à calculer $a_{n-2} + x * s$ et ainsi de suite. On voit que la liste est décrite de droite à gauche, ce qui est possible en lui appliquant la fonction **reversed**. On peut écrire la fonction utilisant cet algorithme :

```
def horner(a,x):
   s=0
   for ai in reversed(a):
      s=ai+s*x
   return s
```

3.3.2 Recherche dans un tableau

3.3.2.1 Cas général

On utilisera à nouveau le type liste pour représenter un tableau.

Une mauvaise méthode pour étudier la présence d'un élément dans un tableau serait de décrire tout le tableau. Si l'élément est présent il est naturel d'arrêter la recherche dès que l'on a déterminé sa présence.

La fonction qui suit exécute une boucle qui est interrompue dès que x apparait, la liste ne sera décrite entièrement que si x est en dernière position ou s'il n'est pas dans la liste.

```
def element(l,x):
    for i in range (len(l)):
        if l[i]==x:
            return True
    return False
```

Deuxième forme de la fonction précédente :

```
def element2(a,x):
    for y in a:
        if y==x:
            return True
    return False
```

Le programme qui suit détermine la présence et dans ce cas l'indice de la première occurence de x dans la liste.

```
def appart(l,x):
    for i,ai in enumerate(l):
        if ai==x:
            return i
    return print("pas de x")
```

Dans ce cas, le programme s'arrête dès la première occurrence de x dans l.

3.3.2.2 Recherche dichotomique dans un tableau trié

On considère un tableau d'entiers ou de réels, que l'on supposera ordonné en ordre croissant. L'algorithme qui va être présenté n'a d'intérêt que dans le cas d'un tableau trié.

On étudie à nouveau la présence de x dans ce tableau en précisant éventuellement sa postion.

Principe : on compare x à la valeur centrale m de l.
— Si x = m, on a trouvé x dans T.
— Sinon, si x ¡ m, on cherche x dans la moitié inférieure du tableau.
— sinon on cherche x dans la moitié supérieure.
Exemple

l=[2,5,7,11,14,16,19,22,28]. On veut étudier la présence de 19 dans ce tableau et sa position.

On recherche 19 dans $l[0:9]$. $l[4] = 14$, le nombre est situé à droite de 14.
On recherche 19 dans $l[5:9]$. $l[7] = 22$, le nombre est situé à gauche de 22.
On recherche 19 dans $l[5:7]$. $l[6] = 19$.

On a donc vérifié que 19 était dans l et que son indice est 6. Trois comparaisons ont été nécessaires pour répondre à la question.

Dans le cas d'un tableau contenant 1000 éléments, le nombre maximum de comparaisons n (dans le pire des cas) serait donné par $2^{n-1} \leq 1000$ soit $n = 10$, ce qui est peu par rapport au nombre d'éléments du tableau.

Pour écrire l'algorithme, on délimite la portion du tableau t dans laquelle la valeur x doit être recherchée à l'aide de deux indices g et d.

On maintient l'invariant suivant : les valeurs strictement à gauche de g sont inférieures à x et les valeurs strictement à droite de d supérieures à x.

Principe :

<u>initialisation</u>

$g \leftarrow 0$
$d \leftarrow longueur - 1$
$trouve \leftarrow faux$

<u>Boucle de recherche</u>

Tant que $trouve = faux$ et que $g \leq d$:

$m \leftarrow partieentiere((g + d)/2)$
$Si\ t[m] = x\ alors$
$trouve \leftarrow vrai$
Sinon
$Si\ x > t[m]\ alors$
$g \leftarrow m + 1$

Sinon
$d \leftarrow m - 1$
FinSi
FinSi

La condition début inférieur ou égal à fin permet d'éviter de faire une boucle infinie si x n'existe pas dans le tableau.

$Fin\ boucle.$

Affichage du résultat
Si trouve Alors
Afficher "La valeur ", x , " est au rang ", m
Sinon
Afficher "La valeur ", x , " n'est pas dans le tableau"
FinSi

Ceci conduit au programme suivant :

```
def rechdicho(l,x):

    """renvoie la position d'une occurrence de x dans l,
supposée trié, si elle existe, et un message sinon"""

    g,d=0,len(l)-1
    t='FALSE'
    while (g<=d) and (t==False):
        m=(g+d)//2
        if l[m]==x:
            t=True
        else:
            if (x>l[m]):
                g=m+1
            else:
                d=m-1
    if t==True:
        print('la valeur ',x,' est au rang ',m)
    else:
        print('la valeur ',x," n'est pas dans le tableau")
```

Montrons que l'algorithme a une complexité en $O(\log_2(n))$.

L'étude préliminaire laisse penser qu'à la k-ième étape on a : $d - g < \dfrac{n}{2^k}$. Montrons ce résultat par récurrence sur k.

Pour $k = 0$, on a $g = 0$ et $d = n - 1$, le résultat est vérifié.

Supposons maintenant l'inégalité vraie au rang k et $g \leq d$. À la fin de la (k + 1)-ième itération, on a soit $g = m + 1$, soit $d = m - 1$.

Dans le premier cas, on a donc $d - E((g + d)/2) \leq d - (g + d)/2 = (d - g)/2 < \frac{1}{2}\frac{n}{2^k} = \frac{n}{2^{k+1}}$. Le second cas se traite de la même manière.

Quand $\dfrac{n}{2^k} < 1$ on a $d - g \leq 0$ c'est à dire que $n < 2^k$ ou encore $k > \log_2(n)$. Ce qui montre que le programme s'arrête avec $k \leq \log_2(n)$

3.3.2.3 Recherche d'un mot dans un texte

Un problème classique en informatique consiste à rechercher, non pas une seule valeur, mais une séquence de valeurs dans un tableau. Cela revient à chercher l'occurrence d'un tableau dans un autre. On souhaite donc écrire une fonction qui, étant donnés deux tableaux m et t, détermine la position de la première occurrence de m dans t, si elle existe, et qui renvoie None sinon ou un message indiquant que le problème n'a pas de solution. Cet algorithme peut également être appliqué à une chaine de caractères.

On utilise deux boucles imbriquées. la première pour i variant de 0 à longueur(t) - longueur(m), puis si le premier caractère de m apparait une boucle pour j variant de i à $i + longueur(m)$ en vérifiant à chaque étape que les caractères sont bien ceux de m.

```
def remot(m, t):
"""renvoie la position de la première occurrence du mot m
dans le texte t, et renvoie None sinon"""
  for i in range(1 + len(t) - len(m)):
      j = 0
      while j < len(m) and m[j] == t[i + j]:
         j += 1
         if j == len(m):
           return i
  return None
```

Cette fonction fait apparaitre seulement la première occurence du mot m s'il existe en plusieurs exemplaires.

La complexité dans le meilleur des cas est celle que l'on obtient lorsque le mot est situé au début de t, elle est de longueur(m). Le pire des cas se présente si tous les caractères du mot m, sauf le dernier, sont répétés dans t, la complexité est alors :
longueur(m)*(longueur(t)-longueur(m)+1).

3.3.2.4 Recherche du maximum dans un tableau de nombres

Le tableau n'est évidemment pas trié.

Le principe consiste à initialiser une variable M, qui représentera le maximum en sortie, par $l[0]$, puis à utiliser une boucle, au cours de laquelle $M = l[i]$ si $l[i] > M$.

Ceci nécessite $n = longueur(l) - 1$ comparaisons et entre une et n affectations.

```
def maxi(l):
"""M est le maximum des nombres de la liste"""
M=l[0]
for i in range(1,len(l)):
if (l[i]>M):
M=l[i]
return M
```

3.3.2.5 Moyenne et variance

Ce paragraphe reprend des calculs traités dans un exercice précédent.

On considère une liste l de nombres dont on veut calculer la moyenne, la variance et l'écart type, qui mesure une dispersion des valeurs par rapport à la moyenne.

Rappel :

Soit $l = [x_1, x_2, \cdots, x_n]$ une liste de nombres. La moyenne et la variance sont respectivement :

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \sigma^2(l) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \overline{x}^2.$$

```
def moyenne(l):
    s=0
    for x in l:
        s=s+x
    return s/len(l)

def variance(l):
    from math import sqrt
    s=0
    for x in l:
        s=s+x**2
    print("la variance est: ")
    v=s/len(l)-moyenne(l)**2
    return v,sqrt(v)
```

Dans le deuxième programme, v est la variance et sa racine carrée et sa racine carrée est l'écart type. C'est $\sigma(l)$ dans la définition.

Remarque : En statistique, dans la théorie des estimateurs, on utilise une variance dans laquelle on divise par $n-1$ au lieu de n, pour éviter ce que l'on appelle le biais. Ces deux valeurs diffèrent peu pour les grandes valeurs de n.

3.4 Résolution d'équations par dichotomie

Soit f une fonction continue et strictement monotone sur l'intervalle $[a, b]$, vérifiant $f(a)\,f(b) < 0$. Le théorème de la bijection permet de donner l'existence et l'unicité de $c \in [a, b]$ tel que $f(c) = 0$.

On va appliquer à la détermination d'une approximation de c une méthode de dichotomie, rappelant celle utilisé pour déterminer la présence d'un élément dans un tableau trié.

Dans les deux cas il s'agit d'une méthode de programmation appelée “diviser pour régner”.

Le principe consiste à faire intervenir deux suites (x_n) et (y_n) de nombres de $[a, b]$ tels que :
1) $f(x_n)\,f(y_n) < 0$. Avec $x_0 = a,\ y_0 = b$.
2) $|y_n - x_n| = \frac{1}{2}|y_{n-1} - x_{n-1}|$.
3) $c \in [x_n, y_n]$ (*ou* $c \in [y_n, x_n]$).

Au bout de n étapes, x_n (ou y_n) est une valeur approchée de c avec une erreur inférieure à $\dfrac{b-a}{2^n}$.

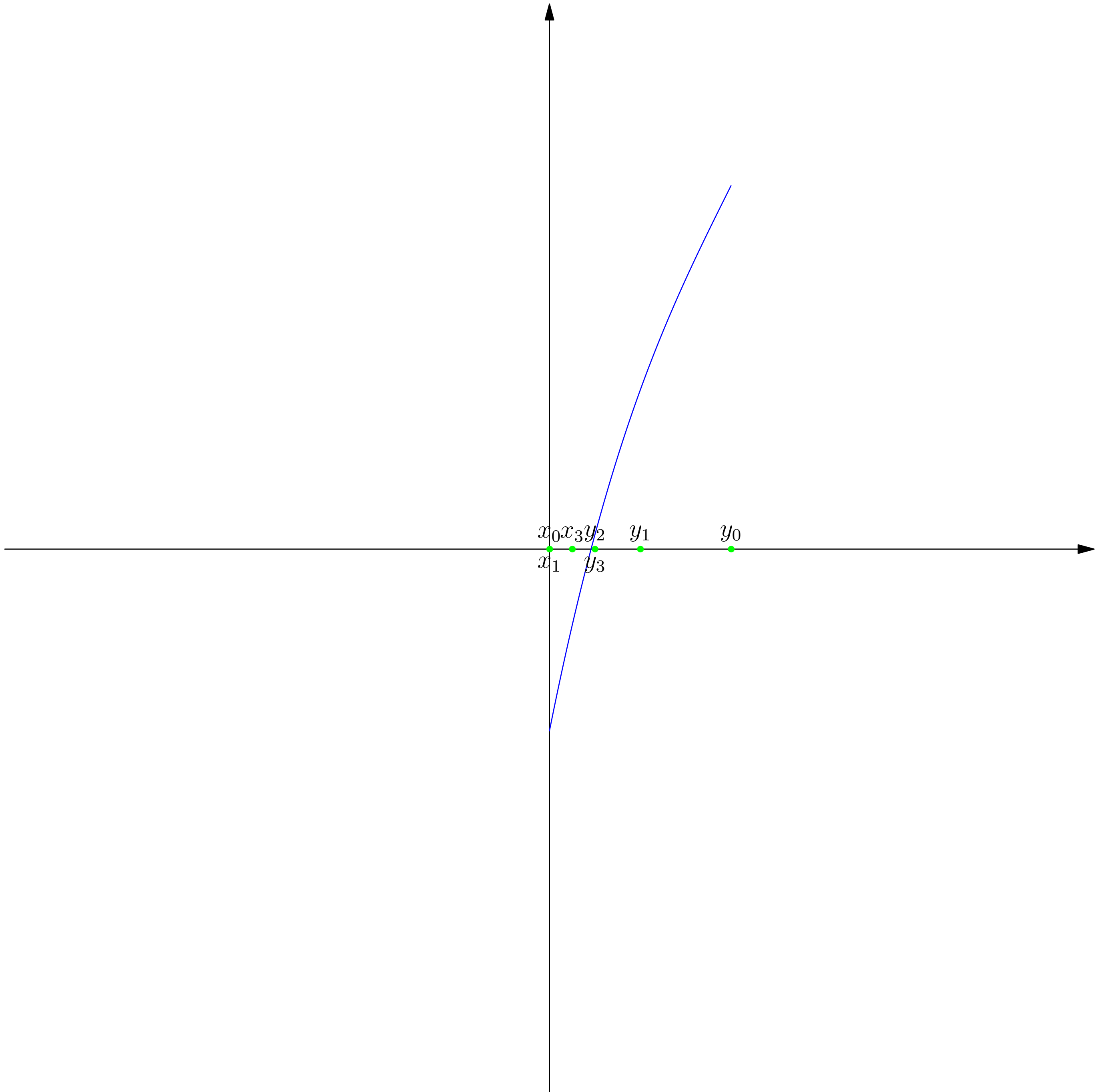
Voici un exemple avec la fonction $f : x \longmapsto x^3 - 3 * x^2 + 5 * x - 1$. Elle est continue et strictement croissante sur $[0, 1]$.

$f(0) = -1,\ f(1) = 1$. On a $x_0 = 0,\ y_0 = 1$.

$f(0.5) = 0.875,\ x_1 = x_0,\ y_1 = 0.5$.

$f(0.25) = 0.078125,\ x_2 = x_1 = x_0,\ y_2 = 0.25$.

$f(0.125) = -0.419921875,\ x_3 = 0.125,\ y_3 = 0.25$. A ce stade on a $0.125 < c < 0.25$.



la fonction suivante est définie de cette manière. Elle est appelée en utilisant les paramètres f, où f est la fonction intervenant dans l'équation, a, b extrémités de l'intervalle et e précision souhaitée. Le programme s'arrête lorsque $|y_n - x_n| \leq e$.

```
Algorithme : f, a, b, e
 Variables locales : g, d, x
 g := a
 d := b

 Tant que |d-g| > e, faire
 x := (d + g)/2
 si f(x) = 0 alors
 Renvoyer x
 Fin
 sinon si f(g).f(x) < 0 alors
 d := x
 sinon g := x
 fin si
 fin Tant que

 Renvoyer x
 Fin
```

Ce qui donne le programme suivant :

```
def dicho(f,a,b,e):
"""Determination d'une solution approchee a e pres de l'equation f(c)=0 sur l'intervalle [a,b]"""
    g=a
    d=b
    while (abs(d-g))>e:
        x=(d+g)/2
        if (f(x)==0): #On envisage le cas ou la valeur exacte de x est determinee.
            return x
        elif (f(g)*f(x)<0):
           d=x
         else:
            g=x
    return x
```

La boucle termine car la suite $(\dfrac{b-a}{2^k})$ converge vers zéro.
ce qui confirme que la valeur retournée par cet algorithme approche c à e près.

On peut ici utiliser comme invariant de boucle : $f(g)\,f(d) \leq 0$, ce qui confirme que la valeur retournée par cet algorithme approche c à e près.

```
def f(x):
return x**3-3*x**2+5*x-1

dicho(f,0,1,10e-10)
0.22908300254493952
```

Le calcul ci-dessus donne une approximation avec 10 décimales exactes de c pour la fonction précédente.

La complexité de cette méthode dépend de la fonction f car il faut exécuter $n = \log_2\left(\frac{b-a}{e}\right)$ calculs de $f(x)$, cette valeur de n est obtenue en remarquant que le nombre n de répétitions de la boucle est le plus petit entier tel que $\frac{b-a}{2^n} \leq e$.

Remarque

Il serait possible d'éviter de placer n dans les paramètres d'appel de la foction et de faire un calcul de cette valeur dans le corps du programme en prenant $n = E\left(\log_2\left(\frac{b-a}{e}\right)\right) + 1$.

Exercice

Evaluer le nombre d'itérations permetteant d'obtenir $y_n - x_n \leq 10^{-p},\ p \in \mathbb{N}^*$.

Ce résultat peut être vérifié en plaçant un compteur de boucle au sein de la boucle while.

Le deuxième algorithme fait intervenir cette remarque. Pour un ϵ donné on calcule le nombre d'itérations n_0 prévues et on introduit une sortie de boucle si soit $y_n - x_n \leq \epsilon$ soit $n > n_0$. cette deuxième version présente l'intérêt de terminer à coup sûr, car compte tenu des approximations de la norme IEEE 754, la relation $f(x_n)\,f(y_n) < 0$ pourrait être fausse sans que la machine le détecte.

```
def dicho2(f,a,b,e):
    """Determination d'une solution approchee a e pres de l'equation f(c)=0 sur l'intervalle [a,b]"""
    g=a
    d=b
    n=1+floor((log(b-a)-log(e))/log(2))
    i=0
    while (abs(d-g))>e and i<=n:
        x=(d+g)/2
        if (f(x)==0):
            return x
        elif (f(g)*f(x)<0):
            d=x
            i+=1
        else:
            g=x
            i+=1
    return x,i
```

Pour éviter une erreur de signe cet algorithme peut être amélioré en plaçant une condition d'arrêt dès que la valeur de $f(x_n)$ devient trop faible.

```
def dicho3(f,a,b,e):
    """Determination d'une solution approchee a e pres de l'equation f(c)=0 sur l'intervalle [a,b]"""
    g=a
    d=b
    n=1+floor((log(b-a)-log(e))/log(2))
    u=1e-12
    i=0
    while (abs(d-g))>e and i<=n and abs(f((g)))>u:
        x=(d+g)/2
        if (f(x)==0):
            return x
        elif (f(g)*f(x)<0):
            d=x
            i+=1
        else:
            g=x
            i+=1
    return x,f(x),i
```

3.5 Calcul approché d'intégrales

Lorsqu'on ne connait pas de primitive explicite pour une fonction continue sur un intervalle $[a, b]$, il est nécessaire d'utiliser des méthodes de calcul approché.

En général ces méthodes consistent à approximer la fonction par des polynômes sur cet intervalle. Nous allons étudier les cas des polynômes de degré zéro et de degré un.

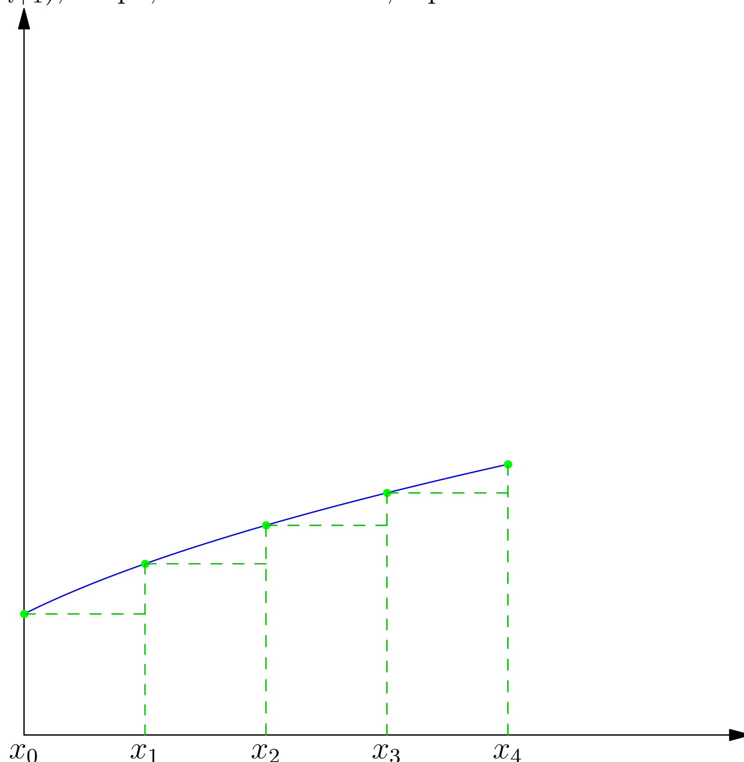
Le premier cas correspond à la méthode des rectangles, le deuxième à la méthode des trapèzes.

3.5.1 Méthode des rectangles

On considère une subdivision $\sigma = (x_0, x_1, \cdots, x_n),\ a = x_0 < x_1 < \cdots < x_n = b$ de l'intervalle $[a, b]$ et une fonction f continue sur cet intervalle.

On va faire un calcul approché de l'intégrale $\displaystyle\int_a^b f(x)dx$.

On représente ci-dessous la fonction $f : x \mapsto \sqrt{x+1}$ sur l'intervalle $[0, 4]$, $\sigma = (0, 1, 2, 3, 4)$. Sur chaque intervalle $[x_i, x_{i+1}]$ le calcul de l'intégrale de f est approximé par le calcul de l'intégrale de la fonction constante prenant la valeur $f(x_i)$ ou $f(x_{i+1})$ sur cet intervalle. La valeur de $\displaystyle\int_{x_i}^{x_{i+1}} f(x)dx$ est approchée par $(x_{i+1} - x_i).f(x_i)$ ou $(x_{i+1} - x_i).f(x_{i+1})$, ce qui, en valeur absolue, représente l'aire d'un rectangle.



Methode des rectangles

Une valeur approchée de l'intégrale dans ces conditions est donnée par :

$\displaystyle\sum_{i=0}^{3}(x_{i+1} - x_i)\, f(x_i) = 1 + \sqrt{2} + \sqrt{3} + 2$, dont une valeur approchée est 6.14.

Une primitive de la fonction f est connue : $F : x \longmapsto \dfrac{2}{3}(x+1)^{\frac{3}{2}}$,ce qui donne une valeur de l'intégrale, à 10^{-2} près de 6.79.

Dans le cas général énoncé au début, une valeur approchée de l'intégrale est donnée par :

$$I_n = \sum_{i=0}^{n-1}(x_{i+1} - x_i)\, f(x_i).$$

En général on fait le choix d'une subdivision régulière, c'est à dire que :

$$x_{i+1} - x_i = \frac{b-a}{n},\ i \in \{1, 2, \cdots n-1\},\ x_i = a + i\,\frac{b-a}{n},\ i \in \{0, 1, \cdots n\}.$$

On obtient le programme suivant où la subdivision est régulière et contient $n+1$ points.

```
def rectangle(f,a,b,n):
"""Calcul approchée de l'intégrale de f entre a et b par la methode des rectangles"""
    h=(b-a)/n
    s=0
    for i in range(n):
        s=s+h*f(a+i*h)
    return s

def f(x):
from math import sqrt
return sqrt(x+1)

>>> f(3)
2.0
>>> rectangle(f,0,4,4)
6.146264369941973
```

On démontre, à l'aide de l'inégalité de Taylor-Lagrange, que $\left|\int_a^b f(x)dx - I_n\right| \leq M_1 \frac{(b-a)^2}{2n}$ où f est de classe C^1 sur $[a,b]$, $M_1 = \sup_{[a,b]} |f'(x)|$, l'erreur est donc en $O(\frac{1}{n})$.

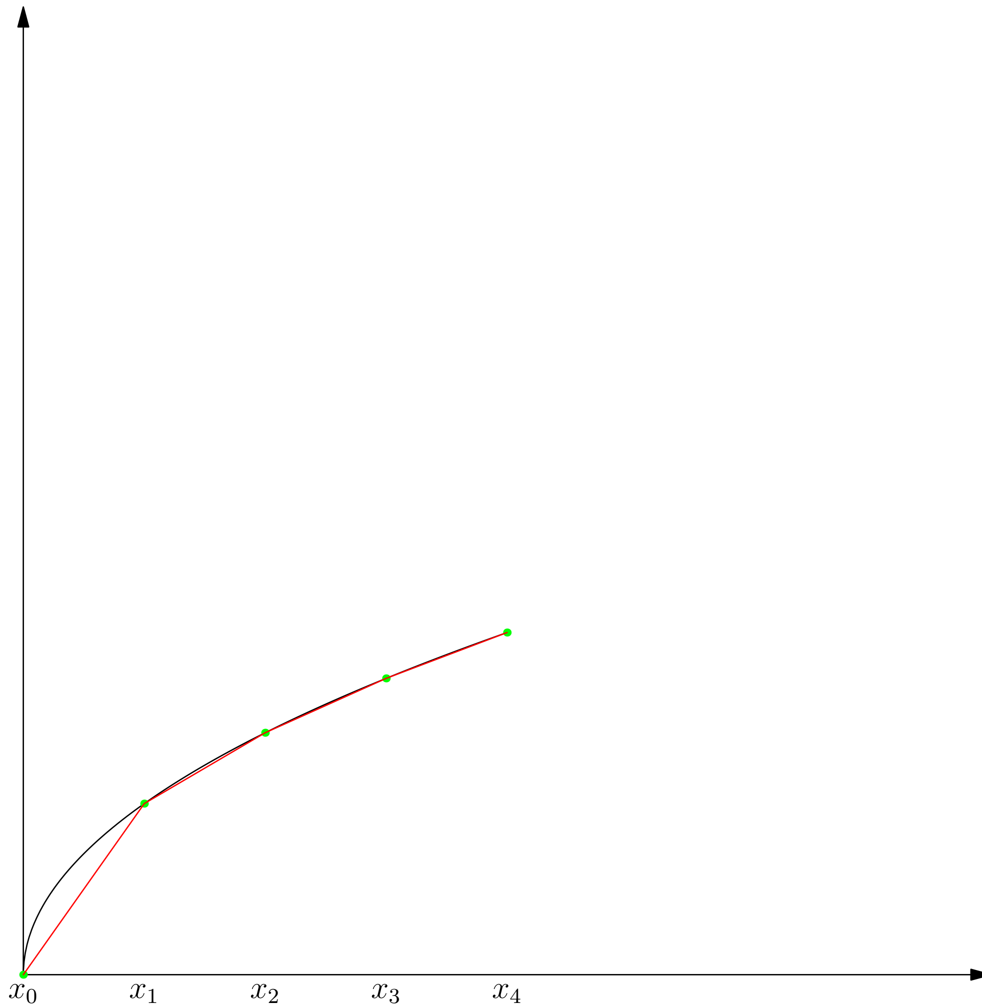
La méthode des rectangles revient à approcher la fonction f par une fonction en escalier.

3.5.2 Méthode des trapèzes

On considère à nouveau une subdivision $\sigma = (x_0, x_1, \cdots, x_n)$, $a = x_0 < x_1 < \cdots < x_n = b$ de l'intervalle $[a,b]$ et une fonction f continue sur cet intervalle.

Dans cette nouvelle méthode, l'arc de courbe compris entre $M_i(x_i, f(x_i))$ et $M_{i+1}(x_{i+1}, f(_{i+1}))$ est approché par le segment de droite passant par ces deux points.

Dans l'exemple suivant la fonction est $f : x \mapsto \sqrt{x}$, l'intervalle $[0,2]$, $\sigma = (0, 1/2, 1, 3/2, 2)$. Les segments de droites sont tracés en rouge.



Methode des trapezes

La valeur de l'aire algébrique du trapèze compris entre les valeurs x_i et x_{i+1} est $\dfrac{f(x_i)+f(x_{i+1})}{2}(x_{i+1}-x_i)$.

Dans le cas général on obtient en faisant le choix d'une subdivision réglière :

$$I_n = h\sum_{i=0}^{n-1}\frac{f(x_{i+1})+f(x_i)}{2} = h\left(\frac{1}{2}(f(a)+f(b))+\sum_{i=1}^{n-1}f(x_i)\right),\text{ où } h=\frac{b-a}{n}.$$

On obtient alors le programme suivant :

```
def trapeze(f,a,b,n):
    """Calcul approche d'integrale par la methode des trapeze"""
    h=(b-a)/n
    s=(1/2)*(f(a)+f(b))*h
    for i in range(1,n):
        s=s+h*f(a+i*h)

En repreant l'exemple traite par la methode des rectangles on obtient:

 trapeze(f,0,4,4)
6.7642983586918675
```

La valeur obtenue ci-dessus est bien plus proche de la valeur exacte que celle obtenue par la méthode des

rectangles. On montre que l'erreur est en $O(\frac{1}{n^2})$ pour une fonction de classe C^2 sur $[a, b]$ plus précisément :

$$\left| \int_a^b f(x)dx \ - \ I_n \right| \leq M_2 \, \frac{(b-a)^3}{12n^2} \text{ où } M_2 = \sup_{[a,b]} |f''(x)|.$$

Remarques

1) Le calcul de la formule des rectangles utilise les valeurs de gauche de chaque intervalle, il serait possible d'utiliser les valeurs de droite avec le même ordre d'erreur.

2) En prenant les valeurs des milieux de chaque intervalle, dans cette même méthode, on améliore sensiblement l'ordre de grandeur de l'erreur, qui est proche de celui obtenu par la méthode des trapèzes. Le programme est quasi identique à celui qui figure ci-dessus, en remplaçant $a + ih$ par $a + (i + 0.5)h$.

3) On dit qu'une méthode de calcul approché d'intégrale est d'ordre m si elle donne un résultat exact pour des polynômes de degré inférieur ou égal à m.
Par construction la méthode des rectangles est d'ordre zéro et la méthode des trapèzes d'ordre un.

3.5.2.1 Exercices

Exercice 1

Les algorithmes suivants calculent et affichent différentes listes de nombres.
Quelle est la complexité de chacun d'entre eux ?

```
def table1(n) :
    for i in range(11) :
        print(i * n)

def table2(n) :
    for i in range(n) :
        print(i * i)

def table3(n) :
    for i in range(n) :
        for j in range(n) :
            print(i * j, end=" ")
    print()
```

Exercice 2

Quelle est la complexité en temps d'un algorithme de division euclidienne procédant par soustractions successives ? Et sa complexité en espace ?

Exercice 3

Déterminer la complexité de l'algorithme de détermination du plus petit commun multiple de deux entiers, présenté en exercice dans le chapitre précédent.

Exercice 4

Ecrire un programme donnant les deux plus grands termes d'un tableau d'entiers contenant au moins deux éléments, en ne parcourant la liste qu'une seule fois.

Exercice 5

Ecrire une fonction **partition(n)** qui construit un tableau x avec n éléments tels que
$x_0 = 0 \leq x_1 \leq x_2 \leq \cdots \leq x_n = 1 = 1$ et où les $x_i,\ i \in [\![1, n-2]\!]$ sont obtenus au hasard uniforme sur $[0\,;\,1]$.
Aide : vous pouvez utiliser les fonctions **random.rand, concatenate**[1] et la méthode **sort** de numpy.

Exercice 6

L'algorithme d'Euclide à l'origine était le suivant. Soit $(a, b) \in \mathbb{N}^{*2}$, $b < a$. On pose $c = a - b$ puis $a_1 = \max(b, c)$, $b_1 = \min(b, c)$, $c_1 = a_1 - b_1$, $a_2 = \max(b_1, c_1)$, $b_2 = \min(b_1, c_1)$.... Les suites (a_k) et (b_k) sont strictement décroissantes et au bout d'un nombre fini d'étapes on obtient $a_k = b_k$ (on ne demande pas de vérifier cette propriété). Cette valeur commune est le pgcd de a et b.
Exemple.

1. Pour ajouter le tableau array([0, 1]) aux n-2 nombres aléatoires.

a	b	c
168	105	63
105	63	42
63	42	21
42	21	21
21	21	

le pgcd de 168 et 105 est 21.
Ecrire une fonction ou une procédure pgcd mettant en oeuvre cet algorithme.

Exercice 7

On veut mettre en oeuvre un algorithme effectuant le produit de deux polynômes. Par exemple
$P = a_0 + a_1X + a_2X^2,\ Q = b_0 + b_1X + b_2X^2.$
$R = PQ = c_0 + c_1X + c_2X^2 + c_3X^3 + c_4X^4 =$
$a_0b_0 + (a_0b_1 + a_1b_0)X + (a_2b_0 + a_1b_1 + a_0b_2)X^2 + (a_2b_1 + a_1b_2)X^3 + a_2b_2X^4.$
Les polynômes peuvent être représentés par des tableaux.
$P = [a_0, a_1, a_2],\ Q = [b_0, b_1, b_2], R = [c_0, c_1, c_2, c_3, c_4]$. Les coefficients c_i peuvent être obtenus par itération, par exemple, en supposant que les éléments de R sont initialisés à 0, on peut calculer c_2 de la maniére suivante :
$c_2 = c_{0+2} \leftarrow c_2 + a_0b_2$
$c_2 = c_{1+1} \leftarrow c_2 + a_1b_1$
$c_2 = c_{2+0} \leftarrow c_2 + a_2b_0$
où la flèche $\leftarrow$ représente l'affectation.
Ecrire une fonction def prod(P,Q, p,q) :
où les tableaux P et Q représentent deux polynômes de degrés respectifs p et q, effectuant le produit de ces deux polynômes. Quelle est le nombre de multiplications et le nombre d'additions effectuées par la procédure ?

Exercice 8

Ecrire une fonction nbrecons(A) :
qui calcule le plus grand nombre de termes consécutifs dans un tableau A.
Exemple $A = [4, 3, 1, 2, 5, 6, 7, 10, 8, 9]$ ici $nbrecons = 3$.

Exercice 9

Il existe de nombreuses méthodes de tri d'un tableaux ou d'une liste de nombres dans l'ordre croissant (ou décroissant), la méthode **sort** de python permet ainsi de trier une liste. (**l.sort()** pour trier la liste l). Le tri à bulle est une de ces méthodes de tri :
L'algorithme parcourt le tableau, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet du tableau, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié. On arrête alors l'algorithme.
Exemple : $l = [5, 1, 4, 2, 8]$.

Première étape :
(51428) $\rightarrow$ (15428) Les éléments 5 et 1 sont comparés, et comme $5 > 1$, l'algorithme les intervertit.
(15428) $\rightarrow$ (14528) Interversion car $5 > 4$.
(14528) $\rightarrow$ (14258) Interversion car $5 > 2$.
(14258) $\rightarrow$ (14258) Comme $5 < 8$, les éléments ne sont pas échangés.

Deuxième étape :
(14258) → (14258) Même principe qu'à l'étape 1.
(14258) → (12458)
(12458) → (12458)
(12458) → (12458)
À ce stade, la liste est triée, mais pour le détecter, l'algorithme doit effectuer un dernier parcours.

Comme la liste est triée, aucune interversion n'a lieu à cette étape, ce qui provoque l'arrêt de l'algorithme.

Ecrire une fonction tri_bulle(l) : permettant de trier un tableau l par cette méthode, deux boucles imbriquées seront nécessaires.

Voici l'algorithme en pseudo code

POUR i de 1 à N FAIRE
    POUR k de 1 à N - i FAIRE
        SI $T[k] > T[k+1]$ ALORS
        $T[k] \leftrightarrow T[k+1]$
        FIN SI
    FIN POUR
FIN POUR

Déterminer la complexité de l'algorithme lorsque le plus petit élément est en dernière position.

Exercice 10

1) Ecrire un programme transformant un entier n écrit en décimal en l'écriture binaire de n considérée comme une chaine de caractère. Rappel : effectue une suite successive de divisions : n par 2, puis le quotient par 2 et ainsi de suite jusqu'à l'obtention d'un quotient nul. La repésentation binaire de n est alors la suite des restes écrite depuis le dernier obtenu jusqu'au premier.
Exemple : 23=2*11+1, 11=2*5+1, 5=2*2+1, 2=2*1+0 1=2*0+1 c'est terminé. l'écriture binaire est alors 10111.
2) Reprendre la question précédente en remplaçant deux par huit.
3) Exponentielle rapide : La cryptologie nécessite le calcul de très grandes puissances. On veut élever l'entier a à la puissance n.
Première étape : décomposer n en binaire considéré comme chaine de caractères.
$n = (c_0\, c_1\, \cdots\, c_{p-1})_2$ en binaire où $c_i \in \{0, 1\}$.

Initialisation : $z \leftarrow 1$
Pour i variant de 0 à $p-1$:
    $z \leftarrow z^2$
    si $c_i == 1$:
        $z \leftarrow z * a$
donner z

Ecrire le programme correspondant à cet algorithme. Montrer que la complexité de cet algorithme est de $\log_2 n$. Calculer 2^{500} en utilisant ce programme. Quel est le nombre maximum de multiplications nécéssaires pour obtenir ce résultat.
Combien de chiffre contient ce nombre écrit en décimal.

Chapitre 4

Calcul numérique

4.1 Pivot de Gauss

4.1.1 Généralités

Un système de n équations à p inconnues, ou système $n \times p$ est un système

$$(S) \left\{ \begin{array}{lcl} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1p}x_p & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2p}x_p & = & b_2 \\ & \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{np}x_p & = & b_n \end{array} \right.$$

dans lequel :

- les a_{ij} sont des nombres réels ou complexes.
- $x_1, x_2, \cdots, x_p$ sont les inconnues réelles ou complexes.
- une solution est un p-uplet $(x_1, x_2, \cdots, x_p)$ vérifiant le système (S). Résoudre (S) revient à déterminer toutes les solutions de ce système.
- Le système homogène associé à (S) est le système dans lequel tous les b_i sont remplacés par 0.
- Un système est impossible, ou incompatible, s'il n'admet pas de solution. Un système est possible, ou compatible, s'il admet une ou plusieurs solutions.
- Deux systèmes sont équivalents s'ils admettent les mêmes solutions.

Représentation matricielle

En notant :

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{np} \end{pmatrix}, \ X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}, \ B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

le système s'écrit sous forme matricielle :

$AX = B$.

L'algorithme du pivot de Gauss permet de résoudre un système linéaire au sens où, partant d'un système à n équations et p inconnues, il va fournir un système équivalent permettant de paramétrer l'ensemble des solutions (s'il est non vide), ou de démontrer qu'il n'y a pas de solution en fournissant une condition nécessaire non compatible.

Cet algorithme est basé sur les opérations élémentaires sur les lignes du système, elles sont au nombre de trois :

— $L_i \longleftarrow L_i + \lambda L_j,\ \lambda \in K^*$. Ajout à une ligne d'un multiple d'une autre ligne.
— $L_i \longleftarrow \lambda L_i,\ \lambda \in K^*$. Multiplication d'une ligne par un scalaire non nul.
— $L_i \longleftrightarrow L_j$. Echange de deux lignes.
Il existe d'autres applications du pivot de Gauss :
— Inverser une matrice (inversible).
— Déterminer L et U deux matrices triangulaires (respectivement inférieure et supérieure) telles que $A \in \mathcal{M}_n(\mathbb{R})$ s'écrive LU.
— Calculer le déterminant de $A \in \mathcal{M}_n(\mathbb{R})$.
— Déterminer le rang de $A \in \mathcal{M}_n(\mathbb{R})$.

4.1.2 Exemple

Soit à résoudre le système linéaire $Ax = b$ avec

$$A = \begin{pmatrix} 2 & 3 & -1 & 3 \\ 4 & 6 & 0 & 2 \\ -1 & -2 & 0 & 5 \\ -1 & 4 & 5 & 2 \end{pmatrix}, \qquad b = \begin{pmatrix} 11 \\ 6 \\ 21 \\ 3 \end{pmatrix}$$

Procédons à l'élimination de Gauss ; la 1ère étape est [1]

$$\begin{pmatrix} 2 & 3 & -1 & 3 \\ 4 & 6 & 0 & 2 \\ -1 & -2 & 0 & 5 \\ -1 & 4 & 5 & 2 \end{pmatrix} \begin{matrix} \urcorner^{-2} \quad \urcorner^{1/2} \quad \urcorner^{1/2} \\ \leftarrow + \\ \longleftarrow + \\ \longleftarrow + \end{matrix} \quad \text{qui donne} \quad \begin{pmatrix} 2 & 3 & -1 & 3 \\ 0 & 0 & 2 & -4 \\ 0 & -1/2 & -1/2 & 13/2 \\ 0 & 11/2 & 9/2 & 7/2 \end{pmatrix}$$

La 2ème étape est

$$\begin{pmatrix} 2 & 3 & -1 & 3 \\ 0 & 0 & 2 & -4 \\ 0 & -1/2 & -1/2 & 13/2 \\ 0 & 11/2 & 9/2 & 7/2 \end{pmatrix} \begin{matrix} \\ \leftrightarrow \quad \urcorner^{11} \\ \leftarrow \\ \longleftarrow + \end{matrix} \quad \text{qui donne} \quad \begin{pmatrix} 2 & 3 & -1 & 3 \\ 0 & -1/2 & -1/2 & 13/2 \\ 0 & 0 & 2 & -4 \\ 0 & 0 & -1 & 75 \end{pmatrix}$$

Enfin la dernière étape consiste en

$$\begin{pmatrix} 2 & 3 & -1 & 3 \\ 0 & -1/2 & -1/2 & 13/2 \\ 0 & 0 & 2 & -4 \\ 0 & 0 & -1 & 75 \end{pmatrix} \begin{matrix} \\ \\ \urcorner^{1/2} \\ \leftarrow + \end{matrix} \quad \text{qui donne} \quad \begin{pmatrix} 2 & 3 & -1 & 3 \\ 0 & -1/2 & -1/2 & 13/2 \\ 0 & 0 & 2 & -4 \\ 0 & 0 & 0 & 73 \end{pmatrix}$$

Les mêmes transformations appliquées au vecteur b,

$$\begin{pmatrix} 11 \\ 6 \\ 21 \\ 3 \end{pmatrix} \begin{matrix} \urcorner^{-2} \quad \urcorner^{1/2} \quad \urcorner^{1/2} \\ \leftarrow + \\ \longleftarrow + \\ \longleftarrow + \end{matrix} \quad \begin{matrix} \\ \leftrightarrow \quad \urcorner^{11} \\ \leftarrow \\ \longleftarrow + \end{matrix} \quad \begin{matrix} \\ \\ \urcorner^{1/2} \\ \leftarrow + \end{matrix} \quad \text{nous donne} \quad \begin{pmatrix} 11 \\ 53/2 \\ -16 \\ 292 \end{pmatrix}.$$

Par remontée du système on trouve

$$x = \begin{pmatrix} 1 \\ -1 \\ 0 \\ 4 \end{pmatrix}.$$

1. Deviner ce que veulent dire les flèches et nombres placés à droite des matrices !

4.1.3 Etude générale et problèmes

Décrivons ce qu'on fait pour un système (3, 3) d'inconnues (x, y, z) « lorsque tout se passe bien », en notant L1, L2 et L3 les trois équations en jeu. On suppose que le coefficient en x de la première équation, disons a, est non nul. On va s'en servir comme pivot pour éliminer les autres occurrences de x. Si on note b et c les coefficients en x des deuxième et troisième lignes, le système constitué des équations
$L_1, L_2' = L_2 - \frac{b}{a}L_1$, et $L_3' = L_3 - \frac{c}{a}L_1$
est alors équivalent au premier (le démontrer n'est pas inutile) et ne fait apparaître x que dans la première équation : en supposant que le coefficient en y de la nouvelle deuxième ligne L_2', disons d, est non nul (c'est alors le nouveau pivot) et en notant e celui de y dans la troisième nouvelle ligne L_3', le système constitué des lignes
$L1, L_2'$ et $L_3' - \frac{e}{d}L_2'$
est équivalent au premier système et triangulaire :
on est ramené à un cas qu'on sait traiter. Lors de la première étape, on ne touche pas à la première ligne. De même, à la deuxième étape, on ne touche ni à la première ni à la deuxième ligne, etc...

Dans l'exemple qui suit, on adopte une notation classique : pour dire qu'on change la seconde ligne
L_2 en $L_2' = L_2 + \alpha L_1$, on préférera noter
$L_2 \longleftarrow L_2 + \alpha L_1$, ce qui signifie : à partir de maintenant, ce qu'on appelle L_2, c'est ce qui désignait avant $L_2 + \alpha L_1$. Après trois opérations de ce type, on parlera donc toujours de L_8 plutôt que de L_8''' .

Exemple

$$\left\{\begin{array}{lcl} x_1 + 2x_2 + 3x_3 + 4x_4 & = & 1 \qquad L_2 \leftarrow L_2 - 2L_1 \\ 2x_1 + 3x_2 + 4x_3 + x_4 & = & 2 \qquad L_3 \leftarrow L_3 - 3L_1 \\ 3x_1 + 4x_2 + x_3 + 2x_4 & = & 3 \qquad L_4 \leftarrow L_4 - 4L_1 \\ x_2 + 2x_3 + 3x_4 & = & 4 \end{array}\right.$$

donne

$$\left\{\begin{array}{rcl} x_1 + 2x_2 + 3x_3 + 4x_4 & = & 1 \\ -x_2 - 2x_3 - 7x_4 & = & 0 \\ -2x_2 - 8x_3 - 10x_4 & = & 0 \qquad L_3 \leftarrow L_3 - 2L_2 \\ -7x_2 - 10x_3 - 13x_4 & = & 0 \qquad L_4 \leftarrow L_4 - 7L_2 \end{array}\right.$$

ce qui donne le système :

$$\left\{\begin{array}{rcl} x_1 + 2x_2 + 3x_3 + 4x_4 & = & 1 \\ -x_2 - 2x_3 - 7x_4 & = & 0 \\ -4x_3 + 4x_4 & = & 0 \\ -4x_3 + 36x_4 & = & 0 \end{array}\right. \quad L_4 \leftarrow L_4 + L_3 \text{ donne le système triangulaire } \left\{\begin{array}{rcl} x_1 + 2x_2 + 3x_3 + 4x_4 & = & 1 \\ -x_2 - 2x_3 - 7x_4 & = & 0 \\ -4x_3 + 4x_4 & = & 0 \\ -40x_4 & = & 0 \end{array}\right.$$

On obtient finalement :
$x_4 = x_3 = x_2 = 0,\ x_1 = 1.$

Décrivons maintenant les différents problèmes qui peuvent arriver en cours de résolution d'un système (3, 3), ainsi que leur solution.

— Le pivot n'est pas là où on veut : si à la première étape le coefficient en x de la première ligne est nul, on peut échanger la première équation avec la deuxième ou la troisième. De même, si à la seconde étape, le coefficient en y (futur pivot) est nul, on peut échanger la deuxième équation avec la troisième, mais pas la première (se souvenir qu'on veut arriver à un système triangulaire : il ne faut pas faire réapparaître x dans les deux dernières équations).
— Il n'y a plus de pivot en une variable : si tous les coefficients en x sont nuls (c'est rare : cela revient à dire que x n'apparaît pas dans le système...), on peut prendre y ou z comme première variable. De même, si

après la première étape, y n'apparaît ni dans la deuxième ni dans la troisième équation, on peut prendre z comme deuxième inconnue.

— Il n'ya plusdepivot : cela signifie que les membres de gauche des équations restantes sont nuls. Selon que les membres de droite correspondants sont nuls ou pas, ces équations vont disparaître, ou bien rendre le système incompatible. Les deux dernières situations ne se produiront pas si le système est de Cramer.

Choix du pivot dans le cas général

Lorsque l'on manipule des fractions, les calculs donnent des résultats exacts. Ce n'est plus le cas lorsque l'on utilise des nombres flottants. Deux nombres peuvent être distincts et avoir des représentations identiques par des flottants.
Par ailleurs la division par des nombres très petits en valeur absolue peut également amener des erreurs de calcul, ce qui peut poser des problèmes dans l'utilisation de pivots de petite valeur absolue.

Les calculs réels sont en fait entachés d'erreurs d'arrondi. Ceci a pour conséquence que le résultat final dépendra fortement du choix des pivots (ou encore de l'ordre des variables éliminées). On peut voir en gros ce qui se passe. On suppose que les variables $x_1, x_2, \cdots, x_{k-1}$ ont été éliminées. Pour éliminer l'inconnue x_k, on multiplie la ligne k par $\frac{a_{ik}}{a_{kk}}$. Si le pivot a_{kk} est petit, toutes les erreurs qui affectent les coefficients a_{ik} seront amplifiées. En conséquence, on adopte toujours la stratégie de «recherche du pivot partiel» ; pour éliminer la variable x_k et quelque soit la valeur de $|a_{kk}|$, on recherche l'élément de plus grand module dans la colonne k et c'est lui que l'on prendra comme pivot effectif, après avoir fait la permutation de lignes nécessaire.

Exemple

$$(S_1)\left\{\begin{array}{lcl} x_1 + x_2 & = & 1 \\ 10^{-k}x_1 + x_2 & = & 0.5 \end{array}\right. \quad (S_2)\left\{\begin{array}{lcl} 10^{-k}x_1 + x_2 & = & 0.5 \\ x_1 + x_2 & = & 1 \end{array}\right.$$

En supprimant x_1 dans chaque système on obtient :

$$(S_1)\left\{\begin{array}{lcl} x_1 + x_2 & = & 1 \\ (1-10^{-k})x_2 & = & 0.5 - 10^{-k} \end{array}\right. \quad (S_2)\left\{\begin{array}{lcl} 10^{-k}x_1 + x_2 & = & 0.5 \\ (1-10^{k})x_2 & = & 1 - 05 \times 10^{k} \end{array}\right.$$

En utilisant des réels double précision on obtient les résultats suivants :

	Système I	Système II
k=10	$x_1 = 0.5,\ x_2 = 0.5$	$x_1 = 0.5,\ x_2 = 0.5$
k=12	$x_1 = 0.5,\ x_2 = 0.5$	$x_1 = 0.499989,\ x_2 = 0.5$
k=14	$x_1 = 0.5,\ x_2 = 0.5$	$x_1 = 0.489600,\ x_2 = 0.5$
k=16	$x_1 = 0.5,\ x_2 = 0.5$	$x_1 = 1.110223,\ x_2 = 0.5$

On constate donc que la résolution du système II donne des résultats erronés dès que k devient suffisamment grand, alors que la résolution du système I reste parfaitement stable.

Remarque

Tester l'égalité de deux flottants n'a presque jamais de sens, est toujours dangereux, et doit donc être évité en général.

4.1.4 Description de l'algorithme

On considère que le système est de Cramer et qu'il a donc toujours une solution.

Comme signalé plus haut, on veut éliminer des variables dans les équations successives. On va donc faire en sorte qu'après k étapes, pour tout i entre 1 et k, la i-ème variable ait disparu de toutes les équations du système à partir de la (i + 1)-ème. Ce sera l'invariant de boucle.
Ainsi, après la (n - 1)-ème étape, le système sera bien sous forme triangulaire.

Dans le pseudo-code qui suit, on résout le système Ax = y. La ligne L_i désigne à la fois les coefficients de A (qui sont dans une matrice, un tableau bidimensionnel) et les seconds membres, qui sont dans une matrice colonne y. Les indexations de tableaux sont « à la Python » : de 0 à n - 1.

Pour i variant de 0 à n-2 faire
    Trouver j entre 1 et n-1 tel que $|a_{ji}|$ soit maximal
    Echanger L_i et L_j ne pas oublier le membre de droite
    Pour k de i+1 à n-1 faire
        $L_k \leftarrow L_k - \dfrac{a_{ki}}{a_{ii}} L_i$

Arrivé ici, le système est sous forme triangulaire, et il n'y a plus qu'à « remonter », via des substitutions. Le résultat est mis dans un tableau x, et il s'agit donc de calculer :

$$x_i = \frac{1}{a_{ii}} \left(y_i - \sum_{k=i+1}^{n-1} a_{ik} x_k \right).$$

ce qui est calculé par l'algorithme suivant :

Pour i de n - 1 à 0 faire
    pour k de i + 1 à n -1 faire
        $yi \leftarrow yi - a_{ik} x_k$
    $x_i \leftarrow \dfrac{y_i}{a_{ii}}$

4.1.4.1 Mise en oeuvre de l'algorithme

Il est préférable de décomposer le problème de la résolution de manière à disposer de programmes simples qui réunis permettront de résoudre le système.

1) Détermination du pivot.
2) Echange éventuel de lignes.
3) Elimination par opération élémentaire.

Remarque

La création et la manipulation de matrices est nettement plus aisée en chargeant le module numpy.

```
from numpy import *

 A=array([[1,2,3],[4,5,6],[7,8,9]])
 A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

 len(A)
3 # Nombre de lignes de la matrice A

len(A[0])
3 # Nombre de colonne de la matrice A
```

Ecrivons les trois programmes réalisant les points précédents.

```
1)
def piv_max(A,i):
    """  recherche du pivot maximal"""
    n=len(A) # le nombre de lignes
    j=i
    for k in range(i+1,n):
        if abs(A[k,i])>abs(A[i,i]):
            j=k
    return j

2)
def echange(A,b,i,j):
     """ echange des lignes i et j """
     m=len(A[0]) # nombre de colonnes
     b[j,0],b[i,0]=b[i,0],b[j,0]
     for k in range(m):
         A[i,k],A[j,k]=A[j,k],A[i,k]

3)
def elimination(A,b,j,i,p):
    """Lj<-Lj+p.Li"""
    m=len(A)
    b[j,0]=b[j,0]+p*(b[i,0])
    for k in range(m):
        A[j,k]=A[j,k]+p*A[i,k]
```

Programme complet

Il est préférable de conserver les matrices initiales et d'en faire des copies qui elles pourront être modifiées. L'instruction B=A ne réalise pas une copie de la matrice A mais représente seulement un alias de A car les matrices comme les listes sont des objets mutables. Une copie véritable de A s'obtient par l'instruction B=deepcopy(A) qui nécessite de charger le module copy.

```
def resolution(A,b):
    """resolution de A.X=b, A matrice inversible"""
    A1,b1=deepcopy(A),deepcopy(b)
    n=len(A1)
    # transformation en systeme triangulaire
    for i in range(n):
        j=piv_max(A1,i)
        if j>i:
            echange(A1,b1,i,j)
        for k in range(i+1,n):
            m=A1[k,i]/(A1[i,i])
            elimination(A1,b1,k,i,-m)
    #resolution systeme
    X=zeros_like(b1)
    for i in range(n-1,-1,-1):
        X[i,0]=(b1[i,0]-sum(A1[i,j]*X[j,0] for j in range(i+1,n)))/A1[i,i]
    return X
```

Remarques

1) La division ne se comporte pas de la même manière dans les version de Python 2 et de Python 3. Dans la version 3 l'expression 1/3 est représenté par un nombre flottant alors que dans les versions 2 ceci représente le quotient dans la division euclidienne de 1 par 3, c'est à dire zéro. Certaines fonctionnalités de Numpy utilisant Python 2, il est préférable d'entrer les données sous forme de nombres flottants.
Dans le cas d'un tableau ou matrice numpy, il suffit que l'une des entrées soit un flottant.

```
A=array([[1.,1],[2,3]])
b=array([[2],[5.]])

 A
array([[ 1.,  1.],
       [ 2.,  3.]])
 b
array([[ 2.],
       [ 5.]])

>>> resolution(A,b)
array([[ 1.],
       [ 1.]])

B=array([[1,1],[2,3]])
c=array([[2],[5]])

resolution(B,c)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    resolution(B,c)
  File "/home/ubuntu/Documents/info/gauss.py", line 42, in resolution
    X[i,0]=(b1[i,0]-sum(A1[i,j]*X[j,0] for j in range(i+1,n)))/A1[i,i]
ValueError: cannot convert float NaN to integer
```

Cet exemple montre que le deuxième calcul, dans lequel les matrices sont entières, conduit à une erreur.

2) Dans le cas d'un tableau ou matrice A à deux dimensions, $len(A)$ représente le nombre de lignes de A.

3) L'instruction $zeros_like(b)$ crée un tableau de la même taille que b que l'on peut ensuite remplir avec d'autres valeurs. L'instruction *ones_like* est également disponible.

Vous pouvez trouver une documentation sur Numpy, Scipy et Matplotlib à l'adresse :
http ://jomellac.free.fr/informatique/intronumpy.pdf.

Utilisation de bibliothèques

Le modules contient de nombreuses applications, dont la résolution de systèmes linéaires, c'est l'application **linalg.solve** qui s'utilise sous la forme **linalg.solve(A,b)** où A est la matrice représentant le système et b le second membre. Cette application transforme automatiquement une matrice d'entiers en matrice de flottants.

```
A=array([[1,1],[2,3]])

b=array([[2],[5]])


linalg.solve(A,b)
array([[ 1.],
       [ 1.]])
```

Exemples de résolutions

```
A=array([[2.,2,-3],[-2,-1,-3],[6,4,4]])

b=array([[2.],[-5],[16]])
#Algorithme du cours

 resolution(A,b)
array([[-14.],
       [ 21.],
       [  4.]])

#Application numpy
 linalg.solve(A,b)
array([[-14.],
       [ 21.],
       [  4.]])
```

Le système suivant n'a pas de solution et malgré tout dans les deux cas, algorithme cours et numpy, ne le détermine pas :

$$\left\{ \begin{array}{rcl} x_1 + \frac{1}{4}x_2 + x_3 & = & 0 \\ x_1 + \frac{1}{3}x_2 + 2x_3 & = & 0 \\ x_2 + 12x_3 & = & 1 \end{array} \right.$$

```
A=array([[1,1./4,1],[1,1./3,2],[0,1,12]])

b=array([[0],[0],[1.]])

resolution(A,b)
array([[ -7.50599938e+14],
       [  4.50359963e+15],
       [ -3.75299969e+14]])

linalg.solve(A,b)
array([[ -7.50599938e+14],
       [  4.50359963e+15],
       [ -3.75299969e+14]])
```

Ce résultat est dû à l'approximation de 1/4 et 1/3, les lignes $L_2 - L_1$ et L_3 sont équivalentes lorsque l'on utilise ces fractions, elles ne le sont plus quand il s'agit simplement d'approximations décimales. Le résultat de cet (ces) algorithmes est donc à consid'érer avec un oeil critique.

<u>Complexité</u>

Phase 1 : recherche du pivot : Pour i fixé il y a $n-i$ comparaisons.
Phase 2 : échange éventuel : au maximum $(2n+2)$ affectations.
phase 3 : opération élémentaire : $2n+2$ affectations et le même nombre de multiplications, divisions et additions (ou soustractions), soit $(n-1-i)(2n+2)$ pour i fixé.
total : $\sum_{i=0}^{n-1} n-i+(2n+2)+(n-1-i)(2n+2) = \sum_{i=0}^{n-1}(2n+3)(n-i) = (2n+3)\dfrac{n(n+1)}{2}$.
phase 4 : remontée : à i fixé on a un maximum de n opérations soit une complexité de l'ordre de n^2.
La complexité de l'ensemble est donc en $O(n^3)$.
Dans les problèmes intervenant en mécanique, dans la méthode des éléments finis, il y a des systèmes pour lesquels n est très grand, il est alors nécessaire d'exploiter la forme particulière de certaines matrices pour pouvoir mener à bien la résolution. C'est le cas des matrices creuses (contenant beaucoup de zéros) ou symétriques ($a_{ji} = a_{ij}$ pour lesquelles des algorithmes ont été développés pour pouvoir approcher des complexités en $O(n^2)$. Par ailleurs des techniques relevant de l'informatique sont également utilisées pour rendre ces calculs possibles, c'est le cas de la parallèlisation des calculs. Plusieurs calculs sont effectués simultanèment par plusieurs processeurs ce qui permet d'accélérer sensiblement les calculsle traitement.
On peut retenir comme ordre de grandeur qu'un ordinateur personnel va réaliser de l'ordre de 10^9 opérations élémentaires dans une minute. Par exemple, un algorithme « en n^2» s'exécutera en temps raisonnable si $n = 10^4$, mais est à proscrire si $n = 10^7$.

Il est nécessaire dans un algorithme de tenir compte de :

— L'impact des erreurs d'arrondi sur les résultats.
— Le temps de calcul,
— Le stockage des données en mémoire.

La première affecte essentiellement la confiance que l'on peut avoir dans les résultats fournis par un programme. Si les deux dernières sont trop critiques, l'algorithme n'a qu'un intérêt théorique.
On l'a vu sur l'exemple du pivot de Gauss, les méthodes numériques peuvent rapidement poser problème de ces trois points de vue.

Une autre amélioration de cet algorithme est ce que l'on appelle le conditionnement d'une matrice, qui nécessite la notion de valeur propre qui est au programme de deuxième année.

4.2 Résolution d'équation par la méthode de newton

Il s'agit de résoudre une équation du type $f(x) = 0$. Une première étude a été réalisée précedemment. Il s'agit de la méthode de résolution de ce type d'équation par dichotomie.

Cette deuxième méthode converge plus rapidement que la précédente vers la solution sous " de bonnes conditions " qui sont les suivantes.

La fonction f est de classe C^1 sur l'intervalle $[a, b]$ et la dérivée ne s'annule pas sur cet intervalle.

Le principe est le suivant. On choisit une abcsisse x_0 'assez proche' de la solution α à déterminer, puis on mène la tangente à la courbe représentative en $A_0(x_0, f(x_0))$, cette tangente coupe l'axe des abscisses en

$x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$ résultat obtenu en remarquant que l'équation de la tangente en A_0 est :

$y - f(x_0) = f'(x_0)(x - x_0)$, il suffit de prendre $y = 0$ pour obtenir ce résultat. On réitère ceci en prenant $A_1(x_1, f(x_1))$, ce qui permet d'obtenir de proche en proche une suite (x_n) définie par la relation de récurrence :

$x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$ avec $\phi(x) = x - \dfrac{f(x)}{f'(x)}$ on a $x_{n+1} = \phi(x_n)$. si cette suite converge vers $l = \phi(l)$ on a

$e_{n+1} = l - x_{n+1} = \phi(l) - \phi(x_n) = \phi'(\xi_n)(l - x_n)$ en utilisant l'égalité

$e_n = (l - x_{n+1}) + (x_{n+1} - x_n) = \phi'(\xi_n)e_n$, on obtient :

$e_n = \dfrac{x_{n+1} - x_n}{1 - \phi'(\xi_n)}$. Ceci montre que le test d'arrêt de l'algorithme correspondant prenant en compte la valeur de e_n est de bonne qualité lorsque $\phi'(x)$ est proche de zéro dans un voisinage de l et très mauvais lorsque $\phi'(x)$ est proche de 1 avec $\phi'(x) < 1$, la suite ne converge pas lorsque $|\phi'(x)| > 1$.

Exemple : $f(x) = x^2 - 2$.

Cette suite est définie par la récurrence :

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right).$$

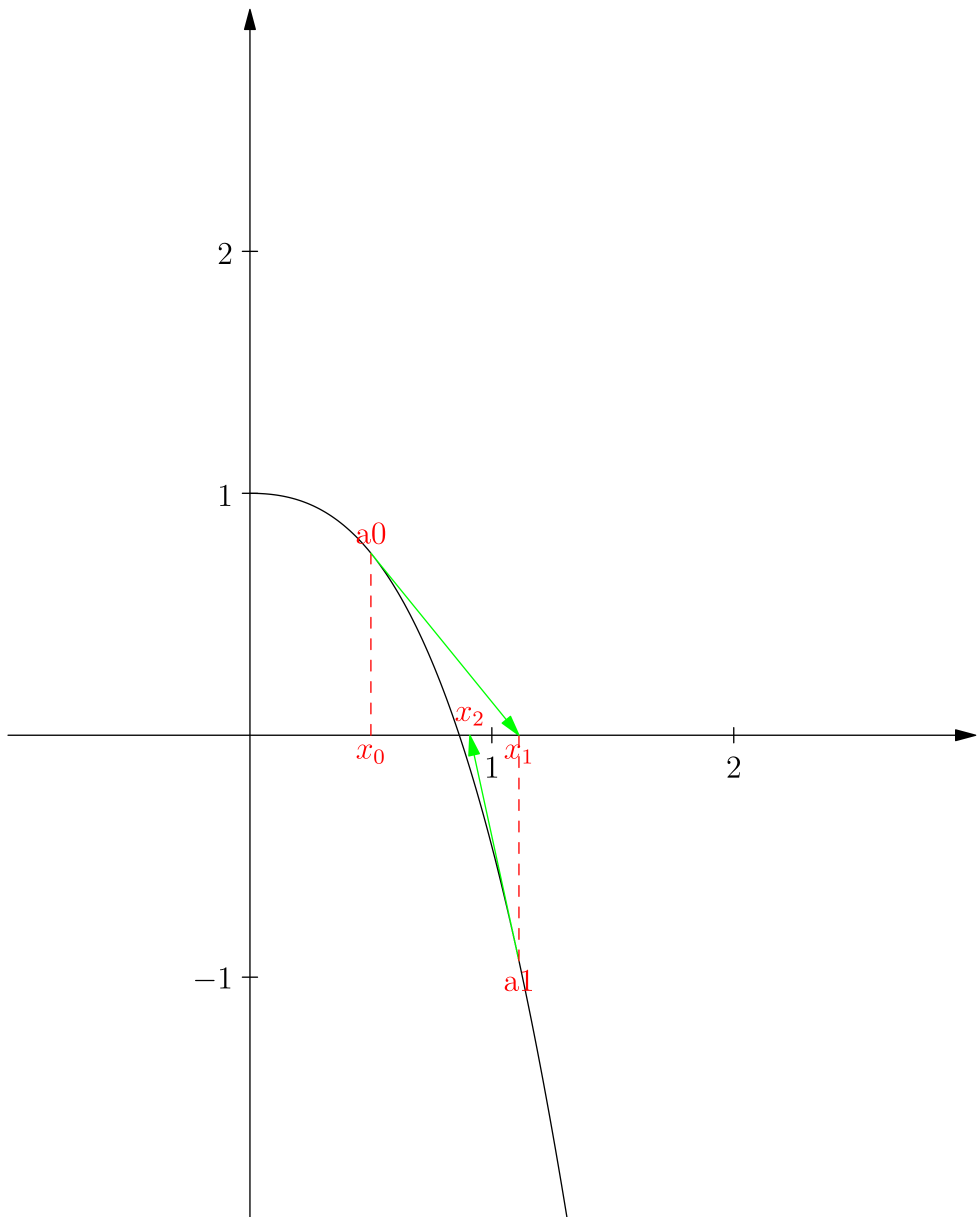
Une étude de cette suite montre sa convergence vers $\sqrt{2}$, qui est bien la valeur positive annulant f.

<u>Remarque</u>

Dans le cas général, en appelant a la solution de $f(a) = 0$ et en appelant $a_n = x_n - l$ on a

$\dfrac{a_{n+1}}{a_n} = \dfrac{f(x_n) - l}{x_n - l} \longrightarrow f'(a)$ en cas de convergence de (x_n) vers a. Ceci montre que la convergence sera d'autant plus rapide que $|f'(a)|$ est petit. Dans l'exemple précédent on peut vérifier que

$a_{n+1} = \dfrac{a_n^2}{2x_n} \leq \dfrac{a_n^2}{2}$ on a par exemple $n \geq 11 \Rightarrow a_n \leq \dfrac{1}{2^{1000}}$, ainsi après onze itérations on a 1000 bits significatifs ce qui montre une convergence très rapide.



$y = \cos(x) - x^3$, Methode de Newton

L'algorithme repose sur un contrôle de la valeur de $|x_{n+1} - x_n|$, en réalisant une boucle dont l'arrêt dépend de la précision souhaitée de la valeur précédente.

Données : f, g, u_0, ϵ (g représente f')
$u \leftarrow u_0$
$v \leftarrow u - f(u)/g(u)$
tant que $|v - u| > \epsilon$ faire
$u \leftarrow v$
$v \leftarrow v - f(v)/g(v)$
Résultat :v

Avant même de parler de complexité d'un algorithme, il est d'usage de démontrer sa terminaison, et sa validité, c'est à dire ici d'une part le fait qu'il n'y aura pas de division par zéro, et d'autre part que le résultat renvoyé r sera tel qu'il existe x_0 zéro de f tel que $|x_0 - r| \leq \epsilon$

Mais ici, on rencontre les problèmes suivants :
— On peut rencontrer des divisions par zéro.
— La terminaison n'est pas assurée.
— Même si un résultat est renvoyé, il peut être éloigné d'un zéro de f.

Cependant, pour une fonction raisonnable, la concavité/convexité locale est la règle : si f est de classe C^2 avec $f''(x_0) \neq 0$, alors f'' est de signe strict constant au voisinage de x_0. Ainsi, si on part de x_0 assez proche d'un zéro vérifiant cette condition, alors la méthode de Newton convergera vers ce zéro.

4.2.1 Méthode de la sécante

La méthode suppose connue la fonction dérivée, si ce n'est pas le cas la tangente sera remplacée par une sécante.

Dans cette méthode $f'(x_n)$ est remplacé par $\dfrac{f(x_n) - f(a)}{x_n - a}$, c'est le coefficient directeur de la sécante (AA_n), et x_{n+1} est l'abscisse de l'intersection de cette droite avec l'axe des abscisses. L'équation de la droite (AA_n) est :

$$y - f(a) = \frac{f(x_n) - f(a)}{x_n - a}(x - a) \text{ ce qui donne}$$

$$x_{n+1} = a - \frac{x_n - a}{f(x_n) - f(a)} f(a).$$

Le principe de l'algorithme est ensuite identique à celui de la méthode de Newton, à ceci prés qu'elle nécessite deux valeurs a et b pour initialiser le programme.

Programmes :

Méthode de Newton.

```
def newton(f,g,x0,e):
    """Resolution de f(x)=0. g est la fonction dérivée de f"""
    u=x0
    v=u-f(u)/g(u)
    while abs(v-u)>e:
        u=v
        v=v-f(v)/g(v)
    return v

def f(x):
return x**2-2
```

```
def g(x):
return 2*x

newton(f,g,2,10e-10)
1.4142135623730951
```

Méthode de la sécante.

```
def secante(f,a,b,e):
    """Resolution de f(x)=0. methode de la secante"""
    u=a
    v=a-((b-a)/(f(b)-f(a)))*f(a)
    while abs(v-u)>e:
        t=v
        u,v=v,a-((t-a)/(f(t)-f(a)))*f(a)
    return v

 newton(sin,cos,3,10e-5)
3.141592653589793

 secante(sin,3,4,10e-5)
3.1415928090391736

pi
3.141592653589793
```

L'exemple précédent montre que la méthode de Newton est sensiblement plus précise que la méthode de la sécante.

Choix d'une méthode

On ne peut pas écrire une méthode universelle de résolution numérique d'équation de la forme $f(x) = 0$. On peut toutefois chercher le meilleur compromis.

— Si on cherche la simplicité et la robustesse, la méthode dichotomique s'impose : la continuité de la fonction et un premier intervalle $[a, b]$ tel que $f(a)f(b) \leq 0$ sont suffisants. La « lenteur » de cette méthode est toute relative quand on veut seulement quelques décimales.
— Si on connaît une bonne approximation d'un zéro et qu'on cherche la rapidité, alors on peut appliquer une méthode de la famille de Newton : la méthode de Newton proprement dite si on connaît f' ; la méthode de la sécante si on ne peut ou ne veut pas évaluer f'.
— Dans des situations intermédiaires, on peut pratiquer des méthodes mixtes, avec une première phase de localisation de zéros par dichotomie, puis des itérations de Newton. Si ces itérations de Newton ne semblent pas converger, on peut reprendre une phase de dichotomie et ainsi de suite.

Exercice

Soit $f \in C^3([a,b])$ et $x_0 \in]a,b[,\ h>0$ tel que $x_0+h,\ x_0-h \in]a,b[$.
1) En utilisant la fonction $\phi : h \mapsto f(x_0+h)-f(x_0-h)$ montrer que

$$\left| \frac{f(x_0+h)-f(x_0-h)}{2h} - f'(x_0) \right| \leq A\,h^3,\ A \in \mathbb{R}_+^*$$

.
2) Ecire une fonction **def derive(f,x,h) :** calculant une valeur approchée de la dérivée à f en x, fonction utile si la dérivée n'est pas connue explicitement.
3) Ecrire une fonction **def newton2(f,h,x_0,e) :** calculant une valeur approchée de α tel que $f(\alpha)=0$ en utilisant la fonction précédente et la fonction newton du cours dans laquelle $g=f'$ sera remplacée par la fonction derive.

4.2.2 Utilisation de Numpy et ou de Scipy

Voir chapitre de compléments concernant ces deux modules.

La plupart du temps, il est déraisonnable de coder soi-même une méthode de résolution qui existe déjà, et a probablement été plus optimisée et testée qu'une fonction « maison ». Les bibliothèques numpy et scipy.optimize proposent les fonctions suivantes :

La fonction **roots** de numpy permet de déterminer les racines d'un polynôme donné par la liste de ses coefficients : $numpy.roots([1,2,-1,-2])$
$array([1.,-2.,-1.])$

Elle fournit même les racines complexes :

$roots([1,1,1])\ array([-0.5+0.8660254j,-0.5-0.8660254j])$.

La méthode dichotomique est implémentée dans scipy et a pour nom **optimize.bisect**
optimize.bisect(math.sin, 3, 4)
3.141592653589214

La méthode de Newton est programmée dans scipy et a pour nom **optimize.newton**. A noter que si on ne donne pas la dérivée, c'est en fait la méthode de la sécante qui est appliquée.

optimize.newton(math.sin, 3, math.cos)
3.141592653589793

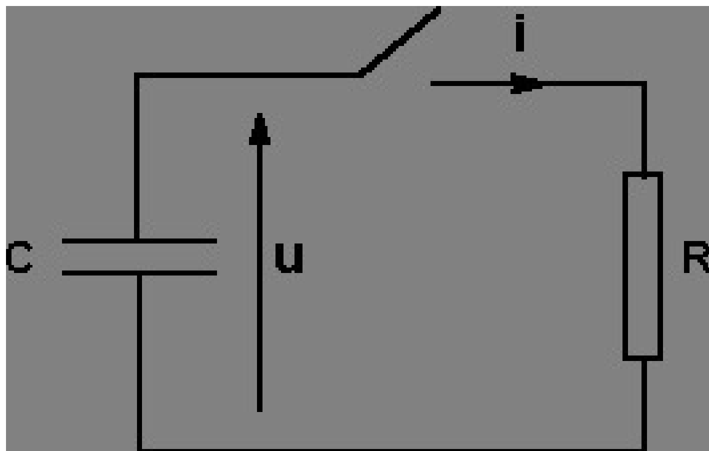
4.3 Résolution numérique d'équations différentielles

De nombreux problèmes conduisent à la résolution d'équations différentielles du premier ordre du type :

$$\frac{dy}{dt} = f(t,y).$$

Lorsqu'il n'est pas possible d'otenir une solution explicite sous forme de fonctions usuelles, on détermine une solution approchée. Il existe plusieurs méthodes permettant de réaliser ceci. L'une d'entre elles est la méthode d'Euler qui sera étudiée ici.

Exemple d'équation



On a $u(t) = Ri(t) = -R\frac{dq}{dt}(t) = -RC\frac{du}{dt}(t)$ d'où l'équation différentielle : $\frac{du}{dt}(t) = \frac{-1}{RC}u(t)$. Toutefois dans cet exemple on dispose d'une solution explicite :

$$u(t) = u_0 \exp\left(\frac{-1}{RC}t\right).$$

4.3.1 Méthode d'Euler

On veut déterminer une solution approchée de l'équation différentielle : $\frac{dy}{dt}(t) = f(t, y(t))$ où $t \mapsto f(t, y(t))$ est une fonction continue sur lintervalle $[a, b]$. avec une condition initiale : $(y(a) = y_0$.

On considère une subdivision régulière $\sigma = (t_0, t_1, \cdots, t_n)$ de $[a, b]$ de pas $h = \frac{b-a}{n}$, on a donc $t_i = t_0 + ih = a + ih$.

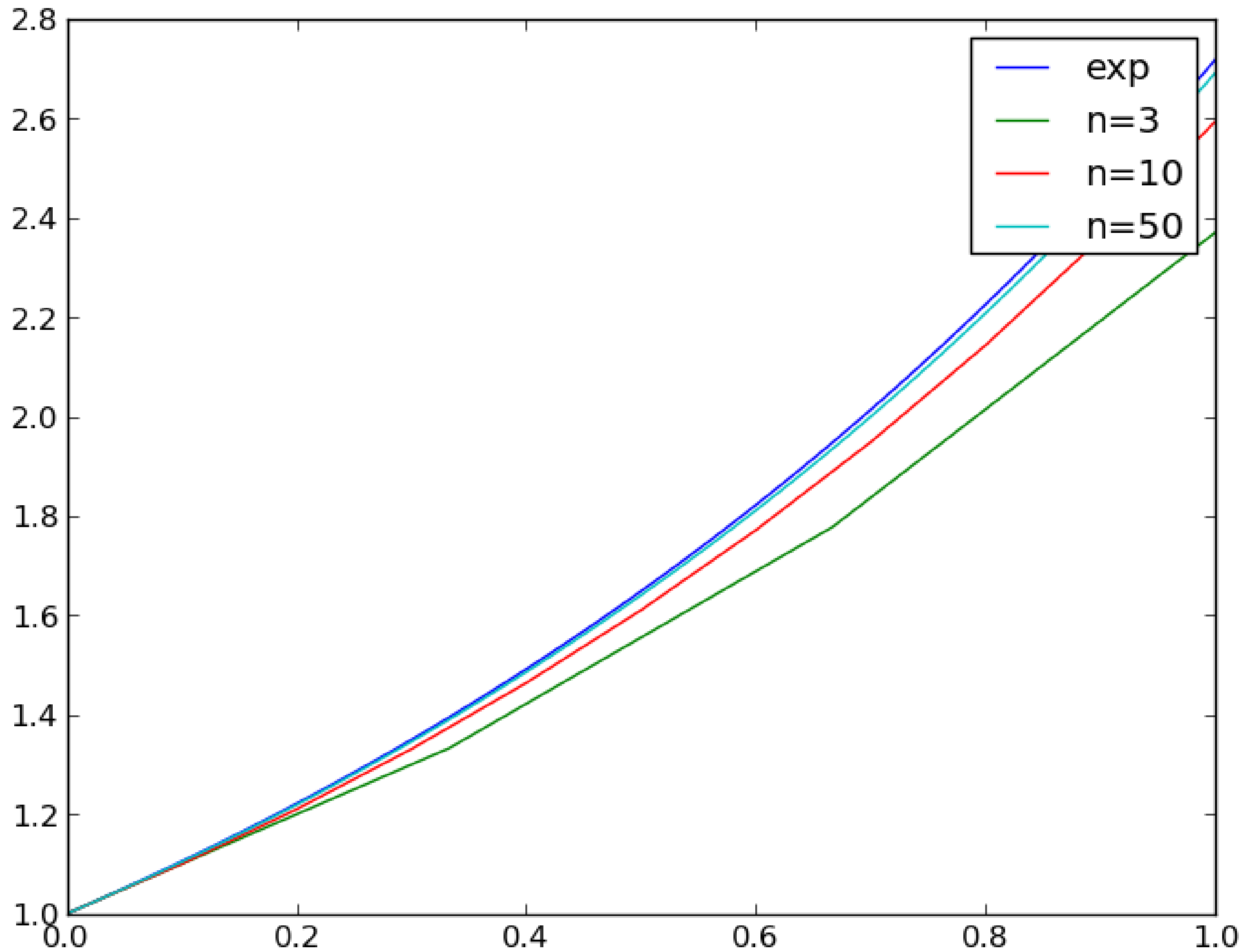
Si h est suffisamment petit on a $y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(t, y(t))dt \simeq hf(t_k, y(t_k))$.

Les approximations sont alors calculées de proche en proche par : $y_{k+1} = y_k + h\, f(t_k, y_k)$.

Graphiquement, cela revient à faire des approximations successives de courbes par des tangentes.

<u>Exemple</u>

Considérons l'équation différentielle $y' = y$ sur $[0, 1]$ dont la solution exacte est $y(t) = \exp(t)$ et représentons en bleu foncé cette solution exacte, en vert la solution correspondant à $n = 3$, en rouge la solution correspondant à $n = 10$ et en bleu clair celle correspondant à n=50.



Comme on pouvait s'y attendre, on constate que plus le pas est petit et plus l'erreur commise est faible. Le temps de calcul lui croit lorsque le pas diminue, il y a donc un compromis à trouver entre la précision souhaitée et le temps de calcul qui doit rester raisonnable.

La définition de l'erreur, appelée aussi erreur de consistance est le maximum des $e_n = |y(t_{n+1}) - y_{n+1}|$, lorsqu'on prend pour les valeurs précédentes des (y_k) les valeurs exactes $y(t_k)$.

Voici un tableau tiré d'un livre déja cité.

n	10^3	10^4	10^5	10^6	10^7
temps	$6,7.10^{-4}$	$6,9.10^{-3}$	$6,8.10^{-2}$	$6,7.10^{-1}$	6,7
erreur	$1,3.10^{-3}$	$1,3.10^{-4}$	$1,3.10^{-5}$	$1,3.10^{-6}$	$1,3.10^{-7}$

Il existe d'autres méthodes de résolution numérique des équations différentielles, par exemple les méthodes de Heun et de Runge Kutta. La première utilise une approximation de l'intégrale par la formule des trapèzes :

$y(t_{k+1}) - y(t_k) = \int_{t_k}^{t_{k+1}} f(t, y(t))dt \simeq \frac{h}{2}\left(f(t_k, y(t_k)) + f(t_{k+1}, y(t_{k+1})\right)$, la deuxième que je ne détaillerai pas, existe en plusieurs versions.

Dans le cas de la méthode de Heun on définit une suite par $y_{k+1} = y_k + \dfrac{h}{2}\left(f(t_k, y_k) + f(t_{k+1}, y_{k+1})\right)$,il s'agit d'une formule implicite qui ne permet pas un calcul direct de y_{k+1} qui intervient également dans $f(t_k, y_k) + f(t_{k+1}, y_{k+1})$. On utilise les étapes suivantes :

$u_{k+1} = y_k + h\, f(t_k, y_k)$, le calcul de cette valeur est basé sur la méthode d'Euler.
$y_{k+1} = y_k + \frac{h}{2}\left(f(t_k, y_k) + f(t_{k+1}, u_{k+1})\right)$.

La méthode de Heun est, en général, nettement plus précise que la méthode d'Euler pour un même pas. Dans l'exemple précédent, avec $n = 10^6$, on ontiendrait une erreur de $4,1.10^{-13}$ à comparer au $1,3.10^{-6}$ de la méthode d'Euler.

Programmes

Méthode d'Euler

Dans le programme suivant les valeurs de t sont regroupées dans le tableau lt et les valeurs de la suite (y_n) dans le tableau ly.

```
def euler(f,a,b,y0,n):
    t=a
    y=y0
    ly=[y0]
    lt=[a]
    h=(b-a)/n
    for i in range(n):
        y=y+h*f(t,y)
        t=t+h
        ly.append(y)
        lt.append(t)
    return lt, ly
```

Méthode de heun

```
def heun(f,a,b,y0,n):
    t=a
    y=y0
    ly=[y0]
    lt=[a]
    h=(b-a)/n
    for i in range(n-1):
        u=y+h*f(t,y)
        y=y+(h/2)*(f(t,y)+f(t+h,u))
        t=t+h
        ly.append(y)
        lt.append(t)
    return lt, ly
```

Le module **scipy** contint une fonction de résolution d'équations différentielles.

La bibliothèque scipy.integrate contient la fonction odeint, qui résout numériquement des équations différentielles. On commence donc par la charger via

from scipy.integrate import odeint

L'utilisation se fait sous la forme : **odeint**(f, y_0, T) où f est la fonction $(t, y) \mapsto f(y, t)$, y_0 est la valeur initiale $y(a)$, $t \in [a, b]$ et T est un tableau de valeurs de t. On remarque que la forme de f fait ici intervenir y avant t. La valeur de sortie est un tableau des valeurs approchées de la solution aux temps précisés dans T.

Exemple

$f(t, y) = y,\ a = 0,\ b = 1,\ T = [0, 0.25, 0.5, 0.75, 1]$, l'équation est donc $y'(t) = f(t, y(t))$.

```
 def f(y,t):
return y

odeint(f,1,[0,0.25,0.5,0.75,1.0])
array([[ 1.        ],
       [ 1.28402541],
       [ 1.64872127],
       [ 2.11700009],
       [ 2.7182819 ]])
```

Le tableau T peut être généré automatiquement de différentes manières :

1) Al'aide d'une boucle **for** :$T = [k/20\ for\ k\ in\ range(21)]$
$T = [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]$.

2) La méthode habituelle est d'utiliser la fonction **linspace** du module numpy qui construit un tableau dont les valeurs sont équidistantes.
T=linspace(a,b,n) construit un tableau de n valeurs équidistantes dont la première est a et la dernière b.

```
from numpy import linspace
 T=linspace(0,1,21)
 T
array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,
        0.45,  0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,
        0.9 ,  0.95,  1.  ])
```

3) Il existe également la fonction **arange** du module numpy qui s'utilise sous la forme :
arange(a,b,h)où h est le pas, b étant exclu comme dans le cas de range. Si on veut un tableau identique aux précédents on écrit :

```
from numpy import arange

 T=arange(0,1.05,0.05)
 T
array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ,  0.25,  0.3 ,  0.35,  0.4 ,
        0.45,  0.5 ,  0.55,  0.6 ,  0.65,  0.7 ,  0.75,  0.8 ,  0.85,
        0.9 ,  0.95,  1.  ])
```

Courbes

Il est possible de représenter graphiquement les solutions obtenues en utilisant le module **pylab** qui contient les modules numpy et math (au moins en partie). **pylab** peut être considéré comme une interface graphique du module **matplotlib**. Etudions quelques exemples :

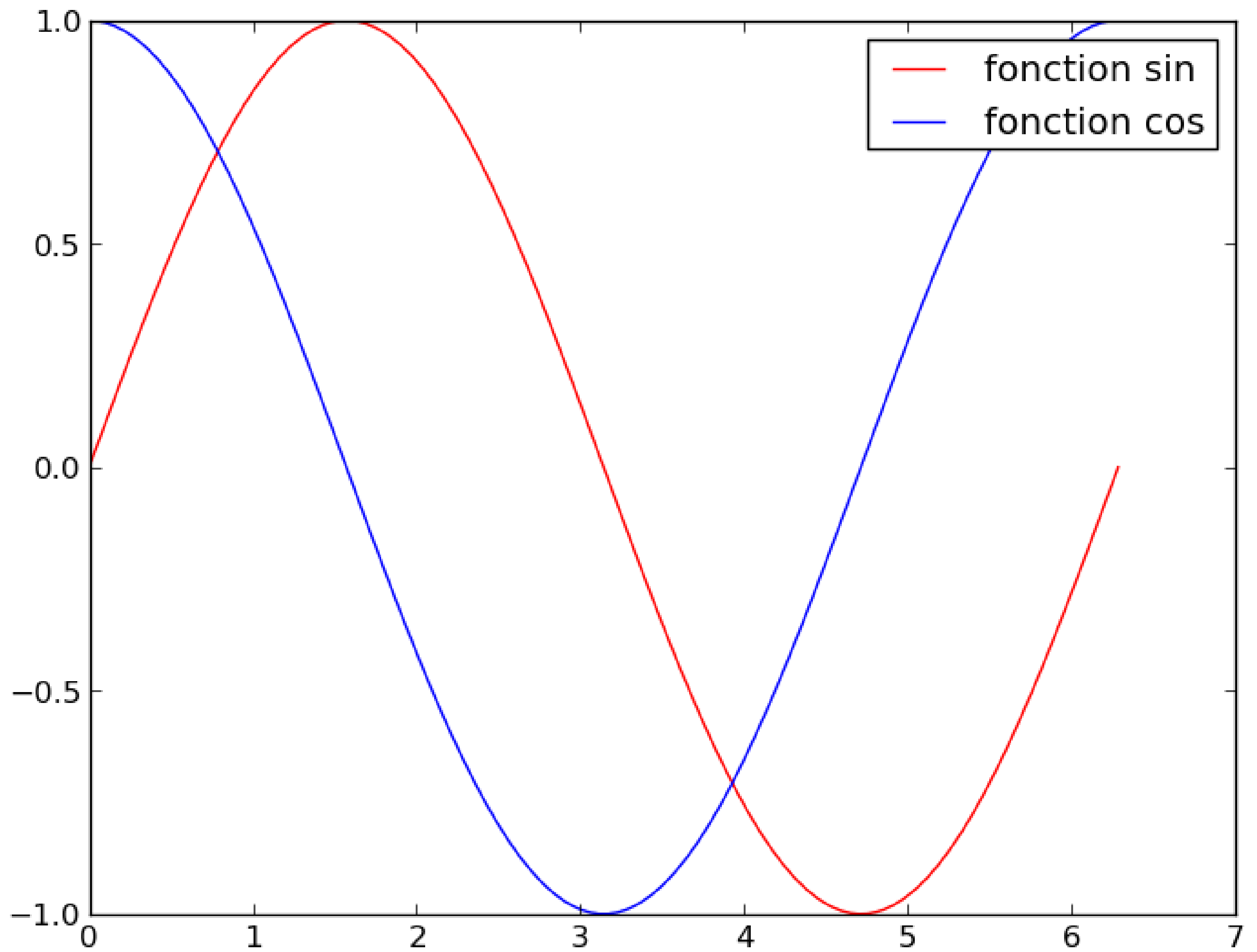
```
x=linspace(0,2*pi,120)

y=sin(x)
 z=cos(x)

plot(x,y,'r',label='fonction sin')
[<matplotlib.lines.Line2D object at 0x7ffdc576c1d0>]

 plot(x,z,'b',label='fonction cos')
[<matplotlib.lines.Line2D object at 0x7ffdc841e510>]
legend()
```

Pour afficher les courbes on termine par l'instruction (la méthode) **show()** et l'on obtient :



Il faut noter que si les fonctions sin et cos avaient été introduites par le module **math** il n'est pas possible de construire un tableau $y = \sin(x)$ à partir d'un tableau x. Dans ce cas x devrait être un réel. On peut alors utiliser la fonction **map** qui permet d'associer une fonction à un tableau (ou une liste) $y = map(\sin x)$ per met de construire un tableau y de même longueur que x dont chaque composante est $y_i = \sin(x_i)$.

Chapitre 5

Bases de données

5.1 Généralités

Qualités souhaitées d'une base de données ou BDD et du système de gestion de base se données ou SGBD, qui est le logiciel permettant l'exploitation aisée d'une base de données.

Il faut pouvoir accéder aux données sans savoir programmer ce qui signifie des langages "quasi naturels".

Efficacité des accès aux données
Ces langages doivent permettre d'obtenir des réponses aux interrogations en un temps "raisonnable". Ils doivent donc être optimisés et, entre autres, il faut un mécanisme permettant de minimiser le nombre d'accès disques. Tout ceci, bien sur, de façon complètement transparente pour l'utilisateur.

Administration centralisée des données
Des visions différentes des données (entre autres) se résolvent plus facilement si les données sont administrées de façon centralisée.

Non-redondance des données
Afin d'éviter les problèmes lors des mises à jour, chaque donnée ne doit être présente qu'une seule fois dans la base.

Cohérence des données
Les données sont soumises à un certain nombre de contraintes d'intégrité qui définissent un état cohérent de la base. Elles doivent pouvoir être exprimées simplement et vérifiées automatiquement à chaque insertion, modification ou suppression des données.

Partageabilité des données
Il s'agit de permettre à plusieurs utilisateurs d'accéder aux mêmes données au même moment. Si ce problème est simple à résoudre quand il s'agit uniquement d'interrogations et quand on est dans un contexte mono-utilisateur, cela n'est plus le cas quand il s'agit de modifications dans un contexte multi-utilisateurs. Il s'agit alors de pouvoir :

— permettre à deux (ou plus) utilisateurs de modifier la même donnée "en même temps"
— assurer un résultat d'interrogation cohérent pour un utilisateur consultant une table pendant qu'un autre la modifie.

Sécurité des données
Les données doivent pouvoir être protégées contre les accès non autorisés. Pour cela, il faut pouvoir associer à chaque utilisateur des droits d'accès aux données.

Résistance aux pannes
Que se passe-t-il si une panne survient au milieu d'une modification, si certains fichiers contenant les données deviennent illisibles ? Les pannes, bien qu'étant assez rares, se produisent quand même de temps en temps. Il faut pouvoir, lorsque l'une d'elles arrive, récupérer une base dans un état "sain". Ainsi, après une panne intervenant

au milieu d'une modification deux solutions sont possibles : soit récupérer les données dans l'état dans lequel elles étaient avant la modification, soit terminer l'opération interrompue.

Un peu d'histoire

1960
Uniquement des systèmes de gestion de fichiers plus ou moins sophistiqués.

1970
Début des systèmes de gestion de bases de données réseaux et hiérarchiques proches des systèmes de gestion de fichiers. Ces systèmes de gestion de bases de données avaient rempli certains des objectifs précédents mais on ne pouvait pas interroger une base sans savoir où était l'information recherchée (on "naviguait") et sans écrire de programmes. Sortie du papier de CODD sur la théorie des relations, fondement de la théorie des bases de données relationnelles.

1980
Les systèmes de gestion de bases de données relationnels apparaissent sur le marché.

1990
Les systèmes de gestion de bases de données relationnels dominent le marché.

Les SGBD permettant d'exploiter une base de données relationnelle sont nombreux, nous étudierons SQL et plus précisèment une variante appelée SQLite, qui sont des SGBD non commerciaux, qui peuvent être téléchargés librement.

Il est possible de résumer les qualités réclamées pour une base de onnées par : Une base de données est un ensemble structiré de données informatiques (chaines de caractères, valeurs numériques, dates...) dans lequel
— Les données sont enregistrées sur un support permanent.
— Les données ne figurent qu'une fois.
— Chaque objet posséde un identifiant unique.
Une base de données relationnelle est formée d'un ensemble de tables, une table étant un ensemble de p-uplets (on dit de tuples) représentée mathématiquement par une relation, c'est à dire un sous ensemble d'un produit cartésien. L'ordre des p-uplets est indifférent et il n'y a aps de répétition.

Introduisons les différentes notions liées aux bases de données relationnelles sur un exemple.

Villes				
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
1	Bordeaux	Aquitaine	239157	33000
2	Mulhouse	Alsace	109588	68100
3	Rennes	Bretagne	207178	35000
4	Sainte-Marie	Franche-Comte	731	25113
5	Sainte-Marie	Bretagne	2100	35600
6	Besancon	Franche-Comte	116914	25000

Regions		
Nom	**Capitale**	**Habitants**
Aquitaine	1	3232352
Bretagne	3	3199066
Alsace	2	1845687
Franche-Comte	4	1171763

Ces deux exemples font apparaitre deux tables ou relations, appelées Villes et Régions. Les colonnes ont un nom et les valeurs d'une même colonne ont toutes un même type. Ce n'est pas le cas pour les lignes. Les colonnes sont appelées **attributs** de la relation, les lignes sont appelées **tuples** ou **entrées** ou **enregistrements** de la relation.
Les valeurs d'un même attribut appartiennent à un même ensemble appelé domaine de l'attribut. Dans la relation **Régions** le domaine de l'attribut **Nom** est l'ensemble des chaines de caractères.
Le domaine de l'attribut **Habitants** de la relation **Villes** est $\mathbb{N}$. Un domaine est décrit par un type, le type entier par exemple est **INT**.

Types usuels en langage SQL :

- **VARCHAR** Permet de coder des chaines de caractères, de longuer variables, mais dont la longueur maximale est fixée. Par exemple **VARCHAR(20)** pemet d'utiliser des chaines de caractère d'au plus vingt caractères.
- **CHAR** désigne les chaines de caractères de longueur fixes. **CHAR(20)** indique que les chaines de caractères utilisées ont exactement vingt caractères.
- **NUMERIC** ou **DEC** permet de coder des nombres décimaux de façon exacte.
- **FLOAT** permet de coder des décimaux en écriture approchée en base deux.
- **INT** permet de coder des entiers relatifs.
- **TEXT** permet de coder des textes de longueur indéterminée.
- **DATE** et **TIME** représentent la date et l'heure.

Définitions

Notation
L'attribut A_k de la relation R est noté $R.A_k$. Ainsi **Villes.Nom** représente l'attribut **Nom** de la relation **Villes**.

Une relation R (représentée par une table) ayant les attributs $A_1,\ A_2,\ \cdots,A_p$ sera représentée symboliquement par le schéma R$(A_1, A_2, \cdots, A_p)$. On notera DOM(A_i) le domaine de l'attribut A_i, c'est à dire l'ensemble auquel appartient cet attribut. Une table vérifiant la relation R est un sous ensemble fini de l'ensemble produit :
DOM$(A_1)\times$DOM$(A_2)\times\cdots\times$ DOM(A_p).

Chaque tuple de la table est élément de cet ensemble.
Si t est un enregistrement de la relation R on note t$([A_k])$ la valeur de l'attribut A_k de cet enregistrement.
On dit que deux relations ont même schéma si elles ont le même nombre d'attributs et si leurs attributs ont le même nom et sont du même type.

Il est à noter que ni les tuples, ni les attributs ne sont ordonnés.

En appelant V la relations "Villes", nous avons cinq attributs et six enregistrements notés $t_1, t_2, \cdots, t_6$.

DOM(Habitants)=INT, DOM(Nom)=VARCHAR(20)

$t_4[1] = 4,\ t_4[2]$ =Sainte-Marie $t_4[3]$ =Franche-Comte, $t_4[4] = 731,\ t_4[5] = 25113$.

Deux enregistrements distincts doivent vérifier $t \neq t' \Rightarrow \exists k,\ t[A_k] \neq t'[A_k]$.

Clé : choix d'un ou plusieurs attribut(s) en relation univoque (injection) avec chaque enregistrement de la relation.

Clé primaire : Choix d'une clé privilégiée parmi toutes les clés possibles. C'est par exemple l'attribut **cle** de la relation **Ville** ou l'attribut **Nom** de la la relation **Regions**.

Ce sera souvent cet attribut noté **cle**, qui est indenté automatiquement comme on le verra plus tard, qui joue le rôle de clé primaire, mais pas toujours. Dans la relation **Villes** l'attribut **Code postal** pourrait être choisi comme clé primaire.

Toute base de données bien conçue évite les redondances en créant s'il le faut plusieurs relations au lieu d'une seule. Ces relations peuvent être reliées entre elles par une **clé étrangère**.

clé étrangère : Fait référence à la clé primaire d'une autre table. Par exemple l'attribut **Capitale** de la relation **Region** est une clé étrangère, qui fait référence à la clé primaire de la relation **Villes**

On peut représenter un lien entre ces deux relations en le représentant par une flèche depuis l'attribut vers la clé primaire, dans l'exemple précédent :
Regions⟶Cle

Schéma d'une relation : C'est l'expression $R(A_1 : Dom(A_1), A_2 : Dom(A_2), \cdots, A_p : Dom(A_p))$. Le schéma d'une relation peut également ne faire intervenir que les attributs, sans préciser leurs domaines respectifs.

Un schéma relationnel décrit les différentes relations de la base, chacune avec ses attributs, et décrit les références entre clés étrangéres et clés primaires des différentes relations d'une base de donnée.

Définition
Un base de données est constituée d'un ensemble de relations ainsi que de leurs schémas relationnels.

5.2 Algèbre relationnelle

5.2.1 Opérateurs ensemblistes

Soit $S = (A_1, A_2, \cdots, A_p)$ p attributs et $R_1(S)$ et $R_2(S)$ deux relations ayant le même schéma.

Union

La réunion $R_1 \cup R_2$ est l'ensemble des enregistrements appartenant à R_1 ou à R_2 auquel on retire les doublons éventuels.

$R_1 \cup R_2 = \{t | t \in R_1 \text{ ou } t \in R_2\}$.

Exemple

Villesbis				
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
1	Bordeaux	Aquitaine	239157	33000
2	Strasbourg	Alsace	271782	67000
3	Rennes	Bretagne	207178	35000
4	Concarneau	Finistere	19048	29900

En notant V_1 respectivement V_2 la relation Villes respectivement Villesbis la relation $V_1 \cup V_2$ est :

Villes union Villesbis				
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
1	Bordeaux	Aquitaine	239157	33000
2	Mulhouse	Alsace	109588	68100
3	Rennes	Bretagne	207178	35000
4	Sainte-Marie	Franche-Comte	731	25113
5	Sainte-Marie	Bretagne	2100	35600
6	Besancon	Franche-Comte	116914	25000
2	Strasbourg	Alsace	271782	67000
4	Concarneau	Finistere	19048	29900

Il faut noter que cle n'est plus une cle primaire, il est possible de la remplacer par Code Postal.

Intersection

Sous les hypothèses précédentes, l'intersection $R_1 \cap R_2$ est l'ensemble des enregistrements qui appartiennent à R_1 et à R_2 simultanément.

$R_1 \cap R_2 = \{t | t \in R_1 \text{ et } t \in R_2\}$.

En reprenant l'exemple précédent la relation $V_1 \cap V_2$ est :

Villes inter Villesbis				
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
1	Bordeaux	Aquitaine	239157	33000
3	Rennes	Bretagne	207178	35000

Différence

Sous ces mêmes hypothèses la différence $R_1 - R_2$ est l'ensemble des enregistrements appartenant à R_1 mais n'apprtenant pas à R_2 .

$R_1 - R_2 = \{t | t \in R_1 \text{ et } t \notin R_2\}$.

La relation $V_1 - V_2$ est donnée par :

Villes - Villesbis				
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
2	Mulhouse	Alsace	109588	68100
4	Sainte-Marie	Franche-Comte	731	25113
5	Sainte-Marie	Bretagne	2100	35600
6	Besancon	Franche-Comte	116914	25000

Produit cartésien

Soit R_1 et R_2 deux relations d'ensembles d'attributs respectifs S_1 et S_2. Le prduit cartésien de R_1 par R_2 est la relation définie par :

$$R_1 \times R_2 = \{t | t[S_1] \in R_1 \text{ et } t[S_2] \in R_2\}$$

Exemple

On considère les deux relations etudiant et enseignant ayant chacune un attribut :

Etudiant
Dupond
Martin
Mallot

Enseignant
Durand
Frison

La relation produit est alors :

Etudiant x Enseignant	
Dupond	Durand
Martin	Durand
Mallot	Durand
Dupond	Frison
Martin	Frison
Mallot	Frison

5.2.2 Projection

La projection permet de récupérer les composantes correspondant à un sous ensemble des attributs d'une relation. Si $S = (A_1, A_2, \cdots, A_p)$ et $X = (A_1, A_2, \cdots, A_m)$ est un sous ensemble de S et $R(S)$ le schéma de la relation R, la projection de R sur X est définie par :

$$\pi_X(R) = \{t[X] | t \in R\}$$

Exemple
Dans la relation Villes, la projection sur $X = \{$ Nom, Regions, Habitants $\}$ donne la relation :

Projection sur X		
Nom	**Region**	**Habitants**
Bordeaux	Aquitaine	239157
Mulhouse	Alsace	109588
Rennes	Bretagne	207178
Sainte-Marie	Franche-Comte	731
Sainte-Marie	Bretagne	2100
Besancon	Franche-Comte	116914

5.2.3 Selection ou restriction

La sélection permet de choisirs certaines entrées dans une table, en imposant des conditions sur les attributs. Les conditions peuvent être réalisées à partir des opérateurs de comparaisons usuels. Si on se donne une condition booléenne F, la sélection de la relation R par la condition F est définie par :

$$\sigma_F(R) = \{t \in R | t \text{ satisfait la condition F } \}$$

Prenons dans la relation Villes la condition F="La ville contient plus de 200 000 habitants" la relation

σ_F(Villes)				
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
1	Bordeaux	Aquitaine	239157	33000
3	Rennes	Bretagne	207178	35000

5.2.4 Jointure

La jointure permet de travailler sur plusieurs tables ayant des relations entre elles.
Etant donné deux relations R_1 et R_2 et une formule de sélection F, la jointure consiste à créer la table sous ensemble de $R_1 \times R_2$ des tuples qui verifient la relation F.

$R_1 \bowtie_F R_2 = \{t \in R_1 \times R_2 | t \text{ vérifie la relation F } \}$.

Exemple

Afficher les villes avec les informations disponibles sur la région à laquelle elles appartiennent.

$F =$ "Region de Villes=Nom de Regions"
La relation obtenue est :

Villes et Regions							
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**	**Nom**	**Capitale**	**Habitants**
1	Bordeaux	Aquitaine	239157	33000	Aquitaine	1	3232352
2	Mulhouse	Alsace	109588	68100	Alsace	2	1845687
3	Rennes	Bretagne	207178	35000	Bretagne	3	3199066
4	Sainte-Marie	Franche-Comte	731	25113	Franche-Comte	4	1171763
5	Sainte-Marie	Bretagne	2100	35600	Bretagne	3	3199066
6	Besancon	Franche-Comte	116914	25000	Franche-Comte	4	1171763

Fonctions d'agrégation

Les fonctions d'agrégation calculent un résultat sur un groupe d'entrées.

Les plus courantes sont :

- Calcul de la somme.
- Calcul de la moyenne.
- Comptage du nombre de tuples.
- Maximum ouminimum.

5.2.5 Division

etant donné deux relations, R_1, d'attributs $A = \{A_1, A_2, \cdots, A_n, B_1, B_2, \cdots, , B_p\}$ et R_2 d'attributs $B = \{B_1, \cdots, B_p\} \subset A$ la relation R notée $R_1 \div R_2$ et appelée division de R_1 par R_2, contient tous les tuples $t \in A_1 \times \cdots \times A_n$ tels que pour tout $s \in B$ $(t, s) \in R_1$
Exemple

On construit la table de tous les enseignants de la relation R_1=**Enseignement** qui enseignent à tous les étudiants de la relation R_2=**etudiants**

Enseignements	
Enseignant	**Etudiant**
Germain	Dubois
Fidus	Pascal
Robert	Dubois
Germain	Pascal
Fidus	Dubois
Germain	Durand
Robert	Durand

Etudiant
Nom
Dubois
Pascal

La relation $R_1 \div R_2$ est la suivante :

$R_1 \div R_2$
Enseignant
Germain
Fidus

5.2.6 Renommage

Il est possible, souvent pour des raisons pratiques afin de lever une ambiguïté, de renommer un attribut d'une relation à l'aide d'un opérateur, dit de renommage.

La relation obtenue après avoir effectué cette opération est alors identique à la relation de départ, mis à part le schéma qui a été changé pour présenter le nouveau nom.

Définition. Soit R = R$(A_1, \cdots, A_n)$ un schéma, $i \in [\![1, n]\!]$ et B un attribut tel que dom(B) = dom(A_i). On note

$$\rho_{A_i \leftarrow B}(R) = R(A_1, \cdots, A_{i-1}, B, A_{i+1}, \cdots, A_n)$$

schéma dans lequel l'attribut A_i est renommé B. Ce renomage peut porter sur un seul ou plusieurs attributs.

5.3 Langage de requête SQL

Les opérations introduites ci-dessus font partie de l'algèbre relationnelle. Elles sont traduites par un langage propre aux bases de données, il en existe plusieurs. Nous allons étudier SQL que l'on peut obtenir librement. L'utilisation de ce langage peut se faire en ligne de commande, sur un terminal ou à partir d'une interface graphique. L'utilisation des bases de données nécessite habituellement un serveur sur lequel se trouve le logiciel de gestion de bases de données et des clients d'où partent les demandes, on dit les requêtes, destinées à la base de donnée. Nous allons utiliser une version de SQL, sqlite, permettant de construire et d'utiliser une base de donnée sans nécessiter l'utilisation d'un serveur. Nous utiliserons une interface graphique, sqliteman, permettant de simplifier les opérations.
Il existe d'autres interfaces graphiques pour sqlite, sqlite data browser pour Linux ou Mac ou Windows et sqlite manager qui est une interface pour firefox.
Si on veut utiliser l'architecture client serveur il est possible d'utiliser phpMyAdmin en parallèle avec le serveur web Apache.

5.3.1 Création d'une table

Création de la table **villes** :

```
CREATE TABLE "Villes" (
    "Cle" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    "Nom" TEXT,
    "Region" TEXT,
    "Habitants" REAL,
    "Code Postal" INTEGER
)
```

Création de la table **regions** :

```
CREATE TABLE regions (
    "Nom" TEXT,
    "Capitale" INTEGER,
    "Habitants" INTEGER,
    "Densite" INTEGER
)
```

Cette opération peut être réalisé par l'utilitaire sqliteman. Une fois que la table est créée les champs doivent être complétés.

5.3.2 Requêtes

Les clauses qui permettent de structurer les interrogations (les requêtes) dans une base de données sont :

SELECT ... FROM ... WHERE

SELECT : Cette clause correspond à l'opérateur de projection ou de sélection.

FROM : Cette clause permet la déclaration de la liste des tables nécessaires à la requête.

WHERE : Cette clause permet de définir les prédicats (conditions) de sélection, formulés selon la syntaxe de la logique propositionnelle.

JOIN ... ON : Assure la jointure de deux tables sur (ON) un critère de jointure.

NOT IN : Permet d'exprimer la négation, de réaliser une différence.

ORDER BY : Permet de trier sur la valeurs des attributs passés en arguments.

UNION : Réunion de deux tables.

RENAME TABLE ... TO : Renommage d'une table.
Il est aussi possible d'utiliser **AS** pour renommer une table ou une colonne.

Il est à noter que la coutume est de noter ces clauses en majuscules mais que ce n'est pas une obligation.

5.3.3 Exemples

5.3.3.1 Sélection

Le symbole * représente tous les enregistrements qui ...
Régions de la table du même nom triées par ordre alphabétique :

SELECT * FROM Regions ORDER BY Nom

Regions		
Nom	**Capitale**	**Habitants**
Aquitaine	1	3232352
Bretagne	3	3199066
Alsace	2	1845687
Franche-Comte	4	1171763

Pour obtenir les entrées triées par ordre décroissant de nombre d'habitants puis, en cas d'égalité, par ordre alphabétique croissant de capitale on utilise :

SELECT * FROM Regions ORDER BY Habitants DESC, Capitale

Nom	**Capitale**	**Habitants**
Aquitaine	1	3232352
Bretagne	3	3199066
Alsace	2	1845687
Franche-Comte	4	1171763

Remarque.
Remarque : contrairement à Python, si une requête est écrite sur plusieurs lignes, l'indentation n'a aucune importance.

Villes dont le nombre d'habitants dépasse les 200 000.

SELECT * FROM Villes WHERE Habitants > 200000

Cle	**Nom**	**Region**	**Habitants**	**Code Postal**
1	Bordeaux	Aquitaine	239157	33000
3	Rennes	Bretagne	207178	35000

5.3.3.2 Projection

Nom et Habitants de la relation Regions :

SELECT Nom, Habitants FROM Regions

Nom	**Habitants**
Aquitaine	3232352
Bretagne	3199066
Alsace	1845687
Franche-Comte	1171763

Afficher les noms des villes de Bretagne.

SELECT DISTINCT Nom FROM Villes WHERE Region = 'Bretagne'

Nom
Rennes
Sainte-Marie

5.3.3.3 Union, intersection, différence

Exemple d'union ensembliste : affichage des villes qui se trouvent en Bretagne ou qui ont plus de 200 000 habitants.

SELECT Nom,Region,Habitants FROM Villes WHERE Region = 'Bretagne' UNION SELECT Nom,Region,Habitants FROM Villes WHERE Habitants > 200000

Nom	**Region**	**Habitants**
Rennes	Bretagne	207178
Sainte-Marie	Bretagne	2100
Bordeaux	Aquitaine	239157

Autre possibilité pour réaliser cette union : utiliser un OR logique.

SELECT Nom,Region,Habitants FROM Villes WHERE Region = 'Bretagne' OR Habitants > 200000

Exemple d'intersection ensembliste : affichage des villes qui se trouvent en Bretagne et qui ont plus de 200 000 habitants.

SELECT * FROM Villes WHERE Region = 'Bretagne' AND Habitants > 200000

Nom	**Region**	**Habitants**
Rennes	Bretagne	207178

Exemple de différence ensembliste : les villes de Bretagne, sauf celles qui ont plus de 200 000 habitants.

SELECT * FROM Villes WHERE Region = 'Bretagne' AND NOT (Habitants > 200000)

Nom	Region	Habitants
Sainte-Marie	Bretagne	2100

5.3.3.4 Fonctions d'agrégation

Les fonctions d'agrégation calculent un résultat sur un groupe d'entrées. Exemple : le nombre moyen d'habitants par région.

SELECT AVG(Habitants) FROM Regions 2311324.0000

Parmi les fonctions d'agrgation les plus usuelles, citons :

- AVG pour le calcul de la somme.
- COUNT pour compter le nombre de tuples.
- MAX et MIN pour une valeur maximum ou minimum
- SUM pour le calcul de la somme.
- certaines fonctions dont la valeur de retour est booléenne et qui servent à des critères de sélections d'autres requêtes : ALL, ANY, EXISTS. Elles apparaissent dans des sous–requêtes.

Exemples
Afficher la ville d'Aquitaine qui a le plus grand nombre d'habitants.

SELECT MAX(Habitants),Nom FROM Villes WHERE Region = 'Aquitaine'

MAX(Habitants)	Nom
239157	Bordeaux

Si on veut afficher uniquement le nom, il est possible d'utiliser une sous requête :

SELECT MAX(Habitants) FROM Villes WHERE Region = 'Aquitaine') de la manière suivante :

SELECT Nom FROM Villes WHERE Habitants = (SELECT MAX(Habitants) FROM Villes WHERE Region = 'Aquitaine')

Nom
Bordeaux

Voici un deuxième exemple d'utilisation de sous requête :
On souhaite connaître toutes les villes qui se trouvent dans la même région qu'une commune nommée Sainte–Marie. Il y a donc deux requêtes portant sur la relation Villes. Pour distinguer les attributs de ces deux requêtes on utilise l'opérateur AS qui permet de renommer les relations de chaque requête :

SELECT Nom, Region FROM Villes AS V1 WHERE EXISTS (SELECT * FROM Villes AS V2 WHERE V2.Region = V1.Region AND V2.Nom = 'Sainte-Marie')

On obtient :

Nom	Region
Rennes	Bretagne
Sainte-Marie	Bretagne
Sainte-Marie	Franche-Comte
Besancon	Franche-Comte

Pour pouvoir différentier les tuples de la requête et de la sous–requête, on appelle V1 la relation Villes dans la requête et V2 dans la sous–requête. Ainsi la clause V2.Region = V1.Region signifie que l'attribut Region doit être le même dans la requête et dans la sous–requête. La fonction d'agrégation EXISTS appliquée 'a la sous–requête est évaluée à Vrai si le résultat de la sous–requuête est non vide, et Faux si le résultat de la sous–requête est vide. Il existe aussi NOT EXISTS.
Dans le langage SQL, la commande EXISTS s'utilise dans une clause conditionnelle pour savoir s'il y a une présence ou non de lignes lors de l'utilisation d'une sous-requête.

Il ne faut pas confondre cette fonction et **IN**. L'opérateur logique IN dans SQL s'utilise avec la commande WHERE pour vérifier si une colonne est égale à une des valeurs comprise dans un ensemble de valeurs déterminés. C'est une méthode simple pour vérifier si une colonne est égale à une valeur OU une autre valeur OU une autre valeur et ainsi de suite, sans avoir à utiliser de multiple fois l'opérateur OR.

5.3.3.5 Jointure

afficher toutes les villes avec les informations disponibles sur la région à laquelle elles appartiennent :
SELECT * FROM Villes JOIN Regions ON Villes.Region = Region.Nom

Villes et Regions							
Cle	**Nom**	**Region**	**Habitants**	**Code Postal**	**Nom**	**Capitale**	**Habitants**
1	Bordeaux	Aquitaine	239157	33000	Aquitaine	1	3232352
2	Mulhouse	Alsace	109588	68100	Alsace	2	1845687
3	Rennes	Bretagne	207178	35000	Bretagne	3	3199066
4	Sainte-Marie	Franche-Comte	731	25113	Franche-Comte	4	1171763
5	Sainte-Marie	Bretagne	2100	35600	Bretagne	3	3199066
6	Besancon	Franche-Comte	116914	25000	Franche-Comte	4	1171763

5.3.3.6 Clauses GROUP BY ET HAVING

GROUP BY Cette clause permet de partitionner le résultat d'une requête et éventuellement d'appliquer ensuite une fonctions d'agrégation sur les parties obtenues.

HAVING La condition **HAVING** en SQL est presque similaire à **WHERE** à la seule différence que **HAVING** permet de filtrer en utilisant des fonctions telles que **SUM(), COUNT(), AVG(), MIN()** ou **MAX()**. **HAVING** est très souvent utilisé en même temps que **GROUP BY** bien que ce ne soit pas obligatoire.

Exemple
SELECT region, AVG(Habitants) FROM villes GROUP BY Region ;

Region	AVG(Habitants)
Alsace	132081.6
Aquitaine	159477.5
Bourgogne	151212
Bretagne	104958.5
Franche-Comte	58826

SELECT Region, AVG(Habitants) FROM villes GROUP BY Region HAVING(AVG(Habitants)>150000) ;

Region	AVG(Habitants)
Aquitaine	159477.5
Bourgogne	151212

Chapitre 6

Numpy : Vecteurs, matrices et tableaux

Il est d'abord nécessaire d'importer la bibliothèque **numpy** par **import numpy** ou **import numpy as np** ou **from numpy import ***. C'est en général la deuxième méthode qui est utilisée. Les fonctions de numpy seront alors accessibles par leur nom qualifié **np.nomfonction**.

6.1 Notions de base.

La bibliothèque **numpy** introduit le type **array** qui est proche d'une liste avec la différence que tous les éléments doivent être de même type.
Les éléments du type **array** sont appelés tableaux, il peuvent être à une ou plusieurs dimension, en pratique 1 ou 2 qui dans ce cas sont des matrices. Vecteurs, matrices lignes, matrice colonne :

Un vecteur est un tableau à une dimension composé d'entiers ou de flottants.

```
import numpy as np # charge le numpy et le renomme np

 >>> # un vecteur
>>> np.array([1,3,5])
array([1, 3, 5])
>>> # une matrice-ligne
>>> np.array([[1,3,5]])
array([[1, 3, 5]])
>>> # une matrice-colonne
>>> np.array([[1],[3],[5]])
array([[1],
[3],
[5]])
```

Matrices :

```
 >>> A = np.array([[5,3,2],[7,4,1]]); A # forme et affiche une matrice d'entiers
array([[5, 3, 2],
       [7, 4, 1]])
```

Les coefficients entiers de A peuvent être convertis au format 'float' ou 'complex' :

```
>>> A.astype(float)
array([[ 5., 3., 2.],
       [ 7., 4., 1.]])
>>> A.astype(complex)
array([[ 5.+0.j, 3.+0.j, 2.+0.j],
       [ 7.+0.j, 4.+0.j, 1.+0.j]])
```

Attention : les indices de lignes et colonnes commmencent à 0. ainsi $A[0,0]$ repésente A_{11}, $A[1,2]$ représente $A_{2,3}$.

Comme pour les listes il est possible de définir des matrices en compréhension ,c'est à dire à partir de formules, sans citer tous les éléments de la matrices :

```
>>> A = np.array([[10*i+j for j in range(10)] for i in range(5)])
>>> print(A) # la matrice de terme général 10i+j, avec 0<=i<=4, et 0<=j<=9
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]]
```

Il est possible d'appliquer une fonction à un tableau ou à une matrice. Il est à noter que **numpy** contient pratiquement tout le module **math** qu'il est donc inutile d'appeler ici pour utiliser la fonction **sqrt**.

```
>>> B=np.sqrt(A)
array([[ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
         2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ],
       [ 3.16227766,  3.31662479,  3.46410162,  3.60555128,  3.74165739,
         3.87298335,  4.        ,  4.12310563,  4.24264069,  4.35889894],
       [ 4.47213595,  4.58257569,  4.69041576,  4.79583152,  4.89897949,
         5.        ,  5.09901951,  5.19615242,  5.29150262,  5.38516481],
       [ 5.47722558,  5.56776436,  5.65685425,  5.74456265,  5.83095189,
         5.91607978,  6.        ,  6.08276253,  6.164414  ,  6.244998  ],
       [ 6.32455532,  6.40312424,  6.4807407 ,  6.55743852,  6.63324958,
         6.70820393,  6.78232998,  6.8556546 ,  6.92820323,  7.        ]])
```

La matrice A a été transformée, chaque terme de la matrice est remplacé par sa racine carrée.

```
B**2
array([[  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.],
       [ 10.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.],
       [ 20.,  21.,  22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.],
       [ 30.,  31.,  32.,  33.,  34.,  35.,  36.,  37.,  38.,  39.],
       [ 40.,  41.,  42.,  43.,  44.,  45.,  46.,  47.,  48.,  49.]])
```

Rétablit la matrice d'origine, à ceci prés que les nombres sont des flottants et non des entiers comme dans A, flottants qui sont apparus dans l'utilisation de la racine carrée.
Remarquer que $E**2$ ne réalise pas un calcul matriciel.

Ici les coefficients sont des flottants : si veut des entiers, il suffit de rajouter l'option **dtype='int'**.
Voici une autre méthode pour calculer la matrice A.

```
>>> B = np.fromfunction(lambda x,y : 10*x+y,(5,10))
>>> print(B)
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.   9.]
 [ 10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [ 20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [ 30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [ 40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]]
```

Il est possible de créer des matrices aléatoires :

Matrice (3,5) d'entiers compris entre 0 et 9.

```
C=np.random.randint(10,size=(3,5))
>>> C
array([[9, 3, 1, 0, 1],
       [4, 7, 5, 1, 9],
       [7, 9, 9, 1, 4]])
```

quatre lignes, cinq colonnes, coefficients dans $[0, 1[$

```
>>> A = np.random.rand(4,5)
>>> print(A)
[[ 0.4672486 0.42452394 0.55652697 0.50122566 0.45039697]
 [ 0.10245706 0.39642783 0.18395313 0.43191611 0.59490124]
 [ 0.97133547 0.06577487 0.33274442 0.39404957 0.34601463]
 [ 0.60141568 0.01085386 0.69123586 0.33062893 0.68429281]]
```

Création de matrices nulles ou de matrices constantes.

```
Matrice nulle de type (3,4)

np.zeros([3, 4])
array([[ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.]])

Matrice composée de 1 de type (3,4)

np.ones([3,4])
array([[ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.]])
```

dimension d'un tableau.

Si A est un tableau de numpy, à une dimension ou une matrice.
np.shape(A) donne le nombre de lignes et de colonnes d'une matrice sous forme d'un tuple, ici un couple.
Pour une matrice **len(A)** donne le nombre de lignes de A et **np.alen(A)** le nombre de colonnes de A. Il faut remarquer que **len(A)** s'utilise sans **np.** contrairement à **np.len(A)**.

size donne le nombre total d'éléments, et par exemple np pour une matrice de type (n,p).

ndim renvoie le nombre d'indices nécessaires au parcours du tableau (usuellement : 1 pour un vecteur, 2 pour une matrice).

```
A=np.random.rand(5,4)
>>> A
array([[ 0.64627153,  0.83166078,  0.52450604,  0.25472639],
       [ 0.18879927,  0.70915753,  0.92600669,  0.62278505],
       [ 0.99826631,  0.13139493,  0.24614402,  0.9075739 ],
       [ 0.86785393,  0.14688378,  0.64664868,  0.54796176],
       [ 0.87425469,  0.01746309,  0.65253954,  0.94076863]])

>>> np.shape(A)
(5, 4)
>>> np.size(A)
20
>>> np.ndim(A)
2
```

Il est possible de considérer de considérer ces fonctions comme attributs d'un tableau, on peut par exemple utiliser **A.shape** au lieu de **np.shape(A)**.

coupe (ou slicing) d'un tableau.

Pour un tableau A de dimension 1 la méthode est identique à celle utilisée pour une liste.

On procède en effectuant une coupe suivant la première et/ou suivant la deuxième dimension.

```
m = np.array([[10*i+j for j in range(8)] for i in range(5)]); m
array([[ 0, 1,   2,  3,  4,  5,  6,  7],
       [10, 11, 12, 13, 14, 15, 16, 17],
       [20, 21, 22, 23, 24, 25, 26, 27],
       [30, 31, 32, 33, 34, 35, 36, 37],
       [40, 41, 42, 43, 44, 45, 46, 47]])

#  élément en position (3,5)
>>> m[3,5]
35
>>> m[3]
# vecteur ligne 4
array([30, 31, 32, 33, 34, 35, 36, 37])
# vecteur colonne 5
>>> m[:,5]
array([ 5, 15, 25, 35, 45])

# lignes 1 à 3, colonnes 2 à 5
m[1:4,2:6]
array([[12, 13, 14, 15],
       [22, 23, 24, 25],
       [32, 33, 34, 35]])
```

```
m[:3,:2]
trois premières lignes deux premières colonnes
array([[ 0, 1],
       [10, 11],
        [20, 21]])


>>> m[2:,4:]
# a partir de la ligne 2 a partir de la colonne 4
array([[24, 25, 26, 27],
       [34, 35, 36, 37],
       [44, 45, 46, 47]])

Les :: correspondent a un pas: une ligne sur deux, une colonne sur trois
m[::2,::3]
array([[ 0, 3, 6],
       [20, 23, 26],
       [40, 43, 46]])
```

6.2 Opérations sur les tableaux ou les matrices

Modification d'une matrice.

Copie d'une ligne ou d'une colonne d'une matrice

```
# Commençons par l'écriture d'un coefficient :
>>> m
array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])

>>> m[1,2] = 999; m

array([[ 0, 1, 2, 3],
       [ 10, 11, 999, 13],
       [ 20, 21, 22, 23]])

# Continuons par l'écriture d'une ligne entière :
>>> m
array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])

>>> m[2] = [6, 7, 8, 9]; m

array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [ 6, 7, 8, 9]])
```

```
# Terminons par l'écriture d'une colonne entière.
>>> m
array([[ 0, 1, 2, 3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])

>>> m[:,1] = [444, 555, 666]; m

array([[ 0, 444, 2, 3],
       [ 10, 555, 12, 13],
       [ 20, 666, 22, 23]])
```

La copie de matrices suit le même principe que la copie de listes python. L'instruction **B=A** ne cée pas une nouvelle matrice mais un alias de A. Toute modification ultérieure de B modifierait A. Une copie véritable de A est obtnue par $B = A.copy()$.

Redimensionnement d'un tableau.

La principale méthode pour modifier la taille d'un tableau est **reshape**. Le redimensionnement d'un tableau est quand même soumis à la contrainte que le nombre total d'éléments doit rester constant. On pourra ainsi passer (dans les deux sens) d'un vecteur de taille n à une matrice de taille (p, q) ou à une matrice de taille (r, s) à condition que n = pq = rs. Si A est un tableau numpy, l'expression **A.reshape(n,p)** renvoie une copie redimensionnée (le contenu de la variable initiale n'est donc pas affecté, comme on le constate ci-dessous).

```
a = np.array(range(6)); a
array([0, 1, 2, 3, 4, 5])

>>> a.reshape(2,3)
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.reshape(3,2)
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.reshape(6,1)
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])

 >>> a
array([0, 1, 2, 3, 4, 5])
```

Transposition d'une matrice.

La fonction (ou la méthode) transpose , ou plus simplement la méthode T , renvoient la transposée d'une matrice :

```
>>> a
array([[ 1, 2, 3, 4],
```

```
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])

>>> a.transpose()
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])

>>> a.T
array([[ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15],
       [ 4,  8, 12, 16]])
```

Opérations usuelles sur les matrices.

Etant donné deux matrices $A, B \in \mathcal{M}_{n,p}(K)$ leur somme est obtenue par $A + B$.

```
>>> C
array([[3, 2, 6],
       [0, 2, 0],
       [0, 1, 7],
       [3, 6, 8],
       [9, 7, 9]])
>>> D
array([[9, 6, 1],
       [3, 8, 1],
       [7, 6, 2],
       [1, 8, 7],
       [3, 4, 9]])
>>> C+D
array([[12,  8,  7],
       [ 3, 10,  1],
       [ 7,  7,  9],
       [ 4, 14, 15],
       [12, 11, 18]], dtype=int32)
```

L'instruction $A + 8$ ajoute 8 à chaque terme de A, L'instruction $A * 8$ multiplie par 8 chaque terme de A.

```
>>> C+8
array([[11, 10, 14],
       [ 8, 10,  8],
       [ 8,  9, 15],
       [11, 14, 16],
       [17, 15, 17]], dtype=int32)
>>> D*8
array([[72, 48,  8],
       [24, 64,  8],
       [56, 48, 16],
       [ 8, 64, 56],
       [24, 32, 72]], dtype=int32)
```

Etant donné deux matrices $(A, B) \in \mathcal{M}_{n,p}(K) \times \mathcal{M}_{p,r}(K)$, le produit $A * B$ est obtenue par **dot(A,B)**, l'instruction $A * B$ n'a de sns que si A et B sont de même type et ne réalise pas un produit matriciel mais donne la matrice dont le coefficient en position (i, j) est le produit des coefficients de A et de B en position (i, j).

```
>>> A
array([[7, 9, 0, 8],
       [7, 2, 7, 5],
       [0, 6, 1, 3],
       [6, 3, 2, 0],
       [8, 9, 9, 2]])
>>> B
array([[5, 0, 7, 6, 9, 4],
       [5, 2, 5, 6, 9, 8],
       [6, 9, 7, 5, 6, 4],
       [4, 6, 7, 9, 7, 3]])
>>> C=dot(A,B) #produit matriciel de A par B
array([[112,  66, 150, 168, 200, 124],
       [107,  97, 143, 134, 158,  87],
       [ 48,  39,  58,  68,  81,  61],
       [ 57,  24,  71,  64,  93,  56],
       [147, 111, 178, 165, 221, 146]])
```

Exercice

Ecrire une fonction **def produit(A,B) :** qui prend en entrée des matrices $(A, B) \in \mathcal{M}_{n,p}(K) \times \mathcal{M}_{p,r}(K)$ et qui renvoie C produit matriciel de A par B.

6.3 Matrices particulières

On a déjà rencontré les matrices **zeros** et **ones**. On peut également citer l'instruction $A.fill(b)$ qui remplit un tableau déjà existant avec la constante b ou les instructions $zeros_like(A)$ et $ones_like(A)$ qui crée une matrice de même type que A composée de zéros (respectivement de 1).

L'instruction **np.arange(n)** crée un tableau (un vecteur) dont les termes sont les entiers de 0 à $n - 1$.

La fonction **np.eye(n,k)** permet de fabriquer la matrice identité ou plus généralement une matrice dont tous les coefficients sont nuls sauf ceux d'une certaine « parallèle » à la diagonale et qui valent 1. Le premier argument (obligatoire) donne le nombre n de lignes. Le second argument (facultatif) donne le nombre p de colonnes (par défaut p = n donc la matrice est carrée). Un argument facultatif nommé k = : : permet de spécifier un décalage de la diagonale de 1 au dessus (si $k > 0$) ou en-dessous (si $k < 0$) de la diagonale principale. Enfin, on peut ajouter un argument pour fixer le type des données (float par défaut).

```
 np.eye(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

>>> np.eye(6,k=-2)
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.]])np.diag(A,k=2)
array([7, 0, 6, 2])
```

La fonction **diag** renvoie la diagonale d'une matrice. Avec un deuxième argument (facultatif) k, on obtient une surdiagonale (si $k > 0$) ou une sous-diagonale (si $k < 0$).
Cette permet aussi de construire des matrices diagonales, dont les termes sont donnés.

```
A=np.random.randint(10,size=(6,6))
>>> A
array([[9, 9, 7, 9, 4, 9],
       [7, 4, 6, 0, 9, 1],
       [6, 3, 4, 0, 6, 8],
       [0, 5, 4, 6, 4, 2],
       [3, 9, 6, 4, 3, 5],
       [1, 8, 4, 9, 5, 7]])
>>> np.diag(A)
array([9, 4, 4, 6, 3, 7])

np.diag(A,k=2)
array([7, 0, 6, 2])

np.diag([1,2,3,4])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Exercice

Ecrire une fonction **def T(n)** qui renvoie la matrice strictement triangulaire d'ordre n donc tous les coefficients sont nuls, sauf ceux situés immédiatement au-dessus de la diagonale et qui valent 1.

Calcul numérique matriciel

Ce paragraphe nécéssite l'utisation du sous module **linalg** de numpy : from numpy.linalg import *
Ceci apporte un certain nombre de fonctions permettant de préciser certains paramètres des matrices.

— Rang d'une matrice A : **rank**(A).
— Inverse d'une matrice inversible : **inv**(A). Vérifier le rang de la matrice avant de l'inverser. Si $A \in \mathcal{M}_n(K)$ et si $\text{rg}(A) < n$ la matrice n'est pas inversible.
— Résolution d'un système $AX = B$: **solve**(A, B), où A est une matrice carrée d'ordre n et B un vecteur colonne à n composantes.

— Calcul du déterminant d'une matrice carrée A : $\mathbf{det}(A)$. Il ne faut pas oublier qu'il s'agit de calculs approchés et que dans le cas où le détemrminant prend une valeur absolue petite il faut aborder le calcul de l'inverse d'une matrice ou de la résolution d'un système avec circonspection.

Exemples

```
B= array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])

        rank(B)
        2

B n'est pas inversible.


G=array([[7, 5, 9],
       [8, 5, 3],
       [7, 9, 7]])

In [11]: det(G)
Out[11]: 213.99999999999994

In [12]: inv(G)
Out[12]:
array([[ 0.03738318,  0.21495327, -0.14018692],
       [-0.1635514 , -0.06542056,  0.23831776],
       [ 0.1728972 , -0.13084112, -0.02336449]])


 H=array([1,2,3])

 solve(G,H)
 array([ 0.04672897,  0.42056075, -0.1588785 ])
```

Chapitre 7

Modules scipy et matplotlib

7.1 scipy

scipy est une bibliothèque de calculs numériques étendant ceux déja présent dans **numpy**. Elles contient les sous modules suivants :

1. **integrate** permettant des calculs approchés d'intégrales et des résolutions d'équations différentielles.
2. **linalg** qui étend le module correspondant de **numpy**.
3. **interpolate** permettant d'interpoler des fonctions, par exemple par les polynômes de Lagrange.
4. Bien d'autres sous modules que l'on peut obtenir par la commande **help(scipy)**

Exemples

calcul approché de $I = \int_0^1 e^{x^2} dx$ en utilisant la fonction **quad** du module **integrate**.

```
 def f(x):
         return exp(x**2)

quad(f,0,1)
(1.4626517459071815, 1.623869645314337e-14)

Le résultat fait apparaitre une valeur approchée de l'intégrale avec un majorant de l'erreur commise.

Voici les fonctions disponibles dans ce module.

 help(integrate)
 Methods for Integrating Functions given function object.

   quad          -- General purpose integration.
   dblquad       -- General purpose double integration.
   tplquad       -- General purpose triple integration.
   fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
   quadrature    -- Integrate with given tolerance using Gaussian quadrature.
   romberg       -- Integrate func using Romberg integration.

 Methods for Integrating Functions given fixed samples.

   trapz         -- Use trapezoidal rule to compute integral from samples.
   cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
   simps         -- Use Simpson's rule to compute integral from samples.
```

```
   romb          -- Use Romberg Integration to compute integral from
                    (2**k + 1) evenly-spaced samples.

   See the special module's orthogonal polynomials (special) for Gaussian
      quadrature roots and weights for other weighting factors and regions.

 Interface to numerical integrators of ODE systems.

   odeint        -- General integration of ordinary differential equations.
   ode           -- Integrate ODE using VODE and ZVODE routines.
```

L'équation différentielle $((t+1)\ln(t+1))y' - y = -\frac{1}{t+1}(\ln(t+1)+1)$. est écrite sous la forme : $y' = \frac{1}{(t+1)\ln(t+1))}y - \frac{1}{(t+1^2)\ln(t+1))}$.

```
 def f(t,y):
    return (1/((t+1)*log(t+1)))*y-(1/(t+1)**2)*(log(t+1)+1)

t=arange(0,4,1.)

y=odeint(f,2,t)
donne
array([[ 2.        ],
       [ 1.91852081],
       [ 2.13649564],
       [ 2.55776486]])
 Les vleurs sont y(0)=2, valeur initiale et les valeurs y(1),y(2),y(3) correspondant aux ordonnées
 des points choisis dant le tableau t.
```

Le module **optimize** de scipy contient la fonction **bisect(f,a,b)** qui détermine une solution de $f(x) = 0$ par dichotomie sur l'intervalle $[a, b]$.
Ce module contient également la méthode **newton** qui détermine une solution de $f(x) = 0$ par la méthode de newton ou de la sécante si la dérivée n'est pas précisée.

```
bisect(lambda x:x**2-2,0,2)
 1.4142135623724243
```

Le module **interpolate** de **scipy** permet d'obtenir des polynômes de Lagrange. La syntaxe est : **lagrange(x,y)** où x et y sont deux tableaux de même dimension, x abscisses et y ordonnées.

```
  x=linspace(-5,5,20)
f=lambda x :1/(x**2+1)
y=f(x)
p=lagrange(x,y)
```

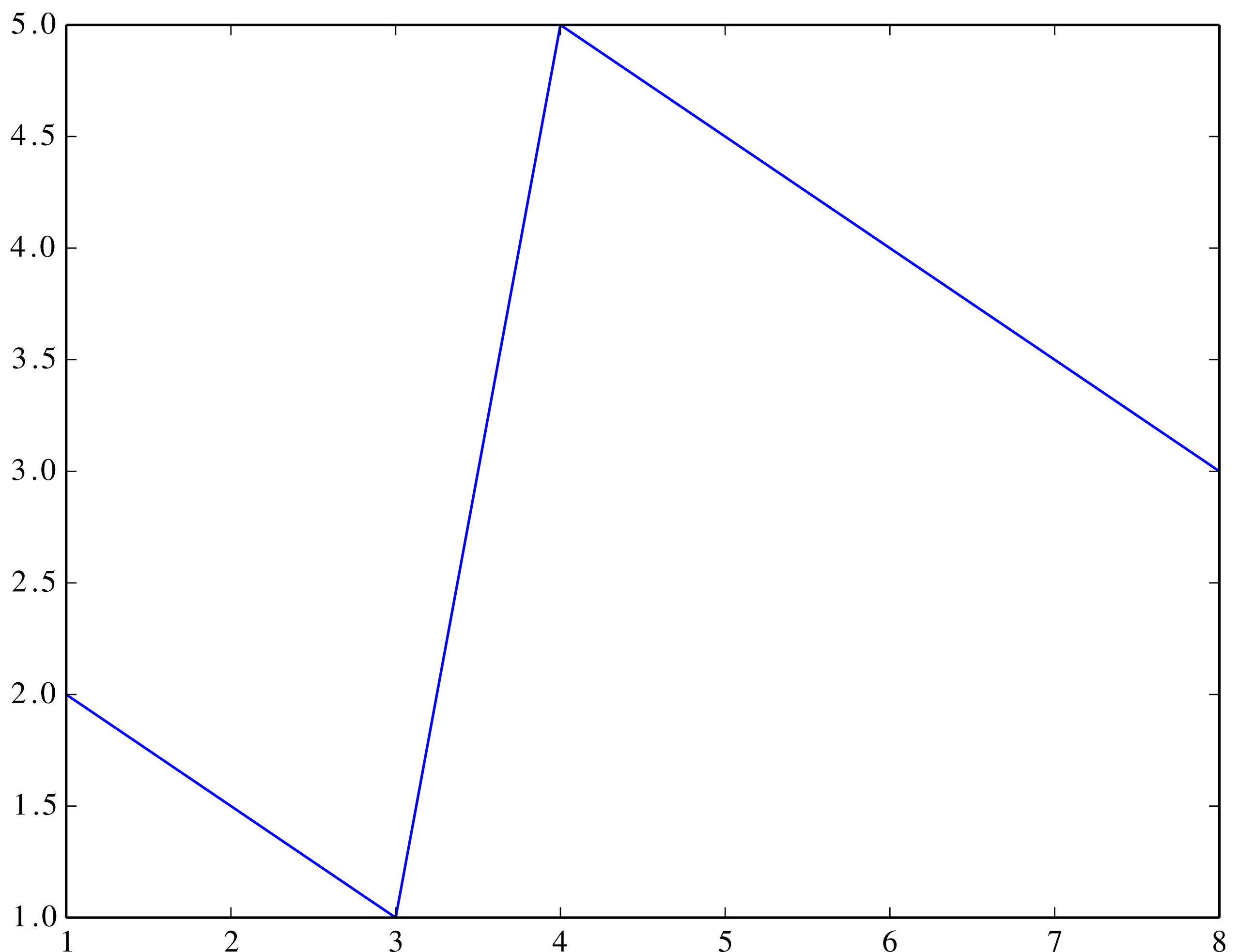
7.2 Tracé de courbes avec matplotlib

Nous allons utiliser la bibliothèque Numpy et le module Pyplot de la bibliothèque Matplotlib.
Le code usuel pour commencer est :
import numpy as np

import matplotlib.pyplot as plt

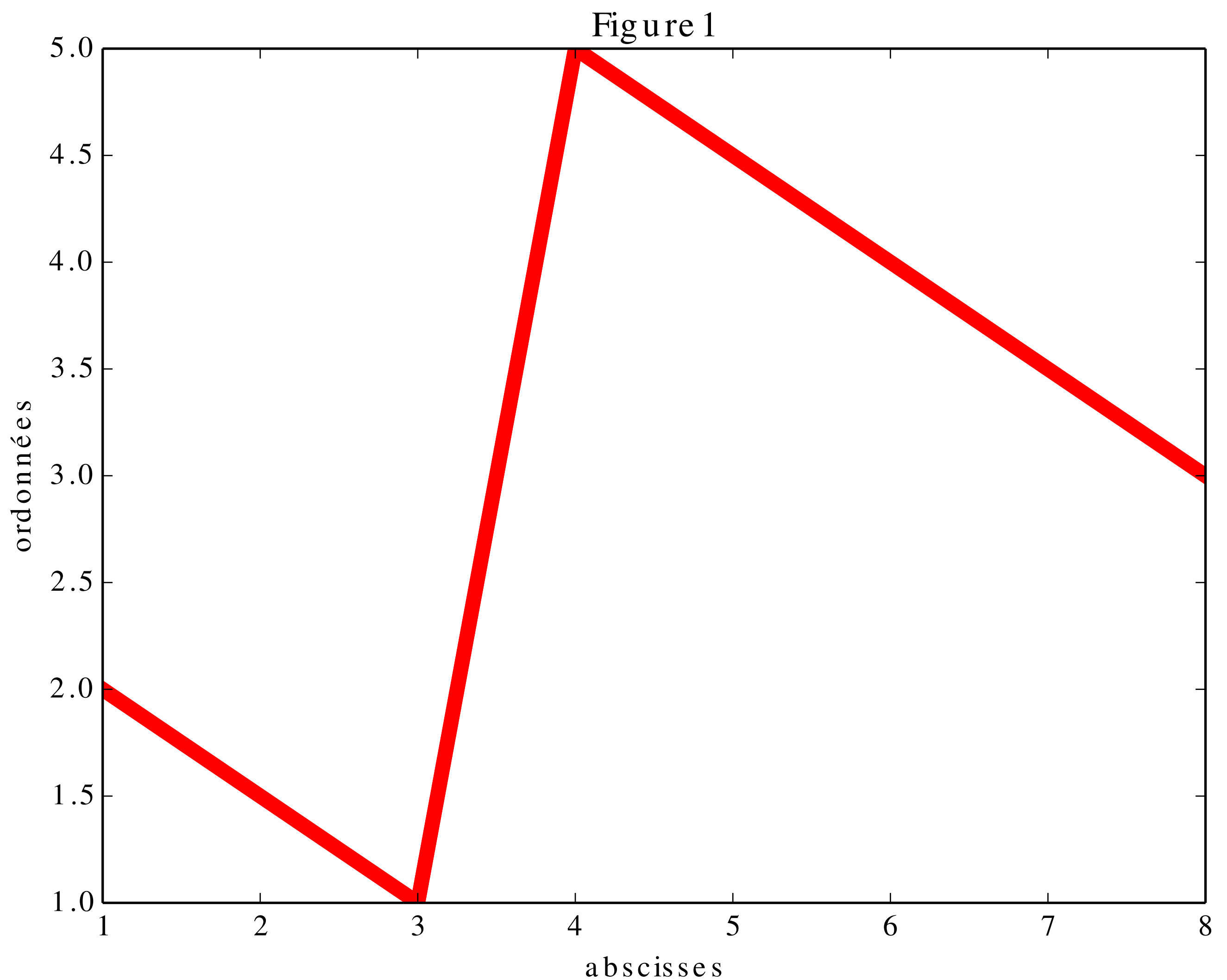
Le principe : on crée une liste d'abscisses et une liste d'ordonnées ; les points sont alors placés et reliés par des segments. Le code est le suivant : $x = [1, 3, 4, 8]$
$y = [2, 1, 5, 3]$

plt.plot(x,y)
plt.savefig('figure') #pour sauvegarder la figure
plt.show()



On change la couleur et l'épaisseur du trait, on ajoute un titre et des étiquettes :
plt.plot(x,y,color='red',linewidth=6)
plt.title("Figure 1")
plt.xlabel("abscisses")
plt.ylabel("ordonnées")

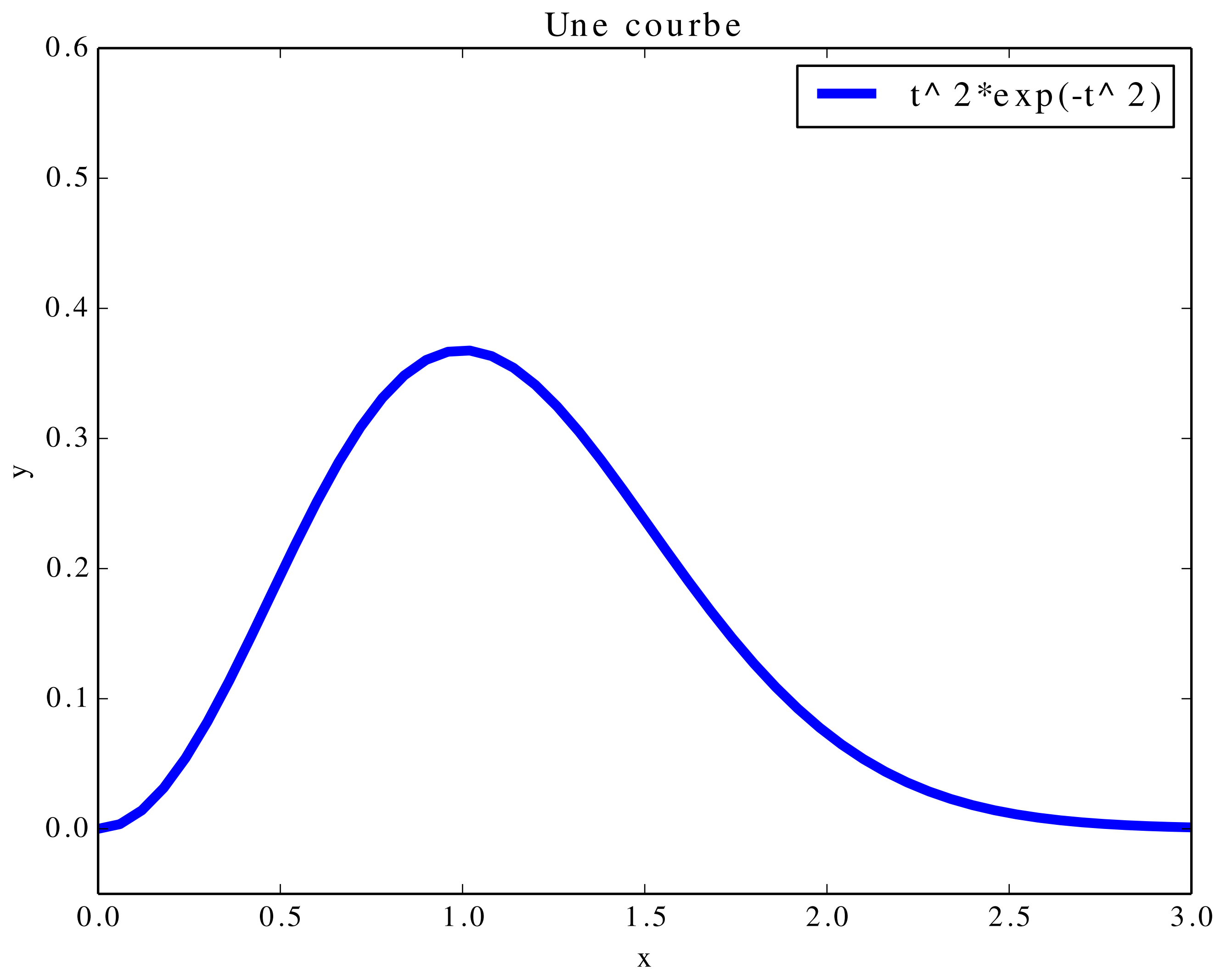
On obtient :



On utilise Numpy pour les fonctions :

```
def f(x):
return x**2*np.exp(-x**2)
x=linspace(0,3,51)
y=f(x)

plt.plot(x,y,linewidth=4)
plt.title(‘‘Une courbe’’)
plt.xlabel(’’x‘‘)
plt.ylabel(’’y‘‘)
plt.legend([’’t^2*exp(-t^2)‘‘])
plt.axis([0,3,-0.05,0.6])
plt.show()
```


Une courbe
t^ 2*exp(-t^ 2)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x