Informatique pour tous

1° année de CPGE

Cours

A. Informatique pour tous – 1° année

A.I. Contexte

L'outil informatique est un outil extrêmement puissant si on le compare à ce qu'un humain est capable de faire aujourd'hui. Il est donc intéressant de savoir programmer au service de soi-même et des autres.

Lorsque l'on parle informatique, on pense le plus généralement aux programmes à notre disposition dans lesquels nous sommes consommateurs d'options pré intégrées et nous nous retrouvons souvent limités à ce que le développeur nous met à disposition.

L'objectif de la formation en informatique pour tous dispensée en CPGE durant 2 ans est de vous permettre de savoir coder par vous-même des algorithmes vous permettant d'obtenir vos propres résultats, quels qu'ils soient.

Utiliser l'outil informatique pour coder repose sur des principes assez simples comme la maîtrise des variables et de quelques boucles (condition, itération, attente autrement appelées if, for et while) et permet très rapidement, selon ce que l'humain veut en faire, de faire des choses très poussées et dont l'accès est impossible autrement (des années de calcul à la main faites en quelques secondes avec un ordinateur).

Lorsque l'on maîtrise un langage de programmation à l'issu de la formation dispensée en CPGE, il devient assez facile de s'adapter à d'autres codes, l'ensemble de la réflexion étant basée sur quelques principes de boucles simples. Souvent, seule la syntaxe sera différente (si on reste au niveau de l'apprentissage CPGE !), et il sera très aisé de passer d'un langage à un autre (exemple entre Matlab, Scilab, Python).

Il est fort probable que vous utilisiez un jour l'informatique pour faire de la simulation numérique de phénomènes physiques dans le cadre de votre travail. Nous allons ici fixer les bases qui seront un pré requis pour la simulation et les résolutions que vous aurez à mettre en œuvre dans vos futures écoles d'ingénieurs.

Savoir programmer dans un langage, cela veut dire deux choses :

1 – indépendamment de tout langage, savoir construire un algorithme réalisant des opérations successives dans un but précis

2 – Savoir transcrire ses idées dans un langage précis adapté au logiciel utilisé. Le moindre caractère aura son importance pour être interprété correctement par la machine en vue d'exécuter une tâche

A.II. Ordinateur – Système d'exploitation – Environnement

Ce paragraphe a pour but de vous familiariser avec l'outil informatique qui va vous permettre de révolutionner le monde dans quelques heures (de cours).

Alors que les idées existaient, c'est l'arrivée de l'outil informatique qui a permis de révolutionner le monde ces dernières décennies. Smartphones, impression 3D en sont de beaux exemples.

Et c'est l'évolution constante des composants dans la « boîte noire » de votre ordinateur qui permet, jour après jour, d'augmenter les puissances de calculs (capacités de stockage, rapidité) qui servent à aller toujours plus loin. Il est donc important de savoir ce qu'il y a dans cette boîte noire, si vous ne l'avez jamais ouverte !

Nous allons donc aborder à la fois l'aspect matériel (Hardware) et l'aspect logiciel (Software).

A.II.1 Principaux composants

Nous allons décrire les composants des ordinateurs « fixes », soit à une unité centrale (donc pas de tablettes, mobiles, ordinateurs portables…) dans lesquels on trouvera des organisations plus ou moins similaires, mais miniaturisées.

Les images de la tour (intérieur et extérieur) des deux prochaines pages sont issues du cours de Clair Gaudy, je l'en remercie !

A.II.2 A l'extérieur

A.II.2.a.i Unité centrale / Tour

Face avant

Lecteur CD/DVD

Face arrière

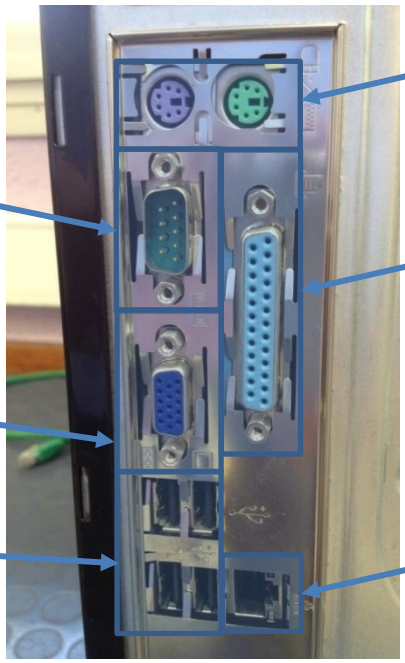
Alimentation

Ports de communication externes

Zone d'installation de cartes (graphique, son Ethernet…)

Sortie carte graphique

A.II.2.a.ii Ports de communication externe



Ports PS/2

Port Série COM1

Port Parallèle

Ports VGA carte graphique intégrée

Ports USB

Port Ethernet

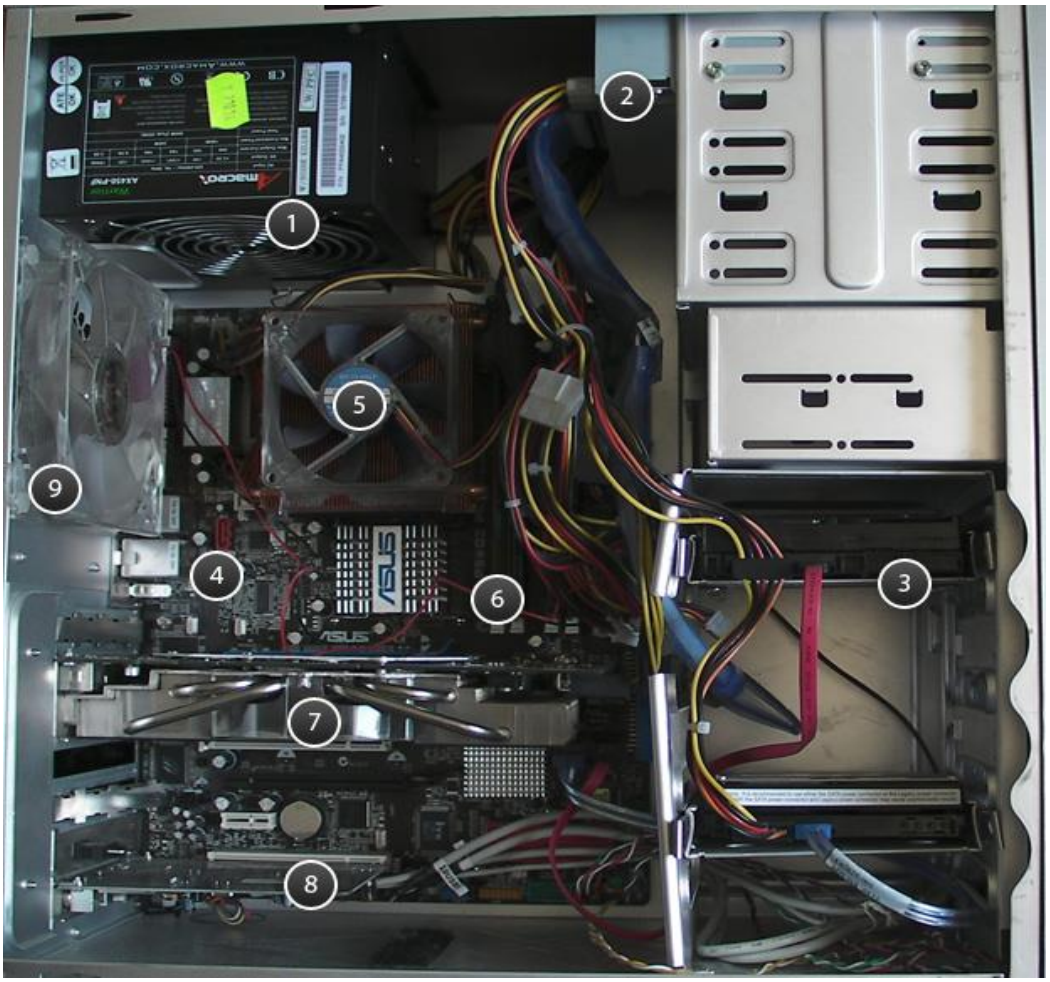
Port DVI

Port HDMI

Port VGA

A.II.3 A l'intérieur

A.II.3.a Vue générale



1	Alimentation
2	Lecteur CD/DVD/Blu-ray
3	Disques durs
4	Carte mère
5	Processeur + Ventilateur
6	Mémoire vive / RAM
7	Carte graphique
8	Autres cartes
9	Ventilateur

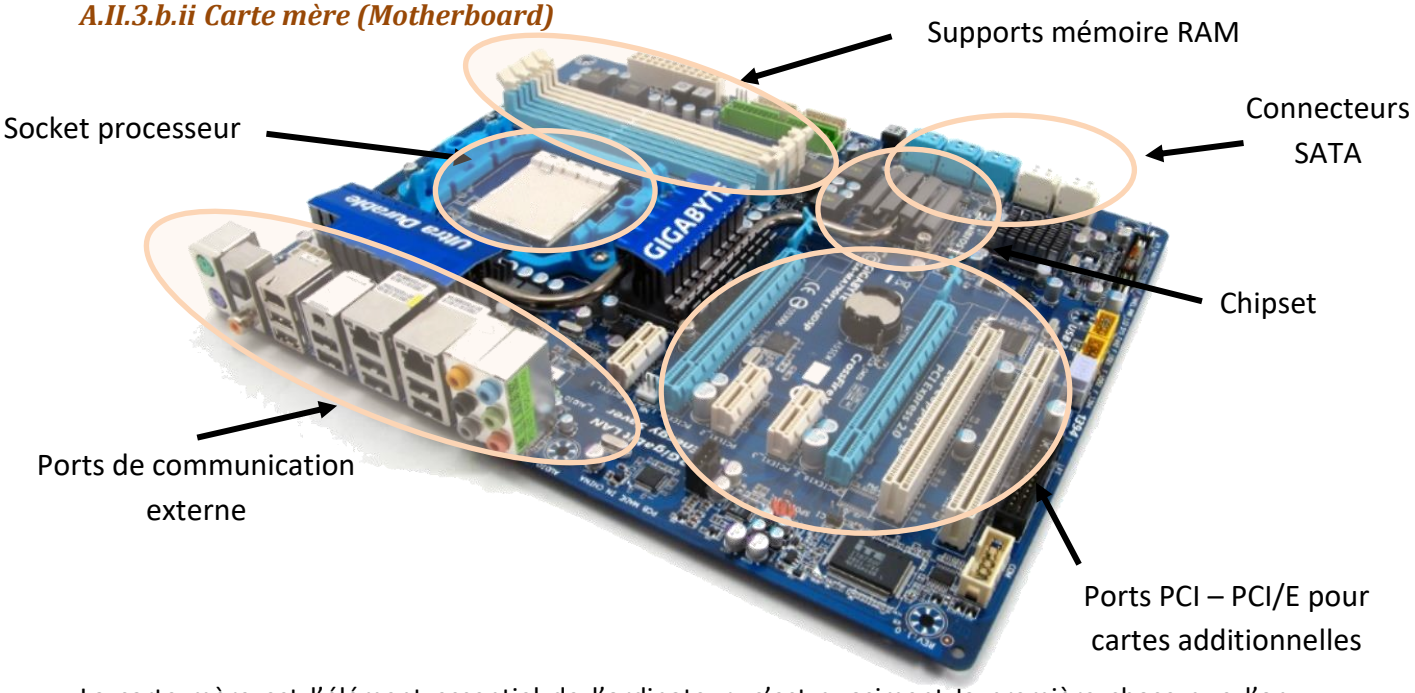
A.II.3.b Composants

A.II.3.b.i Alimentation

L'alimentation fournit la puissance nécessaire au fonctionnement de tous les composants de l'ordinateur. Branchée au secteur (220V), elle adapte la tension du réseau en différentes tensions adaptées à chaque sous système. De l'alimentation partent un grand nombre de fils avec des connectiques diverses, adaptées à chaque sous système.

L'alimentation d'un PC doit être choisie en dernier, afin qu'elle puisse subvenir à la somme des puissances consommées par tous les composants choisis dans l'unité centrale.

A.II.3.b.ii Carte mère (Motherboard)



La carte mère est l'élément essentiel de l'ordinateur, c'est quasiment la première chose que l'on installe dans une unité centrale. Elle contient essentiellement des circuits imprimés et des ports de connexion. On retrouve les ports de communication externes, mais aussi des ports internes multiples, en particulier les ports PCI / PCI Express qui permettent la connexion des cartes graphiques, Ethernet, son etc. qui seront ensuite alimentées par des câbles partant de l'alimentation.

En générale, les cartes mères gèrent de base la connexion internet, le son et ont une carte graphique intégrée. Mais très souvent, on ajoute de nouvelles cartes remplissant les mêmes fonctions mais avec des capacités bien améliorées (Ethernet avec option Wol : Wake on lan, son gérant le 5.1, vidéo gérant la 3D, voir les casques de réalité virtuelle). Par ailleurs, cela permet aussi de s'adapter avec une ancienne carte mère à de nouvelles cartes plus performantes chaque jour qui passe… Il arrive qu'une alimentation secondaire soit nécessaire entre la nouvelle carte et l'alimentation, par exemple pour des cartes graphiques très performantes.

La carte mère contient un Chipset (ensemble de puces électroniques) gérant les différents flux circulants entre les composants de la carte mère et le microprocesseur (Intel Core i3, i7 - 32 ou 64 bits) qui effectue tous les calculs. Il est adapté à un type de microprocesseur donné. Les performances d'une carte mère dépendent étroitement du couple chipset/microprocesseur et le chipset est l'élément déterminant les capacités maximales d'un ordinateur.

Le microprocesseur (CPU) s'adapte à la carte mère via un « socket » ou connecteur spécifique. Il intègre des fonctions de logique combinatoire et séquentielle et de la mémoire. C'est le cœur de l'ordinateur.

Les connecteurs SATA permettent de connecter les disques durs à la carte mère. Ceux-ci seront aussi alimentés par des câbles venant directement de l'alimentation.

La carte mère est pilotée par un logiciel intégré appelé BIOS, qui permet, entre autres, de gérer le fonctionnement de ses composants, et d'installer un système d'exploitation comme Windows.

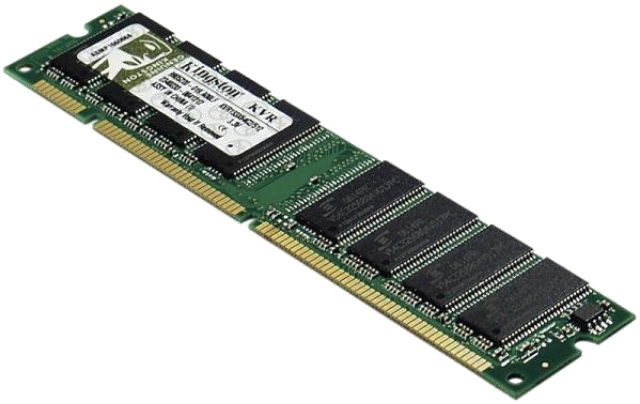
A.II.3.b.iii Refroidissement

Du fait de la dissipation de chaleur des composants en fonctionnement, il est nécessaire d'ajouter des éléments de refroidissement (ventilateurs, radiateurs, système à circulation fluide (caloducs). On trouve généralement

- Un ventilateur pour le processeur associé à un radiateur, un caloduc et une pate thermique qui permettent de refroidir correctement le processeur (Ventirad – Photo de gauche)
- Un ventilateur pour l'alimentation (cf page précédente intégré à l'alimentation)
- Un ou plusieurs ventilateurs dans le boîtier (Photo de droite) afin d'améliorer la circulation dans certains endroits. On peut par exemple ajouter un ventilateur derrière le bloc des disques durs si on utilise plusieurs disques en parallèle dans l'unité centrale.

A.II.3.b.iv Les mémoires

- **Mémoires volatiles**



La mémoire volatile (ou mémoire système) se caractérise par le fait qu'à l'arrêt de l'ordinateur, elle disparaît. C'est la mémoire RAM (Random Access Memory) et elle est ajoutée à la carte mère via des barettes de RAM. Elle se caractérise par une très grande rapidité d'accès et permet de stocker des informations à traiter par le processeur. On verra un peu plus bas l'importance de la version de Windows dans la gestion de cette mémoire.

- **Mémoires non volatiles**

On distingue deux types de mémoire non volatile :

- ROM - Read-Only Memory : C'est une mémoire sans accès en écriture, d'un temps d'accès « raisonnable », initialisée lors de la fabrication de la machine et utilisée pour stocker des programmes, mais pas de données, le BIOS par exemple.
- Mémoire de masse ou disques durs : C'est une mémoire dont l'utilisateur dispose pour stocker son système d'exploitation, ses programmes et ses données. On retrouve :
    - Les disques durs HDD (Hard Drive Disk) : De loin les plus répandus, internes ou externes, ils sont composés d'un disque magnétique en rotation sur lequel sont modifiées des données à l'aide du magnétisme. Leurs performances sont liées à la vitesse de rotation des disques !
    - Les disques SSD (Solid State Drive) : Ces disques sont constitués de composants électroniques uniquement. Ne possédant pas de disques en rotation, ils permettent des vitesses d'accès aux données beaucoup plus importantes que pour les HDD et son pour le moment peu répandu car assez chers.

HDD

SSD

A.II.3.c Ordinateur pour calcul scientifique

Lorsque l'on fait des calculs scientifiques (ou que l'on utilise des jeux demandant beaucoup de puissance, on parle de PC Gamers), on recherche la plus grande rapidité possible car les calculs peuvent durer… plusieurs années !

On recherche donc à avoir un **microprocesseur le plus rapide possible**. Il faut donc une carte mère adaptée au type de processeur recherché.

Par ailleurs, on manipule des données stockées dans la mémoire RAM qui peuvent avoir des dimensions extrêmement grandes (matrices à plusieurs millions de lignes et colonnes). Pour pouvoir les stocker, il faut **beaucoup de mémoire RAM**. Aujourd'hui, on peut souvent aller jusqu'à 24 Go de RAM. Il faut donc choisir une carte mère avec le plus d'emplacement RAM possible et pouvant en plus les gérer !

Mais attention, avoir beaucoup de RAM est un prérequis, mais n'est pas suffisant. En effet, sans rentrer dans les détails, lorsque l'on utilise un ordinateur avec 24 Go de RAM, si la version de Windows utilisée est une version 32 bits, on ne pourra pas profiter de toute la RAM disponible pour faute d'adressage possible sur un codage avec 32 bits ($2^{32} = 4\,294\,967\,296$ adresses différentes, soit 4 Go de RAM utilisable !!!). Il faudra donc veiller à installer une version 64 bits de Windows et les logiciels associés, en version 64 bits. Si les anciens ordinateurs sont souvent en 32 bits, le 64 se démocratise, les logiciels avec !

Enfin, pour pouvoir installer une version de Windows 64, il faut nécessairement que le processeur soit au moins en 64 bits. Pour le savoir, sous Windows : Paramètres – Système – Information système

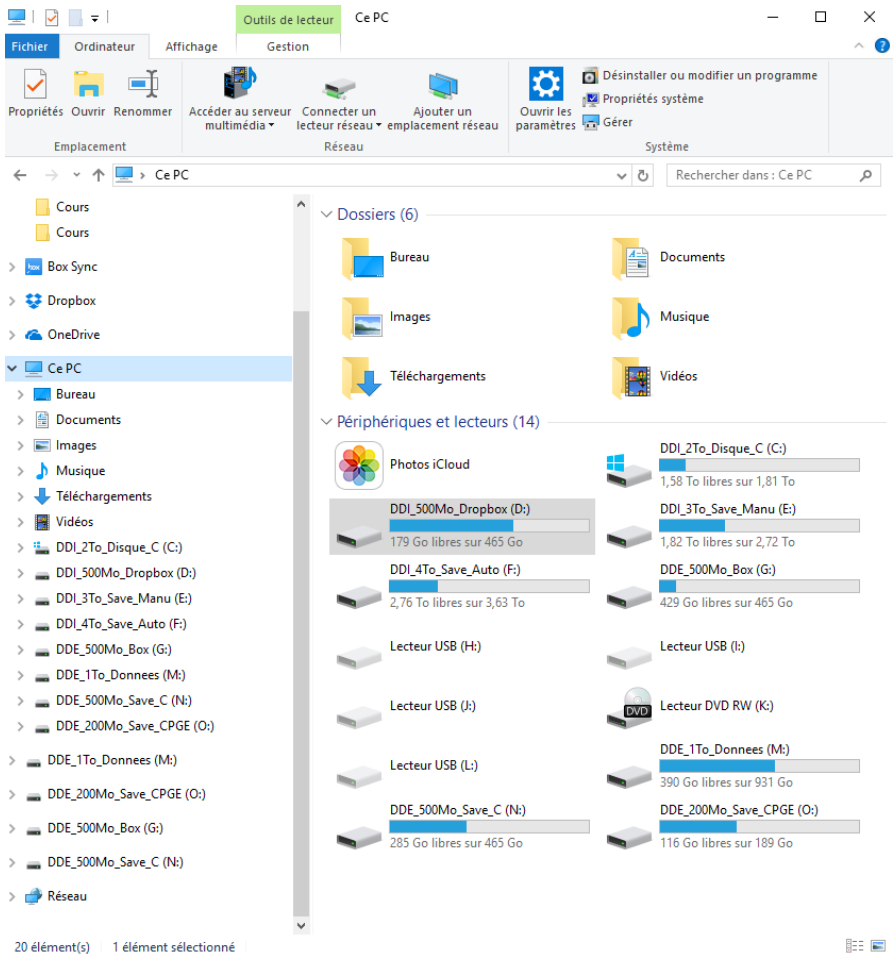
Édition	Windows 10 Professionnel
Version	1607
Version du système d'exploitation	14393.1593
ID de produit	00330-80188-18042-AA392
Processeur	Intel(R) Core(TM) i5-5287U CPU @ 2.90GHz 2.90 GHz
Mémoire RAM installée	8,00 Go
Type du système	Système d'exploitation 64 bits, processeur x64

A.II.4 Manipulation d'un système d'exploitation : Windows et dossiers

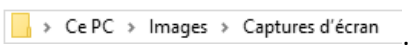
Ce court paragraphe a pour but de vous montrer comment gérer la manipulation des fichiers sous Windows, mais je pense que vous maîtrisez tous déjà parfaitement cela.

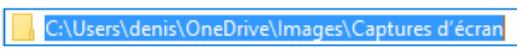
Dans Windows, ouvrons donc un explorateur de fichier à l'aide de l'icône  probablement présente sur le bureau. Sinon, ouvrez le poste de travail !

A quelques détails près, cela ressemble à ça :



Sous la ligne « Ce PC », au lycée, et lorsque vous êtes logués avec votre session (prénom.nom) pour l'identifiant, et votre date de naissance sans / du type 01012000 pour mot de passe (avant de le changer), vous verrez apparaître un dossier à votre nom. C'est ici que vous stockerez vos documents que vous retrouverez d'un ordinateur à l'autre.

Je n'ai pas grand-chose à rajouter, si ce n'est qu'il peut être utile de savoir copier/coller l'adresse d'un répertoire, par exemple, je vais dans 

. Il suffit de cliquer à droite du dernier nom de dossier, ici « Captures d'écran », et on alors le chemin absolu qui apparaît : 

.

A.II.5 Manipulation d'un environnement de développement – Python/Pyzo

A.II.5.a Définition

Un environnement de développement est un ensemble d'outils qui permettent d'éditer un code dans un langage de programmation donné, de le compiler, de le déboguer et de l'exécuter pour en afficher les résultats éventuels.

A.II.5.b Windows et Mac

Vous êtes sous Windows, bien heureux soyez-vous !

Vous êtes sous mac ? Vous pourrez faire 99% de l'année avec Pyzo pour mac. Mais le 1% restant vous fera perdre des heures cruciales.

Un conseil donc : Utilisez l'utilitaire Boot camp, et installez une version de Windows sur votre mac. Il faut une image iso de Windows, et une licence. Cela vous permettra au passage d'utiliser Solidworks en SI.

A.II.5.c Installation logicielle : « Pyzo » + « Miniconda »

Partons du principe que vous disposez d'un ordinateur avec une version récente de Windows installée.

Bien qu'ils soient déjà à votre disposition en salle d'informatique, je souhaite ici vous expliquer comment avoir ces logiciels à votre disposition chez vous.

Il existe plusieurs possibilités pour programmer en Python. Je vous propose de vous rendre sur le site ci-dessous :

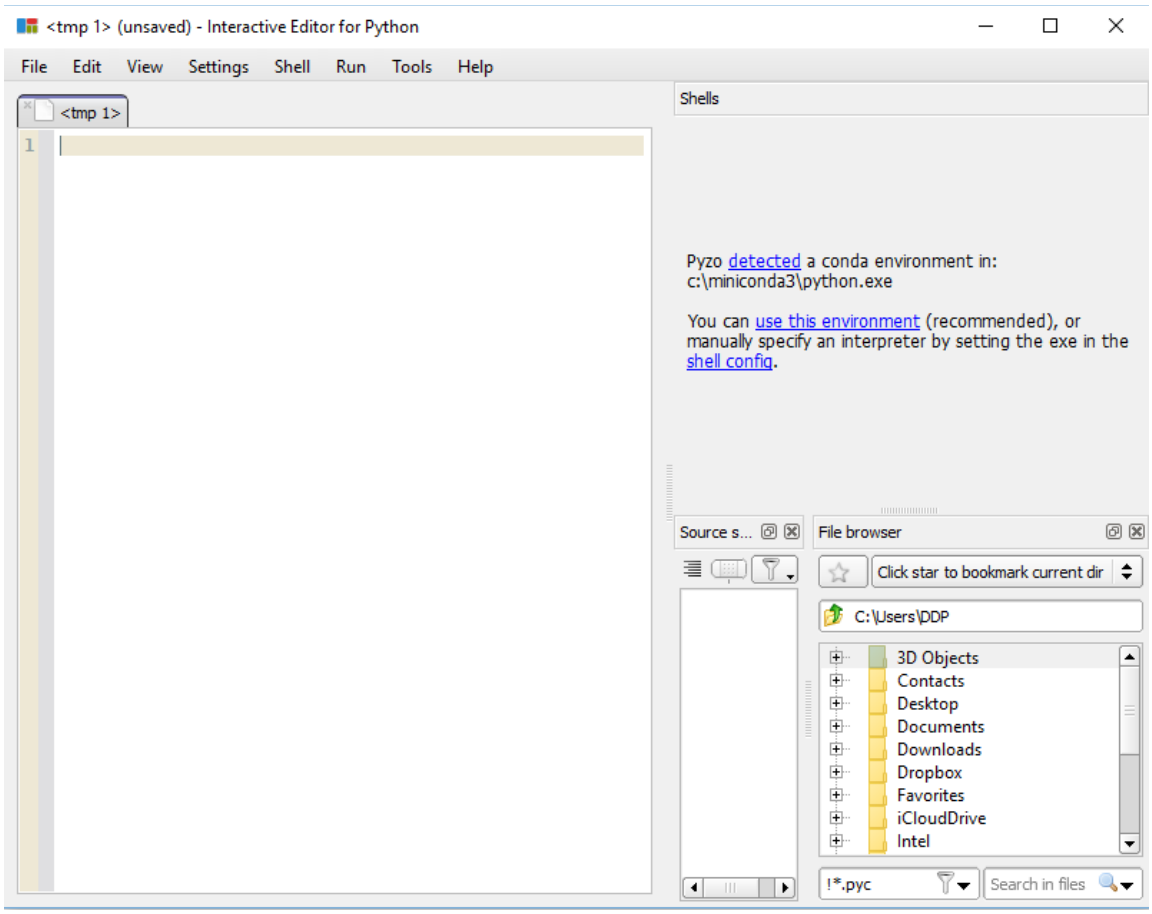
http://www.pyzo.org/start.html

Téléchargez et installez dans le répertoire par défaut « Pyzo », logiciel qui permettra de taper du code dans une interface appropriée.

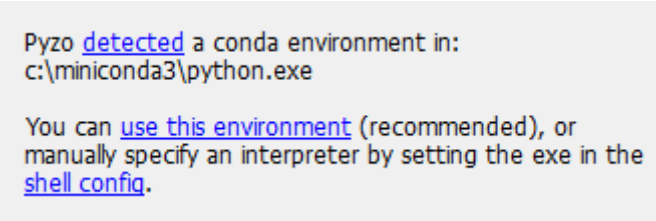
Téléchargez et installez dans le répertoire par défaut « Miniconda », logiciel qui reconnaîtra et exécutera le code en langage python.

Attention, nous utilisons Python 3 et non Python 2 (mais si vous suivez les directives, vous aurez bien Python 3 !)

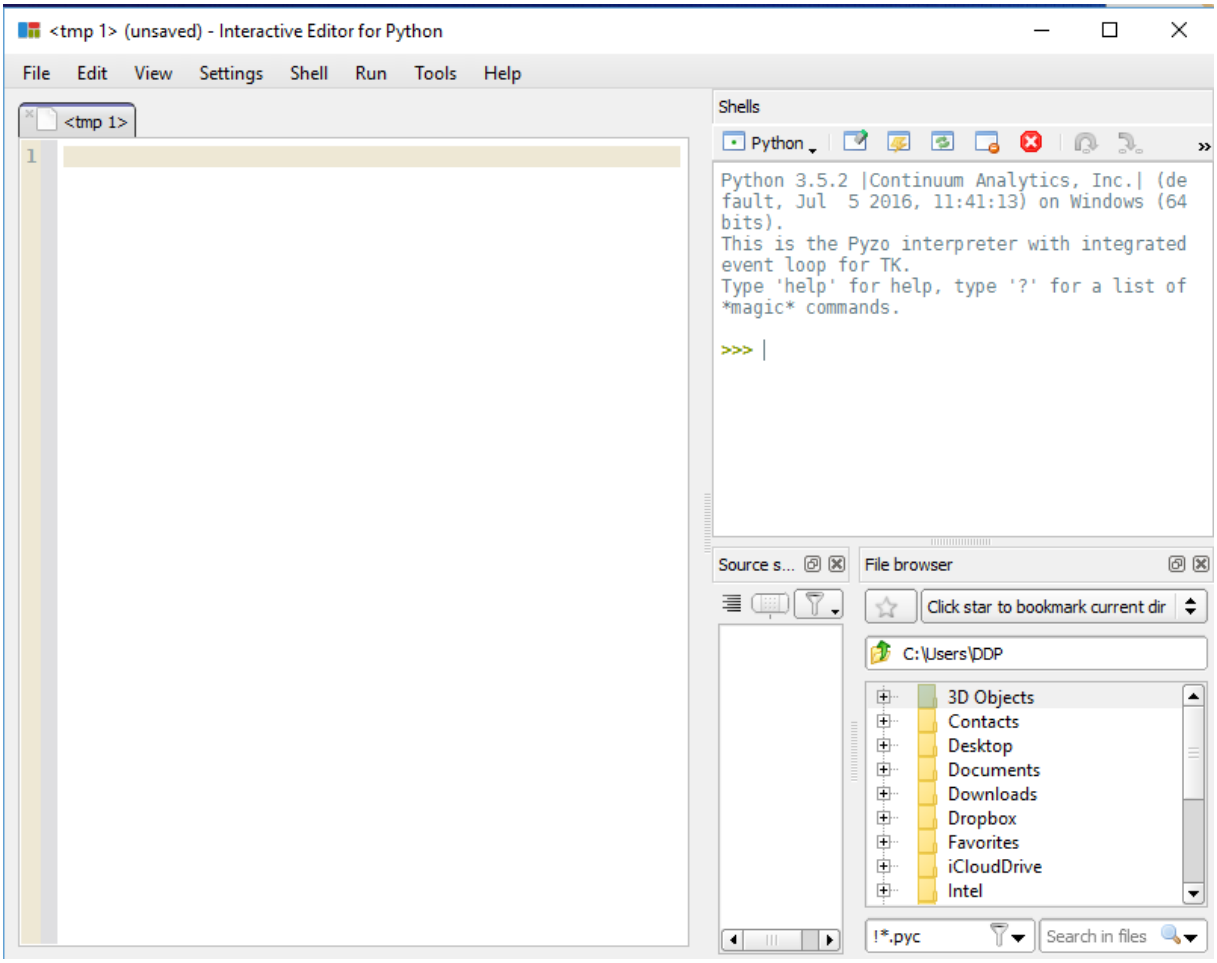
Lorsque tout est installé, lancez « Pyzo » (icône sur le bureau), vous devriez voir apparaître ceci :



Dans la fenêtre à droite, normalement, Pyzo a reconnu un « conda environment », cliquez sur les mots bleus « use this environment » :



Vous voyez alors apparaître ceci :



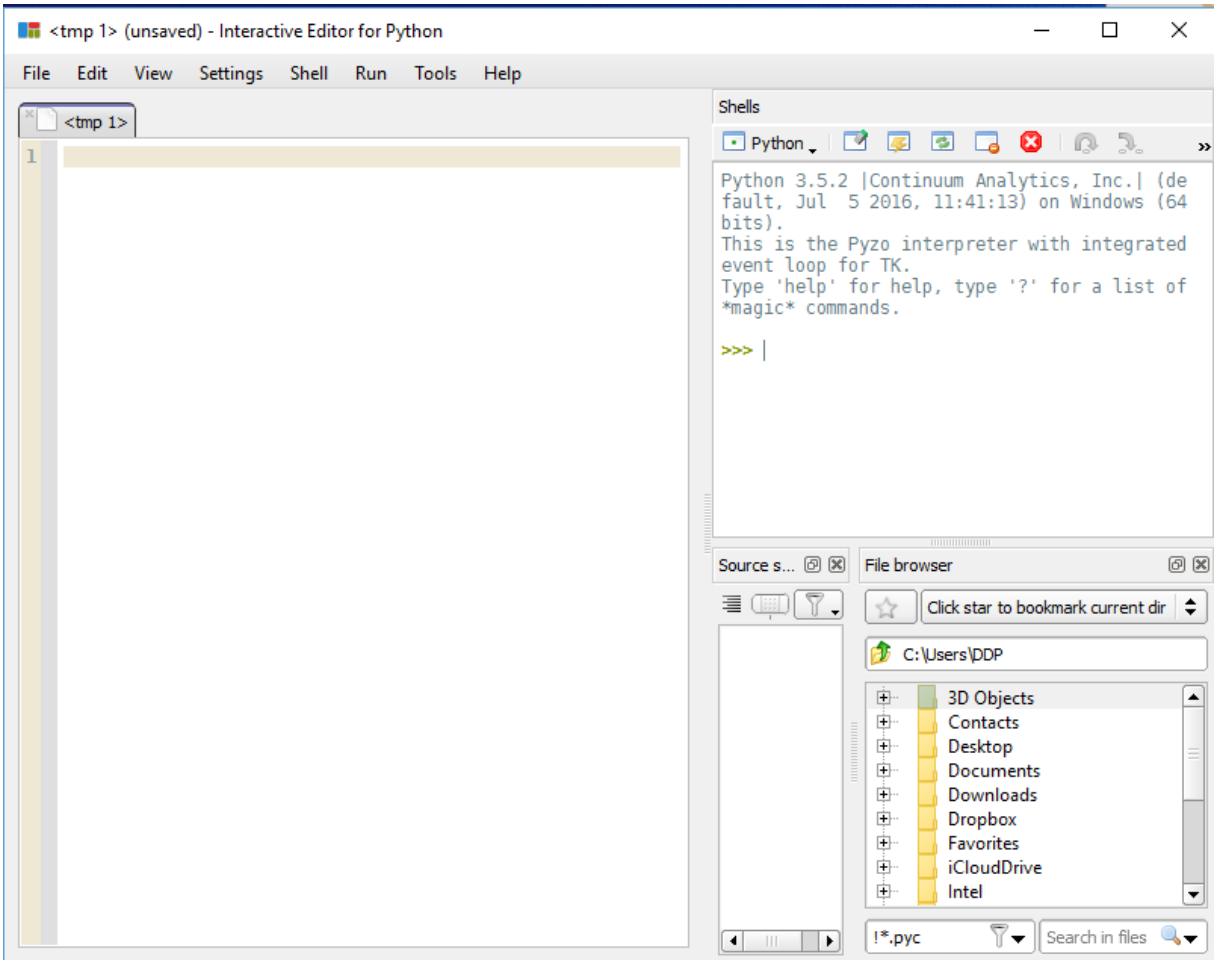
Dans la fenêtre à droite, après les sigles >>>, tapez d'abord **conda install scipy** afin d'installer le module « scipy », pressez « Entrée », attendez, puis lors du message « Proceed ([y]/n) », écrire « y » puis pressez « Entrée » et attendez la fin de l'installation.

Procédez de même avec la phrase : **conda install pyqt matplotlib pandas sympy**

Vous êtes prêts à programmer en Python et à faire du calcul scientifique. Autrement dit, vous avez le piano, reste à apprendre à lire des partitions puis à jouer.

A.II.5.d Premiers pas avec Python sous « Pyzo »

Revenons sur l'affichage de Pyzo :



La partie gauche est un fichier appelé « tmp 1 » dans lequel vous allez pouvoir inscrire du code en vue de l'exécuter intégralement dès que nécessaire. C'est un peu comme écrire une partition. L'écriture de la partition ne fait pas de musique !

A droite, dans le cadre du haut, on voit ce que l'on appelle la « console » ou « Shells », c'est-à-dire la zone où les calculs se font. C'est l'instrument de musique. Tout ce qui y est écrit sera exécuté dès l'appuie sur « Entrée ». Toutefois, à la différence d'un instrument de musique, ne sortira que ce que le programmeur a demandé, et heureusement, car s'il y a des milliers de calculs par secondes, on ne veut que le résultat…

Vous pouvez réorganiser les fenêtres à convenance en cliquant sur leur nom, en laissant appuyé le bouton gauche et en la déplaçant. De même, à l'interface entre fenêtres, vous verrez apparaître une double flèche qui permettra de redimensionner chaque fenêtre.

Lorsque l'on écrit 10+10 dans la console, et que l'on tape entrée, on obtient le résultat :

```
>>> 10+10
20
```

Remarque : il vous sera souvent très utile de pouvoir réécrire une ligne qui a été précédemment exécutée sans avoir besoin d'utiliser la souris pour faire copier/coller. Pour cela, rien de plus simple. Dans la console, sur une nouvelle ligne vierge, il suffit de taper sur la flèche du haut autant de fois qu'il faut remonter les lignes. Exemple : tapez 5+5 puis « Entrée », puis 10+10 et « Entrée », puis 20+20 et « Entrée » :

```
>>> 5+5
10

>>> 10+10
20

>>> 20+20
40
```

Pour pouvoir exécuter la ligne 5+5, il suffit de cliquer 3 fois sur la flèche vers le haut puis « Entrée ».

Intéressons-nous maintenant au fichier à gauche. Créons un fichier personnel dans lequel nous allons entrer du code. Faire « Fichier », « Save as… » puis enregistrer le fichier avec un nom personnel (par exemple « Essai_1 » et dans un répertoire personnel de l'ordinateur, pour moi ce sera sur le bureau.

Il peut être utile d'avoir accès à plusieurs fichiers python dans un même répertoire de travail. Dans la partie en bas à droite, nommée « File browser », cliquez sur le symbole  et allez ouvrir le dossier dans lequel vous avez mis votre fichier. Vous devriez alors le voir apparaître en dessous.

On voit maintenant dans la partie gauche notre fichier, vierge, dans lequel je vous invite à écrire 10+10 puis taper Entrée.

```
1  10+10
2
```

Il ne se passe rien, c'est normal, on a en fait préparé une ligne de code à exécuter lorsqu'on le souhaitera. Pour ce faire, il suffit de taper la touche « F5 » qui

- Enregistre le fichier
- L'exécute dans la console

Dans la console, on voit alors :

```
>>> (executing line 1 of "Essai_1.py")

>>>
```

L'exécution a eu lieu, une nouvelle ligne permet d'écrire du code, il n'y a donc pas eu d'erreurs. L'ordinateur nous informe qu'il a exécuté le fichier mais on ne voit pas le résultat de notre calcul. En effet, la console et le fichier ne se comportent pas de la même façon. La console exécute et donne le résultat du code tapé directement alors que l'exécution d'un fichier exécute chacune de ses lignes sans renvoyer de résultat tant qu'il n'est pas demandé, et heureusement.

Imaginons qu'un calcul prenne un milliard de calculs intermédiaires, heureusement que ce milliard de calculs n'apparaît pas.

Pour afficher le résultat d'un calcul, il faut le demander avec la commande « print() ».

Entrez dans le fichier le code suivant :

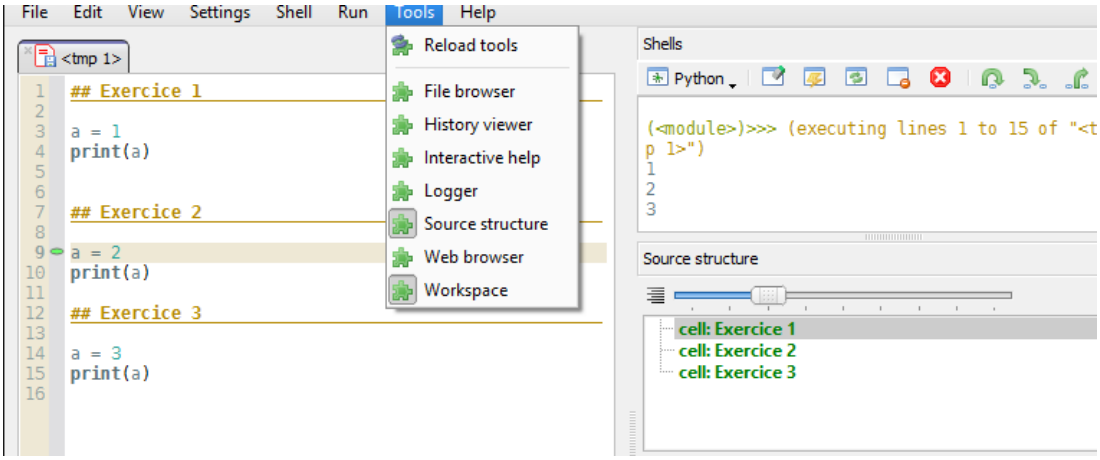
```
1  print(10+10)
2
```

Puis exécutez le fichier avec la touche « F5 », vous verrez alors apparaître le résultat :

```
>>> (executing line 1 of "Essai_1.py")
20
```

Remarques :

- Il est possible de n'exécuter qu'une partie d'un fichier, pour cela il faut sélectionner le code en question et faire Alt + Entrée.
- Il est possible de créer des « cell », c'est-à-dire des parties de programme, par exemple des exercices, que l'on peut exécuter séparément en plaçant le curseur dans la partie concernée, puis en faisant « Ctrl+Entrée ». On peut par ailleurs simplement naviguer d'une « cell » à l'autre et donc la sélectionner en affichant les « cell » existantes en faisant « Tools » puis cliquer sur « Source structure » qui affiche ne fenêtre « Source structure » avec les « cell » existantes. Tous ces éléments sont illustrés ci-dessous :



Nous voilà donc prêts à programmer les programmes les plus fous qui révolutionneront le monde.

Lorsqu'il y a une erreur dans un code, empêchant son exécution, un message apparaît dans la console en rouge. L'erreur est indiquée : N° ligne + position à l'aide d'un ^. Il faut retenir une chose importante :

Un chapeau en début de ligne correspond généralement à une erreur à la ligne précédente. Par exemple, une parenthèse oubliée :

```
a = 1
b = (a + 1
print(a)
```

```
  File "<tmp 1>", line 3
    print(a)
          ^
SyntaxError: invalid syntax
```

A.III. Représentation des nombres en informatique

Comprendre comment sont représentés les nombres en informatique vous permettra peut-être un jour d'éviter de porter la responsabilité de l'explosion d'une fusée comme Ariane 5, dont l'explosion fût d'origine numérique… Ce paragraphe devrait vous en apprendre plus sur ce qu'une machine peut faire et ne pas faire avec les nombres.

Prenons un exemple simple. Calculons avec python :

$$A = 0{,}2 + 0{,}1$$

Le résultat est très simple, n'est-ce pas ?

$$A = 0{,}2 + 0{,}1 = 0{,}3$$

Voyons de résultat sous Python :

```
>>> 0.2+0.1
0.30000000000000004
```

OUPS

Ou encore :

```
>>> 0.3==0.1+0.2
False
```

A.III.1 Code binaire

A.III.1.a Introduction

Un ordinateur manipule des informations binaires, c'est-à-dire à deux états : 0 ou 1

Il est donc nécessaire de traduire un nombre du système en base 10 en un nombre binaire afin de permettre à un système informatique de le manipuler.

Par exemple, le 10 de notre système en base 10 est représenté par le « nombre » 1010 en binaire.

On écrira :

$$10_{(10)} = 1010_{(2)}$$

Le principe est très simple. En base 10, on ajoute un nouveau chiffre à chaque fois que l'on dépasse la valeur 9, 99, 999 etc. puis on l'incrémente. En binaire, on ajoute cette retenue dès que l'on dépasse la valeur 1.

Ainsi, pour les 5 premiers entiers :

Base 10	Base 2
0	0
1	1
2	10
3	11
4	100
5	101

On appelle chaque information valant 1 ou 0 un « **bit** ». Un « **mot »** est une suite de « bits ». Un **octet** est un mot de 8 bits.

Un nombre entier naturel codé sur n bits permettra de représenter 2^n valeurs différentes et pourra donc au maximum correspondre à la valeur en base 10 de $2^n - 1$, le 0 étant inclus. Par exemple, un nombre entier codé sur 8 bits donnera 256 valeurs différentes et ne pourra excéder $2^8 - 1 = 255$. Ce nombre s'écrira en base 2 :

$$255_{(10)} = 11111111_{(2)}$$

A.III.1.b Principe de l'écriture d'un entier dans les bases 2 et 10

Prenons un exemple dans la base 10 :

$$352_{(10)} = 3 * 10^2 + 5 * 10^1 + 2 * 10^0$$

Chaque chiffres (digit) 3, 5 et 2 correspond à une puissance de 10.

En binaire, on respecte le même principe mais avec des puissances de 2 :

$$1010_{(2)} = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

A.III.1.c Transcodage Entier ↔ Binaire

Soient :

- Un nombre en base 10 de digits d_i tel qu'il s'écrive $(d_n d_{n-1} \dots d_1 d_0)_{10}$
- Un nombre binaire de digits b_i tel qu'il s'écrive $(b_m b_{m-1} \dots b_1 b_0)_2$

Ecrivons :

$$d_n 10^n + d_{n-1} 10^{n-1} + \cdots + d_1 10^1 + d_0 10^0 = b_m 2^m + b_{m-1} 2^{m-1} + \cdots + b_1 2^1 + b_0 2^0$$

Le transcodage d'un entier

- Binaire/Base 10 consiste à trouver les digits d_i connaissant les digits b_i
- Base 10/binaire consiste à trouver les digits b_i connaissant les digits d_i

A.III.1.c.i Binaire → Entier

Rien de plus simple, connaissant le nombre en binaire, il suffit de procéder à la somme de chacun de ses digits pris de droite à gauche en multipliant par des puissances de 2 croissantes.

Exemple : $1111101000_{(2)}$

$$1111101000_{(2)} = 1.2^9 + 1.2^8 + 1.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 1.2^3 + 0.2^2 + 0.2^1 + 0.2^0$$

$$1111101000_{(2)} = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3 = 512 + 256 + 128 + 64 + 32 + 8$$

$$1111101000_{(2)} = 1.10^3 + 0.10^2 + 0.10^1 + 0.10^0$$

$$1111101000_{(2)} = 1000_{(10)}$$

A.III.1.c.ii Entier → Binaire

Supposons qu'un nombre entier N s'écrive :

$$N = b_m 2^m + b_{m-1} 2^{m-1} + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

Par définition, les digits b_i ne sont pas divisibles par 2.

Divisons ce nombre par 2 :

$$\frac{N}{2} = (b_m 2^{m-1} + b_{m-1} 2^{m-2} + \cdots + b_2 2^1 + b_1) + \frac{b_0}{2}$$

On a donc :

$$N = 2q_0 + r_0 = 2 * (b_m 2^{m-1} + b_{m-1} 2^{m-2} + \cdots + b_2 2^1 + b_1) + b_0$$

On trouve donc le dernier digit b_0 comme reste de la division euclidienne de N par 2.

De même, on a :

$$q_0 = 2q_1 + r_1 = 2(b_m 2^{m-1} + b_{m-1} 2^{m-2} + \cdots + b_2) + b_1$$

On trouve donc l'avant dernier digit b_1 comme reste de la division euclidienne de q_0 par 2.

On procède alors ainsi jusqu'à ce que :

$$q_{m-1} = 2q_m + r_m = b_m$$

Avec $q_m = 0$, il reste alors b_m.

Récapitulons :

$N = 2q_0 + r_0$	$r_0 = b_0$
$q_0 = 2q_1 + r_1$	$r_1 = b_1$
…	…
$q_{m-1} = 2q_m + r_m$ $q_{m-1} = q_m$	$r_m = b_m$

Exemple : $1000_{(10)}$

	1000	2									
$n_0 =$	0	500	2								
	$n_1 =$	0	250	2							
		$n_2 =$	0	125	2						
			$n_3 =$	1	62	2					
				$n_4 =$	0	31	2				
					$n_5 =$	1	15	2			
						$n_6 =$	1	7	2		
							$n_7 =$	1	3	2	
								$n_8 =$	1	1	2
									$n_9 =$	1	**0**

$$1000_{(10)} = 1111101000_{(2)}$$

A.III.1.d Transcodage Réel base 10 ↔ Réel base 2

En base 10, lorsque nous manipulons des nombres à virgule :

- Les chiffres avant la virgule sont des puissances de 10
- Les chiffres après la virgule sont des puissances de 1/10

Exemple :

$$5{,}375 = 5.10^0 + 3.10^{-1} + 7.10^{-2} + 5.10^{-3}$$

On peut exprimer ce nombre réel comme la somme de sa partie entière et de sa partie décimale :

$$5{,}375 = 5 + 0{,}375$$

En binaire, on va appliquer le même principe. Ainsi, le nombre $101{,}011$ représente le nombre :

$$1.2^2 + 0.2^1 + 1.2^0 + 0.2^{-1} + 1.2^{-2} + 1.2^{-3} = 4 + 0 + 1 + 0 + \frac{1}{4} + \frac{1}{8} = 5{,}375$$

Là aussi, on peut exprimer ce nombre comme somme de sa partie entière et sa partie décimale :

$$101{,}011 = 101 + 0{,}011$$

On pourra remarquer qu'il y a concordance entre partie entière et partie décimale entre les deux écritures :

$$(101)_2 = (5)_{10} \quad ; \quad (0{,}011)_2 = (0{,}375)_{10}$$

A.III.1.d.i Réel base 2 → Réel Base 10

Pour transcoder un nombre binaire réel en nombre réel en base 10, c'est très simple maintenant que l'on a compris le principe de l'écriture d'un nombre binaire réel.

Ecrivons un nombre binaire réel sous la forme :

$$x = (b_n \dots b_0, c_1 \dots c_m)_2$$

Avec b_i et c_i les bits du mot binaire associé au nombre représenté.

On a alors :

$$(x)_{10} = b_n.2^n + \cdots + b_0.2^0 + c_1.2^{-1} + \cdots + c_m.2^{-m}$$

A.III.1.d.ii Réel base 10 → Réel base 2

Soit un nombre réel en base 10 de la forme : $x = (e_n \dots e_0, f_1 \dots f_m)_{10}$

On sait qu'il doit s'écrire sous la forme : $b_{n'}.2^{n'} + \cdots + b_0.2^0 + d_1.2^{-1} + \cdots + d_{m'}.2^{-m'}$

Séparons partie entière et partie décimale : $Ent = e_n \dots e_0$; $Dec = f_1 \dots f_m$

- **<u>Partie entière binaire</u>**

Pour trouver la partie entière du nombre binaire associé, il suffit de transcrire le nombre entier Ent en binaire.

- **<u>Partie décimale binaire</u>**

Concernant sa partie décimale, prenons un exemple pour comprendre :

$$x = (0{,}375)_{10}$$

Pour obtenir le chiffre des dixièmes, on prend la partie entière de $10x$, soit la partie entière de $3{,}75$. On obtient le chiffre 3 et on définit un nouveau nombre correspondant à la partie décimale de $3{,}75$, soit $0{,}75$. On répète alors l'opération jusqu'à ce que la partie décimale soit nulle.

On va en binaire procéder de même mais en multipliant par 2. Ainsi :

Principe	Présentation améliorée
$0{,}375 * 2 = 0{,}750 \rightarrow \begin{cases} Bit\ 0 \\ Nouvelle\ partie\ décimale\ 0{,}75 \end{cases}$ $0{,}750 * 2 = 1{,}5 \rightarrow \begin{cases} Bit\ 1 \\ Nouvelle\ partie\ décimale\ 0{,}5 \end{cases}$ $0{,}50 * 2 = 1 \rightarrow \begin{cases} Bit\ 1 \\ Nouvelle\ partie\ décimale\ 0 \end{cases}$	0,375 | $*2=$ | 0, | 750 0,750 | $*2=$ | 1, | 5 0,50 | $*2=$ | 1, | **0**

C'est terminé :

$$(0{,}375)_{10} = (0{,}011)_2$$

Remarque : on verra dans une remarque dans la suite qu'en passant par une écriture scientifique binaire, ce transcodage est aussi possible. Calculez $0{,}011 = (11)_2.2^{-3}$

A.III.2 Représentation des nombres en machine

Le programme d'IPT limite ce chapitre à la représentation des nombres réels en virgule flottante normalisée sans traiter de cas particuliers. Après avoir vu les limites de la représentation des entiers naturels et une possibilité de représenter des entiers relatifs, nous aborderons donc la représentation associée à la **norme IEEE 754** des nombres à virgule flottante pour représenter les réels en binaire.

A.III.2.a Nombres entiers naturels

Comme nous l'avons vu, un entier naturel codé sur n bits peut avoir une valeur décimale comprise entre 0 et $2^n - 1$.

Les systèmes 32 bits présentent $2^{32} - 1 = 4\,294\,967\,295$ valeurs entières différentes.

Les systèmes 64 bits présentent $2^{64} - 1 = 1{,}8446744007371 * 10^{19}$ valeurs entières différentes.

Comment manipuler des entiers plus grands ? Négatifs ? Ou encore des nombres réels ?

A.III.2.b Nombres entiers relatifs – Codage par excès

Il existe plusieurs manières de coder des entiers relatifs en binaire. Nous n'aborderons ici que le codage par excès car il va nous servir pour le codage des nombres à virgule flottante abordé ensuite.

Le principe consiste à utiliser un entier N entre 0 et $2^n - 1$ et d'associer ses valeurs à des nombres entiers relatifs Z décalés d'un « biais » tel que l'on ait environ la moitié des termes négatifs, et l'autre positifs (il est impossible d'en avoir autant de chaque côté car après avoir enlevé 0, il reste $2^n - 1$ termes à répartir, ce qui est une quantité impaire de chiffres).

Exemple sur 3 bits des deux solutions possibles :

N	0	1	2	3	4	5	6	7
Z	-4	-3	-2	-1	0	1	2	3
Z	-3	-2	-1	0	1	2	3	4

On aura donc environ $\frac{2^n}{2} = 2^{n-1}$ termes négatifs et 2^{n-1} positifs.

Il suffit donc de décaler le nombre entier naturel N en lui soustrayant environ 2^{n-1}. En fait, on soustraira $2^{n-1} - 1$, ce que l'on appelle le « biais » du codage par excès :

$$Z = N - (2^{n-1} - 1)$$

Exemple : Soit un entier naturel N codé sur 8 bits devant coder des entiers relatifs. N permet de coder les entiers naturels de 0 à $2^8 - 1 = 256 - 1 = 255$. En décalant chaque valeur du biais valant $2^{n-1} - 1 = 2^7 - 1 = 127$, on obtient la table d'association suivante :

Entier naturel N	Entier relatif Z associé
0	-127
255	128

Il est ainsi possible de coder sur 8 bits des entiers relatifs associés à des entiers relatifs positifs et négatifs codés sur 7 bits, et d'une manière plus générale sur n bits des entiers relatifs associés à des entiers positifs et négatifs codés sur $n - 1$ bits.

A.III.2.c Nombres à virgule flottante

A.III.2.c.i Principe

Le principe de la norme IEEE 754 qui permet de représenter en binaire des nombres à virgule flottante (on parle de flottants) est basé sur une représentation similaire à notre représentation scientifique.

En effet, en base 10, nous avons appris à représenter les nombres ainsi :

$$\begin{cases} 1200 = 1{,}2.10^3 \\ 0{,}000056 = 5{,}6.10^{-5} \\ -289456{,}5 = -2{,}894565.10^5 \end{cases}$$

Prenons l'exemple de $X = -289456{,}5 = -2{,}894565.10^5$

Pour stocker le moins d'informations possible de ce nombre, on écrit :

$-2{,}894565.10^5$			
$-$	2	894565	5
Signe	Caractéristique	Mantisse	Puissance
	Partie significative		

Remarques :

- Le signe vaut 0 ou 1
- La caractéristique est un chiffre de 1 à 9, elle ne peut pas être nulle car dans ce cas, la notation scientifique n'est pas respectée : $\mathbf{0},022 = \mathbf{0},22.10^{-1} = 2{,}2.10^{-2}$

En binaire, on va utiliser le même principe mais avec des puissances de 2 au lieu de puissance de 10.

A.III.2.c.ii Ecriture « scientifique binaire »

Nous admettrons que l'on peut transformer un nombre binaire sous forme scientifique, par exemple :

$$(1100)_2 = (1{,}1)_2 * 2^3$$

Prenons deux exemples pour montrer que cela fonctionne :

$(1000)_2 = 8$	$(1100)_2 = 12$
$(1000)_2 * 2^0 = 8 * 1 = 8$ $(100{,}0)_2 * 2^1 = 4 * 2 = 8$ $(10{,}00)_2 * 2^2 = 2 * 4 = 8$ $(1{,}000)_2 * 2^3 = 1 * 8 = 8$	$(1100)_2 * 2^0 = 12$ $(110)_2 * 2^1 = 6 * 2 = 12$ $(11)_2 * 2^2 = 3 * 4 = 12$ $(1{,}1)_2 * 2^3 = 1{,}5 * 8 = 12$
$(1000)_2 = 1.2^3$	$(1100)_2 = (1{,}1)_2 * 2^3$

Rappelons que : $(1{,}1)_2 = 1.2^0 + 1.2^{-1} = 1 + \frac{1}{2} = 1{,}5$

Remarque : Un pourra remarquer que la transcription de binaire à virgule à scientifique binaire peut se faire en passant par un entier multiplié par une puissance de 2. Dans l'exemple du paragraphe précédent : $(0{,}011)_2 = (11)_2.2^{-3} = 3.2^{-3} = (0{,}375)_{10}$

A.III.2.c.iii Ecriture binaire du codage à virgule flottante

Reprenons l'exemple précédent :

$-2{,}894565.10^5$			
$+$	2	894565	5
Signe	Caractéristique	Mantisse	Puissance
	Partie significative		

Voyons comment stocker ce nombre sous forme scientifique binaire :

$$(-2{,}894565.10^5)_{10} = (-289456{,}5)_{10} = (-289456 - 0{,}5)_{10}$$
$$= -(100011010101010000)_2 - (0{,}1)_2 = -(100011010101010000{,}1)_2$$

On écrit donc ce nombre binaire sous forme scientifique binaire :

$$-(100011010101010000{,}1)_2 = -(1{,}00011010101010100001)_2.2^{18}$$

Pour stocker ce nombre, on a besoin des informations suivantes :

$-(1{,}00011010101010100001)_2.2^{18}$			
$-$	1	00011010101010100001	18
Signe	Caractéristique	Mantisse	Puissance
	Partie significative		

Comme précédemment, la caractéristique ne peut être nulle. En base 10, elle pouvait valoir un chiffre de 1 à 9. Maintenant, elle ne peut plus être différente de 1. Chouette ! Plus besoin de la stocker, on gagne un bit.

Finalement, il suffit de stocker les informations suivantes :

$-$	$00011010101011000 01$	18
Signe	Mantisse	Puissance

Toutefois, ces informations ne peuvent être stockées sous forme binaire, avec des 0 et des 1.

Les choix suivants sont effectués :

- Le signe est stocké sur un bit : $\begin{cases} s = 0 \Leftrightarrow x > 0 \\ s = 1 \Leftrightarrow x < 0 \end{cases}$
- La mantisse est déjà sous forme binaire
- La puissance est un nombre qui peut être positif ou négatif. Le choix est fait de le coder en binaire par excès, c'est-à-dire en décalant la valeur décimale en une valeur entière par l'opération vue précédemment $Z = N - (2^{n-1} - 1)$ avec n le nombre de bits sur lequel la puissance va être codée.

Pour coder la puissance, nous devons donc maintenant choisir un format de stockage. Nous les aborderons plus tard, pour le moment admettons qu'en 32 bits, on réserve 8 bits pour la puissance, soit un biais de 127.

La puissance 18 se transforme en un entier naturel $18 + 127 = 145$ qui en binaire s'écrit :

$$(145)_{10} = (10010001)_2$$

Les informations binaires à stocker sont donc les suivantes :

1	$00011010101011000 01$	10010001
Signe	Mantisse	Puissance

Le choix est fait de les représenter dans le sens suivant :

1	10010001	$00011010101011000 01$
Signe	Puissance	Mantisse

Dernier détail, en 32 bits, 8 bits sont alloués à la puissance, 1 bit au signe, le reste à la mantisse. Il en reste donc 23. Il faut donc compléter la mantisse de 0 à droite pour avoir 23 bits (a droite, car la mantisse est la partie décimale, donc par exemple $0{,}1 = 0{,}10000$).

Finalement, en 32 bits, on aura une représentation de $-289456{,}5$ par :

Représentation sur 32 bits	1	10010001	00011010101011000010000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)

Ne jamais oublier le bit implicite non décrit dans ce format.

A.III.2.c.iv Formats de la norme

La norme propose différents formats de stockage de flottants, voyons les trois principaux :

	Nombre de bits	Bit signe	Bits exposant	Bits implicite	Bits mantisse	Décalage
Simple précision	32	1	8	1	23	127
Double précision	64	1	11	1	52	1023
Quadruple précision	128	1	15	1	112	16383

Dans les logiciels de programmation, on peut déclarer la manière avec laquelle on veut stocker des nombres, par exemple en écrivant « $float\ a$ » ou « $double\ a$ ».

A.III.2.c.v Exemples de transcodage

- **Réel base 10 – Binaire en virgule flottante**

Nombre entier :

Soit le nombre $x = (2100)_{10}$ que l'on souhaite exprimer en simple précision. Traduisons ce nombre en binaire et mettons le sous forme scientifique binaire :

$$(2100)_{10} = (100000110100)_2 = (1{,}000001101)_2.2^{11} = 1.025390625 * 2^{11}$$

La mantisse vaut donc 000001101 qu'il faut compléter de 0 « à droite » pour avoir 23 bits :

$$00000110100000000000000$$

La puissance 11 est le résultat d'un codage par excès, le nombre associé est donc $11 + 127 = 138 = (10001010)_2$. La puissance vaut donc 10001010 et elle a déjà une taille de 8 bits dans ce cas.

Finalement, on a x codé en binaire simple précision :

	$(2100)_{10}$		
Simple précision 32 bits	0	10001010	00000110100000000000000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)
	01000101000000110100000000000000		

Nombre à virgule :

Soit le nombre $x = (-10{,}125)_{10}$ que l'on souhaite exprimer en simple précision. Procédons comme précédemment : $(-10{,}125)_{10} = (-1010{,}001)_2 = (-1{,}010001)_2.2^3$

$$3 + 127 = 130 = (10000010)_2$$

	$(10{,}125)_{10}$		
Simple précision 32 bits	1	10000010	01000100000000000000000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)
	11000001001000100000000000000000		

- **<u>Binaire en virgule flottante – Réel base 10</u>**

Nombre entier :

Soit le mot binaire 32 bits suivant : $01001101100011101111001111000010$

Simple précision 32 bits	Identification des 3 parties		
	0	10011011	00011101111001111000010
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)

Alors :

- Le signe est positif
- Le nombre n associé à l'exposant vaut $n = 1.2^7 + 0.2^6 + 0.2^5 + 1.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = 155$. Avec le décalage de 127, on obtient l'exposant : $e = 155 - 127 = 28$
- Sans oublier d'ajouter le bit implicite, la partie significative vaut : $(\mathbf{1}, 00011101111001111000010)_2 = 1.1168138980865479$

Finalement, le nombre décimal associé vaut : $x = 1.1168138980865479.2^{28} = 299792448$

Nombre à virgule :

Soit le mot binaire 32 bits suivant : $11000001001001110000000000000000$

Simple précision 32 bits	Identification des 3 parties		
	1	10000010	01001110000000000000000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)

- Le signe est négatif
- Le nombre n associé à l'exposant vaut $n = 130$. Avec le décalage de 127, on obtient l'exposant : $e = 130 - 127 = 3$
- Sans oublier d'ajouter le bit implicite, la partie significative vaut : $(\mathbf{1}, 0100111)_2 = 1{,}3046875$

Finalement, le nombre décimal associé vaut : $x = -1{,}3046875.2^3 = -10.4375$

Remarques suite à ces exemples :

- dans le cas où la mantisse a une taille inférieure à la valeur de la puissance, on peut se ramener à un nombre binaire sans virgule, et la conversion se fait directement d'un binaire à un entier : $(\mathbf{1}, 00011101111001111000010010 1)_2 * 2^{28} = (\mathbf{1}0001110111100111100001001010)_2 = (299792458)_{10}$. On pourra toutefois dans tous les cas utiliser une procédure de conversion d'un binaire à virgule multiplié par une puissance de 2.
- Pour que le résultat en base 10 soit un nombre à virgule, il est nécessaire que la mentisse soit de longueur plus grande que la valeur de l'exposant

A.III.2.c.vi Quelques nombres réservés

Sans rentrer dans tous les détails, certains nombres de la représentation en virgule flottante sont réservés, par exemple en simple précision :

- L'exposant maximum et une mantisse remplie de 0 est réservé au nombre ∞ :

$$\pm\infty \Leftrightarrow \begin{matrix}0\\1\end{matrix} \quad 11111111 \quad 00000000000000000000000$$

- L'exposant maximum et une mantisse non nulle est réservé au nombre NaN dire « Not a Number ». C'est par exemple le résultat d'une division par 0 ou une racine d'un nombre négatif…

$$NaN \Leftrightarrow \begin{matrix}0\\1\end{matrix} \quad 11111111 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

A.III.2.c.vii Notion de représentation normalisée et dénormalisée

Il existe des représentation dites « normalisées » respectant ce que nous venons de voir, et des nombres en notation « dénormalisée ». Lorsqu'un nombre a un exposant binaire nul (plus petite puissance), au lieu d'écrire

$$Valeur = signe * 1,mantisse * 2^{-décallage}$$

On écrit

$$Valeur = signe * mantisse * 2^{-décallage+1}$$

En 32 bits, cela donne : $Valeur = signe * mantisse * 2^{-126}$.

Cela permet de représenter des nombres encore plus petits et d'assurer une certaine continuité de valeurs entre nombre normalisés et dénormalisés, mais……..

Vous n'avez pas besoin de retenir cela, mais cela vous permettra de comprendre les deux valeurs minimales proposées dans le tableau un peu plus bas.

Limites de la représentation en virgule flottante

• Justesse

La justesse (correspondance entre nombre théorique et nombre réel) d'un nombre est associée au nombre de chiffres significatifs qui le décrivent, c'est-à-dire au nombre de bits codant la mantisse auquel on ajoute le bit implicite. La justesse est parfaite si l'on dispose d'un nombre infini de bits ou si, par chance, le nombre représenté possède un nombre de bits de mantisse égale au nombre de bits stockables dans le format choisi.

La quantité de nombre différents représentables est :

32 bits	4 294 967 296
64 bits	18 446 744 073 710 000 000

On représente donc autant de nombre avec la notation en virgule flottante, mais ils ne sont plus limités aux entiers et peuvent dépasser le nombre maximum de $2^n - 1$!!!

• Nombre minimum et maximum

Pour une simple précision :

- La plus grande puissance codée par excès sur 8 bits vaut 255. Elle vaut donc en réalité $255 - (2^7 - 1) = 128$. Toutefois, cette puissance est réservée au nombre NaN. C'est donc au maximum 127. La plus grande partie significative vaut :
$$(1{,}11111111111111111111111)_2 = (1{,}99999988079071)_{10}$$
Le nombre le plus grand vaut donc :
$$1{,}9999998807907104.\, 2^{127} = 3{,}4028234663852886.\, 10^{38}$$
- Le nombre le plus petit en notation normalisée a la puissance la plus faible : -126, et la partie significative :
$$(1{,}00000000000000000000000)_2 = (1)_{10}$$
En notation normalisée, le nombre le plus petit vaut donc :
$$2^{-126} = 1{,}17549435082229.\, 10^{-38}$$
- En revanche, en notation dénormalisée, le plus petit nombre est :
$$(0{,}00000000000000000000001)_2 * 2^{-127+1} = (1)_2 * 2^{-23} * 2^{-126} = 2^{-149}$$

Valeurs normalisées	Valeur min normalisée	Valeur min dénormalisée	Valeur max
Simple précision	$1{,}2.\, 10^{-38}$	$1{,}4.\, 10^{-45}$	$3{,}4.\, 10^{38}$
Double précision	$2{,}2.\, 10^{-308}$	$4{,}9.\, 10^{-324}$	$1{,}8.\, 10^{308}$

Sur un système d'exploitation 32 bits, ouvrez Excel par exemple, et tapez dans une case le nombre : $3{,}5.\, 10^{38}$, vous verrez alors : 

. Par contre, entrez $3{,}4.\, 10^{38}$, vous verrez 

De même sur un système d'exploitation 64 bits, essayez $1{,}8.\, 10^{308}$ et $1{,}7.\, 10^{308}$.

Vous savez maintenant quelle valeur maximale vous êtes capable de traiter avec votre outil informatique.

- **<u>Précision</u>**

Définissons la précision comme l'écart entre deux valeurs représentables en virgule flottante.

Prenons l'exemple des deux valeurs les plus grandes représentables (pour les valeurs minimales, intervient la notion de dénormalisation que je préfère éviter).

Ainsi, les deux plus grands nombres sont :

$$A = (1{,}11111111111111111111111)_2 * 2^{127} = 3{,}4028234663852886.10^{38}$$

$$B = (1{,}11111111111111111111110)_2 * 2^{127} = 3{,}40282326356119.10^{38}$$

L'écart entre A et B vaut :

$$A - B = -2{,}02824096036517.10^{31}$$

Il n'existe pas de nombres en simple précision entre A et B, et ils sont… relativement éloignés.

A.III.3 Conséquences de la représentation des nombres en virgule flottante

A.III.3.a Dépassement de capacité - Overflow

Le dépassement de capacité est atteint lorsqu'un nombre dépasse la valeur limite représentable dans le système choisi.

Un entier naturel binaire codé sur un octet (8 bits) ne peut dépasser 255. Le calcul 255+1 consistant à calculer $11111111 + 00000001$ donne normalement le résultat 100000000 sur 9 bits. La machine retiendra les 8 derniers bits, soit 00000000 et donnera un résultat nul.

Comme nous l'avons vu avec Excel pour un système 64 bits, le dépassement de capacité consistant à écrire le nombre $1{,}8.10^{308}$ renvoie une erreur car ici aussi, il y a dépassement de capacité.

Ariane 5 s'est crashée à cause d'un dépassement de capacité…

A.III.3.b Erreurs d'arrondis

A.III.3.b.i Exemple

Souvenez-vous, au début de ce paragraphe, nous avons abordé l'exemple suivant :

```
>>> 0.2+0.1
0.30000000000000004
```

Sans rentrer dans les détails, les mantisses des nombres utilisés sont de dimension 52 bits (double précision). Les additions sur les nombre en virgule flottante consistent à faire une addition de mantisses après les avoir mises au même exposant. La mise au même exposant consiste à effectuer des décalages de mantisse, ce qui conduit à perdre les bits décalés vers la droite au-delà du 52°, ce qui est à l'origine d'erreurs d'arrondis.

A.III.3.b.ii Conséquence importante – Test « == »

Exécutez ce programme et vous aurez tout compris :

```
a = 0.2
b = 0.1
c = a + b

print('c = ',a,' + ',b,' = ', c)
if c==0.3:
    print('OUI: c==0.3')
else:
    print('NON: c==0.3')

if abs(c-0.3)<=10**(-10):
    print('OUI: (c-0.3)<=10**(-
n)')
else:
    print('NON: (c-0.3)<=10**(-
n)')
```

```
>>> (executing file "<tmp 1>")
c =  0.2  +  0.1  =  0.30000000000000004
NON: c==0.3
OUI: (c-0.3)<=10**(-n)
```

Lorsque l'on travaille avec des flottants, il ne faut pas effectuer de tests d'égalité mais comparer des différences à un nombre très petit. Un exemple à votre portée est de réaliser un programme qui calcul les solutions d'une équation du second degré avec discriminent nul. Le test « $Disc == 0$ » ne fonctionnera jamais à tous les coups…

Exemple souvent rencontré : On veut tester si un nombre est entier :

BIEN	PAS BIEN

`a = 0.1`
`b = 0.2`
`c = 0.3`
 	
`Test = (a+b)/c`
 	
`if abs(Test - int(Test)) < 1e-10:`
`  print("Cool")`
`else:`
`  print("Pas cool")`
 	
`>>> (executing file "<tmp 1>")`
`Cool`
	`a = 0.1`
	`b = 0.2`
	`c = 0.3`
	
	`Test = (a+b)/c`
	
	`if Test == int(Test):`
	`  print("Cool")`
	`else:`
	`  print("Pas cool")`
	
	`>>> (executing file "<tmp 1>")`
	`Pas cool`

L'idée d'utiliser le test « `type(Test) == int` » n'est pas mauvaise mais ne fonctionne pas à tous les coups. En effet, pour que la variable Test soit un entier, il faut qu'elle soit issue d'entiers. Or, ici, si le calcul fonctionnait, le résultat serait 1.0, c'est-à-dire une valeur entière stockée sous forme de flottant. Il faut donc faire attention à ce que l'on manipule.

A.IV. Les bases de la programmation

A.IV.1 Les commentaires – Texte personnel

Il est possible et même très recommandé de commenter ses programmes. Cela consiste à ajouter toutes les précisions utiles à la compréhension du code, sans que ces « phrases » ne soient regardées par Python.

C'est très simple, il suffit avant d'écrire un texte de commentaires, d'écrire « # »

Exemple :

```
1  # Ce programme est un essai réalisé pour le cours d'IPT
2
3  print(10+10)
4
```

Autre solution, très utile pour commenter toute une partie de programme : Sélectionner le texte à mettre en commentaire puis faire « Edit » puis « Comment ». Il est ensuite possible de « décommenter » avec « Uncomment ».

Cela peut être très utile lorsqu'un gros programme bogue. On commente une zone qui ne nous intéresse pas afin d'aller exécuter la seule partie qui pose problème.

On peut créer des zones bien visibles en écrivant « ## » :

```
1 ## Exercice 1
2
3 ## Exercice 2
4
```

Il est enfin possible de créer une zone de commentaires sur plusieurs lignes en utilisant « ''' » et de la terminer en écrivant à nouveau « ''' » :

```
4 '''
5 Ceci est une zone de commentaires
6 On peut y inscrire ce que l'on veut sur plusieurs lignes
7 '''
```

A.IV.2 Les variables

Taper 10+10, c'est bien, mais on le fera très rarement dans un programme. Nous allons manipuler des variables dans lesquelles sont stockées des valeurs.

A.IV.2.a Noms des variables

Une variable est un « mot », c'est-à-dire une chaîne de caractères qui doit commencer par une lettre, sans espaces, en évitant les caractères spéciaux à par underscore « _ », on évitera les accents, à laquelle on associe une valeur, un mot entre guillemets etc. Elle est définie en inscrivant le nom de celle-ci, puis le signe « = » et enfin ce qu'elle doit contenir. Lorsqu'elle est définie, il suffit de taper son nom pour obtenir sa valeur :

Programme	Console lors de l'appel des variables après exécution du programme

`a = 1`
`Nom = "Denis"`
`Variable_1 = 10`
	`>>> Nom`
	`'Denis'`
	
	`>>> a`
	`1`
	
	`>>> Variable_1`
	`10`

Il est fortement recommandé de choisir des noms correspondant à ce que contient la variable pour plusieurs raisons :

- De base, vous serez évaluées en partie à l'écrit, vos programmes doivent être compréhensibles
- D'une manière générale, vos codes doivent être lisibles pour d'autres personnes, il faut donc qu'ils soient facilement compréhensibles
- Lorsque vous travaillerez sur de gros programmes, vous oublierez vite ce que vous avez fait quelques semaines avant et si vous ne respectez pas ce conseil, vous pourriez avoir besoin de repasser beaucoup de temps à comprendre ce que vous avez fait

Il faudra donc par exemple, préférer :

- « Nom_Eleve » plutôt que « n »
- « Coeff_Conductivité » plutôt que « mu »
- « Nb_Eleves » plutôt que « n »

Cela donnera en plus la possibilité d'utiliser la variable « n » dans un autre contexte car vous n'auriez pas pu l'utiliser 2 fois…

Attention, **majuscules et minuscules sont différents** dans python. Si la variable a existe, A n'existe pas et peut avoir une valeur différente.

Lorsqu'une variable existe et qu'on la redéfinie, on écrase sa valeur précédente.

Vous pourrez remarque que dès qu'une variable existe, Pyzo permet en tapant ses quelques premières lettres de l'écrire entièrement, c'est assez pratique et ça évite souvent une faute de frappe pour les longs noms.

Attention, danger ! **Il ne faut surtout pas utiliser de nom de fonctions python** comme noms de variables, prenons les exemples de input (fonction abordée un peu plus loin) et print

Il peut vous arriver d'écrire :

```
a = input = ("Valeur de n")
```
au lieu de
```
a = input("Valeur de n ? ")
```

Ou encore :

```
print = 2
```
au lieu de
```
print(2)
```

Ce sont de grosses erreurs qui ont des conséquences importantes. Supposons avoir écrit ces deux erreurs et avoir exécuté le code en question, puis appelons les fonctions input et print :

`print(2)`	`Traceback (most recent call last):`
	`  File "<console>", line 1, in <module>`
	`TypeError: 'int' object is not callable`

`b = input("Valeur de i ? ")`	`Traceback (most recent call last):`
	`  File "<console>", line 1, in <module>`
	`TypeError: 'str' object is not callable`

Il faut savoir qu'écrire **mot()** consiste à appeler une fonction mot (nous aborderons les fonctions plus tard) en précisant qu'elle n'a pas d'argument (parenthèse vide). Ecrire **mot(2)** appelle la fonction mot avec comme argument la valeur 2. Or, en écrivant les lignes d'erreurs print=2 et input=2, on a défini des variables de type int dont les noms sont print et input. Pyzon nous informe dont que ces variables de type int ne sont pas appelables (callable) car ce ne sont pas des fonctions, et c'est logique.

La solution consiste à supprimer ces variables qui ont le nom de fonctions dans python avec la commande del :

```
del input
del print
```

Ce qui supprime les variables créées par erreur avec des noms de fonctions existantes dans python pour pouvoir à nouveau les utiliser.

Attention donc à ne pas appeler des variables avec des noms de fonctions et à savoir utiliser del si vous l'avez fait par erreur

A.IV.2.b Variables inexistantes

Il faut bien retenir que l'ordinateur ne connaît que ce qu'on définit.

Pour le moment, nous avons créé les variables a, b, c et quelques autres avec des noms plus grands.

Tapez z puis pressez Entrée dans la console. Un message d'erreur apparaît :

```
>>> z
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'z' is not defined
```

C'est normal, la variable z n'étant pas définie, Python nous le dit : « name 'z' is not defined ».

A.IV.2.c Création de variables classiques

A.IV.2.c.i Variables de type entiers et décimaux

Les deux types de variables qui nous intéresseront ici sont les types « integer » pour « entier » et « floating » pour décimal. Les types de variables se créent automatiquement en fonction du nombre entré.

Dans la console, tapez « $a = 10$ » puis validez, ensuite tapez « type(a) » et valider.

```
>>> a=10

>>> type(a)
<class 'int'>
```

« a » est une variable de type « int » pour « integer » ou entier.

Tapez maintenant « b=10.0 » et validez, puis « type(b) et regardez le résultat :

```
>>> b=10.0

>>> type(b)
<class 'float'>
```

Le nombre vaut la même chose, mais c'est maintenant un « float » ou décimal.

On obtient un arrondi de x à n chiffres après la virgule en écrivant : $round(x, n)$

Une opération entre un entier et un décimal donnera toujours un nombre décimal. Tapez dans la console $c = a + b$ puis $type(c)$, vous verrez :

```
>>> c=a+b

>>> type(c)
<class 'float'>
```

Il est possible de transformer/convertir un entier en décimal et inversement, un décimal en entier, soit prendre sa partie entière.

Essayez $float(10)$, il donnera 10.0 - Essayez $int(10.2)$, il donnera 10

A.IV.2.c.ii Variable de type chaînes de caractères

Une chaîne de caractères est un mot composé de lettres. Il faut bien faire la différence entre une variable z et la lettre z, python ne les traite pas de la même manière.

Dans la console, si z n'est pas une variable existante (ie n'est pas visible dans le Workspace), écrivez z et tapez « Entrée », un message d'erreur dit :

```
>>> z
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'z' is not defined
```

Effectivement, la variable z n'a pas été créée, et ne contient aucune donnée « à l'intérieur ».

Tapez maintenant, au choix :

''z'' (guillemets) ou 'z' (apostrophes)

Python renvoie dans les 2 cas la lettre z entre apostrophes.

Définissons la lettre z dans la variable d : tapez $d = 'z'$ puis tapez $type(d)$, vous verrez que d est un « string » ou chaîne de caractères. La variable d contient la lettre z.

Attention, lettres majuscules et minuscules ne sont pas identiques pour Python.

Il est possible de regrouper deux chaînes de caractère avec le symbole « + », par exemple :

```
>>> "Classe_1 " + "Eleve 1"
'Classe_1 Eleve 1'
```

Attention, lorsque l'on utilise deux apostrophes pour créer une chaîne de caractères, il est évidemment impossible d'utiliser ce même apostrophe dans la chaîne de caractère puisque celle-ci mettra fin à ladite chaîne :

```
>>> 'Nous l'avons utilisée'
  File "<console>", line 1
    'Nous l'avons utilisée'
           ^
SyntaxError: invalid syntax
```

Dans ce cas, deux solutions :

- Utiliser des guillemets pour encadrer la chaîne de caractères :

```
>>> "Nous l'avons utilisée"
"Nous l'avons utilisée"
```

- Utiliser l'anti slash (touches Alt+8) devant l'apostrophe qui doit rester du texte et non une délimitation :

```
>>> 'Nous l\'avons utilisée'
"Nous l'avons utilisée"
```

Il est possible de récupérer une lettre en fonction de sa position dans une chaîne de caractères. Pour cela, il faut dans un premier temps définir une variable contenant cette chaîne de caractères puis de demander la lettre en question à l'aide de crochets en précisant la place, attention, en partant de 0 pour la première lettre :

```
>>> Nom = 'Denis'

>>> Nom[2]
'n'
```

Si on demande une lettre au-delà de la taille du mot enregistré dans la variable, ici Nom, l'erreur suivante apparaît :

```
>>> Nom[6]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: string index out of range
```

Pyzo nous prévient que l'index 6 est hors de portée (out of range).

Il est possible de créer une chaîne de caractères contenant à la fois des mots et la valeur d'une variable. Dans ce cas, il est obligatoire de dire à Python qu'il faut considérer la valeur de la variable comme une chaîne de caractères (équivalent à un problème d'homogénéité) à l'aide de la commande str afin de créer une chaîne de caractères comme somme de chaînes de caractères, qui sera affichée à l'aide de la fonction print.

Soit a une variable qui vaut 10, pour créer la phrase « Il y a 10 pommes » telle que si a change, le 10 change, il faut écrire :

```
>>> "Il y a " + str(a) + " pommes"
'Il y a 10 pommes'
```

La commande $str(a)$ permet de transformer la valeur de a en un texte qui sera ajouté au reste du texte.

Erreur à ne pas commettre :

```
>>> "Il y a " + "a" + " pommes"
'Il y a a pommes'
```

Il est possible de créer un texte « à trous » dans lequel seront ajoutées des valeurs précisées à la fin :

```
>>> print("Mon nom est %s et j'ai %s ans" % ("Denis",30))
Mon nom est Denis et j'ai 30 ans
```

La structure est la suivante :

- Texte entre guillemets et %s pour chaque trou
- Ajout d'un % après le texte
- Parenthèse avec chaque élément à intégrer dans les trous, dans l'ordre d'apparition
- Attention : pour afficher le symbole % dans le texte, il faut le doubler en écrivant %%

Remarque : Multiplier une chaîne de caractères par un entier la reproduit :

```
>>> 2 * 'Cou'
'CouCou'
```

A.IV.2.c.iii Variables de type Booléens

Nous aborderons plus précisément les variables booléennes dans la suite du cours, dans l'algorithmique et les conditions.

Nous pouvons toutefois définir dès maintenant deux variables booléennes :

- « True » - Equivalent à « Juste », « Oui », « 1 »
- « False » - Equivalent à « Faux », « Non », « 0 »

Une variable booléenne peut être créée directement, par exemple « a = True », ou être le résultat d'un test, par exemple :

```
>>> a = 2==2

>>> a
True
```

A.IV.2.c.iv La variable « Rien »

Lorsque l'exécution d'un code ne renvoie rien, en réalité, elle renvoie la variable « None ».

Lorsque l'on tape par exemple :

```
>>> a = 2

>>> 2+2
4
```

L'exécution 2+2 renvoie 4 alors que la commande a = 2 ne renvoie rien. En fait, cela renvoie None.

```
>>> None

>>>
```

Il sera donc possible, plus tard, de renvoyer None si un code ne doit rien renvoyer, ou de tester le résultat d'une commande en vérifiant s'il vaut None ou pas.

A.IV.2.c.v Vérification du type d'une variable

On peut vérifier qu'une variable est d'un type donné, par exemple :

```
>>> a = 1

>>> b = 1.2

>>> c = ""

>>> type(a) == int
True

>>> type(b) == int
False

>>> type(b) == float
True

>>> type(c) == str
True

>>> type(c) == str
```

A.IV.2.d Création d'une variable via demande à l'utilisateur

Il est possible de faire en sorte qu'à un moment soit demandé à l'utilisateur d'entrer une variable par lui-même au cours d'un programme, pour cela il suffit d'utiliser la fonction input :

Programme	Console après exécution
`Nb_Eleves = input("Entrer le nombre d'élèves: ")`	`>>> (executing lines 1 to 3 of "Essai_1.py ")` `Entrer le nombre d'élèves:`

Attention, quelle que soit la donnée entrée, le résultat d'un input est une chaîne de caractères (string).

Il suffit alors, dans la console, d'entrer le nombre associé puis validez avec la touche Entrée.

```
Entrer le nombre d'élèves: 10
```

Et enfin, vérifiez ce que contient la variable :

```
>>> Nb_Eleves
'10'
```

ATTENTION : la variable Nb_Eleves est une chaîne de caractères, on le voit à la présence des guillemets. Pour pouvoir l'utiliser dans des calculs, il faut la convertir en flottant par exemple, il faut donc ajouter obligatoirement :

```
>>> Nb_Eleves
'10'

>>> Nb_Eleves = float(Nb_Eleves)

>>> Nb_Eleves
10.0
```

Les erreurs classiquement rencontrées avec le « input » sont associées à l'oublie de transformation de la chaîne de caractères en nombre :

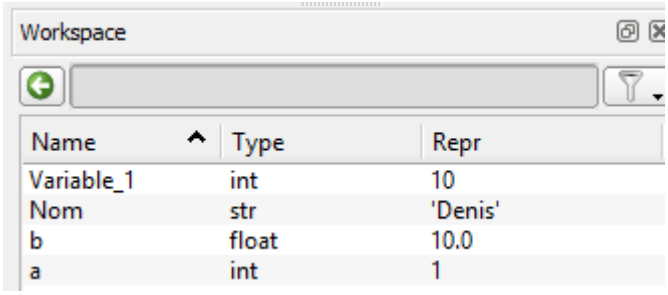
```
>>> a = input("Quel âge as tu ? ")
b = 2 + a
Quel âge as tu ? 10
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> a = input("Quel âge as tu ?")
b = 2 * a
print(b)
Quel âge as tu ?32
3232
```

Remarques :

- lors de l'utilisation d'un input, ajoutez « : » ou « ? » à la fin du message à l'utilisateur afin de voir clairement que l'ordinateur attend une entrée. Une erreur classique est de ne pas se rendre compte que l'ordinateur attend, et de lancer plusieurs fois le programme… Il faudra alors remplir autant de fois la console avec une valeur ! Ou « casser » le code avec les touches « Ctrl + i » autant de fois.
- La commande Input sera utilisée avec parcimonie, on lui préfèrera la création directe de variables dans le fichier de travail.

A.IV.2.e Visualiser les variables créées

Allez dans « Tools » puis cochez « Workspace ». Regardez en bas à droite, une nouvelle fenêtre « Workspace » est apparue et on y voit les variables existantes (noms, types, valeur).



A.IV.2.f Echanger deux variables

Sous Python, on a deux possibilités pour échanger deux variables :

```
bb = a
a = b
b = bb
```

```
a,b = b,a
```

Attention, si la solution de gauche fonctionne quel que soit le programme, celle de droite fonctionne sous Python. Quid des autres …

A.IV.2.g Effacer des variables

Rien de plus simple. Effaçons la variable d créée précédemment et visible dans l'explorateur de variables. Il suffit d'écrire « del a » et de valider pour la voir disparaître.

A.IV.3 Les messages à l'utilisateur

Il est très simple d'afficher un message dans la console, il suffit d'utiliser la fonction $print()$. Il faut alors mettre le message entre guillemets.

Par exemple :

Programme	Console après exécution
`print("Bonjour les élèves")`	`>>> (executing lines 1 to 3 of "Essai_1.py ")` `Bonjour les élèves`

On peut demander au programme d'afficher la valeur d'une variable, par exemple :

Programme	Console après exécution
`a = 10` `print(a)`	`>>> (executing lines 1 to 5 of "Essai_1.py")` `10`

Enfin, il est possible de combiner les deux, soit en créant une variable concaténée (association de string avec +), ou en utilisant des virgules entre chaque type de caractère :

Programme	Console après exécution
`a = 10` `print("Il y a " + str(a) + " pommes")`	`>>> (executing lines 1 to 5 of "Essai_1.py")` `Il y a 10 pommes`
`a = 10` `print("Il y a ",a," pommes")`	`>>> (executing lines 1 to 2 of "<tmp 1>")` `Il y a  10  pommes`

A.IV.4 Mathématiques et imports de librairies

Pour effectuer des additions, soustractions, multiplications, et divisions, il suffit d'utiliser les symboles + - * / disponibles au niveau du clavier numérique.

Il est bon de savoir calculer une puissance :

Pour calculer 2^3, il faut écrire 2**3 :

```
>>> 2**3
8
```

Pour toutes les autres fonctions mathématiques (cos, log, exp,racine (sqrt)…), nous verrons qu'il est nécessaire d'importer des librairies car python ne sait pas, à priori, ce que veut dire cos par exemple.

Il est très utile de connaître la commande du « modulo », qui s'écrit $x[n]$ en mathématiques, et $x\%n$ sous Python. Il donne le reste dans la division euclidienne de x par n. Plus généralement, on a :

$$\begin{cases} a = bq + r \\ q = a//b \\ r = a\%b \end{cases}$$

La valeur absolue s'obtient avec la commande $abs(x)$

Pour utiliser des constantes mathématiques ou des fonctions mathématiques, il faut les importer depuis la librairie « Maths ». Rien de plus simple, il faut écrire « from math import _____ » et préciser ce que l'on veut importer :

```
>>> pi
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'pi' is not defined

>>> from math import pi

>>> pi
3.141592653589793


>>> cos(pi)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'cos' is not defined

>>> from math import cos

>>> cos(pi)
-1.0
```

Remarque : il est possible d'importer toutes les fonctions de « math », en inscrivant :

```
>>> from math import *
```

Toutefois, on se retrouve avec un Workspace plein :

Variable_1	int	10
trunc	builtin_function...	<built-in functio...
tanh	builtin_function...	<built-in functio...
tan	builtin_function...	<built-in functio...
sqrt	builtin_function...	<built-in functio...
sinh	builtin_function...	<built-in functio...
sin	builtin_function...	<built-in functio...

On peut alors utiliser toutes les variables et fonctions simplement en écrivant pi, cos, sqrt…

On peut aussi renommer une fonction, par exemple :

```
>>> from math import sqrt as racine
```

Enfin, on peut importer math entièrement de la manière suivante : import math. Dans ce cas, il faut utiliser les variables en écrivant math.pi… ou les fonctions en écrivant math.cos, math.sqrt …

Résumé :

```
import math
from math import *
from math import sqrt
from math import sqrt as racine
```

A.IV.5 Listes

A.IV.5.a Description

Une liste est un ensemble d'objets mis les uns à côté des autres dans un ordre précis. Elle s'écrit à l'aide de crochets ouverts et fermés encadrant différentes données séparées par des virgules.

A.IV.5.b Créer une liste

Appelons dans la suite L la liste créée.

A.IV.5.b.i Liste vide

Créer une liste vide permet ensuite de lui ajouter des termes, lors d'itérations dans des boucles par exemple. Pour la créer, il suffit d'écrire :

```
L = []
```

A.IV.5.b.ii Liste d'objets divers

Il est possible de créer une liste avec des objets de types différents, prenons l'exemple suivant :

```
L = ['essai',1,True,None,2,'fin']
```

A.IV.5.b.iii Liste d'objets répétés

Pour créer une liste contenant n fois le même objet, il suffit d'écrire :

```
>>> L = 2*['a']

>>> L
['a', 'a']

>>> L = 10*[0]

>>> L
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> L = 2*['a',1]

>>> L
['a', 1, 'a', 1]
```

A.IV.5.b.iv Listes de lettres

Pour créer une liste de lettres, il suffit d'écrire :

$$L = ["a","b","c","d","e","f","g","h"]$$

Toutefois, il est possible d'écrire :

$$L = "abcdefgh"$$

Ces deux objets sont différents :

```
>>> L1 = ["a","b","c","d","e","f","g","h"]

>>> L2 = "abcdefg"

>>> type(L1)
<class 'list'>

>>> type(L2)
<class 'str'>
```

Toutefois, nous verrons que l'on peut utiliser L2 comme L1 en accédant à ses différents termes.

A.IV.5.b.v Ajouter / retirer un élément

Lorsqu'une liste existe, il est possible d'ajouter ou enlever un élément très facilement avec les fonctions pop et append :

```
>>> L
[1, 2, 3]

>>> L.append(10)

>>> L
[1, 2, 3, 10]

>>> L.pop()
10

>>> L
[1, 2, 3]

>>> L.pop()
3

>>> L
[1, 2]
```

$L.append(x)$ ajoute l'élément x dans la liste

$L.pop()$ retire le dernier élément (à droite) et retourne sa valeur, qui peut par exemple alors être utilisée comme nouvelle variable :

```
>>> L
[1, 2]

>>> x = L.pop()

>>> x
2

>>> L
[1]
```

Attention L.pop() renvoie un message d'erreur si la liste est vide puisqu'il n'y a plus d'objets à enlever.

Il est possible d'enlever un élément en début de liste en écrivant $L.pop(0)$

En réalité, on peut enlever n'importe quel objet en écrivant $L.pop(i)$ où i décrit sa position en partant de 0 pour le premier élément.

ATTENTION : écrire L.append(Objet) modifie L en mémoire, mais ne renvoie rien dans la console : « on va à la ligne ». Exécuter L.append(Objet) renvoie en réalité un objet appelé « NONE ». Lorsque l'on écrit ~~L = L.append(Objet)~~, cela revient à écrire L = NONE et L est donc d'abord modifiée (ajout de l'objet), puis supprimée !!!!!

A.IV.5.c Taille

Pour obtenir la taille d'une liste, on utilise la commande $len(L)$

```
>>> L = [1,2,3]

>>> len(L)
3
```

On obtient le nombre d'éléments contenus dans L.

On peut donc accéder au dernier élément ainsi :

$$x = L[len(L) - 1]$$

A.IV.5.d Indices/séparateurs

Au départ, vous aurez du mal à utiliser les listes pour une simple raison. Les indices vont de 0 à n-1 pour n termes. Il faudra vous habituer à travailler avec des indices partant de 0.

A.IV.5.d.i Indices : récupérer un objet

Prenons l'exemple d'une liste de lettres : $L = ["a","b","c","d","e","f","g","h"]$

Pour accéder à l'un des objets de la liste L, il suffit d'écrire $L[i]$ où i est l'indice de l'objet recherché

Le tableau ci-dessous résume les indices positifs et négatifs associés aux éléments de L :

Indice positif	0	1	2	3	4	5	6	7
Objet	$"a"$	$"b"$	$"c"$	$"d"$	$"e"$	$"f"$	$"g"$	$"h"$
Indice négatif	-8	-7	-6	-5	-4	-3	-2	-1

Ainsi :

```
>>> L = ["a","b","c","d","e","f","g","h"]

>>> L[0]
'a'

>>> L[-8]
'a'
```

On pourra accéder simplement au dernier terme d'une liste en écrivant :

$$L[-1] \text{ ce qui est identique à } L[len(L)-1]$$

Il est aussi possible d'accéder à une portion de la liste en créant une nouvelle liste extraite de la première.

Remarque : On peut créer une liste de taille donnée ne comportant pas pour autant de valeurs à l'aide de ***None***. ***None*** est un objet qui « n'est rien ». Ainsi, on peut appeler un terme d'une liste sans pour autant recevoir de message d'erreur si le terme n'existe pas. Avec None, python retourne « None », ou « rien ».

Exemple :

```
>>> L=[None,None,None]

>>> L[1]

>>> L=[]

>>> L[1]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list index out of range
```

A.IV.5.d.ii Séparateurs – récupérer plusieurs objets

Si on veut récupérer les 3 premiers termes, c'est-à-dire de l'indice 0 à l'indice 2 inclus, il faut écrire :

```
>>> L[0:3]
['a', 'b', 'c']
```

Cela peut paraître perturbant, car on aurait voulu logiquement écrire $L[0:2]$. Toutefois, on peut retenir que l'on demande tous les termes entre les « séparateurs » (mot personnel définissant toute séparation d'un objet d'autre chose) précisés. Le tableau ci-dessous présente les numéros des séparateurs en indices positifs et négatifs entre chaque objet :

	0	1	2	3	4	5	6	7	
Objet	"a"	"b"	"c"	"d"	"e"	"f"	"g"	"h"	
	-8	-7	-6	-5	-4	-3	-2	-1	

Ainsi, on a :

```
>>> L[2:4]
['c', 'd']

>>> L[-6:-4]
['c', 'd']
```

On notera que prendre les indices dans le mauvais sens conduit à un résultat vide :

```
>>> L[4:2]
[]

>>> L[-4:-6]
[]
```

Enfin, il est possible de demander tous les termes à partir d'une position ainsi :

```
>>> L[-6:]
['c', 'd', 'e', 'f', 'g', 'h']

>>> L[:3]
['a', 'b', 'c']

>>> L[:-5]
['a', 'b', 'c']

>>> L[3:]
['d', 'e', 'f', 'g', 'h']
```

On remarquera que la liste sera toujours prise dans le sens gauche -> droite lorsque l'on utilise le : seul ($L[:-3]$)

A.IV.5.e Opérations

Vous pourrez utiliser les quelques opérations suivantes :

$L.remove(objet)$	Retire la première occurrence de l'objet en question de la liste en partant de la gauche
$L.reverse()$	Inverse le sens des termes de la liste
$L.count(objet)$	Retourne le nombre de fois où l'objet apparaît
$L.index(objet)$	Retourne l'index de la première occurrence de l'objet
$L.extend(LL)$ $L + LL$	Ajoute la liste LL à la liste L
$L.insert(i,x)$	Ajoute x au niveau du séparateur numéro i
$del\ L[i]$	Supprime le terme d'indice i de L, soit le $(i+1)$° terme

A.IV.5.f Fonction range

A.IV.5.f.i Pour créer une liste

La fonction $range(n)$ est très intéressante car elle génère une boucle invisible permettant de créer des listes assez complexes.

On écrit par exemple :

```
>>> L=[i for i in range(10)]

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

Concrètement, cela veut dire : « créer la liste L contenant les différentes valeurs de i pour i dans la plage contenue entre les séparateurs 0 et n ». Elle contient n termes pour i allant de 0 à n-1

On peut comme cela créer une liste de nombres paires par exemple :

```
>>> L=[2*i for i in range(10)]

>>> L
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Remarques :

- On peut prendre les termes

du séparateur i au séparateur j : $range(i,j)$	entre deux séparateurs en précisant un pas : $range(i,j,p)$

`>>> L = [i for i in range(2,5)]`
 	
`>>> L`
`[2, 3, 4]`
	`>>> L = [i for i in range(10,20,2)]`
	
	`>>> L`
	`[10, 12, 14, 16, 18]`

- On peut aussi directement créer une liste avec range, auquel cas pour afficher les termes, il faut utiliser la commande list(L) :

```
>>> L = range(0,10,1)

>>> L
range(0, 10)

>>> list(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Attention, quelle que soit la forme utilisée, les arguments de range doivent être des nombres de type int (entiers).

A.IV.5.f.ii Pour parcourir une liste

Il est possible de parcourir les éléments d'une liste L en écrivant $for\ i\ in\ L$, par exemple :

`L = [i for i in range(0,11,2)]`
`LL = [2*i for i in L]`
	`>>> L`
	`[0, 2, 4, 6, 8, 10]`
	
	`>>> LL`
	`[0, 4, 8, 12, 16, 20]`

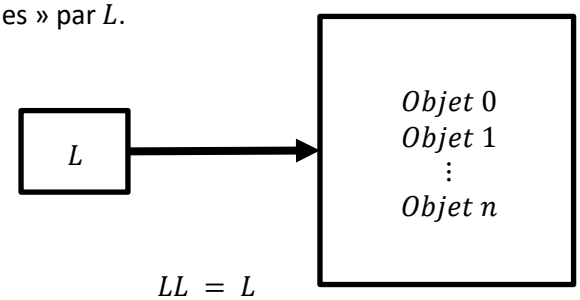
Attention dans les boucles for à l'utilisation de cette commande for i in L à ne pas modifier L ! (cf paragraphe sur la boucle for)

A.IV.5.g Copie de listes

Il est très important de faire attention à la copie de listes.

En effet, une liste L est en fait un ensemble de données en mémoire. Taper L consiste à aller chercher en mémoire les données « pointées » par L.

Lorsque l'on écrit :



$$LL = L$$

On croit créer une nouvelle liste LL, copie de L.

Essayons donc de le faire, et modifions une valeur de la nouvelle liste :

```
>>> L = [1,2,3,4,5]
>>> LL = L
>>> LL
[1, 2, 3, 4, 5]
```

Maintenant que nous disposons de deux listes, modifions la nouvelle liste LL :

```
>>> LL[1]=0
>>> LL
[1, 0, 3, 4, 5]
```

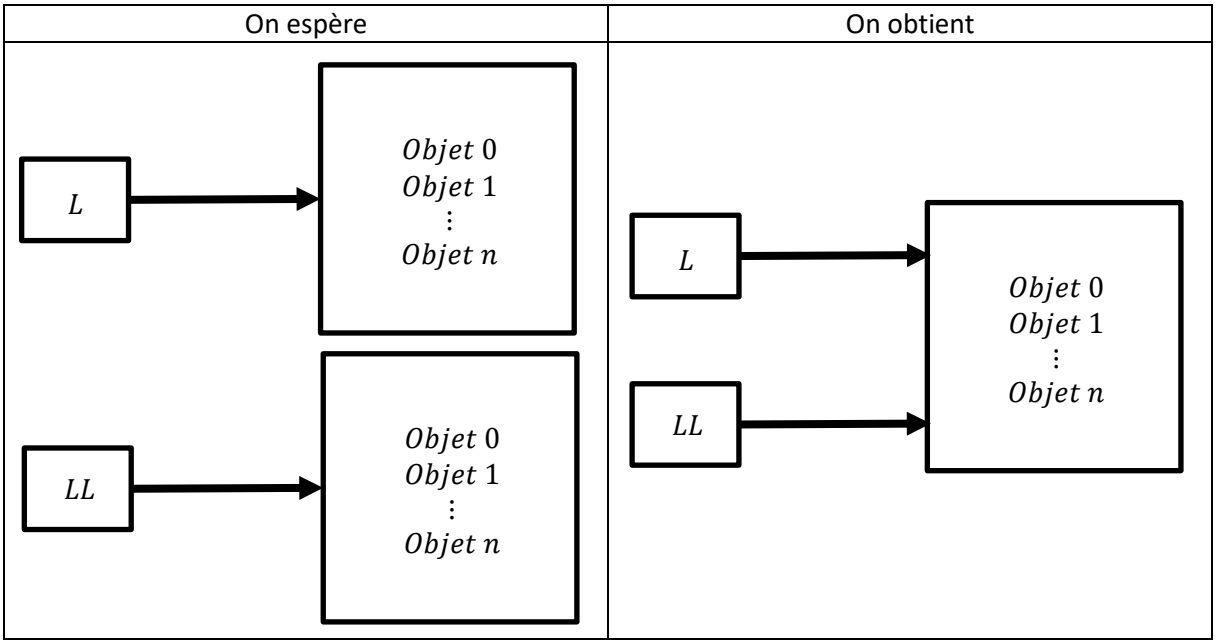
Et maintenant, regardons ce que vaut L :

```
>>> L
[1, 0, 3, 4, 5]
```

OUPS

Il faut faire très attention à cela !

Lorsque l'on écrit $LL = L$:



Remarque : ce n'est pas le cas des variables qui elles sont indépendantes.

Il existe des solutions simples, en faisant une itération sur chaque terme et en l'affectant dans une nouvelle liste vide par exemple.

Sinon, on peut importer dans pythons le module « copy » puis écrire la copie ainsi :

```
>>> import copy
>>> L = [1,2,3,4,5]
>>> LL=L.copy()
>>> LL[1]=0
>>> LL
[1, 0, 3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
```

A.IV.5.h Listes de listes

Il est possible de créer des listes de listes de listes…, pour cela rien de plus simple :

- Créer des listes L1, L2,…,Ln et écrire L=[L1,L2,…,Ln]
- Ou le faire directement : L = [[1,2],[3,4,5],…,[6,7,8,9]] par exemple

On peut alors récupérer

- L'une des listes en écrivant : L[i] ou i est l'indice de la liste à récupérer
- Un terme d'une des sous listes en écrivant L[i][j]…[k] où k est l'indice du terme dans la liste, et ainsi de suite pour les indices précédents si on a une liste de liste de liste… Ne pas écrire L[i,j…], ça ne fonctionne pas avec Python et les listes

Exemple :

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

A.IV.6 Algorithmique

A.IV.6.a Les variables booléennes

L'utilisation des boucles/conditions va nous conduire à devoir effectuer des tests pour obtenir des conditions.

Pour cela, introduisons les variables de type booléennes. Elles valent 1, ou 0, ou plutôt, True (juste) ou False (faux).

Pour obtenir une variable booléenne, il faut effectuer un test :

==	Vérifie l'égalité
!=	Vérifie la différence
>	Comparaison
>=	
<	
<=	
a <= b < c	On peut écrire des inégalités à plusieurs comparaisons

Les résultats de ces tests sont soit « True », soit « False »

On peut effectuer des opérations entre variables booléennes :

and	Fonction « et »
or	Fonction « ou »
not(Cond)	Donne la condition inverse : Not(True)->False Not(False)->True

A.IV.6.b Implémentation d'un compteur

L'utilisation de boucles va souvent être associée à l'utilisation d'un compteur.

Ce compteur sera toujours initialisé avant la boucle :

Compteur = 0

Puis incrémenté, par exemple de 1, à chaque itération, en écrivant au choix :

- Compteur = Compteur + 1
- Compteur += 1

A.IV.6.c Boucles et itérations

A.IV.6.c.i Boucle for – pour

Le principe est de réaliser une ou plusieurs opérations pour une variable allant d'une valeur à une autre.

La syntaxe est la suivante :

```
for i in range(n):
    opérations à effectuer
    opérations à effectuer
    opérations à effectuer
    opérations à effectuer
    opérations à effectuer
```

Attention
i varie du séparateur 0 au séparateur n,
soit dans la plage 0, n-1
Il y a bien n étapes

La fin de cette boucle est prévue d'avance !

La variable i est créé par la boucle, ses valeurs sont imposées par le for et augmentent toutes seules. Inutile donc de l'initialiser avant la boucle, elle sera remplacée :

Code	Résultat

`i = 10`
`for i in range(5):`
`    print(i)`
	0
	1
	2
	3
	4

Et inutile de l'augmenter :

Code	Résultat

`for i in range(5):`
`    print(i)`
`    i += 5`
	0
	1
	2
	3
	4

On peut stopper une boucle for avec la commande « break » :

Code	Résultat

`for i in range(5):`
`    print(i)`
`    if i==3:`
`        break`
	0
	1
	2
	3

Attention, danger : Si on utilise une boucle for en écrivant « `for i in L:` », et si par hasard on modifie L dans la boucle, il y aura des erreurs très compliquées à identifier. Exemples

Code	Résultat

`A = [0,1,2,3,4,5,6,7,8,9,10]`
`for i in range(len(A)):`
`    A.remove(i)`
`print("A la fin: A = ",A)`
	`A la fin: A =  []`

`A = [0,1,2,3,4,5,6,7,8,9,10]`
`for i in A:`
`    A.remove(i)`
`print("A la fin: A = ",A)`
	`A la fin: A =  [1, 3, 5, 7, 9]`

On peut remarquer le comportement insoupçonné du second code !

- Ecrire « `for i in range(len(A)):` » fait varier i de 0 à la longueur initiale de A
- Ecrire « `for i in A:` » incrémente un indice caché j de 1 à chaque itération (j=j+1), i prend la valeur A[j] jusqu'à ce que j+1 corresponde au dernier indice de A actuel augmenté de 1 ! On évitera d'écrire i et on lui préfèrera un nom de variable qui marque la différence avec les indices, par exemple : « `for Terme in A:` »

Remarque : Ecrire « `for i in range(0):` » conduit à la non-exécution du code dans le for !!!

A.IV.6.c.ii Condition if – si

• **Syntaxe**

Le principe est de réaliser une ou plusieurs opérations si une condition est vraie, et éventuellement d'autres opérations si la condition est fausse, voire même différente.

La syntaxe est la suivante :

```
if Condition_1 :
    opérations à effectuer si Condition_1 vaut True
    opérations à effectuer si Condition_1 vaut True
elif Condition_2 : # Optionnel
    opérations à effectuer si Condition_2 vaut True
    opérations à effectuer si Condition_2 vaut True
else : # Optionnel
    opérations à effectuer si ni Condition_1, ni Condition_2 ne sont
vérifiées
    opérations à effectuer si ni Condition_1, ni Condition_2 ne sont
vérifiées

# Suite du programme
```

Ceci n'est pas une boucle, elle est exécutée puis le programme continue.

• **Remarques**

Attention, lorsque vous n'exécutez rien, python n'est pas content. Exemple :

```
L = [2,10,9,3,5,6,1,7]
LL = []

for i in range(len(L)):
    Terme = L[i]
    if Terme > 5:
        # Ne rien faire
    else:
        LL.append(Terme)

print(LL)
```

```
>>> (executing file "<tmp 1>")
  File "<tmp 1>", line 8
    else:
       ^
IndentationError: expected an indented block
```

Il faut donc mettre quelque chose d'inutile pour que ça passe :

```
L = [2,10,9,3,5,6,1,7]
LL = []

for i in range(len(L)):
    Terme = L[i]
    if Terme > 5:
        a = 0 # Ne rien faire
    else:
        LL.append(Terme)

print(LL)
```

Lorsque l'on ne veut que le programme n'exécute rien à une condition, il est préférable de transformer la condition en son opposé et de n'exécuter que ce que l'on veut. On pourra donc ne pas mettre de « else » et le programme fonctionnera parfaitement :

On préfèrera donc écrire :

```
L = [2,10,9,3,5,6,1,7]

for i in range(len(L)):
    Terme = L[i]
    LL = []
    if Terme <= 5:
        LL.append(Terme)

print(LL)
```

Certains logiciels de programmation n'utilisent pas l'indentation pour organiser les boucles. Par exemple, voici la comparaison de deux codes identiques sous Python et Matlab :

Python

```python
def f(x):
    if x == 0:
        return 0
    else:
        return 1/x

print(f(2))
```

Matlab

```matlab
function [ s ] = f( x )
if x == 0
s = 0;
else
s = 1/x;
end
end
```

```matlab
disp(f(1));
```

L'absence d'indentation est contrée par la présence d'un « end » pour marquer la fin de la condition if.

Sous Python, vous risquez de prendre la « mauvaise » habitude de ne pas forcément écrire else en présence de return :

Python

```python
def f(x):
    if x == 0:
        return 0
    return 1/x

print(f(2))
```

Ce code fonctionne sous Python, mais ce n'est pas possible sous Matlab, d'une parce que Matlab ne permet pas de s'arrêter à un « return » mais lit tout le code, et de deux parce que sans indentation et sans else, on exécute tout !

Comme l'objectif est de vous apprendre à programmer, quel que soit le logiciel, pensez à toujours mettre « else ».

A.IV.6.c.iii Boucle conditionnelle while – tant que

Le principe est de réaliser une ou plusieurs opérations tant qu'une condition est vraie.

Le while est utilisé lorsque l'on ne connait pas à priori le nombre d'itérations à effectuer. Sinon, on privilégie la boucle for !

La syntaxe est la suivante :

```
while Condition :
    opérations à effectuer
    opérations à effectuer
    opérations à effectuer
    opérations à effectuer
    opérations à effectuer
    Modification de la condition

# Suite du programme
```

Lorsque l'on rentre dans cette boucle, la condition est vérifiée. Si on ne modifie pas cette condition dans la boucle, le programme y restera indéfiniment ! Pour en sortir, casser le programme avec la commande « Ctrl+i ».

Il faudra donc que la condition soit modifiée lors de la lecture de la boucle jusqu'à ce qu'elle corresponde à ce qui est attendu.

Le code contenu dans la boucle est réalisé une dernière fois lorsque la condition devient vraie.

On peut faire une boucle infinie en écrivant « `while True :` »

A.IV.6.d Exécution de boucles directement dans la console « Shell »

Il est possible d'exécuter des boucles dans la console.

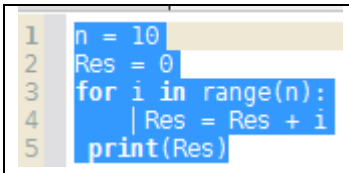
Il suffit :

- Soit de la copier dans le code et de la coller dans la console puis d'appuyer 2 fois sur « Entrée ».
- Soit d'écrire la première ligne (ex : `for i in range(10):`) puis de valider avec « Entrée ». Puis on écrit les lignes de code les une après les autres en validant à chaque fois avec « Entrée ». Pour l'exécuter, appuyer 2 fois sur « Entrée », ou plutôt, valider deux lignes vides

A.IV.6.e Remarque importante

Il est obligatoire de respecter la présence des « : » d'une part, et « l'indentation » d'autre part. En effet, c'est ce décalage des instructions qui va permettre de dire si l'instruction est dans la boucle ou en dehors.

Indenter d'un étage correspond à l'appui une fois sur la touche Tabulation, ou 4 fois la barre espace. Lorsque l'indentation est mauvaise dans les boucles et conditions, on peut le voir en surlignant le code :

	Ligne 4, avant Res, on voit un trait vertical blanc qui correspond à l'indentation qu'il devrait y avoir, on voit qu'il y a un espace en trop ici. On pourra aussi remarque que le print est décalé d'un espace en trop ligne 5.

Dans d'autres langages de programmation, l'indentation peut ne pas avoir d'effet, mais dans ce cas, la boucle contient un terme, par exemple :

if a == 1 then
print('Fin')
endif
	if a == 1 then
	print('Fin')
	ifend

A.IV.6.f Vérification des algorithmes

Face à un algorithme, on doit se poser 3 questions :

- L'algorithme donne-t-il un résultat ?
- Le résultat est-il le bon ?
- Est-ce que le temps mis par l'algorithme est raisonnable / optimisé ?

A ces 3 notions sont respectivement associés 3 termes :

- Terminaison
- Correction
- Complexité

A.IV.6.f.i Terminaison

- **Introduction**

Vérifier la terminaison d'un algorithme consiste à vérifier qu'il a bien une fin. Autrement dit, il faut que le nombre d'itérations soit fini.

Lorsqu'une boucle for est utilisée avec un **compteur non perturbé** (pas de modification dans la boucle de ce qui est après le for, nous traiterons un exemple), on connaît à priori le nombre d'itérations qui sont réalisées, on sait donc que l'algorithme se finit.

Lorsqu'une boucle « while » est utilisée, l'algorithme se termine si la condition d'entrée n'est plus vérifiée. Par construction, la condition est vérifiée lors de l'entrée dans l'algorithme afin d'en exécuter le contenu. Il est donc nécessaire de faire évoluer la condition dans la boucle et d'être sûr qu'elle finira par prendre la valeur « False ».

La non terminaison d'un algorithme conduit le logiciel utilisé à répéter indéfiniment des actions sans fin.

- **Outil d'étude de la terminaison**

On définit un « variant » ou « convergent » de boucle. C'est un nombre qui prend des valeurs dans un ensemble de dimension finie (imposé par la condition) et qui diminue strictement (il ne peut rester identique entre deux itérations) à chaque itération. Il est donc sûr que la boucle se terminera. Si son élaboration est impossible, c'est qu'il y a un problème et qu'il faut revoir l'algorithme.

- **Exemples**

	Programme	Terminaison

for	`n = 100`
	`L = [i for i in range(n)]`
	`Somme = 0`
	`for i in range(len(L)):`
	`    Somme += L[i]`
	`print(Somme)`
		Cet algorithme a un nombre d'itérations prédéterminé. Après 100 itérations, il se termine.
		
		La terminaison est vérifiée

	`L = [0]`
	`for Terme in L:`
	`    L.append(Terme)`
		Danger : On modifie la taille de L dans la boucle !
		La taille initiale de L vaut $T_0 = 1$.
		A chaque itération, la taille de L est incrémentée de 1 : $\forall i, T_i = T_0 + i$ lors du début de lecture de la boucle.
		La boucle basée sur la liste L appelle l'indice i tant que $i < T_i$
		Soit : $i < T_0 + i \Leftrightarrow 0 < T_0$
		Cette condition est toujours vraie si $T_0 > 0$!
		
		La terminaison n'est pas vérifiée
		Rq : si ce n'est pas une liste mais un réel après « in », pas de problèmes même si ce réel est modifié dans la boucle

while	`# Division euclidienne a = bq+r`
	
	`a = 25`
	`b = 7`
	
	`q = 0`
	`r = a`
	`while r > b:`
	`    r = r - b`
	`    q += 1`
	
	`print(q,r)`
		Le reste r **est un variant de boucle**. C'est un entier positif (imposé par la condition `r > b` et `r = r - b`) défini dans l'intervalle $[b, +\infty]$ dont la valeur décroit strictement à chaque itération (`r = r - b`).
		
		La terminaison est vérifiée

	`# Nombre pair -> 0 impaire -> 1`
	
	`N = 23`
	`while N != 1:`
	`    N += -2`
	`print(N)`
		N décroit à chaque itération mais n'évolue pas dans un intervalle fini. La condition `N != 1` définie l'intervalle $N \backslash 1$.
		On ne répond donc pas à la condition de terminaison.
		Si N est impair au départ, la boucle a une fin.
		Si N est pair, la boucle n'a pas de fin.
		
		La terminaison n'est pas vérifiée

A.IV.6.f.ii Correction

- **Introduction**

La correction d'un algorithme, qu'il soit composé ou non d'itérations, consiste à vérifier qu'il donne la valeur attendue. Cela revient à en faire la démonstration.

Vérifier un algorithme présentant une boucle consiste à réaliser une démonstration comme la récurrence en mathématiques.

- **Outil d'étude de la correction**

Appelons $\mathcal{P}$ une propriété quelconque, qui doit être vérifiés tout au long du déroulement de l'algorithme. La propriété $\mathcal{P}$ s'appelle un invariant de boucle.

Trouver la propriété est la phase « difficile ». Il faut de l'intuition, ou écrire l'algorithme pour quelques exemples.

Exemple : Soit l'algorithme suivant :

```
# Factoriel n, n>0

p = n
a = n
for i in range(n-1):
    a = a - 1
    p = p * a
print(p)
```

Trouvons la propriété $\mathcal{P}(i)$ qui définit les valeurs $\begin{cases} a_i = \cdots \\ p_i = \cdots \end{cases}$

$n=1$	RAS
$n=2$	$i=0: \begin{cases} a_0 = n-1 = 2-1 = 1 \\ p_0 = n * a_0 = 2*1 = 2 \end{cases}$
$n=3$	$i=0: \begin{cases} a_0 = a-1 = 3-1 = 2 \\ p_0 = p * a_0 = 3*2 = 6 \end{cases}$ $i=1: \begin{cases} a_1 = a_0 - 1 = 2-1 = 1 \\ p_1 = p_0 * a_1 = 6*1 = 6 = 3*2*1 \end{cases}$
$n=4$	$i=0: \begin{cases} a_0 = a-1 = 4-1 = 3 \\ p_0 = p * a_0 = 4*3 = 12 \end{cases}$ $i=1: \begin{cases} a_1 = a_0 - 1 = 3-1 = 2 \\ p_1 = p_0 * a_1 = 12*2 = 24 = 4*3*2 \end{cases}$ $i=2: \begin{cases} a_2 = a_1 - 1 = 2-1 = 1 \\ p_2 = p_1 * a_2 = 24*1 = 4*3*2*1 \end{cases}$

Avec notre intuition, on peut proposer la propriété suivante :

$$\mathcal{P}(i): \begin{cases} a_i = n - i - 1 \\ p_i = \dfrac{n!}{(n-2-i)!} \end{cases}$$

• **Correction d'une boucle « for »**

Notons $\mathcal{P}(k)$ la propriété après l'itération pour $i = k$

Corriger une boucle « for » consiste à réaliser 3 étapes :

- Initialisation : $\mathcal{P}$ vraie à la première itération : $\mathcal{P}(0)$
- Transmission : Si $\mathcal{P}$ vraie à l'itération i, $\mathcal{P}$ vraie à l'itération $i+1$: $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$
- Sortie : A la dernière itération n, $\mathcal{P}$ doit permettre de démontrer que le résultat obtenu est le résultat attendu : $\mathcal{P}(n) \Rightarrow Résultat$

Remarque : la première itération correspond à $i = 0$

• **Correction d'une boucle « while »**

Notons $\mathcal{P}(i)$ la propriété après la i eme itération.

Corriger une boucle « while » consiste à réaliser 3 étapes :

- Initialisation : $\mathcal{P}$ vraie avant la première itération : $\mathcal{P}(0)$
- Transmission : Si $\mathcal{P}$ vraie avant l'itération i, $\mathcal{P}$ vraie après: $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$
- Sortie : Après la dernière itération n (lorsque la condition devient fausse pour la première fois), $\mathcal{P}(n)$ doit permettre de démontrer que le résultat obtenu est le résultat attendu : $\mathcal{P}(n) \Rightarrow Résultat$

Remarque : $i = 0$ correspond à l'état initial, avant la première itération. $i = 1$ correspond donc à la fin de la première itération, contrairement à la correction de la boucle for où $i = 1$ correspond à la fin de la seconde itération, $i = 0$ étant la fin de la première itération

- **Exemples**

<table>
<tr><th></th><th>Programme</th><th>Correction</th></tr>
<tr><td>Programme normal</td><td>

```
# Partie
positive

X = 10
Y = (X +
abs(X))/ 2
print(Y)
```

</td><td>Il suffit de vérifier que le programme fait ce qu'il annonce
$$\frac{X+\text{abs}\,X}{2}=\begin{cases}\frac{X+X}{2}=X\ si\ X>0\\ \frac{X-X}{2}=0\ si\ X>0\end{cases}$$
Ce programme est correct</td></tr>
<tr><td rowspan="2">for</td><td>

```
# Factoriel n,
n>0

p = n
a = n
for i in
range(n-1):
    a = a - 1
    p = p * a
print(p)
```

</td><td>Vérifions si cette fonction calcule bien $n!$ Si $n>0$
Supposons $n\geq 1$. Notons a_i et p_i les valeurs de a et p à la fin de l'itération i.
Algorithme : à tout moment, on a : $\begin{cases}a_{i+1}=a_i-1\\ p_{i+1}=p_i*a_{i+1}\end{cases}$
Propriété : soit la propriété $\mathcal{P}(i)$: $\begin{cases}a_i=n-i-1\\ p_i=\frac{n!}{(n-2-i)!}\end{cases}$
Initialisation : $i=0, \mathcal{P}(0)$: $\begin{cases}a_0=n-1=n-0-1=n-i-1\\ p_0=n*(n-1)=\frac{n!}{(n-2-0)!}=\frac{n!}{(n-2-i)!}\end{cases}$
Transmission : $i\in[1,n-2]$, $\mathcal{P}(i)$ supposée vraie :
$$\begin{cases}a_i=n-i-1\\ p_i=\frac{n!}{(n-2-i)!}\end{cases}$$
$$\begin{cases}a_{i+1}=a_i-1=n-1-i-1=n-(i+1)-1\\ p_{i+1}=p_i*a_{i+1}=p_i*(a_i-1)=p_i*(n-2-i)=\frac{n!\,(n-2-i)}{(n-2-i)!}\end{cases}$$
$$\begin{cases}a_{i+1}=n-(i+1)-1\\ p_{i+1}=\frac{n!}{(n-3-i)!}=\frac{n!}{(n-2-(i+1))!}\end{cases}$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : $\mathcal{P}(n-2)$ vraie : $p_{n-2}=\frac{n!}{(n-2-(n-2))!}=\frac{n!}{0!}=n!$
Ce programme est correct</td></tr>
<tr><td>

```
# Factoriel n,
n>0

p = 1
for i in
range(n+1):
    p = p *
(i+1)
print(p)
```

</td><td>Vérifions si cette fonction calcule bien $n!$ Si $n>0$
Supposons $n\geq 1$. Notons p_i la valeur de p à la fin de l'itération i.
Algorithme : à tout moment, on a : $p_{i+1}=p_i*(i+2)$
Propriété : soit la propriété : $\mathcal{P}(i): p_i=(i+1)!$
Initialisation : $i=0, \mathcal{P}(0): p_0=1=(0+1)!=(i+1)!$
Transmission : $i\in[1,n]$, $\mathcal{P}(i)$ supposée vraie $p_i=i!$
$$p_{i+1}=(i+1)!*(i+2)=(i+2)!$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : $\mathcal{P}(n)$ vraie : $p_n=(n+1)!$
Ce programme est incorrect – Il faudrait changer n+1 en n</td></tr>
</table>

<table>
<tr><th></th><th>Programme</th><th>Correction</th></tr>
<tr><td rowspan="2">while</td><td>

```
# Somme des n 1° entiers

s = 0
a = 0
while a <= n:
    a += 1
    s += a
print(s)
```

</td><td>
Vérifions si cette fonction calcule bien $\sum_{i=1}^{n} i$
Supposons $n \geq 0$. Notons a_i et s_i les valeurs de a et s à la fin de l'itération i.
Algorithme : à tout moment, on a : $\begin{cases} a_{i+1} = a_i + 1 \\ s_{i+1} = s_i + a_{i+1} \end{cases}$
Propriété : soit la propriété : $\mathcal{P}(i)$: $\begin{cases} a_i = i \\ s_i = \sum_{j=1}^{i} j \end{cases}$
Initialisation : $i = 0$ (avant boucle), $\mathcal{P}(0)$: $\begin{cases} a_0 = 0 = i \\ s_0 = 0 = \sum_{j=1}^{0} j \end{cases}$
Transmission : $i \in [1, n+1]$, $\mathcal{P}(i)$ supposée vraie :
$$\begin{cases} a_i = i \\ s_i = \sum_{j=1}^{i} j \end{cases}$$
$$\begin{cases} a_{i+1} = a_i + 1 = i + 1 \\ s_{i+1} = s_i + a_{i+1} = \sum_{j=1}^{i} j + (i+1) = \sum_{j=1}^{i+1} j \end{cases}$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : $\mathcal{P}(n+1)$ vraie : $p_{n+1} = \sum_{i=1}^{n+1} i$
Remarque : la n° fois où l'itération est effectuée, $a_n = n$, le programme va donc faire une n+1 ° itération

Ce programme est incorrect – Il faudrait changer <=n en <n
</td></tr>
<tr><td>

```
# Factoriel n, n>0

p = 1
a = 0
while a < n:
    a = a + 1
    p = p * a
print(p)
```

</td><td>
Vérifions si cette fonction calcule bien $n!$ Si $n > 0$
Supposons $n \geq 1$. Notons a_i et p_i les valeurs de a et p à la fin de l'itération i.
Algorithme : à tout moment, on a : $\begin{cases} a_{i+1} = a_i + 1 \\ p_{i+1} = p_i * a_{i+1} \end{cases}$
Propriété : soit la propriété : $\mathcal{P}(i)$: $\begin{cases} a_i = i \\ p_i = i! \end{cases}$
Initialisation : $i = 0$ (avant boucle), $\mathcal{P}(0)$: $\begin{cases} a_0 = 0 = i \\ p_0 = 1 = 0! = i! \end{cases}$
Transmission : $i \in [1, n]$, $\mathcal{P}(i)$ supposée vraie :
$$\begin{cases} a_i = i \\ p_i = i! \end{cases}$$
$$\begin{cases} a_{i+1} = a_i + 1 = i + 1 \\ p_{i+1} = p_i * a_{i+1} = i!\,(i+1) = (i+1)! \end{cases}$$
$\mathcal{P}(i+1)$ est donc vraie
Sortie : $\mathcal{P}(n)$ vraie : $p_n = n\,!$
Remarque : la n° fois où l'itération est effectuée, $a_n = n$ après mise à jour, on avait donc bien $a < n$

Ce programme est correct
</td></tr>
</table>

A.IV.6.f.iii Complexité

- **Introduction**

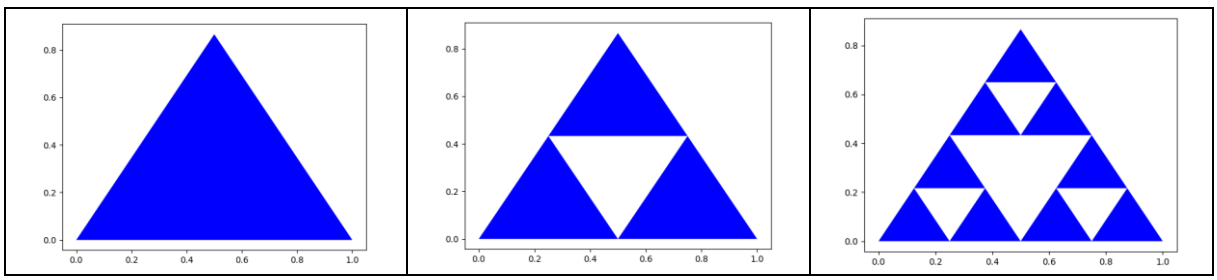
Un algorithme exécute une série d'opérations. Celles-ci mettent un certain temps à être exécutées et peuvent, selon les variables stockées, occuper un espace mémoire important. On parle de complexité en temps et en espace des algorithmes.

Il ne faut pas confondre complexité et capacités des ordinateurs. Un programme complexe en mémoire tournant sur un ordinateur ayant une capacité mémoire importante sera exécuté sans problèmes. Un programme de même complexité en temps ne prendra pas le même temps d'exécution sur deux ordinateurs différents (fréquence de processeur par exemple).

Aujourd'hui, la mémoire n'est généralement pas un problème. On s'intéresse souvent plus à la complexité en temps des algorithmes.

Remarque : Certains simulations numériques peuvent prendre plusieurs mois à plusieurs années de calculs.

Exemple de réalisation récursive que l'on fera en 2° année et qui consomme un temps de calcul en 3^n, n étan t le nombre de sous triangles demandés. Réalisation de l'image : 5 minutes !



• Complexité en temps

On définit un paramètre de complexité qui correspond à un nombre qui va définir en quelques sorte le nombre d'opérations à réaliser.

Prenons l'exemple suivant :

```
Res = 0
n = 100
for i in range(1,n+1): # Somme des termes de 1 à n
    Res = Res + i
print(Res)
```

Appelons t_a le temps mis pour effectuer une affectation de variable (`Res = 0`, `n = 100`, `Res = Res + i`)

Appelons t_o le temps mis pour effectuer une opération sur un entier (augmenter i)

On peut alors estimer le temps total T d'exécution de l'algorithme ci-dessus :

$$T = 2t_a + n(t_o + t_a)$$

Les temps t_a et t_o dépendent du langage de programmation et de l'ordinateur utilisés.

On dira que cet algorithme est de complexité $O(n)$, où que son temps d'exécution augmente linéairement avec le paramètre n.

• Ordres de grandeurs de temps de calcul

L'ordre de grandeur d'un calcul d'ordre 1 est $t = 1 * s = 10^{-6}\ s$. A partir de là, il est assez simple de calculer le temps d'exécution T d'un algorithme de complexité $Of(n)$ en calculant :

$$T = t * f(n)$$

	$n = 1$	$n = 10^5$	$n = 10^{10}$
$O(1)$	$1\ \mu s$	$1\ \mu s$	$1\ \mu s$
$O(\log_2 n)$	$1\ \mu s$	$17\ \mu s$	$33\ \mu s$
$O(n)$	$1\ \mu s$	$0{,}1\ s$	$2{,}7\ h$
$O(n^2)$	$1\ \mu s$	$2{,}7\ h$	$31\ 700\ millénaires$
$O(n^n)$	$1\ \mu s$	Démesuré	

Attention, ce sont des ordres de grandeur. Pour de faibles valeurs de n, un algorithme $O(an)$ peut être moins rapide qu'un algorithme $Of(bn^2)$.

- **<u>Notation de Landau – Grand O</u>**

En mathématiques, vous allez rapidement utiliser la notation $o(n)$ ou « petit o de n », et $O(n)$ ou « grand o de n ». Ils correspondent à une comparaison asymptotique.

- $o(n)$ s'associe à quelque chose de petit devant n au voisinage de l'infini
- $O(n)$ s'associe à quelque chose de dominé par n au voisinage de l'infini

Dans le cadre de l'analyse de la complexité des algorithmes, nous n'utiliserons que la notation $O(f)$ pour dire que cette complexité est de l'ordre de grandeur de la fonction f dans la parenthèse.

Mathématiquement, dire : $f(x) = O\big(g(x)\big)$ signifie $\exists N \in \mathbb{R}^+ \ \& \ A \in \mathbb{R} \ / \forall x > N, \left|\frac{f(x)}{g(x)}\right| < A$

Lorsque qu'une complexité est indépendante de n, c'est donc une constante. On donne alorts le résultat $O(1)$.

Si une complexité est en $O(n)$ et si par exemple $n = 100$, NE SURTOUT PAS dire $O(100)$, ce qui est équivalent à $O(1)$.

Evidemment, lorsque l'on demande une complexité, on attend la fonction de croissance le plus petit possible pour décrire le comportement temporel du temps de calcul maximum. Ainsi, dire que (presque) tout code est de complexité $O(n^n)$ n'est pas faux…….

Ainsi :

$f(n)$	$C(n)$	Preuve
10	$O(1)$	$\frac{10}{1} = 10$
$2n$	$O(n)$	$\frac{2n}{n} = 2$
$2n^2$	$O(n^2)$	$\frac{2n^2}{n^2} = 2$
$2n + 3n^2$	$O(n^2)$	$\frac{n + 3n^2}{n^2} \underset{+\infty}{\sim} \frac{3n^2}{n^2} = 3$
n	$O(n^3)$	$\frac{n}{n^3} \underset{+\infty}{\sim} \frac{1}{n^2} \underset{+\infty}{\to} 0$ A éviter toutefois
$2^n + 3^n$	$O(3^n)$	$\frac{2^n + 3^n}{3^n} = \left(\frac{2}{3}\right)^n + 1 \underset{+\infty}{\sim} 1$
2^{n+1}	$O(2^n)$	$\frac{2^{n+1}}{2^n} = 2$
$\log n$	$O(\ln n)$	$\frac{\log n}{\ln n} = \frac{\frac{\ln n}{\ln 2}}{\ln n} = \frac{1}{\ln 2}$

- **Exemples simples**

Soient les algorithmes suivants, dont le résultat est identique :

Code Python	Complexité temporelle

`n = 100`
`Res = n*2*(1 + n)*n/2`
`print(Res)`
	$O(1)$

`Res = 0`
`n = 100`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    Res = Res + 2*n*i`
`print(Res)`
	$O(n)$

`Res = 0`
`n = 100`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    for j in range(1,n+1):`
`        Res = Res + i + j`
`print(Res)`
	$O(n^2)$

Si l'on fait tourner ces 3 algorithmes à la suite pour $n = 10^4$ par exemple, on verra immédiatement le temps mis par l'ordinateur pour donner les différentes solutions.

- **Exemples plus complexes (2° année)**

$$u_0 = 1, n \geq 1, u_{n+1} = \begin{cases} u_n + 1 \; si \; u_n < 1 \\ \frac{u_n}{2} \; sinon \end{cases}$$

Code Python	Complexité temporelle

`def rec(n):`
`    if n==0:`
`        return 1`
`    else:`
`        Un_m1 = rec(n-1)`
`        if Un_m1 < 1:`
`            Un = Un_m1 + 1`
`        else:`
`            Un = Un_m1 / 2`
`        return Un`
	$a_n = 0$
	$a_{n+1} = a_n + 1$
	$O(2^n)$

`def rec(n):`
`    if n==0:`
`        return 1`
`    else:`
`        if rec(n-1) < 1:`
`            Un = rec(n-1) + 1`
`        else:`
`            Un = rec(n-1) / 2`
`        return Un`
	$a_n = 0$
	$a_{n+1} = 2a_n + 1$
	$O(2^n)$

- **Outil d'analyse de complexité : time.clock()**

Importons un outil qui permet de définir le temps actuel afin de calculer le temps mis par l'ordinateur pour effectuer chacun des algorithmes vus plus haut :

Code exécuté	Résultat (temps en secondes)

`import time`
 	
`tic = time.clock()`
`n = 10000`
`Res = n*2*(1 + n)*n/2`
`print(Res)`
`toc = time.clock()`
`print("Temps 1: ",toc - tic)`
 	
`tic = time.clock()`
`Res = 0`
`n = 10000`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    Res = Res + 2*n*i`
`print(Res)`
`toc = time.clock()`
`print("Temps21: ",toc - tic)`
 	
`tic = time.clock()`
`Res = 0`
`n = 10000`
`for i in range(1,n+1): # Somme des termes de 1 à n`
`    for j in range(1,n+1):`
`        Res = Res + i + j`
`print(Res)`
`toc = time.clock()`
`print("Temps 3: ",toc - tic)`
	`10001000000000.0`
	`Temps 1:  2.716859745532929e-05`
	`10001000000000`
	`Temps21:  0.001440841284676253`
	`10001000000000`
	`Temps 3:  12.678871102513739`

Attention, ces temps dépendent de l'ordinateur utilisé. On voit clairement que pour le même calcul, les temps évoluent fortement en fonction de la méthode de programmation choisie. Il conviendra donc toujours de réfléchir à la meilleure manière de programmer, dès que les temps de calculs deviennent importants.

- **Complexité en log**

Prenons un dernier exemple d'algorithme dont la complexité est en $O(\log_2 n)$. Soit une liste L délimitant des intervalles du type : $L = [0,10,20,30,40,50,60,70,80,90,100]$ ou d'une manière générale, des intervalles de largeur l, dont la limite basse est 0 et la limite haute $l * n$:

$$L = [l * i \; for \; i \; in \; range(n+1)]$$

On cherche alors dans quel intervalle se trouve une valeur x quelconque. On souhaite donc déterminer si $L[i] \leq x < L[i+1]$ afin de remonter à i et $i+1$. Nous allons utiliser le principe de dichotomie, c'est-à-dire que nous allons diviser la plage d'étude en 2, puis voir si x est supérieur ou inférieur à la valeur médiane. On redéfinit alors le nouvel intervalle du bon côté et on continue.

```
n = 10
l = 10
L = [l*i for i in range(n+1)]
x = 50
import copy
L_Mod = L.copy()
while len(L_Mod) >= 3:
    print(L_Mod)
    Separateur_Med =
int(len(L_Mod)/2)
    Lg = L_Mod[0:Separateur_Med+1]
    Ld = L_Mod[Separateur_Med:]
    L_Med = L_Mod[Separateur_Med]
    if x >= L_Med:
        L_Mod = Ld
    elif x < L_Med:
        L_Mod = Lg
print(L_Mod)
```

On part de n intervalles, on divise t fois la plage d'intervalles jusqu'à ce que l'intervalle final ait une largeur égale à 1. On a donc :

$$\frac{n}{2^t} = 1$$

On cherche le nombre de divisions à réaliser, t :

$$2^t = n$$
$$e^{t \ln 2} = n$$
$$t \ln 2 = \ln n$$
$$t = \frac{\ln n}{\ln 2} = \log_2 n$$

Soit finalement :

$$O(\log_2 n)$$

On peut aussi dire :

$$O(\ln n)$$

A.IV.7 Fonctions

A.IV.7.a Utilité

Une fonction est une portion de code qui peut être « appelée », exécutée, à n'importe quel endroit d'un code informatique, à partir du moment où elle a été définie, c'est-à-dire lue une fois et mise en mémoire par l'ordinateur.

Elle peut exécuter une ou plusieurs opérations en prenant ou non des arguments en entrée, et en renvoyant ou non des variables en sortie.

Elle prend en entrée des variables bien définies, et renvoie le résultat voulu. On peut donc aisément copier/coller une fonction faite par quelqu'un d'autre à partir du moment où l'on maîtrise ses entrées et sorties, et ce sans modifier les variables déjà existantes (notion de variables locales vue par la suite).

A.IV.7.b Les fonctions par l'exemple

Supposons que nous souhaitons déterminer les diviseurs de 3 nombres. Mettons en œuvre ce programme avec et sans fonctions.

Sans fonctions

```
a = 10
b = 5
c = 200

Diviseurs_a = []
Diviseurs_b = []
Diviseurs_c = []

for i in range(1,a+1):
    if a%i == 0:
        Diviseurs_a.append(i)

for i in range(1,b+1):
    if b%i == 0:
        Diviseurs_b.append(i)

for i in range(1,c+1):
    if c%i == 0:
        Diviseurs_c.append(i)
```

Avec fonctions

```
def f_Diviseurs(Nombre):
    Diviseurs = []
    for i in range(1,Nombre+1):
        if Nombre%i == 0:
            Diviseurs.append(i)
    return Diviseurs

a = 10
b = 5
c = 200

Diviseurs_a = f_Diviseurs(a)
Diviseurs_b = f_Diviseurs(b)
Diviseurs_c = f_Diviseurs(c)
```

Sans fonctions, on réécrit le code qui détermine les diviseurs autant de fois qu'il faut afin d'avoir toutes les listes nécessaires.

Lorsque l'on utilise une fonction, celle-ci a pour rôle de déterminer tous les diviseurs d'un nombre en entrée et de renvoyer cette liste. Ainsi, on l'exécute en début de code puis à chaque appel, elle donne les diviseurs du nombre qu'elle a en argument. On voit clairement l'intérêt de simplification par l'utilisation des fonctions.

Et il est parfaitement possible d'appeler une fonction dans une fonction, voire de créer une sous-fonction locale dans une fonction.

A.IV.7.c Syntaxe

A.IV.7.c.i Création de la fonction

- **Définition d'une fonction**

Pour définir une fonction, on écrit la première ligne suivante :

```
def Nom_Fonction(Argument_1, Argument_2,…,Argument_n):
```

def permet de définir une fonction dont le nom est Nom_Fonction (sans espaces et éviter les accents). Personnellement, j'aime bien appeler les fonctions par « f_Nom_Fonction » afin de bien faire la différence entre d'éventuelles variables et les fonctions associées.

Voici quelques exemples liés aux noms mal choisis :

```
def Maximum(L):
    return max(L)

L = [0,1,2]
Maximum = Maximum(L)

LL = [4,5,6]
Maximum = Maximum(L)
```

```
>>> (executing file "<tmp 1>")
Traceback (most recent call last):
  File "<tmp 1>", line 8, in <module>
    Maximum = Maximum(L)
TypeError: 'int' object is not callable
```

On a appelé la fonction Maximum et le maximul de L avec le même nom. Deux problèmes :

- On ne sait plus si Maximum est la fonction ou un nombre

On remplace la fonction par un nombre qui cause le bug lors d'un nouvel appel de la fonction qui nous dit qu'un int n'est pas une fonction…

```
L = [1,2,3]

M = max(L)
print(M)

def max(L):
    return 10

M = max(L)
print(M)

max = 50
M = max(L)
print(M)
```

```
>>> (executing file "essai.py")
3
10
Traceback (most recent call last):
  File "C:\Users\Denis DEFAUCHY\Desktop\essai.p
y", line 13, in <module>
    M = max(L)
TypeError: 'int' object is not callable
```

Au premier appel de max, la fonction python est utilisée, le résultat est bon.
Au second appel de max, la fonction python a été remplacée par notre fonction du même nom
Au 3° appel de max, la variable max a pris la place des fonctions, on ne peut plus appeler la fonction.

Après le nom de la fonction viennent des parenthèses dans lesquelles on définit si nécessaire les noms des variables qui seront les paramètres d'entrée de la fonction. Il est tout à fait possible de laisser ces parenthèses vides si aucun argument ne doit être défini.

Enfin, il ne faut pas oublier les « : » comme pour les boucles.

Remarque : On peut définir très simplement une fonction mathématique :

```
g = lambda x: x*2
```

```
>>> g(2)
4
```

- **Arguments optionnels**

Pour mettre des arguments optionnels, on définit la fonction aussi :

```
def f(*args):
    Somme = 0
    for arg in args:
        Somme += arg
    return Somme

print(f(2,1))
```

`*args` permet de dire que les arguments peuvent avoir n'importe quelle taille.

L'appelle `(f(2))` renvoie 2, `(f(2,1))` renvoie 3…

Cela revient au même que de demander une liste en argument, et d'appeler la fonction pour une liste dans laquelle on choisit les arguments :

```
def f(L):
    Somme = 0
    for Terme in L:
        Somme += Terme
    return Somme
```

- **Arguments prédéfinis**

Il est simple de définir des arguments pré définis. On leur affecte une valeur dans la parenthèse de définition de la fonction :

```
def f(x,a=0,b=0):
    return a*x+b
```

Ainsi, si les arguments ne sont pas définis lors de l'appel de la fonction, ils seront égaux à la valeur de base, ici 0.

```
>>> f(2)
0

>>> f(2,1,1)
3
```

• **Contenu**

Ensuite, il faut indenter le contenu de la fonction et mettre le code que l'on souhaite. Nous verrons dans le prochain paragraphe comment manipuler les variables dites locales ou globales liées aux fonctions.

Attention : un contenu de fonction vide empêche l'exécution d'un programme. Ecrire au minimum quelque chose du genre a=0.

• **Champs « aide »**

On peut définir un champ d'aide dans une fonction, c'est-à-dire un texte qui sera retourné si on tape dans la console « `help(f_Fonction)` »

Exemple :

```
def f_Fonction(x):
    ''' Cette fonction prend en argument un reel et renvoie son carre'''
    return x^2
```

Rq : Ne pas mettre d'accents dans le champs d'aide

Lors de l'exécution de la fonction help(), on obtient :

```
>>> help(f_Fonction)
Help on function f_Fonction in module __main__:

f_Fonction(x)

    Cette fonction prend en argument un reel et renvoie son carre
```

Le fonction « help » fonctionne sur toutes les fonctions de python.

- **Renvoie d'un objet**

Il n'est pas obligatoire qu'une fonction renvoie un objet (une valeur, une liste…). Elle peut par exemple enlever un terme dans une liste avec $L.pop$. Si l'on souhaite que la fonction renvoie un résultat, il suffit d'inscrire à la fin :

```
return Objet
```

Attention, ce que vous mettrez après ce return ne sera pas exécuté, return marque la fin de la fonction.

Pour retourner plusieurs objets :

Ne fonctionne pas	Fonctionne
`return Objet_1` `return Objet_2`	`return Objet_1,Objet_2` `return [Objet_1 Objet_2]`

Ainsi, lors de l'appel d'une fonction $f\_Fonction$ qui renvoie deux variables en sortie et quelle que soit la méthode proposée ci-dessus, on écrira lors de l'appel de la fonction :

```
a,b = f_Fonction(...)
```

```
v = f_Fonction(...)
a = v[0]
b = v[1]
```

ATTENTION : si vous appelez uniquement la fonction `f_Fonction(...)`, les variables a et b ne seront pas créées !!!

Remarque : si la fonction ne retourne rien, elle retourne en réalité un résultat qui s'appelle « None ».

- **Fin d'exécution**

On peut arrêter l'exécution d'une fonction simplement en inscrivant return et rien derrière. La fonction ne renvoie alors rien, ou plutôt, renvoie l'objet None.

A.IV.7.c.ii Appel d'une fonction – Ordre des arguments

Supposons que l'on ait défini la fonction suivante :

```
from math import sqrt
def f(x,y,z):
    Norme = sqrt(x**2+y**2+z**2)
    return Norme
```

Lors de l'appelle de la fonction, il suffit d'écrire par exemple :

```
f(1,2,3)
```

Attention à l'ordre des arguments de la fonction, en effet si l'on écrit :

```
X = 1
Y = 2
Z = 3

f(Z,Y,X)
```

La fonction va associer la valeur Z à x, Y à y et X à z !!!

A.IV.7.d Notions de variables locales et globales

A.IV.7.d.i Structure générale d'un code avec fonctions

En général, lorsque l'on crée un code avec des fonctions, on réserve tout un premier espace au début du fichier pour définir toutes les fonctions.

Ensuite, on créer le code à proprement parler, qui résout le problème traité, et qui fait appel aux fonctions précédemment créées.

```
## Définition des fonctions

def f_Fonction_1(loc_n):
    ''' Aide '''
    loc_Var = 1
    ...
    return loc_Var

def f_Fonction_2(...):
    ...
    ...
    ...

def f_Fonction_3(...):
    ...
    ...
    ...

## Programme

Sol = f_Fonction_1(...)+f_Fonction_2(...)/f_Fonction_3(...)
```

A.IV.7.d.ii Variables locales

Une variable dans la parenthèse des arguments de la fonction lors de son écriture ou créée dans une fonction est une variable locale. Dans l'exemple donnant la structure générale d'un code, `loc_Var` et `loc_n` sont des variables locales.

Une variable locale n'existe que dans la fonction dans laquelle elle est créée. Une variable locale est créée :

- A l'appel de la fonction s'il y a présence d'arguments, elle a alors le nom de l'argument tel qu'il est écrit dans la ligne `def f_Fonction(loc_n):`. Concrètement, il y a création d'une variable dans la fonction et on lui affecte la valeur associée lors de l'appel de la fonction
- Lors de la création de toute variable directement dans la fonction

On fait ce que l'on veut des variables locales tant que l'on reste dans le cadre de la fonction considérée. Elles n'existent plus à la sortie des fonctions. Tout se passe comme si, lorsque l'on appelle une fonction, on ouvrait un second Pyzo dans lequel on exécute le code de la fonction et l'on définit de nouvelles variables. A la différence près que ce second Pyzo a le droit d'aller chercher des variables existantes dans le premier (abordé plus tard), l'inverse étant impossible. A la fin de l'exécution de la fonction, c'est comme si l'on fermait le second Pyzo et toutes les variables créées dans celui-ci disparaissent.

D'une manière générale, j'aime bien mettre le nom précédé du `loc_` indiquant que la variable est créée dans la fonction afin de retenir que c'est une variable locale qui ne sera pas utilisable autre part et modifiée uniquement dans la fonction. Nous verrons qu'il n'est pas indispensable de respecter cette notation mais au début, cela vous permettra de faire attention à ce que vous manipulez.

Remarque : Attention aux noms choisis pour les variables locales. Voici un exemple d'erreur à ne pas faire :

```
def f(min):
    L = [1,2,3]
    return min(L)

f(10)
```

```
>>> (executing file "<tmp 1>")
Traceback (most recent call last):
  File "<tmp 1>", line 5, in <module>
    f(10)
  File "<tmp 1>", line 3, in f
    return min(L)
TypeError: 'int' object is not callable
```

Lorsque l'on définit l'argument min dans la fonction, on déclare que min sera une variable locale. La fonction native de python min pour minimum se retrouve donc remplacée par la variable min qui est dans le cas de l'appel de f(10) un entier. Lorsque l'on appelle min(L), on essaie d'appeler un entier…

A.IV.7.d.iii Variables globales

Toutes les variables créées dans le code principal, autrement dit toute variable qui n'est pas créée dans une fonction, est une variable globale que l'on retrouvera dans le « Workspace ». Dans l'exemple de la page précédente, `Sol` est une variable globale.

Il est possible d'appeler une variable globale dans une fonction sans la mettre en argument :

```
## Définition des fonctions

def f_Fonction():
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction()
print(Sol)
```

Si maintenant on veut changer la valeur de n localement dans la fonction :

```
## Définition des fonctions

def f_Fonction():
    n = n + 1
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction()
print(Sol)
```

Pyzo nous renvoie une erreur : UnboundLocalError: local variable 'n' referenced before assignment

Solution : si l'on veut modifier une variable globale dans une fonction **sans** la modifier en tant que variable globale, il faut la faire devenir locale en la mettant en argument de la fonction comme proposé ci-dessous :

```
## Définition des fonctions

def f_Fonction(loc_n):
    loc_n = loc_n + 1
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction(n)
print(Sol)
```

Mais attention : à la sortie de la fonction, cette variable n'est pas modifiée ! n vaudra toujours 10.

Autrement dit, dans l'exemple ci-dessous :

```
## Définition des fonctions

def test(x):
    x = 10

## Programme

x = 1
test(x)
print('x: ',x)
```

Le résultat affiché sera :

```
x:  1
```

C'est très important de s'en souvenir !!!

Si l'on souhaite modifier une variable globale dans une fonction et que cette modification soit maintenue après exécution de la fonction, il faut :

- Soit qu'elle soit locale dans la fonction en la mettant en argument, puis faire en sorte que la fonction retourne cette valeur de variable globale et que lors de l'appelle de la fonction, on affecte la nouvelle valeur de la variable à l'ancienne (`n = f_Fonction(n)`)

```
## Définition des fonctions

def f_Fonction(loc_n):
    loc_n = loc_n + 1
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme, loc_n

## Programme

n = 10
Sol, n = f_Fonction()
print(Sol)
```

- Soit la déclarer comme variable globale à l'intérieur de la fonction en écrivant `global n` et alors la modifier dans la fonction

```
## Définition des fonctions

def f_Fonction():
    global n
    n = n + 1
    loc_Somme = 0
    for i in range(n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Sol = f_Fonction()
print(Sol,n)
```

Dans ce code, on a déclaré la variable n comme une variable globale en écrivant `global n`. Ainsi, la variable n utilisée dans la fonction est la variable globale et à la fin de l'exécution du code, n vaut réellement n+1 !

A.IV.7.d.iv Gestion des variables locales et globales - Risques

• **Variables locales/globales et noms associés**

Supposons que l'on appelle une fonction `f_Fonction(a,b,c)` dans le programme principal (pas depuis une fonction) et que l'on définit la fonction avec `def f_Fonction(a,b,c):`. Il faut faire attention car les variables a, b et c ne sont pas vues de la même manière dans les deux cas :

`def f_Fonction(a,b,c):`	`Sol = f_Fonction(a,b,c)`
a, b et c sont des variables locales	a, b et c sont des variables globales
La modification de ces variables dans la fonction ne va pas changer les variables globales !	
Préférer leur donner des noms différents afin d'en être conscient : `loc_a,loc_b,loc_c`	

Prenons un exemple. Soit le code suivant, proprement rédigé avec distinction des noms des variables locales et globales :

```
## Définition des fonctions

def f_Fonction(loc_n,loc_Somme): # On ajoute les loc_n premiers entiers à
loc_Somme
    for i in range(loc_n):
        loc_Somme += i
    return loc_Somme

## Programme

n = 10
Somme = 0
Sol = f_Fonction_1(n,Somme)
print(Sol,Somme)
```

Prenons maintenant le code suivant où variables locales et globales ont le même nom :

```
## Définition des fonctions

def f_Fonction(n,Somme):
    for i in range(n):
        Somme += i
    return Somme

## Programme

n = 10
Somme = 0
Sol = f_Fonction(n,Somme)
print(Sol,Somme)
```

Dans le premier code, tout va bien, on différencie variables locales et globales. Somme vaut 0 à la fin.

Dans le second code, on a fait exprès de ne pas faire attention aux noms des variables locales dans la définition de la fonction. Ainsi, en particulier, `loc_Somme` est devenu `Somme`. Lors de l'appel de la fonction `f_Fonction`, on pense que la variable globale `Somme` qui porte le même nom que la variable locale `Somme` de la fonction changent toutes les deux. Mais à la fin, le `print(Somme)` renvoie une valeur nulle ! C'est normal.

On essaiera donc toujours de bien faire attention à changer les noms dans les variables locales, par exemple comme je le propose avec `loc_` de manière à ne pas se tromper dans notre interprétation de ce qu'il se passe.

• **Listes – Attention – Tout est global !**

Le cas des listes est quelque peu problématique vis-à-vis des variables locales/globales. En effet, rappelons que lorsque l'on écrit $LL = L$, les deux pointeurs L et LL pointent vers la même liste en mémoire.

Cela a un effet important dans une fonction, donnons l'exemple suivant :

```
## Définition des fonctions

def f_Fonction(loc_L):
    for i in range(n):
        loc_L[i] = 0

## Programme

n = 10
L = [i for i in range(n)]
Sol = f_Fonction(L)
print(Sol)
```

On croirait que la fonction a modifié la liste locale `loc_L` qui serait une copie de `L` alors qu'en réalité, elle a modifié les blocs mémoire vers lesquels pointe la liste `loc_L` qui sont les mêmes que ceux de la liste `L`.

Encore pire :

```
## Définition des fonctions

def f_Fonction(loc_L):
    loc_LL = loc_L
    for i in range(n):
        loc_LL[i] = 0
    return loc_LL

## Programme

n = 10
L = [i for i in range(n)]
Sol = f_Fonction(L)
print(Sol)
```

En créant la variable locale `loc_LL`, on espère créer une nouvelle liste que l'on va pouvoir modifier. Sauf que : La liste en variable globale L a été modifiée aussi…

On retiendra que pour les listes, tout est global et une égalité entre listes, comme on le savait, fait pointer vers les mêmes blocs mémoire.

Et le cauchemar :

```
## Définition des fonctions

def test(L):
    L = [1,2,3,4,5,6]

## Programme

L = [1,2,3]
test(L)
print('L: ',L)
```

Le résultat de ce code est…

```
L:  [1, 2, 3]
```

On a vu précédemment que modifier des variables en argument dans une fonction modifiait les variables locales mais pas les variables globales associées.

On vient de voir que les listes pointaient vers une adresse mémoire et que leur modification modifiait la variable globale associée.

Dans le code proposé ci-dessus, c'est le premier cas qui s'applique. Ecrire `L = [1,2,3,4,5,6]` consiste à traiter L comme une variable locale. On n'a pas demandé de modifier des valeurs en mémoire de L initiale mais de créer une « nouvelle » liste L. Elle devient locale dès que l'on écrit L = …

A retenir donc : Pour modifier une liste dans une fonction, il faut obligatoirement modifier ses valeurs avec par exemple L.pop(), L.append(), L[i] = 1 ...

- **Fonctions, variables locales et globales**

Soit l'exemple suivant :

```
k = 2

def f(x):
    return k*x

def g(x):
    k = 10000
    Val = 2 * f(x)
    return Val

Sol = g(3)
print(Sol)
```

La fonction f fait appel à une variable globale k.

La fonction g fait appel à la fonction f. Dans la fonction g, on définit k=10000. Dans l'ordinateur, une nouvelle variable k différente de la variable globale est créée. Elle est locale, uniquement pour g. Lors de l'appel de g(2), on fait appel à f qui recherche la variable globale k qui vaut 2. Le k=10000 est donc totalement sans effets ☹

Pour corriger cela, deux solutions :

Méthode 1 : il suffit de modifier f pour qu'elle prenne comme variable d'appel la valeur de k :

```
def f(x,k):
    return k*x

def g(x):
    k = 10000
    Val = 2 * f(x,k)
    return Val

Sol = g(3)
print(Sol)
```

La définition de k en dehors des fonctions devient alors inutile et ce code prendra bien en compte la valeur k=10000.

Méthode 2 : déclarer k global et lui affecter la valeur adéquate pour qu'elle soit utilisée dans f :

```
def f(x):
    return k*x

def g(x):
    global k
    k = 10000
    Val = 2 * f(x)
    return Val

Sol = g(3)
print(Sol)
```

A.IV.7.e Débogage des erreurs

La manipulation de variables locales dans les fonctions rend assez délicat le débogage. En effet, en cas d'arrêt du code, toutes les variables locales sont effacées et on ne peut pas simplement déterminer où est l'erreur.

Il y a donc plusieurs solutions, les deux meilleures étant :

- Créer le code de la fonction directement dans le programme afin de manipuler des variables globales accessibles après arrêt du code puis mettre le code dans une fonction lorsque tout fonctionne
- Mettre des print(…) un peu partout dans les fonctions afin d'afficher les variables lors de l'exécution de la fonction pour trouver le problème

A.IV.7.f Mise en mémoire des fonctions

A partir du moment ou une fonction a été mise en mémoire, on peut directement l'utiliser « à la main » dans la console « Shells ».

Pour mettre une fonction en mémoire à la main, il suffit de la copier dans le code et de la coller dans la console puis d'appuyer 2 fois sur « Entrée ». On peut ensuite l'utiliser.

A.IV.7.g Pile d'exécution (stack)

A.IV.7.g.i Principe

A chaque fois qu'une fonction est appelée, cet appel est stocké dans ce que l'on appelle la pile d'exécution.

Lorsqu'une fonction Fonction_1 qui appelle une seconde fonction Fonction_2 qui appelle une 3° fonction Fonction_3 est exécutée, voici schématiquement ce qu'il se passe dans la pile d'exécution :

Appel fonction 1	Appel fonction 2	Appel fonction 3
Fonction_1 - Var_Loc_1	Fonction_2 - Var_Loc_2 Fonction_1 - Val_Loc_1	Fonction_3 - Var_Loc_3 Fonction_2 - Var_Loc_2 Fonction_1 - Val_Loc_1

A chaque étape, la pile stocke les fonctions en cours dans l'ordre d'exécution ainsi que les variables locales associées. A partir du moment où la dernière fonction Fonction_3 est appelée, elle exécute son script puis la fonction Fonction_2 peut terminer son travail, puis la fonction Fonction_1 et c'est terminé.

On peut simplement voir cette pile d'exécution en intégrant une erreur dans la fonction 3. La console nous renvoie un « Traceback »

Exemple :

```
def Fonction_1(n):
    return Fonction_2(n)

def Fonction_2(n):
    return Fonction_3(n)

def Fonction_3(n):
    return (1/n)
```

La console nous renvoie :

```
>>> Fonction_1(0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 39, in Fonction_1
    return Fonction_2(n)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 42, in Fonction_2
    return Fonction_3(n)
  File "C:\Users\DDP\Dropbox\Privé\Professionnel\Cours\CPGE\Chapitres\Info IPT\IPT 2\TP\TP2 - Récursivité\TP2 - Récursivité.py", line 45, in Fonction_3
    return (1/n)
ZeroDivisionError: division by zero
```

On voit clairement qu'après appelle de Fonction_1 a été appelée Fonction_2, qui a appelé Fonction_3, lieu de l'erreur.

A.IV.8 Tracés de courbes

Nous allons apprendre à tracer des courbes. Il ne me semble pas utile de détailler l'utilité de ce paragraphe…

Il est possible d'aller loin dans les possibilités qu'offrent les options de python. Nous ne développerons ici que les bases utiles le plus généralement. L'utilisateur qui souhaitera aller plus loin pourra trouver tout ce dont il a besoin sur internet en tapant simplement ce qu'il souhaite effectuer.

Il existe différentes méthodes pour tracer des courbes, je vous en propose une que j'ai adoptée et qui fonctionne bien.

Sachez que la création de courbes sous Python nécessite d'avoir deux listes, l'une pour les abscisses, l'autre pour les ordonnées. On n'écrit donc pas comme dans les calculatrices graphiques la fonction directement. C'est là une différence assez importante qu'il faut avoir comprise.

A.IV.8.a Import de la librairie

Pour tracer des courbes, il est nécessaire d'importer la librairie associée en écrivant :

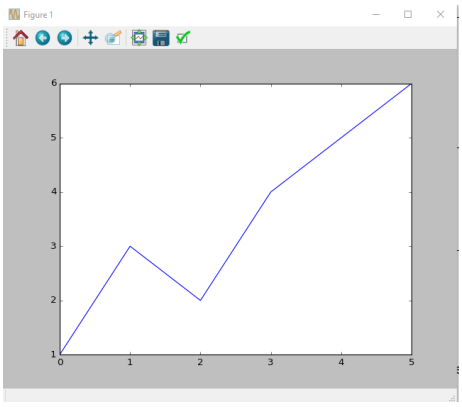
```
import matplotlib.pyplot as plt
```

Le fait de rajouter « `as plt` » permet par la suite de ne pas écrire pyplot mais juste plt. En fait, on change le nom de la fonction `pyplot` en `plt`.

A.IV.8.b Un exemple basique

```
X = [0,1,2,3,4,5]
Y = [1,3,2,4,5,6]
plt.plot(X,Y)
plt.show()
```

Le résultat est le suivant :



« `plt.plot(X,Y)` » met en mémoire un graphique affiché grace à la commande « `plt.show()` ».

A.IV.8.c Les options

Il existe une multitude d'options permettant de changer les couleurs, styles de traits, épaisseurs, titres des graphiques, titres des axes etc… Voyons ici les plus importants.

Lors de la création du plot, on peut préciser :

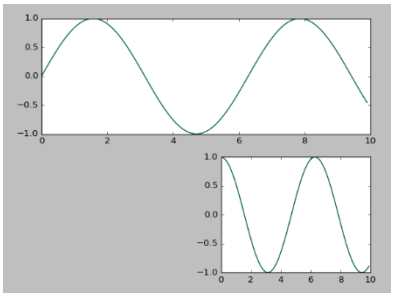
`plt.plot(X, Y, linewidth=2.0)`	On règle ainsi l'épaisseur du trait
`plt.plot(X, Y, 'ro')`	Crée un nuage de points
`plt.plot(X, Y, '--')`	Le trait est en pointillés
`plt.plot(X,Y,'r')`	On précise la couleur associée à la courbe *b: blue - g: green - r: red - c: cyan - m: magenta - y: yellow - k: black - w: white*
`plt.plot(X,Y,label="Texte")` `plt.legend()`	Ajout d'une légende à la courbe y(x) Ligne nécessaire pour afficher la légende

Après avoir créé un graphique via la commande « `plt.plot(X,Y)` » :

`plt.xlim(-2,2)`	Définit l'intervalle des abscisses de la figure
`plt.ylim(-2,2)`	Définit l'intervalle des ordonnées de la figure
`plt.axis([0, 6, 0, 20])`	Définit l'intervalle des abscisses (0 à 6) puis des ordonnées (0 à 20)
`plt.axis('equal')`	Redéfini les intervalles d'abscisses et ordonnées pour avoir un repère orthonormé
`plt.grid(True)`	Affiche la grille
`plt.title('Droite Y=X')`	Définit un titre au graphique
`plt.xlabel('Abscisses')`	Définit le nom des données en abscisses
`plt.ylabel('Ordonnées')`	Définit le nom des données en ordonnée
`plt.show()`	Affiche le graphique
`plt.close()`	Ferme la dernière figure créée/appelée/ouverte
`plt.close('all')`	Ferme toutes les figures
`plt.clf()`	Vide la dernière figure créée/appelée/ouverte sans la fermer (lors d'une simulation, il est beaucoup plus rapide de vider que de fermer/ouvrir)

Il est possible de faire apparaître plusieurs graphiques dans la même figure, par exemple :

```
from math import cos
from math import sin
n = 100
X = [i/10 for i in range(n)]
Y1 = [sin(X[i]) for i in range(n)]
Y2 = [cos(X[i]) for i in range(n)]
plt.subplot(211)
plt.plot(X,Y1)
plt.subplot(224)
plt.plot(X,Y2)
plt.show()
```



211 veut dire : séparation en 2 lignes et 1 colonnes, choix de la première cellule parmi 2
224 veut dire : séparation en 2 lignes et 2 colonnes, choix de la dernière cellule parmi 4

Google vous donnera le reste…

A.IV.8.d Gestion de plusieurs figures

Il n'existe pas une seule façon de faire. Personnellement, j'aime bien avoir la main sur les figures que je crée afin de pouvoir :

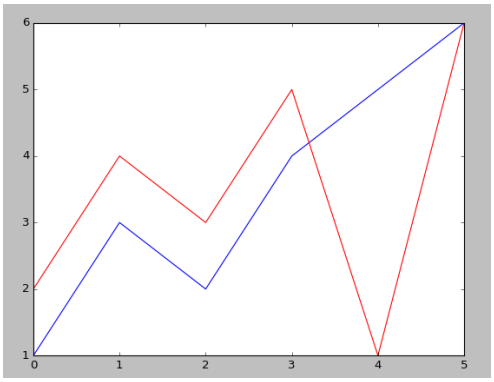
- En ouvrir autant que je le souhaite en parallèle
- Mettre à jour l'une d'elles
- Fermer l'une d'elles

Si on écrit :

```
X = [0,1,2,3,4,5]
Y = [1,3,2,4,5,6]
plt.plot(X,Y,'b')
plt.show()

X = [0,1,2,3,4,5]
Y = [2,4,3,5,1,6]
plt.plot(X,Y,'r')
plt.show()
```

On obtient :



Les deux courbes sont tracées sur le même graphique…

C'est pourquoi, à chaque fois que je fais une figure, j'utilise en premier lieu la commande :

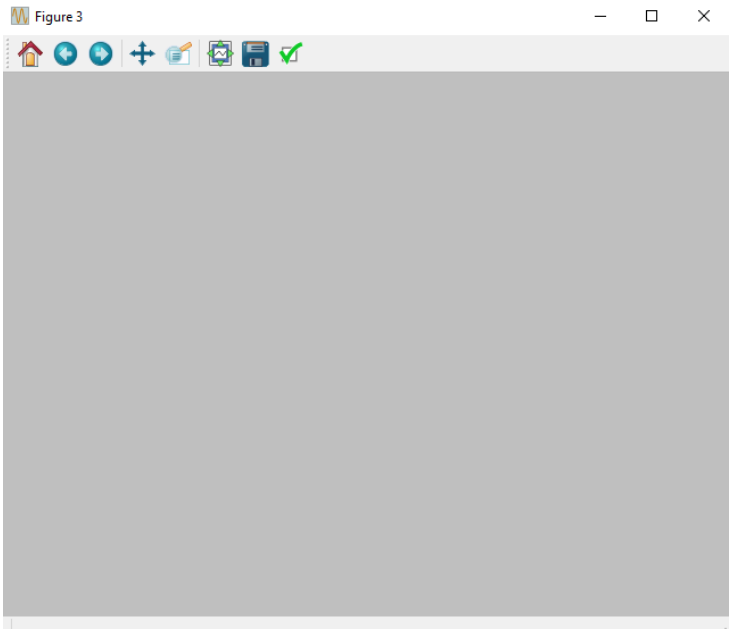
```
plt.figure(i)
```

Où `i` est un nombre entier qui définit un numéro de figure.

Ainsi, le code suivant ouvre la figure 3 :

```
plt.figure(3)
plt.show()
```

On obtient alors :



Il est alors très simple de fermer une figure parmi toutes les figures ouvertes à l'aide de la commande :

```
plt.figure(3)
plt.close()
```

De même, il est possible d'afficher une seconde courbe à la figure 3 en rappelant la figure en question :

```
plt.figure(3)
plt.plot(…)
plt.show()
```

Ou encore de vider la figure 2 en écrivant :

```
plt.figure(2)
plt.clf()
```

A.IV.8.e Objet figure

Il est possible de créer un objet associé à une figure, pour cela il suffit d'écrire par exemple :

```
Fig_1 = plt.figure(1)
```

Il n'est plus alors obligatoire de rappeler la figure avec « `plt.figure(1)` » pour la fermer par exemple, il suffit d'écrire « `plt.close(Fig_1)` ».

Cela peut avoir d'autres intérêts que nous ne détaillerons pas ici.

A.IV.8.f Le tout en une fonction

Idéalement, si vous souhaitez afficher une courbe, il est intéressant de créer une fonction qui prend en argument les deux listes de la courbe en question, et le numéro de la figure associée.

Ainsi, vous écrirez :

```
# Import librairie

from matplotlib import pyplot as plt

# Fermeture d'éventuelles fenêtres ouvertes

plt.close('all')

# Définition de la fonction

def f_courbe(X,Y,N_Fig,Legende):
    plt.figure(N_Fig)
    # Options à définir
    plt.plot(X,Y,Label=Legende)
    plt.xlabel('Abscisses')
    plt.ylabel('Ordonnées')
    plt.legend()
    plt.show()

# Tracé

X = [1,2,3]
Y = [0,1,2]
f_courbe(X,Y,1,'Legende')
```

A.IV.9 Matrices

A.IV.9.a Contexte

Une matrice est généralement représentative d'un système linéaire et c'est pour les systèmes linéaires que je vais les aborder. En effet, il est très courant d'obtenir un système linéaire lorsque l'on résout des équations scientifiques sur un solide ou un fluide par exemple.

Comme vous le savez sans doute, on peut assez simplement passer d'un système linéaire à un système matriciel équivalent, exemple :

$$\begin{cases} a_1x + b_1y + c_1z = d_1 \\ a_2x + b_2y + c_2z = d_2 \\ a_3x + b_3y + c_3z = d_3 \end{cases} \Leftrightarrow \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

Avec a_i, b_i, c_i, d_i des coefficients réels constants.

On l'écrira souvent :

$$KU = F \quad ; \quad K = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \quad ; \quad U = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad ; \quad F = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}$$

La solution de ce système s'obtient alors « très simplement » par inversion de la matrice K, si elle est inversible évidemment :

$$U = K^{-1}F$$

En effet :

$$KU = F \Leftrightarrow K^{-1}KU = K^{-1}F \Leftrightarrow U = K^{-1}F$$

Car $K^{-1}K = I$

Très simplement est entre guillemets, car selon les problèmes traités, son inversion numérique n'est pas toujours juste… Mais vous aurez l'occasion de voir ça dans vos études futures.

A.IV.9.b Fonctions de Numpy

Voyons ici les principales fonctions utiles de Numpy pour la résolution de systèmes linéaires. Au préalable, importer numpy : `import numpy as np`

Création d'un vecteur	$F = \begin{bmatrix}1\\2\\3\end{bmatrix}$; $H = \begin{bmatrix}0\\0\\0\end{bmatrix}$
	$G = \begin{bmatrix}0\\\vdots\\10\end{bmatrix}, 50\ termes$
		`F = np.array([1,2,3])`
		`G = np.linspace(0,10,50)`
		
		`H = np.zeros([3])`
		`H = np.zeros([3,1])`
		Δ Utilisation différente de H selon la syntaxe choisie

Création d'une matrice	$K = \begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix}$	`K = np.zeros([3,3])`
	$K = \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}$	`K = np.eye(3)`
	$K = \begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	`K = np.ones([3,3])`
	$K = \begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix}$	`K = np.array([[1,4,7],[2,5,8],[3,6,9]])`
Accès à un terme d'une matrice ou d'un vecteur	$K = \begin{bmatrix}1&4&7\\2&5&8\\3&\mathbf{6}&9\end{bmatrix}$ $F = \begin{bmatrix}0\\\mathbf{0}\\0\end{bmatrix}$; $G = \begin{bmatrix}0\\\mathbf{0}\\0\end{bmatrix}$	`K[2,1]` Si `F = np.zeros([3])` → `F[1]` Si `G = np.zeros([3,1])` → `G[1,0]`
Copie d'une matrice ou d'un vecteur		`B = np.copy(A)`
Transposition d'une matrice		`K.T`
Produit matriciel Produit scalaire	$\begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}30\\36\\42\end{bmatrix}$ $UV = \begin{bmatrix}1\\2\\3\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} = 14$	`np.dot(K,F)` ≠ `np.dot(F,K)` `np.dot(U,V)` = `np.dot(V,U)`
Récupération d'une ligne ou d'une partie de matrice	$K = \begin{bmatrix}1&4&7\\\mathbf{2}&\mathbf{5}&\mathbf{8}\\3&6&9\end{bmatrix}$	`K[1,:]`
	$K = \begin{bmatrix}\mathbf{1}&\mathbf{4}&7\\\mathbf{2}&\mathbf{5}&8\\3&6&9\end{bmatrix}$	`K[0:2,0:2]`
Nombres de lignes et colonnes d'un array		`l,c = np.shape(K)`
Nombre de termes d'un array		`np.size(K)`
Produit termes à termes	$U * V = \begin{bmatrix}a\\b\\c\end{bmatrix} * \begin{bmatrix}d\\e\\f\end{bmatrix} \Rightarrow \begin{bmatrix}ad\\be\\cf\end{bmatrix}$	`U*V`
Inversion d'une matrice		`np.linalg.inv(K)`
Résolution d'un système		`[x,y,z] = np.linalg.solve(K,F)`
Calcul d'un déterminant		`np.linalg.det(K)`

A.IV.9.c Exemple de résolution

Il existe plusieurs modules permettant de traiter des matrices sous Python. J'ai choisi de vous parler de « Numpy ».

Essayons de résoudre le système suivant :

$$\begin{cases} x + 2z = 1 \\ 3y + 4z = 2 \\ 5y = 3 \end{cases}$$

Soit :

$$KU = F \quad ; \quad K = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 4 \\ 0 & 5 & 0 \end{bmatrix} \quad ; \quad U = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad ; \quad F = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

On peut écrire :

Import de Numpy	`import numpy as np`

Création de la matrice K	`K = np.zeros([3,3])`
	
	`K[0,0] = 1`
	`K[0,2] = 2`
	`K[1,1] = 3`
	`K[1,2] = 4`
	`K[2,1] = 5`

Création du vecteur F	`F = np.array([1,2,3])`
Résolution	`[x,y,z] = np.linalg.solve(K,F)`

A.IV.9.d Remarques

Attention, écrire `B = ([1,2,3])` n'est pas identique à `F = np.array([1,2,3])`.

Dans le premier cas, vous n'arriverez pas à écrire `V[i,:]` car la dimension de B est uniquement 3 (pas 3,1), alors que dans le second, cela fonctionnera.

On peut créer une matrice à l lignes, c colonnes, contenant des nuplets, en écrivant :

```
Mat = np.zeros((l,c,n))
```

A.IV.10 Lecture et écriture dans un fichier

A.IV.10.a Contexte

La lecture ou l'écriture dans un fichier texte peut servir à réaliser une multitude de tâches. Toutefois, celle que vous serez amenés à réaliser le plus souvent consistera à extraire de fichiers textes des listes de données numériques issues d'un système de mesure quelconque permettant l'exportation au format texte.

Selon les logiciels utilisés, les données seront regroupées et séparées différemment. Souvent, à chaque temps de mesure ou chaque mesure sera associée une ligne.

Pour chaque ligne, les différentes données peuvent être séparées par différents « séparateurs », virgule, point-virgule, espace, tabulation… Nous allons donc apprendre à les extraire

Il est possible de faire beaucoup de choses, nous verrons ici uniquement comment :

- Lire et extraire des données sous forme de listes
- Créer un fichier et y ajouter les valeurs d'une liste

A.IV.10.b Préliminaires sur la gestion des fichiers

Dans toute la suite, j'entends par « chemin » une variable de type string qui contient soit :

- Le nom d'un fichier avec son extension :« fichier.txt »
- Le nom du fichier précédé du chemin où il est enregistré, c'est-à-dire l'ensemble des dossiers depuis C:\ jusqu'au lieu où il se trouve, chemin dans lequel on aura pris soin de doubler tous les antislash \, par exemple : « C:\\Users\\denis\\Desktop\\Exemple\\fichier.txt »

Dans le premier cas, on parle de chemin **relatif**, dans le second, on parle de chemin **absolu**.

Le chemin absolu fonctionne toujours, mais ne permet pas de faire tourner un code sur deux ordinateurs différents sans le modifier à chaque fois. Le chemin relatif est bien plus pratique, mais pour fonctionner, il faut :

- Que le fichier python qui l'appelle soit :
    - Enregistré et pas juste ouvert dans Pyzo comme nouveau document (temporaire on ne sait où dans l'ordinateur)
    - Présent dans le même répertoire que le fichier appelé, d'où la notion de chemin relatif
- Exécuter le fichier Python avec la commande F5 (exécuter entièrement le code) ce qui permet à Pyzo de savoir où chercher le document en chemin relatif

Autrement dit, les deux actions suivantes ne fonctionneront pas en relatif :

- N'exécuter qu'une portion d'un code avec Alt+Entrée
- Exécuter une commande quelconque directement dans la console

Par ailleurs, il semble que sur le réseau du Lycée, les chemins relatifs ne fonctionnent pas.

A.IV.10.c Lecture d'un fichier

A.IV.10.c.i Ouverture

Pour « ouvrir » un fichier texte sous python, on peut écrire :

```
fichier = open(Nom_Fichier, "r")
```

Quelques explications :

- A la variable `fichier` est associé le fichier texte
- L'option r veut dire « read ». On ouvre donc le fichier en lecture uniquement.
- `Nom_Fichier` est une variable contenant le chemin du fichier à ouvrir :
  - En relatif : dans le même dossier sont présent le code python et le fichier texte de nom « Texte.txt », alors il suffit d'écrire avant le code ci-dessus :
    ```
    Nom_Fichier = "Texte.txt"
    ```
  - En absolu : on veut ouvrir un fichier dans l'ordinateur à un endroit spécifique, on précise alors le chemin complet, par exemple :
    ```
    Nom_Fichier = "C:\\Users\\denis\\Desktop\\Texte.txt"
    ```
    Veiller à doubler les anti slashs

Remarque : on pourrait ne pas passer par la variable `Nom_Fichier` mais écrire directement :

```
fichier = open("Texte.txt", "r")
```

A.IV.10.c.ii Lecture des lignes

• **<u>Lecture ligne par ligne</u>**

On peut alors parcourir chacune des lignes du fichier à l'aide d'une boucle for ainsi :

```
for Ligne in fichier:
```

A la variable `Ligne` est alors associée une ligne du fichier au format str. Cette écriture permet de parcourir toutes les lignes du fichier, les unes après les autres. Attention, `Ligne` n'est pas un nombre, c'est le contenu d'une ligne du fichier !

Un retour à la ligne est spécifié en fin de ligne par `"\n"`. Ainsi, on peut tester la présence d'une ligne vide (hormis la dernière) à l'aide de la commande : « `if Lignes[i] == '\n':` ». Cela peut permettre de mettre fin à une lecture de fichier dont la fin contient plusieurs lignes vides par exemple. Pour la dernière, on peut écrire : « `if Lignes[i] == '':` ».

Si besoin, on peut ajouter un compteur afin d'identifier les numéros de lignes pour n'en traiter qu'une partie.

- **Lecture complète**

Il est possible de récupérer directement une liste des lignes d'un fichier en écrivant :

```
Liste_Lignes = fichier.readlines()
```

Il faudra toutefois faire le post traitement de chaque ligne par la suite.

Chaque élément de la liste `Liste_Lignes` est alors une chaîne de caractères contenant une des lignes du texte ouvert. Attention, la fin de chaque ligne, sauf la dernière s'il n'y a pas de retour à la ligne, contient un « \n ».

Exemple : soit le fichier texte suivant :

```
Cours d'informatique
10.2
Ligne finale
```

On remarquera que le fichier se termine après le mot « finale » sans nouvelle ligne vide, c'est-à-dire sans retour à la ligne.

En exécutant le code readlines(), on obtient :

```
>>> Liste_Lignes
["Cours d'informatique\n", '10.2\n', 'Ligne finale']
```

Il faut donc savoir enlever le « \n ». Deux cas de figure :

- Il n'y a qu'une valeur numérique avec une virgule de type « . » reconnue par python : utiliser float :

```
>>> float(Liste_Lignes[1])
10.2
```

- La virgule est « , » (on verra plus tard comment la remplacer par « . ») ou c'est un texte : le « \n » est vu comme un seul et unique caractère à la fin de la ligne, il suffit donc de récupérer dans la chaîne de caractères de n termes ses n-1 premiers :

```
>>> Ligne_0 = Liste_Lignes[0]

>>> Ligne_0
"Cours d'informatique\n"

>>> Ligne_0[:len(Ligne_0)-1]
"Cours d'informatique"
```

A.IV.10.c.iii Découpage des données de chaque ligne

Il faut alors s'adapter au cas de figure afin d'en extraire les données importantes. La fonction utile dans notre cas est la fonction « **.split** ». Voici des exemples d'utilisation :

Ligne	Commande	Résultat
`Ligne = "10 20 30"`	`Ligne.split()`	`['10', '20', '30']`
`Ligne = "10;20;30"`	`Ligne.split(";")`	

Sans arguments, il y a un découpage chaque fois qu'il y a un ou plusieurs espaces.

Avec argument, il y a découpage chaque fois que l'argument mis entre guillemets est rencontré.

On obtient alors une liste de strings. On peut alors récupérer les valeurs associées en appelant chaque terme de la liste et en le transformant en entier (int) ou flottant (float).

A.IV.10.c.iv Suppression du retour à la ligne

Chaque ligne d'un fichier contient à la fin un retour à la ligne `\n`. La dernière ligne peut d'ailleurs ne pas en avoir…

Ainsi, pour le retirer, deux solutions :

- `Ligne = Ligne[0,len(Ligne)-1]` qui ne fonctionnera que si la dernière ligne contient un retour à la ligne aussi
- `Ligne = Ligne.strip()` toujours utilisable ☺

A.IV.10.c.v Post traitement : transformer des caractères

La dernière chose importante à savoir faire est la transformation d'éventuelles virgules en points (la virgule sous python est le point …). Il suffit d'utiliser la fonction « **.replace()** »

Ligne	Commande	Résultat
`Ligne = "1,2;2,3;3,1"`	`Ligne = Ligne.replace(",",".")`	`Ligne = "1.2;2.3;3.1"`

A.IV.10.c.vi Fermeture

Après avoir lu un fichier texte, il faut le refermer sous Python avec la fonction « **.close** »:

```
fichier.close()
```

Remarque : un fichier non fermé ne se supprime plus via Windows…

A.IV.10.d Création et écriture dans un fichier

A.IV.10.d.i Ouverture

Commençons par ouvrir le fichier, voyons ici deux modes d'ouverture :

Mode ajout : ouvre le fichier et ajoute du texte	Mode écrasement : ouvre le fichier en écrasant son contenu
`fichier = open(Nom_Fichier, "a")`	`fichier = open(Nom_Fichier, "w")`

Remarque : si le fichier n'existe pas, il est créé

A.IV.10.d.ii Ajout de lignes

Pour ajouter du texte à un fichier texte, il suffit d'utiliser la commande « **.write** »

```
fichier.write(str(Liste[i]))
```

Si ce doit être une ligne, on ajoute `"\n")` à la fin pour indiquer un renvoie à la ligne :

```
fichier.write(str(Liste[i]) + "\n")
```

Pour ajouter uniquement un renvoie à la ligne, il suffit donc d'écrire :

```
fichier.write("\n")
```

A.IV.10.d.iii Fermeture

Pour finaliser un fichier texte, il faut le fermer avec la commande « **.close** »:

```
fichier.close()
```

Remarque : un fichier non fermé ne se supprime plus via Windows…

A.IV.11 Ecrire un code commenté et lisible

Maintenant que vous savez programmer, il est nécessaire que vous appreniez à rédiger proprement un code.

A.IV.11.a Conseils

- Utiliser des noms de variables clairs et parlant
- Aérez votre code avec lignes vides et espaces
- Organiser les lignes logiquement les unes après les autres
- Commenter : # Cette ligne réalise …
- Organiser : créer différentes parties :
    - # Import des librairies
    - # Définition des fonctions
    - # Programme
- A l'écrit, marquer l'indentation par des traits verticaux

A.IV.11.b Exemples

Voici deux exemples de fonctions réalisant le même calcul, vous comprendrez vite la différence.

A.IV.11.b.i Mauvaise rédaction

```
from math import cos
from math import pi
N = 100
a = []
c = []
for i in range(N):
    b = (2*pi/N)*i
    a.append(abs(cos(b)))
    c.append(b)
import matplotlib.pyplot as plt
plt.close('all')
plt.plot(c,a)
plt.show()
def f(a):
    b = 0
    for i in range(len(a)):
        b += a[i]
    return b/len(a)
d = f(a)
e = [d for i in range(N)]
plt.plot(c,e)
plt.show()
```

A.IV.11.b.ii Bonne rédaction

```
## Code proposé

'''
Discrétisation de la fonction abs(cos(x)) sur N points et calcul de sa
moyenne
Auteur: Denis DEFAUCHY
'''

## Import des librairies

from math import cos
from math import pi
import matplotlib.pyplot as plt

## Définition des fonctions

def f_Moyenne(Liste):
    Nb_Termes = len(Liste)
    Somme = 0
    for i in range(len(Liste)): # Calcul de la somme des Nb_Termes de la
liste Liste
        Terme = Liste[i]
        Somme += Terme
    Moyenne = Somme/Nb_Termes # Calcul de la moyenne de la liste Liste
    return Moyenne

## Programme

N = 100 # Nombre de points de la discrétisation
Liste_X = []
Liste_abs_cos = []

for i in range(N): # Création de la liste des valeurs de la fonction
abs(cos(x))
    X = (2*pi/N)*i # Création de la liste des abscisses sur une période
    Liste_X.append(X)
    abs_cos = abs(cos(X))
    Liste_abs_cos.append(abs_cos)

Moyenne = f_Moyenne(Liste_abs_cos) # Calcul de la moyenne de la fonction
abs(cos(x))
Liste_Moyenne = [Moyenne for i in range(N)] # Création de la liste pour
tracer la moyenne

# tracé des courbes

plt.close('all') # Fermeture des courbes déjà ouvertes
plt.plot(Liste_X,Liste_abs_cos) # Ajoute de la courbe de abs(cos(x))
plt.plot(Liste_X,Liste_Moyenne) # Ajout de la courbe de la moyenne
plt.show() # Affichage du graphique
```

A.IV.11.c Pourquoi tout cela

- En prépa
  - Vous allez être évalués à l'écrit…
  - Vous ne serez pas là pour expliquer ce que fait votre code
  - Les correcteurs partiront du principe que les enseignants vous apprennent à bien rédiger – S'ils se cassent la tête à comprendre un code, ils passeront sans mettre de points même si c'est juste, partant de ce principe
- Ensuite
  - Dans le milieu professionnel, votre code doit être lisible par les autres
  - Et surtout, par vous-même plusieurs mois ou années après…

A.V. Application des bases

A.V.1 Listes et chaînes de caractères

Le programme d'IPT mentionne le calcule d'extrema d'une liste, de sa moyenne et de sa variance. Nous aborderons ces algorithmes en TD/TP, nous ne donnerons donc ici que les principes de ces algorithmes.

A.V.1.a Extrema – Moyenne – Variance

A.V.1.a.i Recherche d'un extrema

Le principe de la recherche d'un minimum/maximum dans une liste est très simple. On peut se rapporter à un paquet de copies dans lequel on cherche la note la plus basse/haute.

Pour un maximum, par exemple, on réalise les opérations suivantes :

- On retient la première note du paquet, par exemple 12
- On passe les copies les unes après les autres, et dès qu'il y a une note supérieure à la première, par exemple 14, on change la note enregistrée dans notre mémoire en 14, et on continue
- A la fin du paquet, la note maximale est en mémoire

A.V.1.a.ii Calcul de la moyenne

Calculer une moyenne consiste à sommer les termes d'une liste les uns après les autres, puis à diviser le résultat obtenu par le nombre de termes. Nous avons eu l'occasion de mettre en pratique ce calcul dans le TP sur le lissage d'une courbe de poids obtenue par une balance Withings.

A.V.1.a.iii Calcul de la variance

La variance est la moyenne de l'écart au carré entre chaque valeur d'une liste et sa moyenne.

Soit N le nombre de termes appelés $L_i, i \in (1, N)$ d'une liste L de moyenne M :

$$V = \frac{1}{N}\sum_{i=1}^{N}(L_i - M)^2$$

En termes de programmation, il suffit donc par exemple de :

- Créer une fonction de calcul de moyenne permettant de calculer la moyenne M de L
- Créer une fonction qui calcul une nouvelle liste L' correspondant à la liste L à laquelle on a soustrait M et on élève au carré le résultat
- Appeler la fonction de calcul de moyenne sur L'

A.V.1.b Recherche d'un mot

A.V.1.b.i Principe

Soit une chaîne de caractères $Texte$ de T termes, par exemple "$Je\ suis\ sûr\ que\ je\ suis\ Suisse$"

On recherche dans cette chaîne de caractère un $Motif$ de M lettres, par exemple "$suis$"

Le principe de cette recherche consiste à :

- Placer le motif tel que sa première lettre soit à une position d'indice i allant de 0 à $T - M$ inclus, soit $range(T - M + 1)$!
- Créer un indice j initialisé à 0 pour chaque valeur de i
- Créer une boucle qui incrémente j tant qu'il y a concordance entre le caractère en position i du texte et le caractère en position j du motif
- Incrémenter un compteur chaque fois que la concordance est vérifiée pour les M caractères du motif
- Renvoyer le compteur à la fin

A.V.1.b.ii Exemple

	J	e		s	u	i	s		s	û	r		q	u	e		j	e		s	u	i	s		S	u	i	s	s	e
$i = 0$	s	u	i	s																										
$i = 3$				s	u	i	s																							
$i = 8$									s	u	i	s																		
$i = 8$																									s	u	i	s		
$i = 24$																											s	u	i	s

Voyons les détails ci-dessous :

- Lorsque $i = 0$, dès la première valeur de $j = 0$, la lettre 0 du Texte « J » n'est pas la même que la lettre 0 du motif « s », on passe au i suivant
- Lorsque $i = 3$
  - à la première valeur de $j = 0$, la lettre 0 du Texte « s » concorde avec la lettre 0 du motif « s », on incrémente j qui vaut 1
  - la lettre 1 du Texte « u » concorde avec la lettre 1 du motif « u » , on incrémente j qui vaut 2
  - la lettre 2 du Texte « i » concorde avec la lettre 2 du motif « i » , on incrémente j qui vaut 3
  - la lettre 3 du Texte « s » concorde avec la lettre 3 du motif « s » , on incrémente j qui vaut 4
  - j étant supérieur au dernier indice du motif, on incrémente le compteur (occurrence du motif) et on passe au i suivant
- Et ainsi de suite

Remarque : Attention, lettres majuscules et minuscules sont des caractères différents, l'algorithme appliqué à l'exemple ci-dessus renverra 2 !

A.V.2 Dichotomie

Le principe de la dichotomie est de diviser le domaine de recherche par 2 à chaque itération.

A.V.2.a Recherche dans un tableau trié

A.V.2.a.i Principe

Soit une liste (un tableau) L de N éléments. Appelons E l'élément recherché supposé présent une seule fois dans L.

Le principe de la recherche par dichotomie dans une liste triée est le suivant :

- Définir la plage d'indices de recherche notée $[I_g, I_d]$ (pour Indice gauche, Indice droite) qui au départ vaut $[0, N-1]$
- Déterminer l'indice milieu I_m de la plage de recherche : $I_m = \text{int}\left(\frac{I_g+I_d}{2}\right) = (I_g + I_d)\%2$
- Déterminer la valeur E_m de l'élément de la liste L à l'indice I_m
- Comparer la valeur E_m à la valeur recherchée E et, la liste étant triée :
  - Si $E = E_m$, l'élément recherché a été trouvé, son indice est I_m
  - Si $E < E_m$, définir la nouvelle plage de recherche à $[I_g, I_m - 1]$
  - Si $E > E_m$, définir la nouvelle plage de recherche à $[I_m + 1, I_d]$
- Répéter la procédure jusqu'à ce que l'élément soit obtenu (on a supposé qu'il existait)

A.V.2.a.ii Exemple

Soit le tableau trié suivant :

1	3	7	11	15	22	25	31	36	47	52	59	66	71	82	83	94	100	110	111
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

Recherchons la valeur $E = 25$.

D'après les notations précédentes : $I_g = 0, I_d = 19 \Rightarrow I_m = 9 \Rightarrow E_m = 47$. Comme $E < E_m$, le nouvel intervalle de recherche est $[0,8]$

1	3	7	11	15	22	25	31	36	47	52	59	66	71	82	83	94	100	110	111
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

$I_g = 0, I_d = 8 \Rightarrow I_m = 4 \Rightarrow E_m = 15$. Comme $E > E_m$, le nouvel intervalle de recherche est $[5,8]$

1	3	7	11	15	22	25	31	36	47	52	59	66	71	82	83	94	100	110	111
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

$I_g = 5, I_d = 8 \Rightarrow I_m = 6 \Rightarrow E_m = 25$. Comme $E = E_m$, on a trouvé l'élément recherché. Son indice est $I_m = 6$ et c'est le 7° élément de L

1	3	7	11	15	22	25	31	36	47	52	59	66	71	82	83	94	100	110	111
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*

A.V.2.a.iii Complexité

Comme nous l'avons vu lors du calcul de complexités en temps des algorithmes dans le cas d'une dichotomie sur une liste de N termes, le nombre d'opérations n permettant à la fin d'aboutir à un seul terme dans la liste est solution de l'équation :

$$\frac{N}{2^n} = 1 \Leftrightarrow 2^n = e^{n\ln 2} = N \Leftrightarrow n = \frac{\ln N}{\ln 2}$$

Ainsi, la complexité d'un algorithme en dichotomie est en $o(\ln N)$. C'est bien évidemment moins couteux qu'une recherche sur les termes de la liste les uns après les autres, qui dans le pire des cas est en $o(N)$.

A.V.2.b Recherche du zéro d'une fonction

A.V.2.b.i Contexte

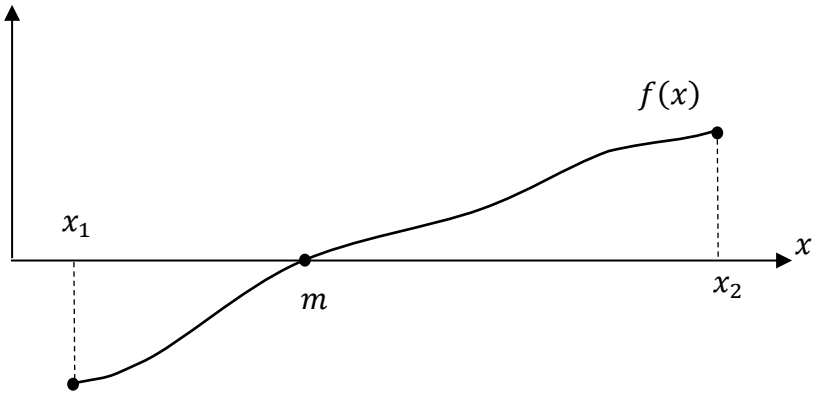
Soit une fonction $f(x)$ définie et continue sur un intervalle $[x_1, x_2]$ telle qu'il existe m tel que :

$$\forall a < m, \forall b > m, f(a)f(b) < 0$$

On se placera ici dans le cas où la fonction est monotone sur l'intervalle d'étude $[x_1, x_2]$.

Autrement dit, la fonction possède un signe sur $[x_1, m]$ et le signe opposé sur $[m, x_2]$.

Exemple :



A.V.2.b.ii Objectif

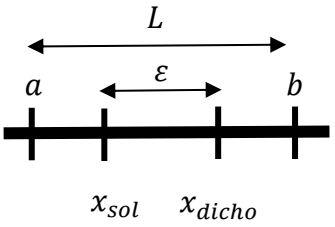
On souhaite déterminer une solution approchée de l'équation :

$$f(x) = 0$$

Appelons x_{sol} la solution exacte de cette équation. Par dichotomie, nous allons déterminer x_{dicho} tel que :

$$x_{sol} \in [a, b], x_{dicho} \in [a, b]$$

$$L = |b - a| \leq Critere \Rightarrow \varepsilon = |x_{dicho} - x_{sol}| \leq ou < Critere$$



Où $Critere$ est un réel définissant la précision de la solution obtenue.

A.V.2.b.iii Principe

Le principe de cette recherche par dichotomie est le suivant :

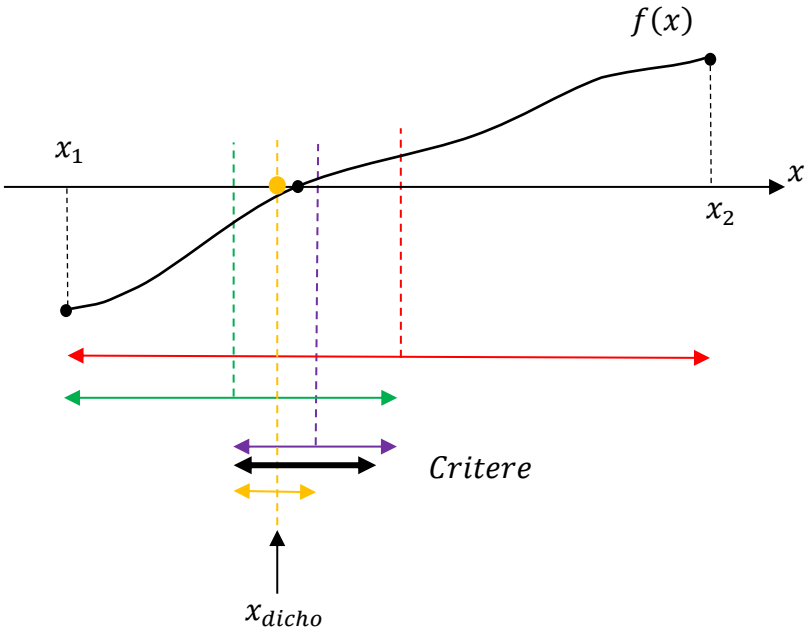
- Vérifier l'existence de la solution (facultatif) sur $[x_1, x_2]$
- Déterminer l'abscisse au centre de l'intervalle x_m et calculer son image par $f_m = f(x_m)$
- Identifier dans quel intervalle $[x_1, x_m]$ ou $[x_m, x_2]$ se trouve la solution de $f(x) = 0$
- Définir le nouvel intervalle de recherche $[x_1, x_2]$ comme celui dans lequel la solution existe
- Continuer tant que l'intervalle de recherche est de largeur supérieure au critère précisé
- Renvoyer une valeur de x dans l'intervalle obtenu lorsque le critère est vérifié, en général on renverra la valeur au centre x_m

Remarque : la solution $f(x) = 0$ existe sur l'intervalle $[x_1, x_2]$ si et seulement si $f(x_1)f(x_2) \leq 0$ – Le « ou égale » est important

A.V.2.b.iv Exemple

Soit un critère tel que la longueur de l'intervalle final soit inférieure à la longueur de la flèche ci-dessous :

On procède par itérations successives à la division par 2 de l'intervalle de recherche :



A.V.2.b.v Remarque

Il est envisageable de proposer un critère d'arrêt sur les ordonnées $|f(x)|$ plutôt que sur les abscisses.

A.V.3 Newton

La méthode de Newton est une seconde méthode de résolution d'équation de la forme $f(x) = 0$ qui converge plus vite que la méthode de dichotomie vers la solution recherchée.

A.V.3.a Contexte

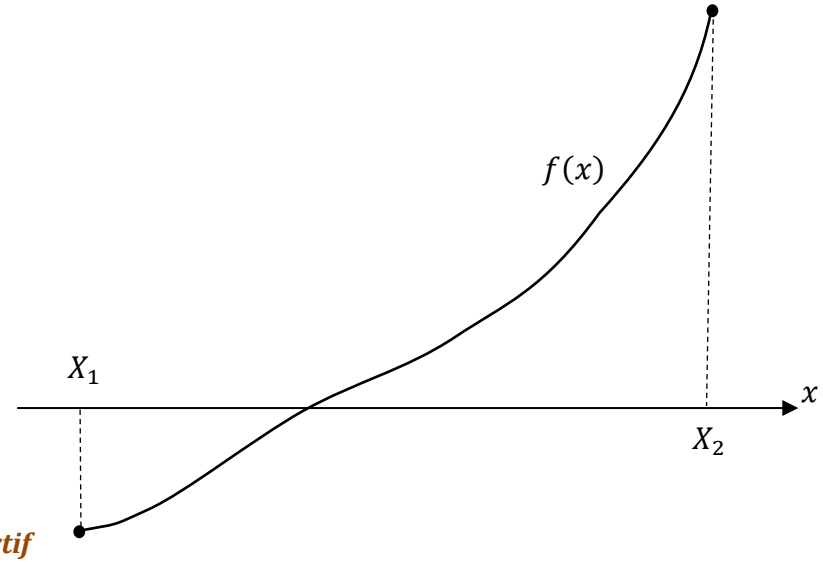
Soit une fonction $f(x)$ définie, continue et dérivable sur un intervalle $[X_1, X_2]$ telle qu'il existe m tel que :

$$\forall a < m, \forall b > m, f(a)f(b) < 0$$

On se placera ici dans le cas où la fonction est monotone sur l'intervalle d'étude $[X_1, X_2]$.

Autrement dit, la fonction possède un signe sur $[X_1, m]$ et le signe opposé sur $[m, X_2]$.

Exemple :



A.V.3.a.i Objectif

On souhaite déterminer une solution approchée de l'équation :

$$f(x) = 0$$

Contrairement à la résolution par Dichotomie vue au paragraphe précédent, nous ne sommes pas en mesure d'être sûrs que la solution approchée par la méthode de Newton est à une distance précise de la solution réelle.

Nous allons suivre l'écart entre deux solutions successives et proposer un critère d'arrêt lorsque deux solutions successives sont « assez proches ».

A.V.3.a.ii Principe

Le principe de la méthode de Newton consiste à approcher la courbe de $f(x)$ avec sa tangente $T_0(x) = f(x_0) + f'(x_0)(x - x_0)$ en un point initial x_0 choisi arbitrairement. On déterminer alors l'abscisse du point d'intersection $(x_1, 0)$ de cette tangente avec l'axe des abscisses, soit $T_0(x_1) = 0$.

On procède alors ainsi par itérations :

$$x_{i+1}/T_i(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) = 0$$

Soit :

$$f(x_i) + f'(x_i)x_{i+1} - x_i f'(x_i) = 0$$

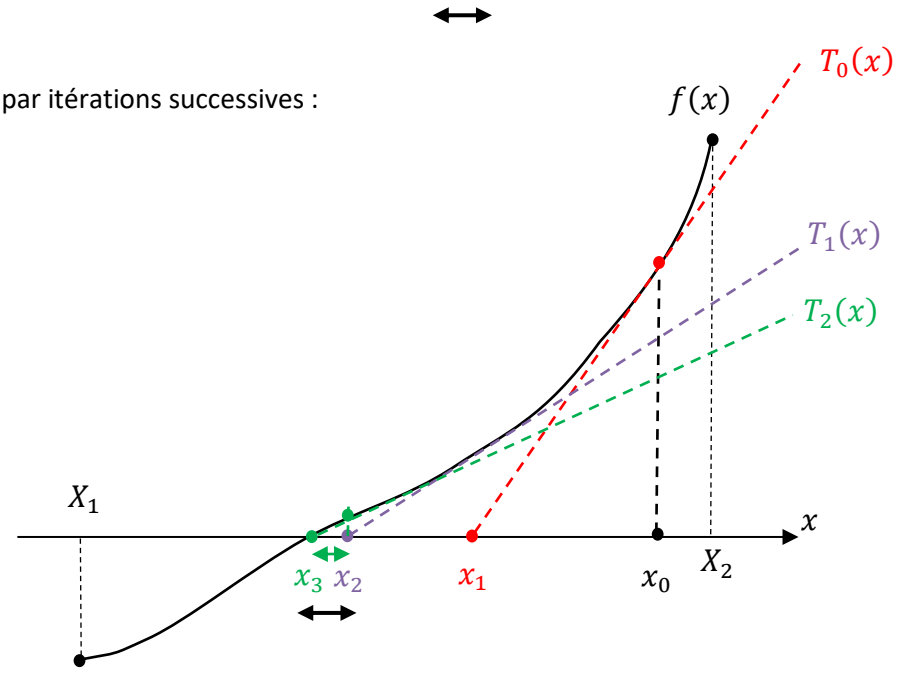
$$x_{i+1} = \frac{x_i f'(x_i) - f(x_i)}{f'(x_i)} = x_i - \frac{f(x_i)}{f'(x_i)}$$

On procède alors ainsi jusqu'à ce que :

$$|x_{i+1} - x_i| \leq ou < Critere$$

A.V.3.a.iii Exemple

Soit un critère tel que la longueur de l'intervalle final soit inférieure à la longueur de la flèche ci-dessous :



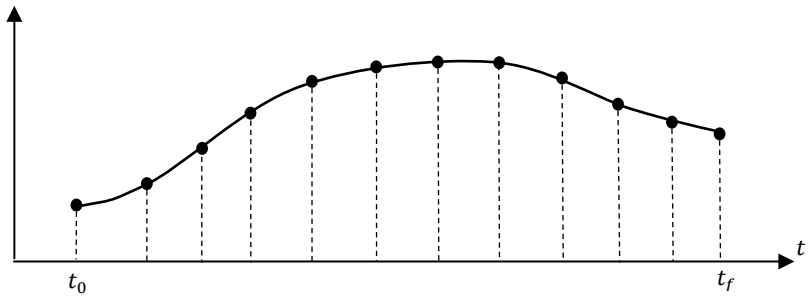
On procède par itérations successives :

A.V.3.a.iv Remarque

Il est envisageable de proposer un critère d'arrêt sur les ordonnées $|f(x)|$ plutôt que sur les abscisses.

A.V.4 Intégration numérique

Soit le signal échantillonné e suivant contenant n valeurs à partir du temps t_0 :



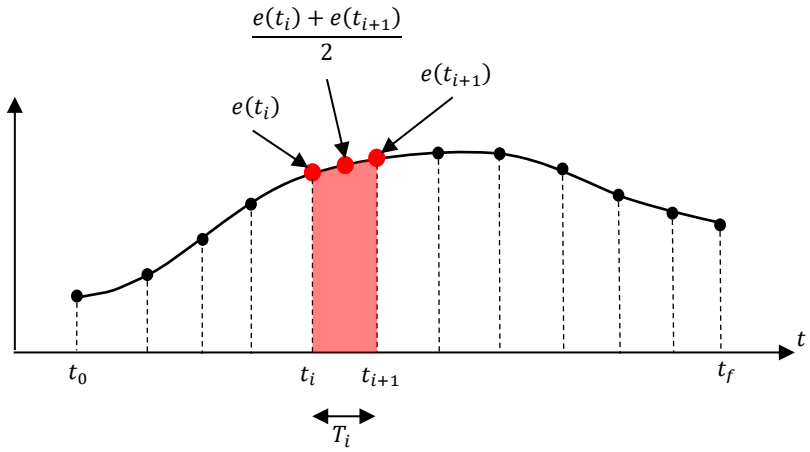
On veut :

$$\int_{t_0}^{t_f} e(t)dt$$

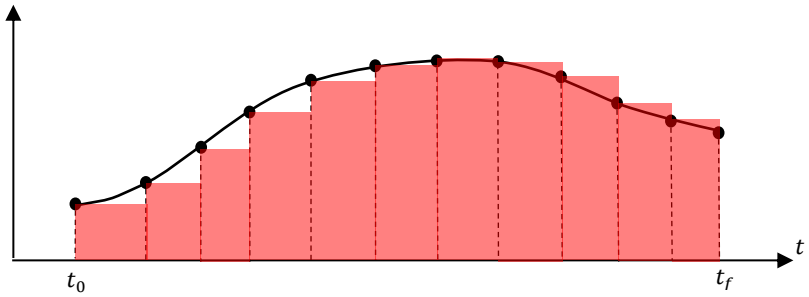
Appelons t_i et t_{i+1} les temps de part et d'autre de chaque intervalle.

On va sommer les aires rectangles sur chaque intervalle $[t_i, t_{i+1}]$, de largeur $T_i = t_{i+1} - t_i$ et de hauteur, soit :

- Valeur à gauche :
$$e(t_i)$$
- Valeur à droite :
$$e(t_{i+1})$$
- Valeur centrée :
$$\frac{e(t_i) + e(t_{i+1})}{2}$$



A.V.4.a Valeur à gauche – Méthode des rectangles

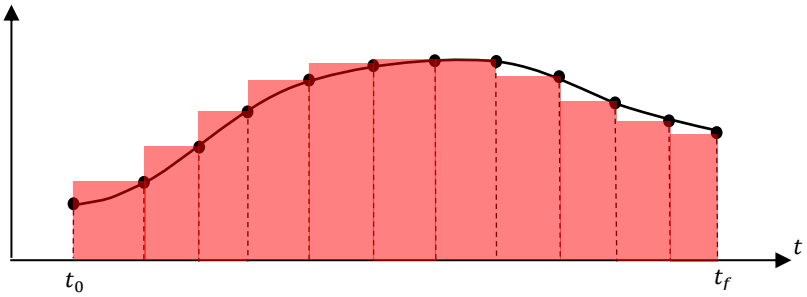


$$\mathcal{A} = \int_{t_0}^{t_f} e(t)dt = \sum_{i=0}^{n-1} T_i e(t_i) = \sum_{i=0}^{n-1} (t_{i+1} - t_i) e(t_i)$$

Attention à ne bien prendre que $n-1$ valeurs !

On voit qu'il y a sous-estimation de la courbe lorsqu'elle est croissante et surestimation lorsqu'elle est décroissante.

A.V.4.b Valeur à droite - Méthode des rectangles



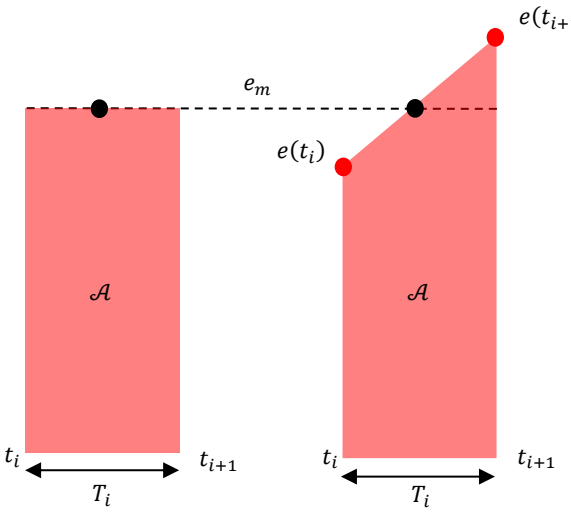
$$\mathcal{A} = \int_{t_0}^{t_f} e(t)dt = \sum_{i=0}^{n-1} T_i e(t_{i+1}) = \sum_{i=0}^{n-1} (t_{i+1} - t_i) e(t_{i+1})$$

Attention à ne bien prendre que $n-1$ valeurs !

On voit qu'il y a surestimation de la courbe lorsqu'elle est croissante et sous-estimation lorsqu'elle est décroissante.

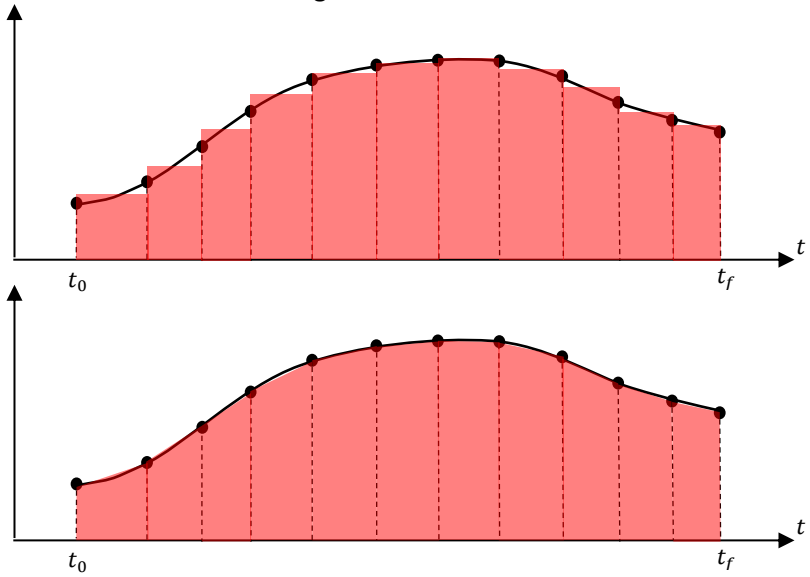
A.V.4.c Valeur centrée – Méthode des trapèzes

Le calcul d'aires en tenant compte de la valeur centrée sur le segment de largeur T_i revient calculer les aires de trapèzes, d'où le nom de méthode des trapèzes :



$$e_m = \frac{e(t_i) + e(t_{i+1})}{2}$$

Les deux surfaces ci-dessus ont des aires égales.



$$\mathcal{A} = \int_{t_0}^{t_f} e(t)dt = \sum_{i=0}^{n-1} T_i \frac{e(t_i) + e(t_{i+1})}{2} = \sum_{i=0}^{n-1} (t_{i+1} - t_i) \frac{e(t_i) + e(t_{i+1})}{2}$$

Attention à ne bien prendre que $n - 1$ valeurs !

On voit que cette méthode compense à peu près la surestimation et la sous-estimation de e sur chacun des intervalles de largeur T_i. Elle sera donc privilégiée.

A.V.4.d Remarque

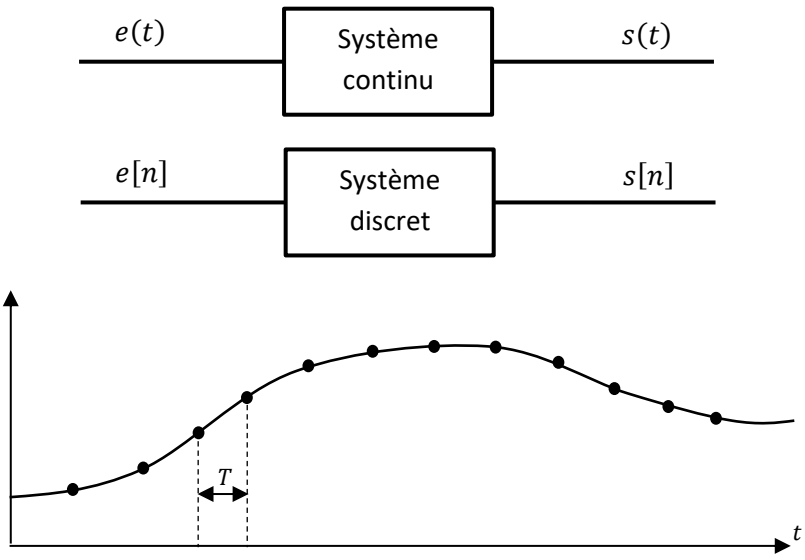
Plus T_i sera petit, soit plus il y aura de points, plus les résultats seront proches de la réalité.

A.VI. Simulation physique de phénomènes

A.VI.1 Discrétisation des problèmes

Jusqu'ici, nous avons étudié des systèmes continus. Les variables traitées étaient des fonctions continues du temps.

Dans bon nombre de systèmes, les variables sont échantillonnées et on ne connaît qu'une liste de valeurs à différents temps. On parle de variables discrètes.



Cette figure représente un signal continu en fonction du temps sur lequel ont été prises des valeurs à différents temps espacés d'un temps T supposé ici constant et appelé « période d'échantillonnage ».

A.VI.2 Problèmes dynamiques à une dimension

A.VI.2.a Dérivation de variables discrètes – Euler - Taylor

Soit la variable discrète e.

Voyons comment déterminer une approximation de ses dérivées successives. On parle de différences finies, de méthode d'Euler, de développement de Taylor.

A.VI.2.a.i Dérivée première

- **Euler explicite**

Proposons le développement de Taylor de la variable e à l'ordre 1

$$e(t+T) = e(t) + T\frac{de(t)}{dt} + o(T)$$

On a donc :

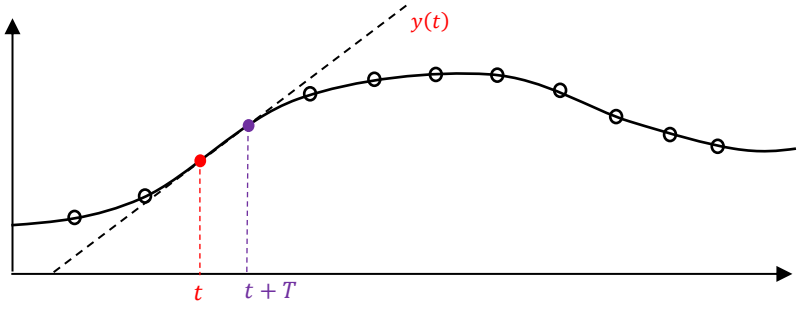
$$e(t+T) \approx e(t) + T\frac{de(t)}{dt}$$

$$T\frac{de(t)}{dt} = e(t+T) - e(t)$$

$$\frac{de(t)}{dt} = \frac{e(t+T) - e(t)}{T}$$

Cette approximation de la dérivée correspond à une méthode Euler Explicite.

Cela revient à approcher la dérivée à un instant t en utilisant la valeur à l'instant t et la valeur à l'instant $t + dt$



$$y'(t) = \frac{e(t+T) - e(t)}{T}$$

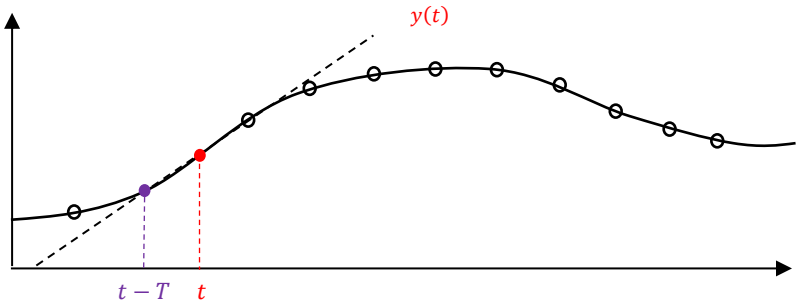
La dérivée à l'instant t est approchée à l'aide de la valeur en t et après

Comme $e(t+T) = Ty'(t) + e(t)$, on trouve la prochaine valeur de $e(t+T)$ en fonction uniquement des anciennes, on dit que ce calcul est explicite.

• **Euler implicite**

On peut aussi associer la dérivée en $t+dt$ à cette expression :

$$y'(t+T)=\frac{e(t+T)-e(t)}{T}$$



Cela revient à écrire :

$$y'(t)=\frac{e(t)-e(t-T)}{T}$$

Cette approximation de la dérivée correspond à une méthode Euler implicite.

La dérivée à l'instant t est approchée à l'aide de la valeur en t et avant.

Comme $e(t+T)=Ty'(t+T)+e(t)$, on trouve la prochaine valeur de $e(t+T)$ en fonction de la dérivée à ce nouveau temps $y'(t+T)$, on dit que ce calcul est implicite.

• **Bilan**

Euler explicite	Euler implicite
$y'(\boldsymbol{t})=\frac{e(\boldsymbol{t+T})-e(\boldsymbol{t})}{T}$	$y'(\boldsymbol{t+T})=\frac{e(\boldsymbol{t+T})-e(\boldsymbol{t})}{T}$ $y'(t)=\frac{e(t)-e(t-T)}{T}$
Nouvelle valeur dépendant de l'ancienne dérivée : Explicite	Nouvelle valeur dépendant de la nouvelle dérivée : Implicite

A.VI.2.a.ii Dérivée seconde

On trouve plusieurs méthodes pour estimer le dérivée seconde d'une variable discrète.

- **Taylor à l'ordre 2**

Proposons deux développements de Taylor de la variable e à l'ordre 2

$$e(t+T) = e(t) + T\frac{de(t)}{dt} + \frac{T^2}{2}\frac{d^2e(t)}{dt^2} + o(T^2)$$

$$e(t-T) = e(t) - T\frac{de(t)}{dt} + \frac{T^2}{2}\frac{d^2e(t)}{dt^2} + o(T^2)$$

Faisons la somme de ces deux expressions :

$$e(t+T) + e(t-T) = 2e(t) + T^2\frac{d^2e(t)}{dt^2}$$

Soit :

$$\frac{d^2e(t)}{dt^2} = \frac{e(t+T) - 2e(t) + e(t-T)}{T^2}$$

Cette approximation calcul la dérivée seconde de manière centrée autour du temps t

- **Double Euler explicite**

Ecrivons l'expression de la dérivée seconde de e avec la méthode d'Euler Explicite vue précédemment :

$$\frac{d^2e(t)}{dt^2} = \frac{\frac{de(t+T)}{dt} - \frac{de(t)}{dt}}{T}$$

On a par ailleurs :

$$\frac{de(t)}{dt} = \frac{e(t+T) - e(t)}{T}$$

Soit

$$\frac{d^2e(t)}{dt^2} = \frac{\frac{e(t+2T) - e(t+T)}{T} - \frac{e(t+T) - e(t)}{T}}{T}$$

$$\frac{d^2e(t)}{dt^2} = \frac{e(t+2T) - e(t+T) - e(t+T) + e(t)}{T^2}$$

$$\frac{d^2e(t)}{dt^2} = \frac{e(t+2T) - 2e(t+T) + e(t)}{T^2}$$

On remarque ici que l'approximation de la dérivée est faite avec les valeurs en t et après.

- **Double Euler implicite**

Ecrivons l'expression de la dérivée seconde de e avec la méthode d'Euler Explicite vue précédemment :

$$\frac{d^2e(t)}{dt^2} = \frac{\frac{de(t)}{dt} - \frac{de(t-T)}{dt}}{T}$$

On a par ailleurs :

$$\frac{de(t)}{dt} = \frac{e(t) - e(t-T)}{T}$$

Soit

$$\frac{d^2e(t)}{dt^2} = \frac{\frac{e(t) - e(t-T)}{T} - \frac{e(t-T) - e(t-2T)}{T}}{T}$$

$$\frac{d^2e(t)}{dt^2} = \frac{e(t) - 2e(t-T) + e(t-2T)}{T^2}$$

On remarque ici que l'approximation de la dérivée est faite avec les valeurs en t et avant.

• **Mélange implicite – explicite**

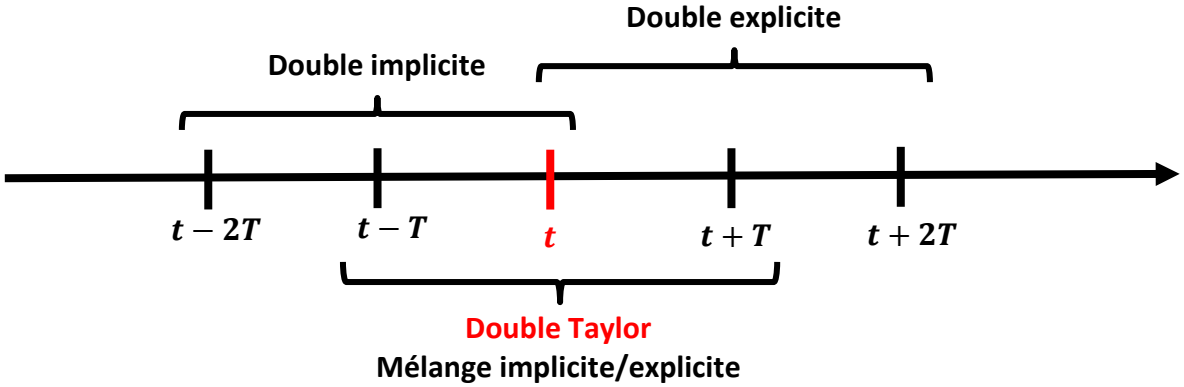
On peut aussi mélanger les méthodes :

Implicite sur la dérivée seconde et explicite sur la dérivée première	Explicite sur la dérivée seconde et implicite sur la dérivée première
$$\frac{d^2e(t)}{dt^2}=\frac{\frac{de(t+T)}{dt}-\frac{de(t)}{dt}}{T}$$ $$\frac{de(t)}{dt}=\frac{e(t)-e(t-T)}{T}$$ $$\frac{d^2e(t)}{dt^2}=\frac{\frac{e(t+T)-e(t)}{T}-\frac{e(t)-e(t-T)}{T}}{T}$$ $$\frac{d^2e(t)}{dt^2}=\frac{e(t+T)-2e(t)+e(t-T)}{T^2}$$	$$\frac{d^2e(t)}{dt^2}=\frac{\frac{de(t)}{dt}-\frac{de(t-T)}{dt}}{T}$$ $$\frac{de(t)}{dt}=\frac{e(t+T)-e(t)}{T}$$ $$\frac{d^2e(t)}{dt^2}=\frac{\frac{e(t+T)-e(t)}{T}-\frac{e(t)-e(t-T)}{T}}{T}$$ $$\frac{d^2e(t)}{dt^2}=\frac{e(t+T)-2e(t)+e(t-T)}{T^2}$$

Dans les deux cas, on retrouve l'approximation centrée issue de l'application de deux développements de Taylor à l'ordre 2.

• **Conclusion**

La dérivée en t s'exprime en fonction de valeurs de la fonction en :



On veut généralement trouver $f(t+T)$ connaissant $f(t)$ et $f(t-T)$. On préfèrera donc écrire proprement un double développement de Taylor

A.VI.2.a.iii Dérivées d'ordre supérieurs

On pourra procéder de la même manière pour obtenir des dérivées d'ordres supérieurs

A.VI.2.a.iv Précision des solutions

Plus T sera petit, plus les résultats seront proches de la réalité, la dérivée réelle étant la limite de nos formules quand T tend vers 0…

A.VI.2.b Equations du premier et second ordre

A.VI.2.b.i Equation du premier ordre

Soit l'équation $y'(t) = f\big(t, y(t)\big)$

Discrétisons $y'(t)$ avec la méthode d'Euler explicite :

$$y'(t) = \frac{y(t+dt) - y(t)}{dt}$$

On a donc :

$$y(t+dt) = y(t) + y'(t)dt$$

Ou encore :

$$y(t+dt) = y(t) + f\big(t, y(t)\big)dt$$

A partir du moment où vous avez défini la fonction $f\big(t, y(t)\big)$, il reste alors à programmer une résolution entre deux valeurs de temps t_1 et t_2 en définissant la valeur initiale de $y(t_1)$.

A.VI.2.b.ii Equation du second ordre

Soit l'équation

$$y''(t) = f\big(t, y(t), y'(t)\big)$$

Nous pourrions résoudre cette équation comme vu précédemment en introduisant un développement de Taylor à l'ordre 2 pour $y''(t)$. Voici comment faire autrement.

Soit le vecteur $V(t) = \begin{pmatrix} y(t) \\ y'(t) \end{pmatrix}$

On peut proposer une discrétisation de $y'(t)$ et $y''(t)$ à l'ordre 1 :

$$y'(t) = \frac{y(t+dt) - y(t)}{dt} \quad ; \quad y''(t) = \frac{y'(t+dt) - y'(t)}{dt}$$

On a alors :

$$\begin{cases} y(t+dt) = y(t) + y'(t)dt \\ y'(t+dt) = y'(t) + y''(t)dt = y'(t) + f\big(t, y(t), y'(t)\big)dt \end{cases}$$

$$V(t+dt) = V(t) + dt \begin{pmatrix} y'(t) \\ f\big(t, y(t), y'(t)\big) \end{pmatrix}$$

A partir du moment où vous avez défini la fonction $f\big(t, y(t), y'(t)\big)$, il reste alors à programmer une résolution entre deux valeurs de temps t_1 et t_2 en définissant la valeur initiale de $y(t_1)$ et $y'(t_1)$.

Exemple :

Soit une masse soumise à la gravité dont on veut déterminer les positions successives en chute libre sous l'action de la gravité et d'une force de frottement fluide. L'application du PFD donne :

$$mg - \mu v = ma$$

$$ma + \mu v = mg$$

$$m\frac{d^2z(t)}{dt^2} + \mu\frac{dz(t)}{dt} = mg$$

$$mz''(t) + \mu z'(t) = mg$$

$$mz''(t) = mg - \mu z'(t)$$

On a donc :

$$f\big(t, z(t), z'(t)\big) = mg - \mu z'$$

On trouve alors la nouvelle position $z(t+dt)$ en $t+dt$, connaissant $z(t)$ et $z'(t)$, en calculant :

$$\begin{pmatrix} z(t+dt) \\ z'(t+dt) \end{pmatrix} = \begin{pmatrix} z(t) \\ z'(t) \end{pmatrix} + dt \begin{pmatrix} z'(t) \\ f\big(t, z(t), z'(t)\big) \end{pmatrix}$$

A.VI.2.c Modélisation par équations aux différences

A la différence des méthodes vues au paragraphe précédent pour la résolution d'équations simples de la forme $y'(t) = f\big(t, y(t)\big)$ ou $y''(t) = f\big(t, y(t), y'(t)\big)$, on va pouvoir ici résoudre tous types d'équations différentielles ☺. Au concours, c'est plus généralement ce type de fonctions simples qui sont proposées.

La méthode que nous allons aborder ici est toutefois utilisable aussi pour les fonctions ci-dessus !

A.VI.2.c.i Principe

Soit un système répondant à une équation différentielle à coefficients constants du type :

$$\sum_i^{N_s} a_i \frac{d^i s(t)}{dt^i} = \sum_i^{N_e} b_i \frac{d^i e(t)}{dt^i}$$

Avec s et e des variables discrètes.

Supposons que e est une entrée connue et s une sortie recherchée.

Pour déterminer, à tout temps $t = t_0 + kT$, on exprimer chacune des dérivées avec les méthodes vues précédemment (Euler, Taylor, différences finies) afin d'obtenir une relation entre divers états de l'entrée et de la sortie et on exprime ensuite à l'aide de cette équation la valeur recherchée en fonction des valeurs connues de l'entrée et des valeurs précédentes de la sortie.

Autrement dit, une équation aux différences est de la forme :

$$s(t + dt) = f\big(t, s(t), s(t - dt), s(t - 2dt)\big) \dots$$

On veillera à traiter correctement les conditions initiales.

A.VI.2.c.ii Exemple

- **Equation récurrente**

Reprenons l'exemple de la masse en chute libre étudiée avec la méthode précédente, dont l'équation était :

$$mg - \mu v = ma$$

$$ma + \mu v = mg$$

$$m\frac{d^2z(t)}{dt^2} + \mu\frac{dz(t)}{dt} = mg$$

On choisit d'exprimer les dérivées première et seconde de z à l'aide d'un schéma Euler Explicite pour la dérivée première et Euler centré pour la dérivée seconde :

$$\frac{dz(t)}{dt} = \frac{z(t+T) - z(t)}{T}$$

$$\frac{d^2z(t)}{dt^2} = \frac{z(t+T) - 2z(t) + z(t-T)}{T^2}$$

On obtient donc l'équation aux différences suivante :

$$m\frac{z(t+T) - 2z(t) + z(t-T)}{T^2} + \mu\frac{z(t+T) - z(t)}{T} = mg$$

On obtient alors la position à tout instant connaissant les positions antérieures :

$$\frac{mz(t+T)}{T^2} - \frac{2mz(t)}{T^2} + \frac{mz(t-T)}{T^2} + \frac{\mu z(t+T)}{T} - \frac{\mu z(t)}{T} = mg$$

$$\left(\frac{m}{T^2} + \frac{\mu}{T}\right)z(t+T) - \frac{2m + \mu T}{T^2}z(t) + \frac{m}{T^2}z(t-T) = mg$$

$$\frac{m + \mu T}{T^2}e(t+T) = \frac{2m + \mu T}{T^2}e(t) - \frac{m}{T^2}e(t-T) + mg$$

$$\boxed{z(t+T) = \frac{2m + \mu T}{m + \mu T}z(t) - \frac{m}{m + \mu T}z(t-T) + \frac{T^2}{m + \mu T}mg}$$

- **Prise en compte des conditions initiales**

Attention, à l'instant initial $t = t_0$, on a $z(t_0)$.

Pour calculer la valeur suivante, il faut $z(t_0)$ et $z(t_0 - T)$. Deux solutions :

- La vitesse initiale est nulle, alors on sait qu'avant t_0, $z(t) = z(t_0)$ et on peut donc affecter la valeur en $t_0 - T$ égale à la valeur en t_0
- La vitesse initiale est non nulle, on a alors :

$$\frac{dz(t_0)}{dt} = V_0 = \frac{z(t+T) - z(t)}{T}$$

On suppose qu'on a aussi :

$$\frac{dz(t_0)}{dt} = V_0 = \frac{z(t) - z(t-T)}{T}$$

On impose alors :

$$z(t-T) = z(t_0) - V_0 T$$

- **Remarque : Regroupement de termes dépendant de z dans le second membre**

Il serait possible de résoudre l'équation en intégrant la force de frottement fluide en second membre :

$$ma = mg - \mu v$$

A chaque instant, on calcule donc la valeur de $mg - \mu v$ que l'on appellera A. Pour calculer v à chaque étape, on utilise Euler explicite à l'ordre 1 :

$$v = \frac{dz(t)}{dt} = \frac{z(t+T) - z(t)}{T}$$

L'équation est donc :

$$ma = A \Leftrightarrow a = \frac{A}{m} \Leftrightarrow \frac{z(t+T) - 2z(t) + z(t-T)}{T^2} = \frac{A}{m}$$

$$\Leftrightarrow z(t+T) - 2z(t) + z(t-T) = \frac{AT^2}{m}$$

$$\Leftrightarrow z(t+T) = \frac{AT^2}{m} + 2z(t) - z(t-T)$$

Cela peut être très utile en présence de termes non linéaires, par exemple une force en carré de la vitesse (frottements fluides réalistes…).

A.VI.2.d Résolution d'équations et précision

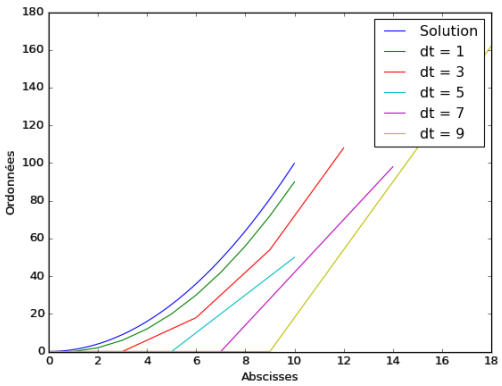
Comme nous l'avons dit, nos approximations de dérivées ne seraient exactes que si le pas de temps était infiniment nul. Ce n'est évidemment pas le cas, et plus le pas de temps est petit, plus les temps de calcul deviennent élevés…

Il faut donc être conscient que la méthode de résolution par discrétisations d'Euler induit des erreurs ! Et celles-ci se cumulent.

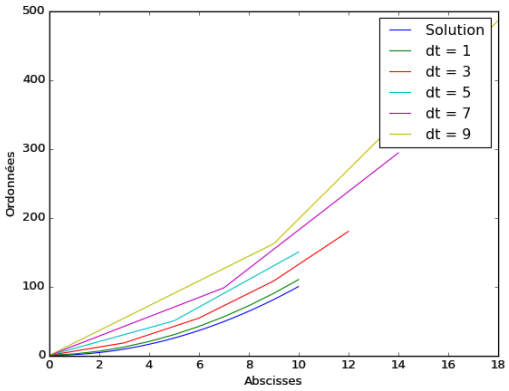
Soit la fonction $f(t) = t^2$.

Voici le résultat de la résolution de l'équation $f'(t) = 2t$ par la méthode d'Euler explicite ($y(t+dt) = y(t) + y'(\boldsymbol{t})dt$) et implicite ($y(t+dt) = y(t) + y'(\boldsymbol{t+dt})dt$) décrite au paragraphe précédent entre 0 et 10 pour des pas de temps différents :

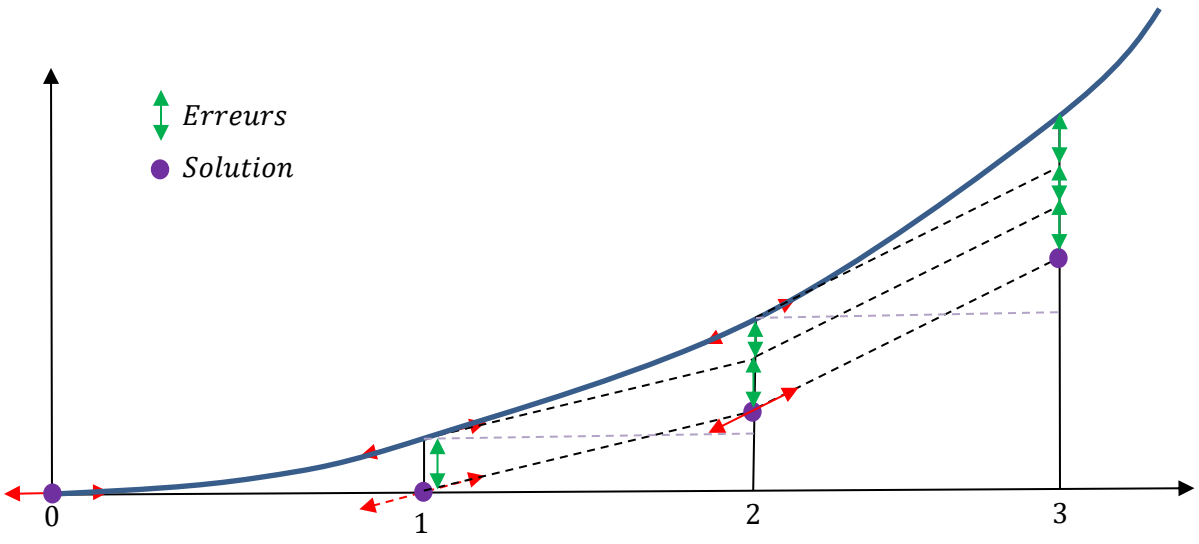
Euler Explicite



Euler Implicite



On remarque bien que plus le pas de temps est petit, plus la courbe tend vers la solution réelle. On peut noter que pour le pas de temps de 9, la première valeur calculée est très fausse. La suivante sera calculée à partir de cette fausse valeur… L'erreur se cumule ! Voici une illustration de ce qu'il se passe en explicite :



A.VI.3 Problèmes discrets multidimensionnels

Maintenant que nous avons vu comment résoudre des problèmes à une dimension (une variable), voyons comment traiter des problèmes multidimensionnels.

Supposons un problème dont les équations se mettent sous la forme d'un système linéaire à n équations et n inconnues inversible, c'est-à-dire admettant une solution unique, ou encore dit de Cramer.

A.VI.3.a Mise sous forme matricielle

Soit le système linéaire de n équations à m variables d'entrée $e_i(t)$ et n variables de sortie $s_i(t)$:

$$\begin{cases} a_{11}s_1(t) + a_{12}s_2(t) + \cdots + a_{1n}s_n(t) = b_{11}e_1(t) + b_{12}e_2(t) + \cdots + b_{1m}e_m(t) \\ a_{21}s_1(t) + a_{22}s_2(t) + \cdots + a_{2n}s_n(t) = b_{21}e_1(t) + b_{22}e_2(t) + \cdots + b_{2m}e_m(t) \\ \vdots \\ a_{n1}s_1(t) + a_{n2}s_2(t) + \cdots + a_{nn}s_n(t) = b_{n1}e_1(t) + b_{n2}e_2(t) + \cdots + b_{nm}e_m(t) \end{cases}$$

Avec a_{ij} et b_{ij} des coefficients constants.

On peut alors traduire ce système d'équations sous forme matricielle :

$$K_E E(t) = K_S S(t)$$

Avec :

$$E(t) = \begin{bmatrix} e_1(t) \\ e_2(t) \\ \vdots \\ e_n(t) \end{bmatrix} \quad ; \quad K_E = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ & & \vdots & \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{bmatrix}$$

$$S(t) = \begin{bmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_n(t) \end{bmatrix} \quad ; \quad K_S = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & & \vdots & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

Connaissant les entrées, l'objectif est alors de déterminer les sorties.

On peut donc se ramener à la résolution du système suivant :

$$K_S S(t) = B$$

Avec $B = K_E E(t)$ un vecteur dans lequel tout est connu.

A.VI.3.b Méthode de Gauss avec recherche partielle des pivots

A.VI.3.b.i Exemple

Soit le système suivant :

$$\begin{cases} x+y+2z=10 \\ x+3y+4z=20 \\ x+5y+z=30 \end{cases}$$

Notre objectif est d'obtenir un système échelonné, c'est-à-dire dans lequel une première ligne contient une seule inconnue, une seconde ligne contient cette précédente inconnue et une autre, et ainsi de suite. Il sera alors aisé de déterminer chacune d'entre elles les unes après les autres.

- **Raisonnement sur le système**

On appelle pivot la première variable de la première ligne dont le coefficient est non nul. Dans notre cas, le pivot est donc la variable x :

$$\begin{cases} \boldsymbol{x}+y+2z=10 \\ x+3y+4z=20 \\ x+5y+z=30 \end{cases}$$

On soustrait à chacune des lignes suivantes la première ligne afin d'y faire disparaître le pivot en multipliant si besoin par coefficient :

$$\begin{cases} \boldsymbol{x}+y+2z=10 \\ x-x+3y-y+4z-2z=20-10 \\ x-x+5y-y+z-2z=30-10 \end{cases} \Leftrightarrow \begin{cases} x+y+2z=10 \\ 2y+2z=10 \\ 4y-z=20 \end{cases}$$

On procède ainsi pour chaque nouveau système à partir de la ligne suivante. Ainsi, le nouveau pivot du second système est y :

$$\begin{cases} x+y+2z=10 \\ 2\boldsymbol{y}+2z=10 \\ 4y-2*2y-z-2*2z=20-2*10 \end{cases} \Leftrightarrow \begin{cases} x+y+2z=10 \\ 2y+2z=10 \\ -5z=0 \end{cases}$$

On peut alors résoudre le système car une équation donne une première inconnue puis chacune des autres permet de trouver une inconnue supplémentaire :

$$\begin{cases} x+y+2z=10 \\ 2y+2z=10 \\ -5z=0 \end{cases} \Leftrightarrow \begin{cases} x+y=10 \\ 2y=10 \\ z=0 \end{cases} \Leftrightarrow \begin{cases} x+5=10 \\ y=5 \\ z=0 \end{cases} \Leftrightarrow \begin{cases} x=5 \\ y=5 \\ z=0 \end{cases}$$

- **Raisonnement matriciel**

$$\begin{cases} x+y+2z=10 \\ x+3y+4z=20 \\ x+5y+z=30 \end{cases} \Leftrightarrow \begin{bmatrix} 1 & 1 & 2 \\ 1 & 3 & 4 \\ 1 & 5 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$$

Les opérations successives sur le système se traduisent ainsi :

$$\begin{bmatrix} \mathbf{1} & 1 & 2 \\ 1-1 & 3-1 & 4-2 \\ 1-1 & 5-1 & 1-2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 20-10 \\ 30-10 \end{bmatrix} \Leftrightarrow \begin{bmatrix} \mathbf{1} & 1 & 2 \\ 0 & 2 & 2 \\ 0 & 4 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 20 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 2 \\ 0 & \mathbf{2} & 2 \\ 0 & 4-2*2 & -1-2*2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 20-2*10 \end{bmatrix} \Leftrightarrow \begin{bmatrix} 1 & 1 & 2 \\ 0 & \mathbf{2} & 2 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 2 \\ 0 & 2 & 2 \\ 0 & 0 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 0 \end{bmatrix}$$

On voit qu'il est donc possible d'obtenir une matrice triangulaire, dite échelonnée, traduisant le même système linéaire que le système initial :

$$\begin{bmatrix} 1 & 1 & 2 \\ \mathbf{0} & 2 & 2 \\ \mathbf{0} & \mathbf{0} & -5 \end{bmatrix}$$

A.VI.3.b.ii Méthode générale

• **Préliminaires**

Soit le système $S(t) = B$ inversible. Appelons L_i la ligne i du système $K_S S(t) = B$ et C_i la colonne i de la matrice K_S. Notre objectif est de procéder à des opérations permettant d'arriver à un système échelonné, c'est-à-dire à une forme de matrice K_S dont un triangle (inférieur ou supérieur) est plein (diagonale incluse) et ou le reste est vide.

Théorème de Gauss-Jordan : Tout système linéaire se ramène à un système échelonné équivalent en utilisant 3 types d'opérations élémentaires :

- Permutation de 2 équations : $L_i \leftrightarrow L_j$
- Permutation de l'ordre des inconnues : $C_i \leftrightarrow C_j$
- Remplacement d'une équation (transvection) : $L_i \leftarrow L_i + \lambda L_j$

Attention : ces opérations doivent être réalisées les unes après les autres, et non en même temps. En effet, il est alors possible de transformer le système et de ne plus avoir les mêmes solutions. Imaginons par exemple remplacer en même temps chaque ligne d'un système à n équations par le somme de toutes les lignes… On obtient alors n fois la même équation… Le système n'est plus inversible.

• **Notations**

Soit la matrice $K_S = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \vdots & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$, le vecteur $B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$ et le vecteur solution $S = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}$

• **Algorithme de transformation**

Attention aux indices pris ci-dessous comme allant de 1 à n (0 à n-1 dans Python).

A la ligne $i \in [1, n-1]$ du système dont la matrice représentative est $\begin{bmatrix} a_{ii} & a_{i,i+1} & \dots & a_{in} \\ & \vdots & & \\ a_{ni} & a_{n,i+1} & \dots & a_{nn} \end{bmatrix}$:

- Si $a_{ii} \neq 0$: On remplace les lignes $L_j, i \in [i+1, n]$ telles que : $L_j \leftarrow L_j - \frac{a_{ji}}{a_{ii}} L_i$ – C'est-à-dire qu'il faut modifier à la fois K_S et B. Cette étape s'appelle la « transvection »
- Si $a_{ii} = 0$:
  - On permute deux lignes (même système) ou deux colonnes (ie inconnues) afin d'obtenir un coefficient a_{ii} différent de 0 – Préférer permuter des lignes afin de ne pas modifier l'ordre des inconnues et donc de simplifier la résolution
  - On procède comme proposé lorsque $a_{ii} \neq 0$

Remarque : si $a_{ii} = 0$, et si le système traité est inversible, le sous système $\begin{bmatrix} a_{ii} & a_{i,i+1} & \dots & a_{in} \\ & \vdots & & \\ a_{ni} & a_{n,i+1} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} b_i \\ b_{i+1} \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} b_i \\ b_{i+1} \\ \vdots \\ b_n \end{bmatrix}$ de $n-i+1$ équations à $n-i+1$ inconnues ne peut présenter une première ligne ou colonne nulle, il existe donc obligatoirement un terme non nul dans la première ligne et la première colonne.

La matrice ainsi obtenue s'écrit sous la forme :

$$K_S' = \begin{bmatrix} a_{11}' & a_{12}' & \dots & a_{1n}' \\ 0 & a_{22}' & \dots & a_{2n}' \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & 0 & a_{nn}' \end{bmatrix}$$

Tel que :

$$K_S' \, S(t) = B'$$

- **<u>Algorithme de résolution</u>**

Ce qui suit n'est valable qu'en cas de permutations de lignes dans la méthode précédente, sinon l'ordre des inconnues peut avoir changé.

- Ligne n :

$$a_{nn}' s_n = b_n'$$
$$\Rightarrow s_n = \frac{b_n'}{a_{nn}'}$$

- Ligne $n-1$:

$$a_{n-1,n-1}' s_{n-1} + a_{n-1,n}' s_n = b_{n-1}'$$
$$\Rightarrow s_{n-1} = \frac{b_{n-1}' - a_{n-1,n}' s_n}{a_{n-1,n-1}'}$$

- Ligne $n-2$:

$$a_{n-2,n-2}' s_{n-2} + a_{n-2,n-1}' s_{n-1} + a_{n-2,n}' s_n = b_{n-2}'$$
$$\Rightarrow s_{n-2} = \frac{b_{n-2}' - a_{n-2,n}' s_n - a_{n-2,n-1}' s_{n-1}}{a_{n-2,n-2}'}$$

Donc, d'une manière générale, à la ligne i en résolvant de n à 1 :

$$s_i = \frac{b_i' - \sum_{k=i}^{n} a_{i,k}' s_k}{a_{ii}'}$$

A.VI.3.b.iii Complexité

- **Mise sous forme échelonnée**

En supposant qu'il n'est pas nécessaire d'effectuer de permutations de lignes, la transformation du système général en système échelonné s'effectue en N opérations (ensemble d'aditions, soustractions, multiplications et ou divisions dont le nombre est fixe) telles que :

Pour chaque ligne i du système de 1 à $n-1$, on effectue N_i opérations avec :

- Une opération aux $(n-i)$ lignes en dessous et aux $(n-i+1)$ termes de chacune de ces lignes de K
- Une opération aux $(n-i)$ lignes en dessous de B

$$N_i = \sum_{j=1}^{n-i}(n-i+2) = (n-i)(n-i+2)$$

Soit au total :

$$N = \sum_{i=1}^{n-1} N_i = \sum_{i=1}^{n-1}(n-i)(n-i+2) = \sum_{i=1}^{n-1}(n^2 - ni + 2n - ni + i^2 - 2i)$$

$$= \sum_{i=1}^{n-1}(n(n+2) - 2(n+1)i + i^2)$$

$$= (n-1)n(n+2) - 2(n+1)\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} i^2$$

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$\sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6}$$

Soit :

$$N = (n-1)n(n+2) - 2(n+1)\frac{n(n-1)}{2} + \frac{(n-1)n(2n-1)}{6}$$

$$N = (n-1)n\left[(n+2) - (n+1) + \frac{(2n-1)}{6}\right] = (n-1)n\left[1 + \frac{(2n-1)}{6}\right]$$

$$N = \frac{(n-1)n(2n+5)}{6}$$

La complexité de la transformation en système échelonné dans le meilleur des cas est en :

$$O(n^3)$$

A ce nombre, dans le pire des cas, il faut effectuer pour chaque ligne i un test sur les $(n-i)$ lignes en dessous puis une inversion afin d'avoir un terme K_{ii} non nul, ce qui ajoute pour chaque ligne N_i' opérations telles que :

$$N_i' = \left(\sum_{j=1}^{n-i} 1\right) + k = n - i + k$$

Avec k coût de l'échange des $(n-i+1)$ termes à priori non nuls de K et du terme de B : $k = (n-i+1)+1$

$$N_i' = n - i + n - i + 2 = 2(n+1) - 2i$$

Et donc un coût supplémentaire de :

$$N' = \sum_{i=1}^{n-1} N_i' = \sum_{i=1}^{n-1} 2(n+1) - 2i = \sum_{i=1}^{n-1} 2(n+1) - 2\sum_{i=1}^{n-1} i$$

$$N' = 2(n-1)(n+1) - 2\frac{n(n-1)}{2}$$

$$N' = 2(n-1)(n+1) - n(n-1) = n(n-1)(2n+2-n) = n(n-1)(n+2)$$

La nombre d'opérations dans le pire des cas étant la somme $N + N'$, la complexité globale reste inchangée, en $O(n^3)$.

- **<u>Résolution triangulaire</u>**

A chaque ligne du système, une opération est réalisée… Complexité en $O(n)$.

- **<u>Bilan</u>**

La méthode de résolution de systèmes linéaires par pivot de Gauss est de complexité $O(n^3)$ quel que soit le système inversible initial.

A.VII. Bases de données

A.VII.1 Définition

Une base de données (BDD) est, comme son nom l'indique, un outil de stockage de données. Elle permet la collecte, le stockage et l'utilisation des données associées.

Le système de gestion des bases de données s'appelle le « SGBD » pour « Système de Gestion de Base de Données ». Il permet d'accéder aux données de la base de données par l'intermédiaire de requêtes simples qui cachent une manipulation complexe des données.

A.VII.2 Systèmes de gestion des données

Le programme d'IPT propose que vous ayez quelques notions sur l'origine des bases de données relationnelles et leur contexte d'utilisation. Ce paragraphe, sans rentrer dans les détails, vous apportera les quelques informations à connaître.

A.VII.2.a Peer to peer

Prenons un premier exemple qui doit vous parler. Le peer to peer est un moyen que possède chaque particulier pour récupérer ou envoyer des données qu'il possède sur sa machine. Chaque utilisateur est donc aussi serveur de données et un logiciel remplissant à la fois les fonctions de client et de serveur gère les échanges.

A.VII.2.b Client-Serveur

L'architecture client-serveur se décompose, évidemment, en deux parties :

- Client : gère la présentation et la partie applicative (ce que l'on fait des données)
- Serveur : stocke les données de façon cohérente

Le client possédant des droits d'accès envoie des requêtes au serveur qui les traite et retourne les résultats. Le serveur intègre alors un logiciel de gestion de base de données (SGBD) ainsi qu'un espace de stockage de données.

Le langage SQL (Structured Query Langage) est classiquement utilisé dans les requêtes entre client et serveur.

L'avantage de ce type d'architecture est que le client n'a pas à connaître ou chercher à maîtriser la manière qu'à le serveur de gérer les données (dimension des disques, répartition des données, sauvegarde…). La qualité de ce type d'architecture est liée à la notion de bases de données relationnelles et au langage SQL (Structured Query Langage). Le modèle relationnel que nous allons aborder est né en 1970 (E. Codd).

Sans rentrer dans les détails, voici quelques avantages et inconvénients de cette architecture :

- Fiabilité et performance du fait de beaucoup de travaux réalisés en presque 60 ans
- Inconvénients : sécurité (tous les clients accèdent à la base de données) et coût

A.VII.2.c Architecture trois/tiers

L'architecture trois-tiers est un peu plus évoluée que l'architecture client/serveur puisqu'au lieu de diviser le travail en deux, il est divisé en trois. Celles-ci sont indépendantes, ce qui permet de changer de mode de stockage, le type de traitement des données ou d'interface homme machine sans pour autant modifier les autres étages. Les étages sont toutefois dépendants entre eux puisqu'ils doivent communiquer.

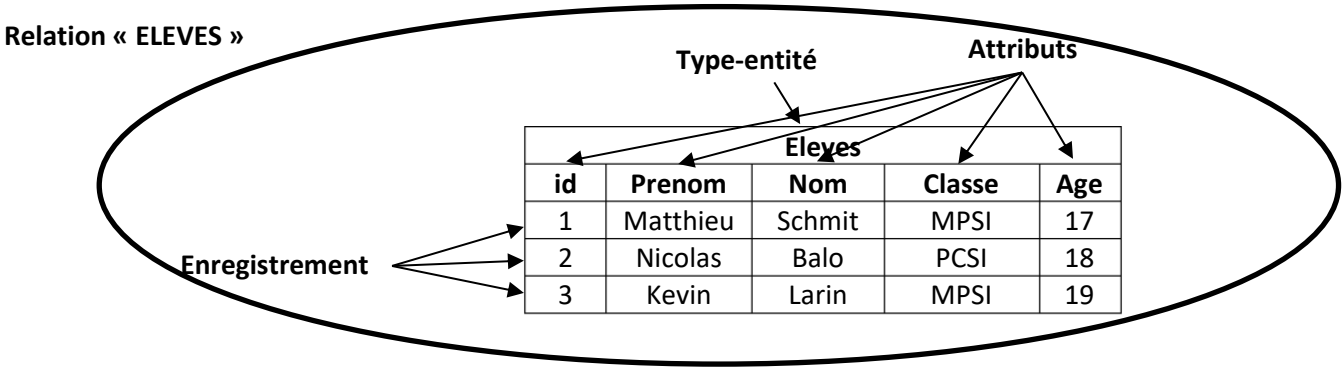
Le gros avantage de cette architecture est que le client ne nécessite qu'une installation minime, souvent un navigateur web, pour traiter ses données, le serveur applicatif étant différencié du client. Evolution de l'architecture client/serveur, elle continue à utiliser les outils de bases de données relationnel et de langage SQL. Cette architecture présente un gain de sécurité puisque les clients n'ont plus chacun accès à la base de données. C'est directement le serveur applicatif qui y accède et il ne donne pas les identifiants d'accès aux clients.

Le principe de l'architecture trois/tiers est d'avoir 3 niveaux de traitements (Présentation, Traitement, Accès aux données) :

- Couche d'accès aux données : Le système de stockage de données peut prendre plusieurs formes différentes, par exemple l'utilisation de bases de données relationnelles.
- Couche de traitement : C'est la couche dans laquelle il y a la partie fonctionnelle de l'application (logique applicative).
- Couche de présentation : Cette couche correspond à l'interface homme/machine. Elle peut avoir différentes formes adaptées aux utilisateurs pour réaliser le même travail. Elle relaie les requêtes de l'utilisateur vers la couche de traitement et renvoie à l'utilisateur les traitements de cette même couche.

A.VII.3 Vocabulaire

Le programme d'IPT propose d'aborder le concept de bases de données relationnelles, c'est-à-dire dans laquelle l'information est organisée dans des tableaux à deux dimensions appelés des **relations** ou des **tables**. Les lignes de ces tables sont appelées **n-uplets** (tuples en anglais) ou **enregistrements**, et les colonnes sont associées à des **attributs**.



Un **Type-entité** définit la caractéristique de la relation, dans notre cas ce sont des « ELEVES ». Une **entité** est une occurrence de son type-entité. A chaque enregistrement d'une relation correspond donc une entité du type-entité, ici nos 3 élèves Matthieu, Nicolas et Kevin.

Le **domaine** d'un attribut, fini ou infini, est l'ensemble des valeurs qu'il peut prendre. Le domaine de l'attribut « Age » est l'ensemble des entiers de 0 à 130 (actuellement) par exemple.

Le **degré** d'une relation est son nombre d'attributs, 5 dans notre exemple (id, Prenom, Nom, Classe, Age).

Un **schéma de relation** précise le nom d'une relation ainsi que la liste des attributs avec leur domaine : Eleves(id : Entier, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Age : Entier). On appelle **Schéma relationnel** l'ensemble des schémas de relation.

On appelle « **clé primaire** » une ou plusieurs données permettent d'identifier de manière unique un enregistrement dans une table. On ne peut avoir deux fois le même enregistrement, il existe donc toujours une clé primaire. On la souligne dans le schéma de relation: Eleves(id : Entier, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Age : Entier). On ajoute souvent un identifiant « id » en première colonne, entier unique qui permet d'être clé primaire.

Une **base de données relationnelle** est constituée par l'ensemble des n-uplets des différentes relations du schéma relationnel.

A.VII.4 Exemple support du cours

Soient les deux tables suivantes :

Eleves					
id	**Prenom**	**Nom**	**Classe**	**Age**	**Ville**
1	Steph	ANE	MPSI	17	Paris
2	Marc	IMBUT	PCSI	18	Marseille
3	Jo	NID	MPSI	19	Montpelier
4	Rayan	AIR	MP	19	Paris
5	Tom	DESAVOIE	PCSI	18	Nice
6	Jerry	CANE	PSI	19	Paris
7	Paul	AINE	MP	19	Marseille
8	Jean	NAIMAR	PCSI	18	Toulouse
9	Jack	OUILLE	MPSI	18	Montpelier
10	Sam	OURAIL	PC	20	Paris

Profs					
id	**Titre**	**Prenom**	**Nom**	**Classe**	**Salle**
1	Mr	Denis	DEFAUCHY	MPSI	D21
2	Mr	Paul	YMERE	PCSI	A05
3	Mme	Sylvie	DEFOU	MP	B18
4	Mr	Georges	AITTE	PC	C15
5	Mme	Marie	AGE	PSI	E8

La base de données relationnelle est donc :

- Eleves(id : Entier, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Age : Entier,Ville : Chaîne)
- Profs(id : Entier, Titre : Chaîne, Prenom : Chaîne , Nom : Chaîne , Classe : Chaîne , Salle : Chaîne)

A.VII.5 Généralités

Nous serons ici amenés à détailler le langage SQL (Structured Query Language) pour interroger les bases de données, c'est-à-dire pour réaliser des **Requêtes** sur une base de données.

A.VII.5.a Manipulations et requêtes

Les bases de données se trouvent sous différentes extensions (csv, sql, xml, xls, bd…). On en trouve par exemple beaucoup concernant la France ici : https://www.data.gouv.fr/fr/

La gestion des bases de données est totalement indépendante du logiciel Python. En effet, le langage SQL peut être directement utilisé dans des logiciels comme « Sqlite browser » gratuit et disponible ici : http://sqlitebrowser.org/

Je vous recommande d'importer un fichier .csv en faisant :

- Création d'une nouvelle bdd et enregistrement
- Ne pas remplir de tables (Cancel)
- Fichier – Importer – Table vers un fichier csv – Afficher toutes les extensions – Ouvrir le fichier
- Choisir les bonnes options d'importation en visualisant le résultat

Ecriture des requêtes dans l'onglet « Exécuter le SQL »

Nous allons toutefois voir dans ce cours comment traiter les bases de données sous Python, même s'il est possible d'en être totalement indépendant.

A.VII.5.b Généralités de langage

Les instructions SQL dans un code doivent impérativement être écrites en lettres majuscules et se finir par un point-virgule « ; ».

Quelques symboles peuvent vous être utiles pour effectuer des recherches particulières :

- « * » permet de dire que l'on recherche tous les attributs
- « % » désigne une chaîne de caractères quelconque – « A% » désignera un mot commençant par A – « %a% » un mot contenant a…
- « _ » désigne un caractère quelconque
- « '…' » désigne un champ au format texte
- « 'mm/jj/aaa' » désigne un champ au format date

A.VII.5.c Structure d'une manipulation de bases de données sous Python

La gestion d'une base de données sous Python s'organise ainsi :

```
import sqlite3
BDD = sqlite3.connect('BDD.db')
cursor = BDD.cursor()
cursor.execute("""…""")
cursor.executemany("""…""")
BDD.commit()
BDD.close()
```

C'est normal qu'à ce stade, vous ne compreniez pas grand-chose à ce code ! En quelques mots, et nous développerons tout cela dans les prochains paragraphes :

- On importe la librairie de bases de données SQL sous Python
- On ouvre ou crée la base de données BDD.db
- On crée un outil « cursor » qui va manipuler la base de données
- On exécute alors différentes requêtes, et les « … » sont des requêtes que vous apprendrez à manipuler par la suite
- On applique les modifications au fichier BDD.db
- On ferme la base de données

Remarque : Dans `cursor.execute("""…""")`, il faut un texte sous forme de string, on peut donc utiliser `",'` ou `'''` ou `"""`. J'ai choisi ici les `"""` (équivalent à `'''`) pour les raisons suivantes :

- Pas de problèmes pour utiliser une seule Guillement pour définir un texte dans le texte : `"""("A","B")"""`
- La couleur du texte sous Python est bleue et permet d'identifier les codes SQL facilement

A.VII.5.d Premiers pas – Création - Ouverture

A.VII.5.d.i Librairie sous Python

Nous allons manipuler les bases de données sous Python avec la librairie sqlite3. Il faut donc écrire :

```
import sqlite3
```

A.VII.5.d.ii Création/ouverture d'une base de données

Que la base de données existe ou non, une commande permet de l'ouvrir ou de la créer. Il faut écrire :

```
BDD = sqlite3.connect('BDD.db')
```

Où `'BDD.db'` est la base de données à ouvrir ou à créer, dans le répertoire de travail (on peut préciser un chemin absolu) comme pour les fichiers textes.

Concrètement, on demande à Python de se « connecter » à la base de données `'BDD.db'` qu'il ouvre ou crée si elle n'existe pas.

A.VII.5.d.iii Création de l'outil de manipulation de la base de données

Pour manipuler la base de données BDD, il est nécessaire de créer un outil sous Python qui va permettre de travailler dessus. Il faut écrire :

```
cursor = BDD.cursor()
```

Ne l'oubliez pas !

A.VII.5.d.iv Création/suppression de relations/tables

- **Création d'une relation**

Nous pouvons maintenant créer des relations à l'aide d'une requête « CREATE TABLE ». Créons les deux tables de l'exemple proposé précédemment :

```
cursor.execute("""CREATE TABLE Eleves(id INTEGER PRIMARY KEY,Prenom
TEXT,Nom TEXT,Classe TEXT,Age INTERGER,Ville TEXT)""")
cursor.execute("""CREATE TABLE Profs(id INTEGER PRIMARY KEY,Titre
TEXT,Prenom TEXT,Nom TEXT,Classe TEXT, Salle TEXT)""")
```

Quelques explications :

- Pour exécuter une commande, il faut écrire `cursor.execute("""  """)` avec entre les guillemets les commandes du langage SQL.
- Pour créer une table, on écrit : `CREATE TABLE` suivi de son nom, puis entre parenthèses, on précise le nom de chaque attribut suivi de son type en majuscules (TEXT pour un texte, INTEGER pour un entier…).
- La mention `PRIMARY KEY` est optionnelle pour nos applications simples – Elle impose en tout cas que le champ soit unique pour chaque n-uplet et il sera donc impossible d'ajouter un n-uplet ayant le même id dans notre exemple

Ça y est, les tables sont initialisées, encore vides, et pas encore enregistrées dans le fichier associé à la base de données.

Remarque : si une table existe, il est possible de préciser de créer la table si elle n'existe pas afin d'éviter une erreur si elle existe en précisant : `CREATE TABLE IF NOT EXISTS`

- **Lister les relations d'une base de données (pour info)**

```
cursor.execute("""SELECT name FROM sqlite_master WHERE type='table';""")
print(cursor.fetchall())
```

A écrire tel quel sans rien changer !

- **Lister les attributs d'une relation (pour info)**

```
cursor.execute("""PRAGMA table_info(Eleves);""")
print(cursor.fetchall())
```

- **Suppression d'une relation**

Il suffit d'écrire :

```
cursor.execute("""DROP TABLE Eleves; """)
```

On supprime ainsi la table « Eleves ».

A.VII.5.d.v Ajout/modification/suppression d'enregistrements

Il existe plusieurs manières de procéder, voyons les deux plus classiques.

- **Ajout ligne par ligne**

On peut ajouter les donnes ligne par ligne, voici deux possibilités :

```
cursor.execute("""INSERT INTO Eleves VALUES
(1,"Steph","ANE","MPSI",17);""")
```

Cette méthode très simple ajoute à la table « Eleves » le nuplet `(1,"Steph","ANE","MPSI",17);"""`. Attention, il doit avoir la même taille que lors de la définition de la relation (CREATE TABLE).

```
cursor.execute("""INSERT INTO Eleves VALUES (?,?,?,?,?)
;""",(1,"Steph","ANE","MPSI",17))
```

Cette méthode un peu moins évidente au premier abord se comprend ainsi : Ajouter à la table « Eleves » les 5 valeurs (?, ?, ?, ?, ?) précisées dans le nuplet un peu plus loin… Ce fonctionne tant qu'il y a au moins 2 éléments par n-uplet. Je la précise surtout pour que vous compreniez l'écriture de l'ajout d'une liste au paragraphe suivant.

- **Ajout d'une liste de lignes**

On crée une liste de nuplets puis on l'ajoute à la table :

```
Liste_Eleves =
[(1,"Steph","ANE","MPSI",17,"Paris"),(2,"Marc","IMBUT","PCSI",18,"Marseil
le")]
cursor.executemany("""INSERT INTO Eleves VALUES
(?,?,?,?,?);""",Liste_Eleves)
```

On notera qu'il faut alors utiliser la commande « executemany » et non plus « execute ». Comme précisé précédemment, il faut au moins 2 attributs pour utiliser la méthode (?, ?…).

- **Modification d'un enregistrement : UPDATE SET**

```
cursor.execute('''UPDATE Eleves SET Nom = New_Nom, Prenom = New_Prenom
WHERE id = 2 ; ''')
```

On met à jour le nom et le prénom de l'élève d'id 2.

- **Suppression d'un enregistrement : DELETE**

```
cursor.execute('''DELETE FROM Eleves WHERE id = 3 ;''')
```

On supprime l'élève d'id 3.

A.VII.5.d.vi Validation des modifications dans le fichier 'BDD.db'

Toutes les modifications qui sont faites sur la base de données sont en réalité réalisées dans la mémoire de Python, et non dans la base de données elle-même. Il faut donc, pour que le fichier de la base de données soit modifié, exécuter une commande « d'enregistrement » en exécutant la ligne :

```
BDD.commit()
```

A.VII.5.d.vii Fermeture de la base de données

Comme pour les fichiers textes, il faut obligatoirement refermer la base de données après utilisation sous Python.

```
BDD.close()
```

A.VII.6 Requêtes SQL au programme

A.VII.6.a.i Requêtes et résultats associés

Maintenant que la base de données est créée, on peut effectuer des requêtes. Voici quelques exemples :

- Combien d'élèves viennent de paris
- Quel âge moyen ont les élèves en 2° année
- En quelle salle doivent aller les élèves de MPSI pour voir leur prof principal
- Etc

La structure des requêtes est toujours à peu près la même :

```
cursor.execute("""SELECT … FROM … WHERE …;""")
```

Remarque : Il y aura quelques variantes que nous allons voir avec des exemples.

On pourra alors récupérer les résultats de la requête à l'aide de l'une des commandes suivantes :

```
Resultats = cursor.fetchall()
Resultat = cursor.fetchone()
```

Remarque : cursor contient le résultat de la requête précédente uniquement.

La variable Resultat(s) contiendra alors l'ensemble des résultats de la requête avec :

- Pour fetchone, le premier résultat
- Pour fetchall, tous les résultats

Il restera à utiliser Python classiquement pour traiter ces résultats, par exemple :

```
Resultats = cursor.fetchall()
for Resultat in Resultats:
    print("Le premier eleve s'appelle : %s %s, il est en classe de %s" %
(Resultat[2], Resultat[3], Resultat[4]))
```

A.VII.6.a.ii Opérateurs de base

- **Projection : SELECT FROM**

La projection permet de n'afficher qu'une partie des attributs (colonnes) d'une relation.

Objectif	Requête
Obtenir toutes les informations d'un tableau	`SELECT * FROM Eleves ;` *Toutes les informations concernant les élèves*
Sélectionner certains attributs dans un tableau	`SELECT Nom,Prenom FROM Profs ;` *Noms et prénons des profs*
Obtenir toutes les occurrences d'un attribut en supprimant les doublons	`SELECT DISTINCT Age FROM Eleves ;` *Tous les âges des élèves*

- **Sélection (ou restriction) : WHERE**

La sélection permet de n'afficher qu'une partie des enregistrement (lignes) d'une relation. On utilise alors le mot **WHERE**.

Les opérateurs de sélection utilisables sont :

- « = » ou « != » utilisables avec tous types de données
- « > », « < », « >= », « <= » utilisables uniquement avec des données numériques
- LIKE, BETWEEN, IN, AND, OR, NOT

Exemples :

Objectif	Requête
Récupérer les attributs des enregistrements possédant un attribut spécifique	`SELECT Nom,Prenom FROM Eleves WHERE Classe = "MPSI" ;` *Noms et prénoms des élèves de MPSI*
	`SELECT DISTINCT Age FROM Eleves WHERE Classe LIKE "MPSI" OR Classe LIKE "PCSI" ;` *Tous les âges des élèves de première année sans doublons*
	`SELECT DISTINCT Age FROM Eleves WHERE Age BETWEEN 18 AND 19 ;` *Tous les âges des élèves de première année sans doublons*

- **Jointure : JOIN ON**

La jointure consiste à trouver des données en lien avec plusieurs tables. Par exemple :

- « Dans quelle salle doivent aller les élèves pour voir leur prof ? ».
- « Dans quelle salle doivent aller les élèves de MPSI pour voir leur prof ? ».

Il faut identifier les élèves de la classe MPSI dans la table « Eleves », chercher le prof de la classe MPSI dans la table « Profs », et associer la salle aux élèves concernés.

Le programme d'IPT propose de ne se limiter qu'aux jointures simples utilisant Join…ON…=….

Voici comment écrire les requêtes proposées :

```
SELECT Eleves.Prenom, Eleves.Nom, Profs.Salle FROM Eleves JOIN Profs ON
Eleves.Classe = Profs.Classe ;

SELECT Eleves.Prenom, Eleves.Nom, Profs.Salle FROM Eleves JOIN Profs ON
Eleves.Classe = Profs.Classe WHERE Eleves.Classe = "MPSI" ;
```

Comme il faut préciser ce que l'on sélectionne parmi les deux relations, on précise les attributs en ajoutant le nom de la table et un point avant chacun d'entre eux.

Quelques explications : La jointure **(Eleves JOIN Profs ON Eleves.Classe = Profs.Classe)** crée une nouvelle table de la basée sur le point commun de la classe de la forme :

Profs						**Eleves**					
id	**Titre**	**Prenom**	**Nom**	**Classe**	**Salle**	**id**	**Prenom**	**Nom**	**Classe**	**Age**	**Ville**
1	Mr	Denis	DEFAUCHY	MPSI	D21	1	Steph	ANE	MPSI	17	Paris
1	Mr	Denis	DEFAUCHY	MPSI	D21	3	Jo	NID	MPSI	19	Montpelier
1	Mr	Denis	DEFAUCHY	MPSI	D21	9	Jack	OUILLE	MPSI	18	Montpelier
2	Mr	Paul	YMERE	PCSI	A05	2	Marc	IMBUT	PCSI	18	Marseille
2	Mr	Paul	YMERE	PCSI	A05	5	Tom	DESAVOIE	PCSI	18	Nice
2	Mr	Paul	YMERE	PCSI	A05	8	Jean	NAIMAR	PCSI	18	Toulouse
3	Mme	Sylvie	DEFOU	MP	B18	4	Rayan	AIR	MP	19	Paris
3	Mme	Sylvie	DEFOU	MP	B18	7	Paul	AINE	MP	19	Marseille
4	Mr	Georges	AITTE	PC	C15	10	Sam	OURAIL	PC	20	Paris
5	Mme	Marie	AGE	PSI	E8	6	Jerry	CANE	PSI	19	Paris

Il reste alors comme d'habitude à écrire : SELECT … FROM **(table de jointure)** WHERE …

Remarques :

- Il pourrait y avoir plusieurs profs ayant la même classe, la jointure va créer autant de lignes qu'il le faut pour que tous les élèves de MPSI soient en face de chaque prof de MPSI.
- Vous remarquerez que rien ne change si on intervertit les 2 tables droite/gauche. Ce qui peut changer si l'instruction est écrite selon Eleves JOIN Profs ou Profs JOIN Eleves, c'est l'ordre de l'organisation des classes, mais les requêtes sur cette table seront inchangées.
- Ecrire **Eleves JOIN Profs** sans condition ON revient à créer une table dans laquelle pour chaque n-uplet d'une des tables initiales, on associe tous les n-uplets de l'autre table, ce qui revient à faire un produit cartésien CROSS JOIN que nous verrons dans un prochain paragraphe

A.VII.6.a.iii Opérateurs ensemblistes

- **Union – Intersection - Différence**

Les trois opérateurs usuels en bases de données sont l'union, l'intersection et la différence. Ces opérateurs s'utilisent entre deux tables (relations) de même structure.

Le principe de ces opérateurs consiste à ajouter un mot (UNION, EXCEPT, INTERSECT) entre deux requêtes SELECT.

Exemple : Soient les deux tables suivantes contenant la référence et le prix d'objets d'un magasin :

T1	
Ref	**Prix**
A27	12
C32	17
E15	18
Z7	20

T2	
Ref	**Prix**
C32	17
A27	15
B20	20
A12	21

Obtention d'une table de l'ensemble des produits Rq : UNION ne garde pas les doublons	`SELECT * FROM T1 UNION SELECT * FROM T2 ;` `[('A12', 21), ('A27', 12), ('A27', 15), ('B20', 20), ('C32', 17), ('E15', 18), ('Z7', 20)]`
Enlever les produits de T1 présents dans T2 (même ref & prix)	`SELECT * FROM T1 EXCEPT SELECT * FROM T2 ;` `[('A27', 12), ('E15', 18), ('Z7', 20)]`
Trouver les produits présents dans les deux tables (même ref & prix)	`SELECT * FROM T1 INTERSECT SELECT * FROM T2 ;` `[('C32', 17)]`

Remarque : on pourra bien-sûr ajouter des restrictions (WHERE) pour affiner les recherches.

- **Produit cartésien : CROSS JOIN**

Soient les deux tables suivantes contenant les groupes sanguins d'une population de 4 mâles et 4 femelles d'un laboratoire :

MALES	
Groupe	**Resus**
A	-
AB	+
B	+
B	-

FEMELLES	
Groupe	**Resus**
B	-
B	-
AB	+
A	-

On souhaite créer une base de données de tous les mixages possibles afin d'étudier les groupes possibles des enfants issus de tous les croisements possibles.

Il suffit d'utiliser l'opérateur CROSS JOIN :

```
SELECT * FROM MALES CROSS JOIN FEMELLES ;
```

Voici les premières lignes du résultat obtenu :

A	-	B	-
A	-	B	-
A	-	AB	+
A	-	A	-
AB	+	B	-
AB	+	B	-
…	…	…	…

Remarque : comme vu pour la jointure, on obtient le même résultat avec :

Il suffit d'utiliser l'opérateur CROSS JOIN :

```
SELECT * FROM MALES JOIN FEMELLES ;
```

- **Division cartésienne**

Soient les deux tables suivantes désignant pour l'une l'ensemble des lâchés machines des pilotes du club (Nom pilote, Identification Avion) et l'autre regroupant l'ensemble des avions du club :

Lachers	
Nom	**Avion**
DEFAUCHY	XH
SMITH	RL
DEFAUCHY	RL
DEFAUCHY	PQ
AMAR	XH
OLAF	PQ
SMITH	PQ
SMITH	XH

Avions	
Avion	**Type**
XH	DR420
RL	DR480
PQ	Sportstar

On cherche les noms des pilotes étant lâchés sur tous les avions.

```
SELECT Nom FROM Lachers GROUP BY Nom HAVING COUNT(*) = (SELECT COUNT(*)
FROM Avions) ;
```

On cherche dans la table « Lachers » regroupée par noms de pilotes ceux qui ont un nombre distinct d'avions égal au nombre distinct d'avions de la table « Avions ».

Oui, si vous avez bien compris, en réalité le nom des avions de la table « Avions » n'a pas d'importance. Si un pilote de la table « Lachers » est lâché sur 3 avions différents et s'il y a 3 avions différents au club, alors il ce pilote sera inclus dans le résultat de la requête.

Il faut donc veiller à ne pas faire n'importe quoi et bien définir les deux tables ensemble avec des données cohérentes.

Le résultat de la requête proposée sera « DEFAUCHY - SMITH »

A.VII.6.a.iv Fonctions d'agrégation

Voyons dans le tableau suivant quelques exemples de ce que l'on peut faire :

Minimum MIN()	Age min des élèves : `SELECT MIN(Age) FROM Eleves ;`
Maximum MAX()	Age max des élèves : `SELECT MAX(Age) FROM Eleves ;`
Somme SUM()	Somme des âges des élèves : `SELECT SUM(Age) FROM Eleves ;`
Moyenne AVG()	Moyenne des âges des élèves : `SELECT AVG(Age) FROM Eleves ;`

Comptage
COUNT
	Nombre d'élèves :
	`SELECT COUNT(*) FROM Eleves ;`
	Nombre délèves de 17 ans :
	`SELECT COUNT(*) FROM Eleves WHERE Age = 17 ;`

Pour utiliser ou afficher le résultat, on écrit après la ligne `cursor.execute("""…""")` :

```
Res = cursor.fetchone()
print(Res[0])
```

A.VII.7 Création d'une nouvelle bdd avec les résultats d'une requête

Voici comment créer une nouvelle relation basée sur le résultat de la requête réalisée. Voici le code permettant de créer une nouvelle relation basée sur les résultats de la requête réalisée (on pourra utiliser ce code pour toutes les requêtes et pas uniquement dans le cadre des unions, intersections et différence) :

```
# Opération d'union, différence, intersection
cursor.execute("""SELECT … FROM … WHERE … UNION SELECT … FROM … WHERE …
;""")
Res_OP = cursor.fetchall()
# Nouvelle base de données
cursor.execute("""CREATE TABLE T_OP(Ref Text,Prix Integer);""")
cursor.executemany("""INSERT INTO TUnion VALUES (?,?);""",Res_OP)
# Affichage du résultat
cursor.execute("""SELECT * FROM T_OP;""")
print(cursor.fetchall())
```

HAVE FUN WITH PYTHON