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 :

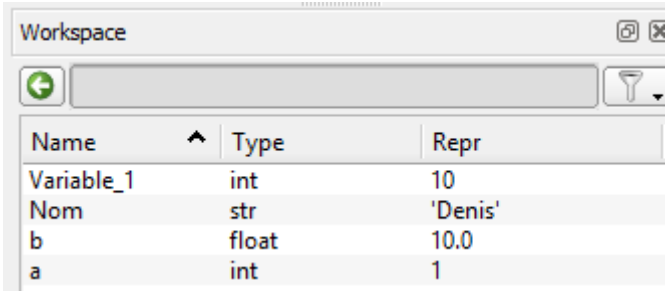
```
>>> a = input("Quel âge as tu ? ")
b = 2 + a
Quel âge as tu ? 10
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> a = input("Quel âge as tu ?")
b = 2 * a
print(b)
Quel âge as tu ?32
3232
```

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]`

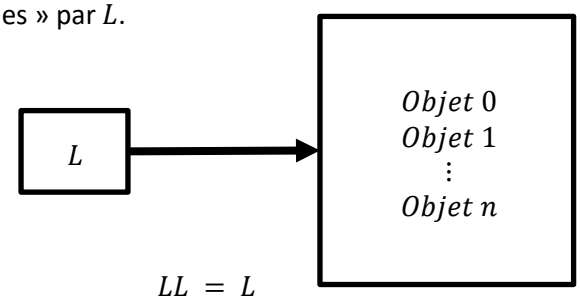
Attention dans les boucles for à l'utilisation de cette commande for i in L à ne pas modifier L ! (cf paragraphe sur la boucle for)

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]
```

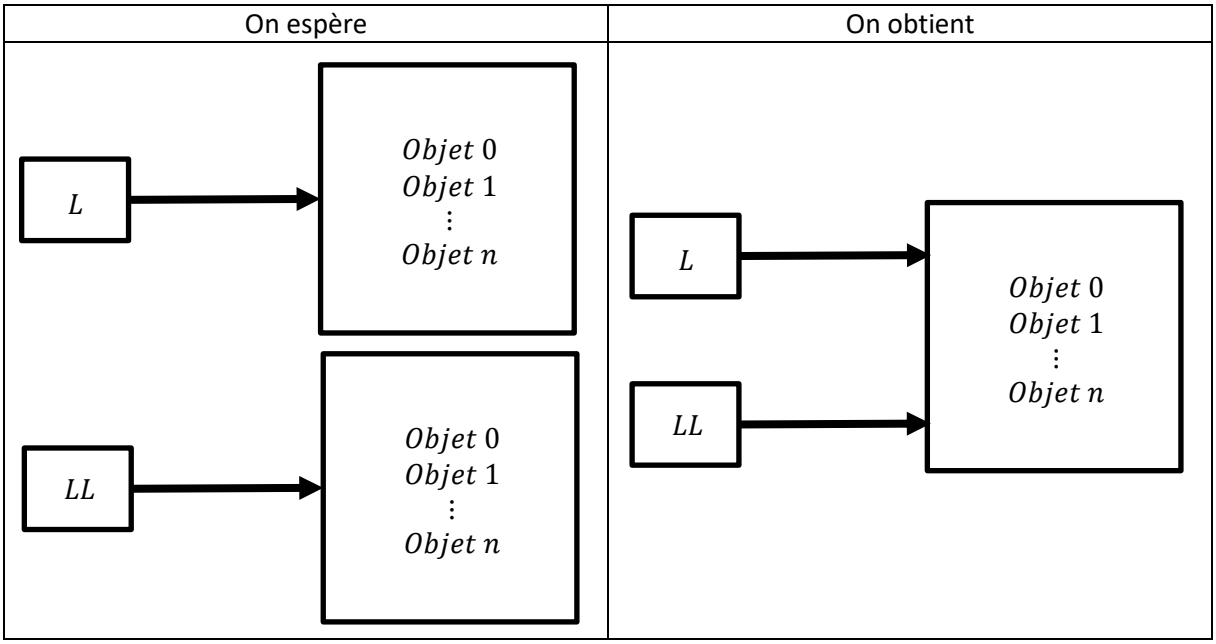
Et maintenant, regardons ce que vaut L :

```
>>> 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.

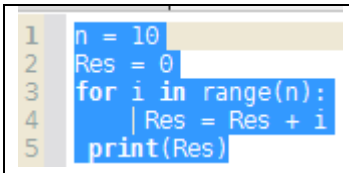
Il suffit :

- 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**

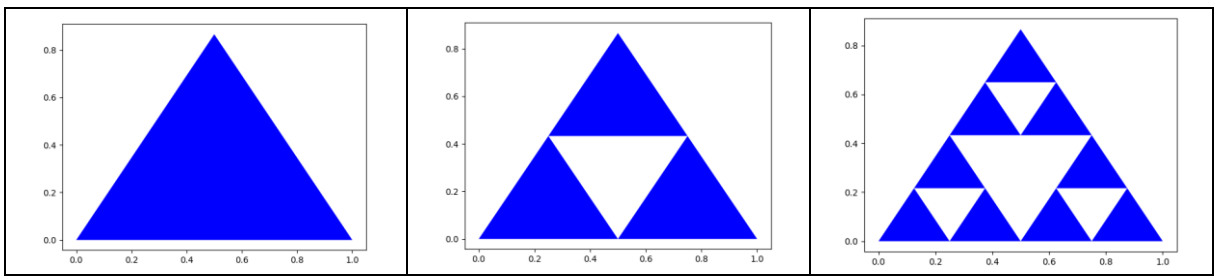
Un algorithme exécute une série d'opérations. Celles-ci mettent un certain temps à être exécutées et peuvent, selon les variables stockées, occuper un espace mémoire important. On parle de complexité en temps et en espace des algorithmes.

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 :

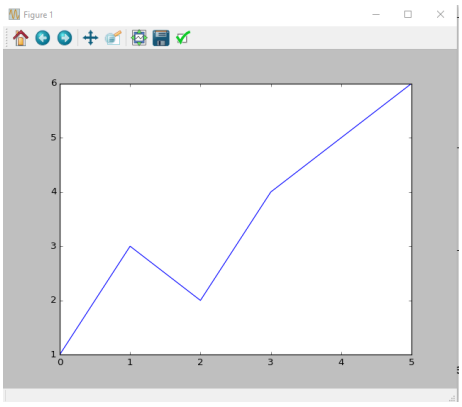
```
import matplotlib.pyplot as plt
```

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 :

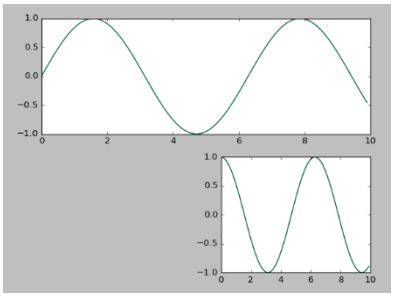
`plt.plot(X, Y, linewidth=2.0)`	On règle ainsi l'épaisseur du trait
`plt.plot(X, Y, 'ro')`	Crée un nuage de points
`plt.plot(X, Y, '--')`	Le trait est en pointillés
`plt.plot(X,Y,'r')`	On précise la couleur associée à la courbe *b: blue - g: green - r: red - c: cyan - m: magenta - y: yellow - k: black - 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 :

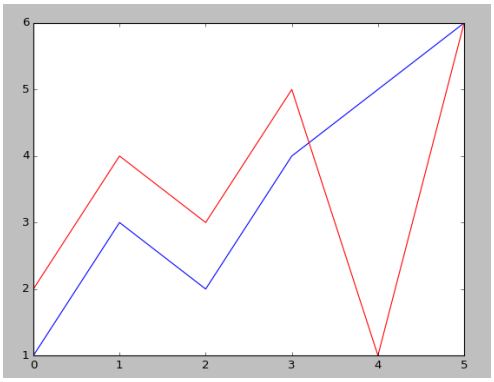
- En ouvrir autant que je le souhaite en parallèle
- Mettre à jour l'une d'elles
- Fermer l'une d'elles

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 :

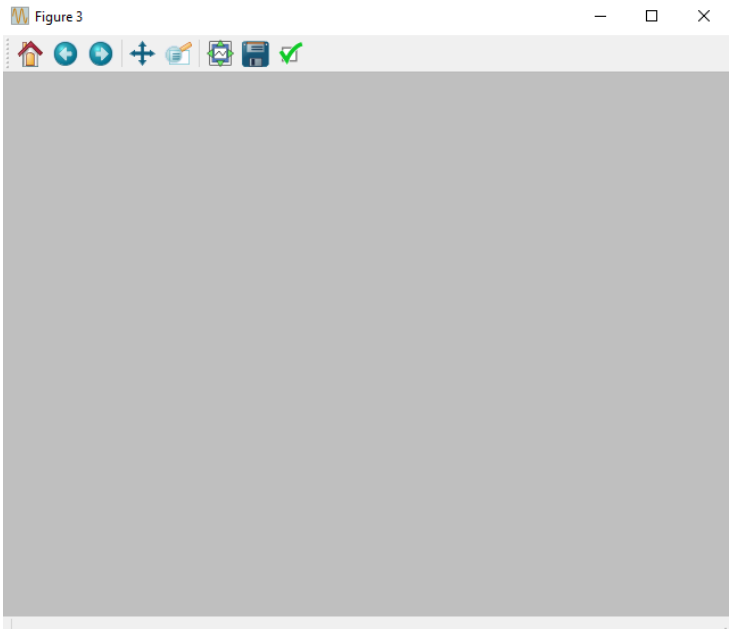
```
plt.figure(i)
```

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…