Python
A la découverte de ce fabuleux langage. . .

A. BOISSARD - A.DEMOLLIENS

Chapitre 0

Table des matières

Chapitre 1

Premiers pas en python...

1 Installation de python

Le langage python est un langage simple, puissant et gratuit. Il existe de nombreuses distributions chacune avec quelques avantages... et inconvénients !

Si vous n'avez pas de distribution de python installée sur votre ordinateur, il faut commencer par en installer une.

a Vous n'avez pas de distribution python installée et vous êtes utilisateur de Windows

Nous vous proposons de télécharger une distribution **WinPython**. C'est une version portable, très puissante, possédant différents IDE (Idle, Spyder, Pyzo) et de nombreux modules que nous utiliserons par la suite. Cette version se limite aux utilisateurs de Windows. Elle peut être installée sur un ordinateur de lycée sans avoir de droits d'administrateur !

Connectez-vous à l'adresse **www.chimsoft.com/download/carnot**, puis téléchargez l'archive **Win python** (environ 650 Mo !). Copiez-la sur votre machine et extrayez le contenu par exemple à la racine du disque C. Pour la suite nous considérerons que le dossier de WinPython est $C : \backslash WinPy$.

L'installation est terminée !

b Et pour mac ?

A titre personnel, j'utilise l'environnement Spyder sous Anaconda. L'installation se fait à l'adresse : **anaconda.com** puis downloads. Le programme **Anaconda Navigator** est alors installé sur votre disque et permet de lancer **Spyder**.

2 Environnement de développement

2.1 Quel IDE choisir ?

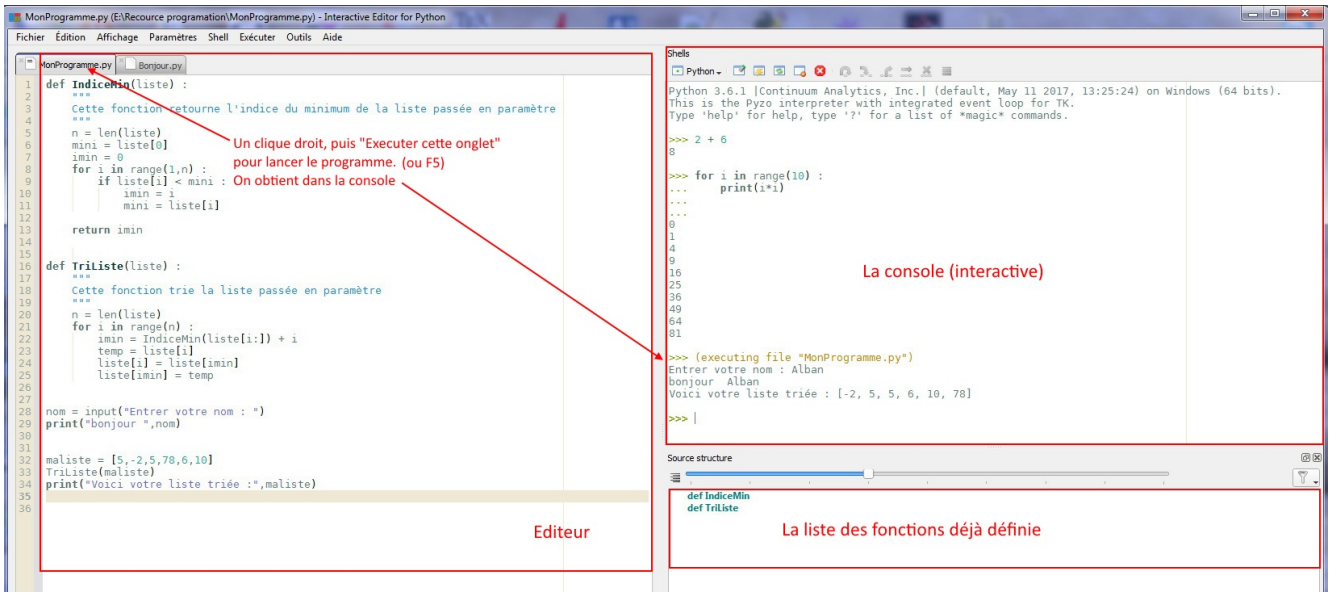
L'avantage de Winpython est (entre autres) de proposer plusieurs Ide. Dans $C : \backslash WinPy$, vous aurez le choix entre :

- **IDLE** : Il est très basique, offre peu de fonctionnalités, mais n'a besoin d'aucune configuration, est extrêmement rapide à démarrer et facile à prendre en main. Il faut, par contre, ajouter une à une les bibliothèques dont on a besoin. Son interface est différente des deux distributions suivantes.

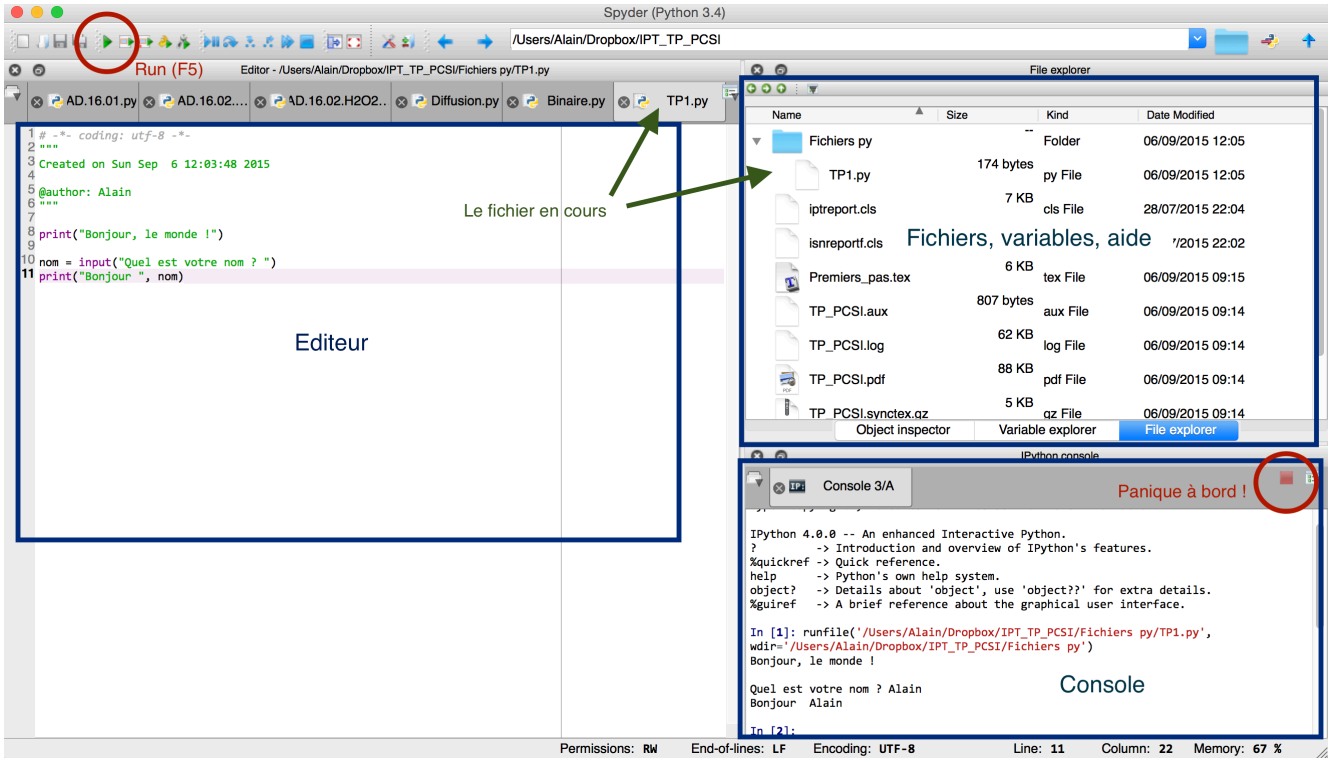
- **Spyder** : Environnement performant, simple d'emploi également, nécessite rarement l'installation de bibliothèques python car de très nombreuses sont déjà importées... au prix d'un démarrage un peu lent et d'une prise en main peut-être un peu plus longue.
- **Pyzo** : Un compromis entre les deux. C'est la configuration retenue au niveau des lycées pour l'apprentissage de python.

2.2 Un éditeur pour éditer et une console pour consulter !

Lorsque charge le programme `Pyzo`, on obtient



Et avec `Spyder` :



Que doit-on repérer en priorité ?

- Une zone **Editeur** qui sert... à éditer le programme dont vous souhaitez l'éxécution ; ce sera votre terrain de jeu !

- Une zone **Console** dans laquelle seront affichés les résultats de vos calculs, les graphes, très souvent... les messages d'erreur ! C'est donc une zone d'interface entre le programme et l'utilisateur.
- La touche F5 vous permet de lancer l'exécution du programme. Avec Spyder, il se peut qu'une console ne soit pas ouverte. Cliquez sur l'onglet `Ouvrir une console I-python` du menu `Console`.
- Parfois votre programme va se bloquer (suite à une erreur de programmation !) ; on peut arrêter son exécution à l'aide du bouton « Terminer » dans les icônes de la console (pyzo) ou en cliquant sur le carré rouge (spyder).
- Une fenêtre en bas à droite garde une trace des fonctions et variables définies dans votre programme (pyzo). Un explorateur de variables est également présent avec Spyder

2.3 La console comme « super calculatrice »... à éviter !

On peut se servir de la console comme d'une calculatrice ou d'un environnement pour réaliser quelques tests simples.

D'un point de vue pédagogique, nous déconseillons de montrer aux élèves que l'on peut à la fois faire des choses dans l'éditeur et dans la console. La logique dans le traitement que l'on cherche à réaliser n'est pas toujours la même et peut être source d'erreur.

Autant leur dire : L'éditeur c'est fait pour éditer et la console pour consulter !

3 Premiers programmes

3.1 Hello world !

Comme il se doit, tout tutoriel doit commencer par afficher ce message.

Pour afficher une information (du texte ici) dans la console l'instruction est ***print()***. Votre premier programme est donc :

```
1 print("Bonjour le monde !")
```

Saisir et exécuter ce premier programme (flèche verte dans la barre d'icône ou équivalent dans la barre de menu). Le message doit s'afficher dans la console.
Une petite variante consisterait à affecter la chaine de caractères (entre guillemets) à une variable *message* ici et d'afficher le contenu de la variable. L'opérateur d'affectation en python est $=$.

```
1 message = "Bonjour tout le monde !"
2 print(message)
```

Une remarque importante, ici : lors de l'affectation, on ne précise pas, en langage python, le type de la variable.

3.2 Petits calculs

On souhaite maintenant réaliser des calculs avec python. Les 4 opérations arithmétiques de base sont implémentées ainsi que **a**b** (pour élever a à la puissance b), **a %b** pour le reste de la division

entière de a par b.

Recopier et exécuter le programme suivant :

```
1 x = 3
2 y = (x + 2)**2
3 print("x = ", x, "y = ", y, "z = ", z)
4
5 # un autre calcul
6 q = int(7 % x)
7 r = 7 % x
8 print("Lors de la division entière de 7 par ", x, " le
    quotient vaut ", q, " et le reste ", r)
```

Le texte derrière le symbole dièse (ou hashtag pour faire d'jeunes) # est un commentaire. Il n'est pas pris en compte par python, mais permet de rendre plus clair un programme par le lecteur. On peut également librement sauter des lignes dans un programme.

Signalons que l'on peut également mettre en commentaire plusieurs lignes de code en l'insérant entre deux séries de 3 guillemets doubles """ et """.

```
1 x = 3
2 """
3 y = (x + 2)**2
4 print("x = ", x, "y = ", y, "z = ", z)
5 """
6
7 q = int(7 % x)
8 r = 7 % x
9 print("Lors de la division entière de 7 par ", x, " le
    quotient vaut ", q, " et le reste ", r)
```

3.3 Utilisation de bibliothèques

On souhaite maintenant calculer la racine carrée d'un nombre. Le langage python ne dispose pas, de base, de la fonction racine carrée. Pour pouvoir en bénéficier, il faut utiliser une bibliothèque de fonctions (que l'on appelle aussi module). Dans notre cas, nous allons importer la bibliothèque « *math* ». Pour cela, il faut écrire, en tête du programme, la ligne suivante : « `from math import *` »

```
1 from math import *
2
3 x = 2
4 r = sqrt(nombre)
5 print("Racine carrée de", x, " = ", r, " ou ", round(r,3))
```

L'instruction ***round(nombre, nb_digit)*** permet d'afficher le nombre *nombre* avec la précision souhaitée.

Chapitre 2

Structures de données

1 Types de données en python

1.1 Principaux types de données

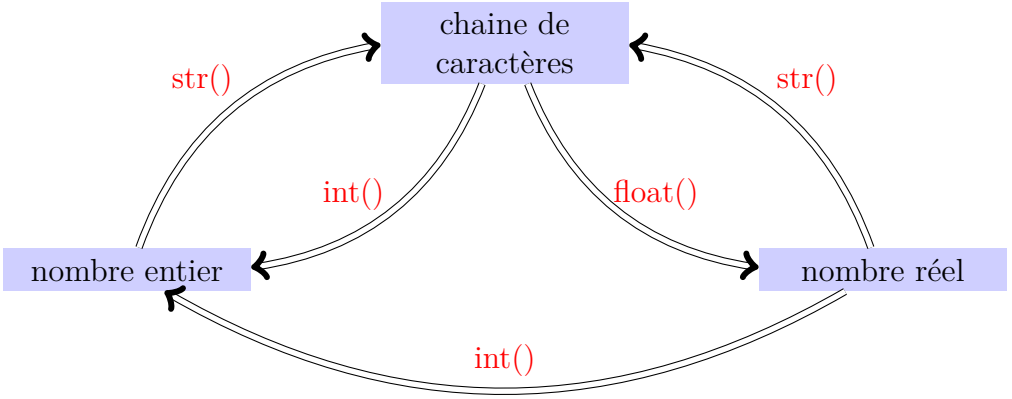
Le langage de programmation python utilise plusieurs *types de données*. Les plus courants sont les suivants :

Types de données	`Nom en python`
Nombre entier	`int`
Nombre réel	`float`
Variable booléenne : vrai ou faux (True/False)	`bool`
Liste (ordonnée)	`list`
Chaîne de caractères (du texte...)	`str`

En python, les variables ne sont pas typées (ou plus précisément, le type des variables n'est pas précisé). On ne peut toutefois pas faire n'importe quoi... par exemple, si on tente de faire des opérations mathématiques sur une variable de type chaîne de caractères, on obtient une erreur (même si le texte est constitué de chiffres).

1.2 Conversion de type

L'instruction ***str()*** permet de convertir un nombre entier ou réel en la chaine de caractères correspondante. Pour peu que la chaine de caractères représente effectivement un nombre entier ou réel, les instructions ***int()*** et ***float()*** permettent de convertir une chaine de caractères en nombre.



1.3 Entrée/sortie et affectation : input, print, =

Recopier le programme suivant :

```
1 nom = input("Ecrire votre nom : ")
2 print("Bonjour ", nom)
```

L'instruction ***input("message")*** permet, dans la console, d'afficher *message* et de demander à l'utilisateur d'entrer une chaine de caractères.
Exécuter le programme suivant :

```
1 x = input("Entrer un nombre : ")
2 x = x + 2
3 print(x)
```

Un message d'erreur (plus ou moins compréhensible) apparait dans la console. En effet, on tente ici d'ajouter un nombre à une chaine de caractères. Le programme suivant permet de rectifier cette erreur très « classique » !

```
1 ch = input("Entrer un nombre : ")
2 x = float(ch)
3 x = x + 2
4 print(x)
```

2 Structure de listes

Savoir manipuler les listes est primordial en python ! En exagérant un peu, « en python, tout est liste ! ».

2.1 Listes python

Sous Python, on peut définir une liste comme une collection ordonnée d'éléments séparés par des virgules, l'ensemble étant enfermé dans des accolades. Les différents éléments ne sont pas forcément de même type.
Chaque élément est repéré par son indice (ou index, ou rang), mais attention, le premier élément est celui d'indice 0, le deuxième d'indice 1 etc...

ATTENTION - ATTENTION - ATTENTION - ATTENTION

L'indice commence à 0 !

ATTENTION - ATTENTION - ATTENTION - ATTENTION

Recopier et exécuter le programme suivant :

```
1 liste = ["je","tu","il"]
2 print("la liste est : ", liste)
3 print("elle contient", len(liste), "éléments")
4 print("L'élément d'index 1 est : ", liste[1])
5 print("C'est le 2ème élément ! ")
6 liste.append("nous")
7 print("la liste est maintenant: ", liste)
8 print("elle contient maintenant", len(liste), "éléments")
```

```
9  del(liste[0])
10 print("la liste est maintenant: ", liste)
```

Observer les résultats...

2.2 Fonctions de base de manipulation de listes

`maliste = []`	crée une liste vide nommée « maliste »
`maliste.append(9)`	ajoute l'élément « 9 » en queue de liste
`len(maliste)`	renvoie le nombre d'éléments (la longueur) de la liste
`maliste[i]`	renvoie l'élément de rang i contenue dans maliste

Il existe de nombreuses autres fonctions comme par exemple :

- del(maliste[i]) qui supprime l'élément d'indice i de la liste nommée ;
- maliste.sort() qui trie la liste par ordre croissant ;
- ...

Nous reviendrons en détail sur la manipulation de listes dans le chapitre 5 lorsque nous aurons vu les instructions itératives.

2.3 Chaîne de caractères en python

Une chaîne de caractère est simplement une liste dans laquelle chaque élément est un caractère. On peut donc utiliser les fonctions vues précédemment, mais python fourni également des fonctions spéciales pour traiter les chaînes de caractères, par exemple l'opérateur « + ».
Les chaînes de caractères se notent entre les symboles ″″ou ''.
Exemple :

```
1 ch1 = "Isaac"
2 ch2 = "Newton"
3 print(ch1[0],ch1[1],ch1[2])
4 ch3 = "Monsieur" + ch1 + ch2
5 ch4 = "Monsieur " + ch1 +" "+ ch2
6 print(ch3, "de longueur :",len(ch3))
7 print(ch4, "de longueur :",len(ch4))
```

L'opérateur + concatène deux chaînes de caractères, c'est à dire les rassemble en une seule. On remarque également que les espaces sont des caractères. On peut déclarer une chaîne vide de la manière suivante : chaîne = ″″.

Chapitre 3

Avec des si...

Il s'agit de permettre l'exécution d'une série d'instructions uniquement si une condition est vérifiée et, éventuellement, d'en exécuter d'autres si la condition n'est pas vérifiée.

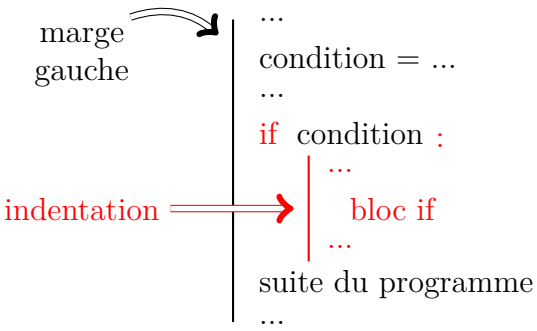
1 Implémentation

En python, *si* se traduit par ***if*** et *sinon* se traduit par ***else*** .

Pour spécifier les lignes de codes qui seront exécutées ou non ne fonction de la condition, une syntaxe particulière (et généralisée par la suite) est utilisée en python :

1. la ligne spécifiant le test à réaliser se termine par **:**
2. les lignes du « bloc de code » à exécuter sont indentées (4 caractères par défaut) ;
3. la fin de l'indentation spécifie la fin de ce bloc de code.

La gestion de l'indentation est capitale en python !
La structure générale de l'implémentation de l'instruction conditionnelle est :



Soit, par exemple, le programme suivant :

```
1 test = True # à changer en False !
2 if test :
3     print("bloc de code exécuté")
4     print("si la condition test")
5     print("est vérifiée")
6 print("Ligne toujours exécutée")
```

Un « bloc ***else*** » peut être ajouté .
Voici un exemple en python, à reproduire et observer :

```
1 n=int(input("Enter un nombre entier"))
2 if n < 10 :
3     print(n, " est plus petit que 10")
```

```
4 else :
5     print(n, " est plus grand ou égale à 10")
6 print("Ceci est écrit systématiquement")
```

On peut aussi faire des tests successifs avec le mot clef ***elif***, qui signifie « sinon si »

```
1 n=int(input("Enter un nombre entier"))
2 if n < 10 :
3     print(n, " est plus petit que 10")
4 elif n==10 :  # Attention : le test d'égalité s'écrit "=="
5     print(n, " est égale à 10")
6     print("c'est gagné !")
7 else :
8     print(n, " est plus grand que 10")
9 print("Ceci est écrit systématiquement")
```

Voici les opérateurs de comparaison en Python

opérateur de comparaison	signification
==	égal
!=	différent
<=	inférieur ou égal
<	strictement inférieur
>=	supérieur ou égal
>	strictement supérieur

L'instruction pour tester une égalité n'est pas « ~~if a = b :~~ » mais « if a == b : » . Le symbole « = » est réservé aux affectations de valeurs ; l'égalité se teste à l'aide de l'instruction ==

2 Opérations booléennes

2.1 Variables booléennes

En python, on peut utiliser des **variables booléennes**. Ce sont des variables qui n'ont que deux valeurs possibles : vraie (`True`) ou fausse (`False`)
Les valeurs booléennes sont très pratiques.

2.2 Les opérations logiques (dites « booléennes »)

Dans le tableau suivant, a et b sont des variables booléennes. (ou des expression contenant un test)

expression logique	rôle
a and b	Vraie si a **et** b sont vraies
a or b	vraie si a **ou** b (ou les deux) sont vraies
not(a)	si a est vraie, not(a) est fausse, et inversement

Chapitre 4

Boucles

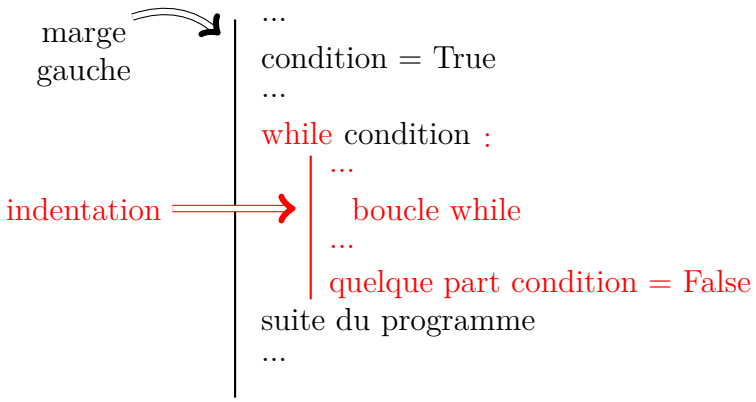
Il s'agit de répéter un certain nombre de fois une série d'instructions.

1 Tant que...

1.1 Objectif

Répéter certaines instructions tant qu'une condition est vraie (ou plus généralement tant qu'une opération booléenne portant sur un ensemble de conditions est vraie).
En python, *tant que* se traduit par *while* (en minuscule).

La structure générale de l'implémentation d'une telle structure itérative est :



Il faut obligatoirement initialiser la condition avant la boucle while !

Il faut obligatoirement modifier la condition dans la boucle while !

Voici un exemple pour comprendre le fonctionnement de la boucle `while` :

```
1 n = 0
2 while (n <= 21) :
3     print(n)
4     n = n + 3
5 print("c'est fini")
```

1.2 Boucle while sur test logique ou sur compteur de boucle

L'utilisation d'une boucle `while` comme compteur de boucle est un cas particulier fréquemment rencontré (par exemple, nous le verrons, pour le tracé de courbes). Il est bon de retenir sa syntaxe

générale.

Utilisation d'un test

```
flag = True
while flag :
    ...
    instructions à éxécuter
    ...
    if (condition) :
        flag = False
suite du programme
```

Utilisation d'un compteur de boucle

```
i = debut
while i <= fin :
    ...
    instructions à éxécuter
    ...
    i = i + pas
suite du programme
```

1.3 Pièges et difficultés

Le plus grand piège est de ne pas modifier les paramètres dans la boucle while... on obtient alors une boucle infinie (ou jamais parcourue) puisque la condition est toujours (ou jamais) vérifiée.
Exemple : Afficher les entiers successifs de 0 à 100.

```
1 x=0
2 while x <= 100 :
3     print(x)
4 x = x + 1 # Erreur ! cette instruction n'est pas dans la
    boucle, qui sera donc infinie ! il manque une tabulation !
```

Bien sûr, si on initialise mal les paramètres ou si on code mal la condition d'arrêt, le programme ne fera pas ce que l'on souhaite.

2 Pour...

La principale utilité de ce type de boucle est de répéter certaines instructions un certain nombre de fois. Ce nombre étant connu avant même la première itération.

2.1 Implémentation

L'implémentation d'origine de la boucle *Pour* en python, est de parcourir les éléments d'une liste. L'instruction :

for t ***in*** T ***:***

permet de récupérer successivement dans la variable t tous les éléments initialement contenus dans la liste (ou tableau) T.
Par exemple le script suivant permet d'imprimer dans la console les différents jours de la semaine.

```
1 semaine = ["dimanche", "lundi", "mardi", "mercredi", "jeudi",
    "vendredi", "samedi"]
2 for jour in semaine :
3     print(jour)
```

Le problème, ici, est que l'on ne peut pas accéder à l'indice du jour dans le tableau semaine.
L'instruction ***in range(****debut*, *fin*, *pas****)*** permet de générer une liste de valeurs **entières** : [*debut*, $debut + pas$, $debut + 2pas$, ... dernière valeur **strictement inférieure** à *fin*].

En combinant les deux instructions on va pouvoir récupérer soit une liste d'indices, soit une liste de valeurs.

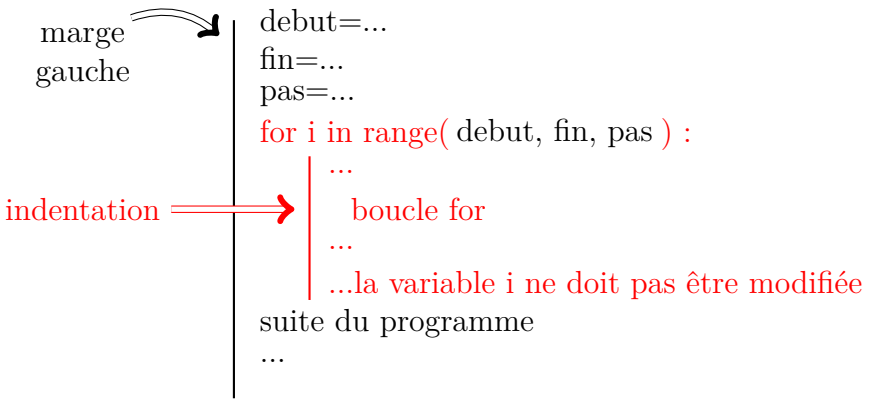
La boucle *Pour* se construit alors à partir de l'instruction :

for** i **in range(**début, fin* (valeur exclue), *pas* (= 1 par défaut)) :***

ATTENTION : les variables *début*, *fin* et *pas* sont obligatoirement des ENTIERS.

ATTENTION : La valeur *fin* n'est pas atteinte !

La structure générale de l'implémentation d'une telle structure itérative est :



Si l'incrément (ou le pas) vaut 1, on n'est pas obligé de le spécifier. On rencontre également une instruction du type for i in range(5). Dans ce cas, le début vaut forcément 0, on précise uniquement la *fin* (non atteinte) et le pas vaut 1. Dans le cas présent, *i* vaut successivement 0, 1, 2, 3, 4.
Par exemple :

Instruction	Valeurs successives de i
for i in range(0, 4)	0 ; 1 ; 2 ; 3
for i in range(-10, 10, 2)	-10 ; -8 ; -6 ; -4 ; -2 ; 0 ; 2 ; 4 ; 6 ; 8
for i in range(10, 5, -1)	10 ; 9 ; 8 ; 7 ; 6

2.2 Pièges et difficultés

- La principale erreur dans l'utilisation de l'instruction« `for i in range(debut, fin):` » est de croire que la valeur `fin` va être atteinte !
- La seconde erreur fréquente et de vouloir utiliser des incréments réels, afin de tracer une courbe par exemple.
- Comme pour toutes les structures de contrôle, il ne faut pas oublier les « : » à la fin de la ligne ; et bien sûr, comme ailleurs en python, seules les instructions indentées convenablement seront exécutées !
- Signalons enfin qu'il ne faut pas modifier la valeur du « compteur de boucle » à l'intérieur de la boucle.

2.3 Bilan : while ou for ?

A priori, la réponse est assez simple :

- si on connaît le nombre d'itérations : on utilise une boucle « for » ;

- si on ne connaît pas (ou pas facilement) le nombre d'itérations : on utilise une boucle « while » !

ou alors

- pour parcourir les éléments d'une liste on utilise la boucle « for » ;
- pour parcourir des nombres réels, pour tracer une courbe par exemple, on utilise la boucle « while » !

Parfois, les deux solutions sont possibles sans pouvoir privilégier l'une ou l'autre.

Chapitre 5

Ah, les listes !

1 Quelques rappels...

Sous Python, on peut définir une liste comme une collection ordonnée d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets. Les différents éléments ne sont pas forcément de même type.
Chaque élément est repéré par son indice (ou index, ou rang), mais attention, le premier élément est celui d'indice 0, le deuxième d'indice 1 ..., le dernier a l'indice len(liste)-1.

`maliste = []`	crée une liste vide nommée « maliste »
`maliste.append(9)`	ajoute l'élément « 9 » en queue de liste
`len(maliste)`	renvoie le nombre d'éléments (la longueur) de la liste
`maliste[i]`	renvoie l'élément de rang i contenue dans maliste

2 Opérations sur les listes

2.1 Accès aux éléments d'une liste

La boucle « for » est particulièrement bien adaptée aux listes de valeurs. Soit L une liste :

- si on a besoin de connaître **l'indice** de l'élément de la liste L, ce qui est toujours le cas si on souhaite modifier des éléments de cette liste, on parcourt ce dernier à l'aide de l'instruction :

$$\texttt{for } i \texttt{ in range(0, len(}L\texttt{)) :}$$

- si seules **les valeurs** des éléments de la liste L nous intéressent, on peut utiliser l'instruction :

$$\texttt{for } x \texttt{ in } L \texttt{ :}$$

On peut toujours utiliser la première version à la place de le deuxième, mais pas l'inverse !

2.2 Modification, ajout d'éléments à une liste

Par exemple, on dispose de la liste L = [3, 5, 18, 7, 12]

- On souhaite ajouter 1 à tous les éléments de la liste. On veut donc modifier les éléments de cette liste, et on doit utiliser la première version :

```
1 L = [3, 5, 18, 7, 12]
2 for i in range(0, len(L)):
3     L[i] = L[i] + 1
```

- On souhaite maintenant créer une nouvelle liste qui contient le carré des éléments de L. Dans ce cas l'indice des éléments ne nous sert à rien. On peut utiliser :

```
L = [3, 5, 18, 7, 12]
C = []  # C est une liste vide
for x in L :
    C.append(x**2) # on y ajoute le carré des éléments de L
```

3 Parcours d'une liste

Soit, par exemple, le calcul de la somme des éléments d'une liste :

Sans accéder aux indices

```
T = [1, 4, 10, 2, 8, 5]
s = 0
for t in T :
    s = s + t
print("Somme : ", s)
```

Avec accès aux indices

```
T = [1, 4, 10, 2, 8, 5]
s = 0
for i in range(len(T)) :
    s = s + T[i]
print("Somme : ", s)
```

On voit, ici, toute la puissance de l'instruction `for i in range(len(T))` qui permet de parcourir tous les éléments de la liste et d'y accéder par l'indice i...

Le programme suivant implémente :

- le calcul de la moyenne des éléments d'une liste ;
- la recherche du maximum ;
- la recherche du maximum et de l'indice de ce maximum (supposé unique).

```
Notes = [8, 12, 4, 16, 10, 8, 18, 15]
Noms = ["a", "b", "c", "d", "e", "f", "g", "h"]

""" calcul de la moyenne """
s = 0
for i in range(len(Notes)):
    s = s + Notes[i]
moy = s / len(Notes)
print ("La moyenne est : ", moy)
""" recherche du maximum """
maxi = Notes[0] # ou maxi = 0 si on veut
for i in range(len(Notes)) :
    if Notes[i] > maxi :
        maxi = Notes[i]
print ("La note maximale est : ", maxi)
""" recherche du maximum et de son indice (supposé unique) """
maxi = Notes[0] # ou maxi = 0 si on veut
indiceMax = 0 # ou -1
for i in range(len(Notes)) :
    if Notes[i] > maxi :
        maxi = Notes[i]
        indiceMax = i
print (Noms[indiceMax], " a la meilleure note : ", maxi)
```

Chapitre 6

Fonctions et représentation graphique

1 Les fonctions en Python

1.1 Objectif

Dès qu'un programme s'étoffe un peu, il devient peu lisible s'il est écrit « linéairement ». On préfère en général le décomposer en plusieurs sous programme, nommés **fonctions** (ensemble d'instructions) appelés par le programme principal.
Parfois certaines actions doivent être effectuées plusieurs fois... les écrire dans un sous programme est alors un choix judicieux.
Les avantage de définir des fonctions sont :

- le programme est plus lisible, et se comprend plus facilement ;
- le code est réutilisable ;
- le programme est moins long, si on fait plusieurs fois la même chose, on définit une fonction une seule fois pour cette tâche ;
- le programme est plus facile à debugger et à améliorer.

1.2 Implémentation

Voici un exemple de fonction qui calcule la moyenne des valeurs d'une liste, **passée en paramètre**, et renvoie cette moyenne.

```
1  """
2      Définition d'une fonction moyenne qui reçoit une liste L
           en paramètre et retourne la moyenne calculée
3  """
4  def moyenne(L): # Au sien de la fonction, la liste passée en
       paramètre se nommera L
5      n = len(L)
6      somme = 0
7      for i in range(n):
8          somme = somme + L[i]
9      moyenne = somme/n
10     return moyenne # on renvoie le contenu de la variable
           moyenne au programme principal
11
12 """
13     Programme principal
14 """
15 L1 = [5, 12, 8, 6, 11, 14, 12, 9, 16, 10, 8, 12]
16 L2 = [8, 13, 11, 9, 10, 11, 15, 12, 13, 10, 11, 9]
```

```
17
18 # on passe la liste L1 à la fonction moyenne et on stocke le
       résultat dans la variable moyenne1
19 moyenne1 = moyenne(L1)
20 # on passe la liste L2 à la fonction moyenne et on stocke le
       résultat dans la variable moyenne2
21 moyenne2 = moyenne(L2)
22
23 print("Moyenne 1 : ", moyenne1, "  moyenne 2 : ", moyenne2)
```

- Les fonctions sont définis par leur nom grâce à l'instruction `def maFonction(x,y) :`
  - le mot clef `def` signal à Python que l'on va définir une fonction ;
  - `maFonction` est le nom que l'on souhaite donner à la fonction ;
  - `x, y...` sont les **paramètres** que l'on souhaite transmettre à la fonction. Ces paramètres, en nombre quelconques sont introduits entre parenthèses et séparés par des virgules.
- Dans la fonction, l'instruction `return` renvoie au programme principale une valeur (ou une liste, ou une chaine de caractères etc.) et stop l'exécution de la fonction. Il n'est pas obigatoire qu'une fonction contienne une instruction `return`, auquel cas la fonction ne renvoie rien au programme principale (on parle alors de procédure)
- On appelle une fonction dans le corps du programme principal par « `maFonction(2,3)` » par exemple.
- Dans le cas d'une fonction, la(les) valeur(s) retournée doivent être stockées dans des variables : « `z = maFonction(2,3)` » ; la variable `z` contient alors le résultat du calcul effectué par `maFonction` lorsqu'elle reçoit les valeurs 2 et 3 comme paramètres.

1.3 Pièges et difficultés

La principale difficulté, au début, est de bien comprendre la notion de paramètres à passer à une fonction ou à une procédure. Comme sur votre calculatrice, il ne suffit pas d'écrire *sin*, par exemple, pour que le sinus du nombre auquel on pense soit calculé !
L'erreur classique, ensuite, consiste à supposer que du moment qu'on appelle une fonction avec les bons paramètres, le résultat du calcul est stocké quelque part... Le calcul est bien effectué mais la valeur de retour est perdue si on ne la stocke pas dans une variable : il faut absolument, dans le cas d'une fonction, avoir une structure du type : `variable_résultat = maFonction(paramètres)`
.
Abordons, ici, un cas plus délicat ; celui où une fonction retourne plusieurs valeurs. On ne peut pas exécuter plusieurs « `return` » de suite, car cette instruction met fin à l'exécution de la fonction. Par contre, on peut écrire « `return valeur1, valeur2` » ; et on récupèrera les résultats sous la forme « `variable1, variable2 = maFonction(paramètres)` ».

Par exemple, la fonction suivante retourne le périmètre et la surface d'un carré.

```
1 def carre(a):
2     p = 4*a
3     s = a*a
4     return p, s
5
6 perimetre, surface = carre(3)
7 print("Périmètre : " + str(perimetre) + " surface : " +
      str(surface))
```

2 Application : tracé de courbes

2.1 Bibliothèque matplotlib

Cette puissante bibliothèque permet de tracer quasiment tout ce que l'on veut. De nombreux exemples sont disponibles sur Internet.

La structure du programme minimal est la suivante :

```
1 import matplotlib.pyplot as plt
2
3 X = [] #liste de valeurs pour les abscisses
4 Y = [] #liste de valeurs pour les ordonnées
5
6 """ création des listes X et Y de même dimension """
7 ...
8
9 plt.plot(X, Y
```

Quelques remarques :

- la bibliothèque est trop volumineuse pour être chargée en mémoire, on préfère l'importer sous forme d'alias (plt en l'occurence) ligne 1 ;
- le corps du programme permet de créer deux listes X et Y de même dimension ;
- l'instruction plt.plo(X,Y) permet de tracer la courbe.

Le tracé de la courbe apparait soit dans la console, soit dans une fenêtre... éventuellement cachée en arrière plan ; surveillez bien la barre d'icones en bas de votre écran !

2.2 Représentation graphique d'une fonction

Une des principales applications des fonctions est quand même de pouvoir définir... une fonction ! et tracer sa courbe représentative.
Soit, par exemple, la représentation de la fonction $sin(k \times x)$ entre 0 et $2 * \pi$.

```
1  from math import sin, pi
2  import matplotlib.pyplot as plt
3
4  X = [] #liste de valeurs pour les abscisses
5  Y = [] #liste de valeurs pour les ordonnées
6  dX = pi/100 #incrément des valeurs sur x
7
8  """ cette partie est sous votre "responsabilité" """
9  def sinkx(k,x): #on définit la fonction que l'on veut étudier
10     return sin(k*x)
11
12 x = 0 #valeur de x en cours
13 while x <= 2*pi :#parcourt boucle tant que x est <= à 2 pi
14     X.append(x) # on complète le tableau des abscisses
15     Y.append(sinkx(4, x)) # et celui des ordonnées
16     x = x + dX # on n'oublie pas d'incrémenter la valeur de x
17
18 """ tracé de la courbe """
19 plt.plot(X, Y)# on a, au minimum, besoin des deux listes de
     valeurs X et Y
```

```
20 plt.show()
```

On se reportera à l'annexe de ce document, et à l'aide en ligne de Python, pour enrichir éventuellement ce graphe.

2.3 Exploitation d'un tableau de valeurs

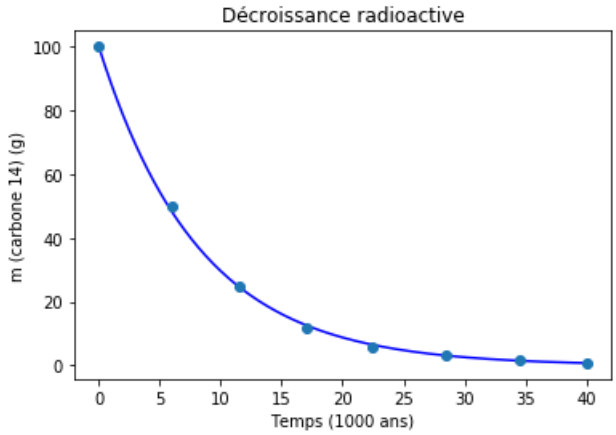
On dispose, par exemple, d'un tableau de valeurs donnant la quantité N de carbone 14 en fonction du temps (en milliers d'années).

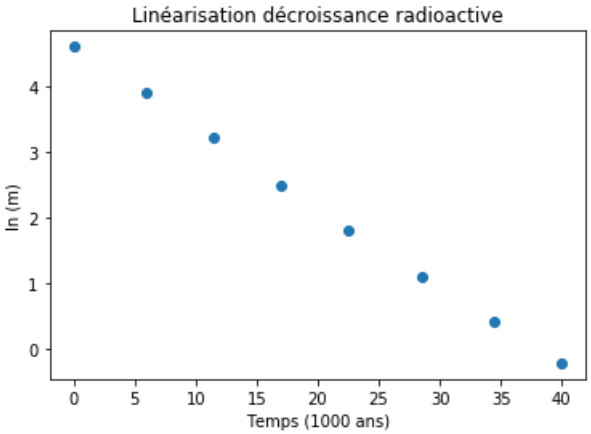
T (10^3 ans)	0	6	11.5	17	22.5	28.5	34.5	40
N (g)	100	50	25	12	6	3	1.5	0.8

On souhaite maintenant réaliser quelques traitements numériques.

```
1  from math import log, exp
2  import matplotlib.pyplot as plt
3
4  T = [0, 6, 11.5, 17, 22.5, 28.5, 34.5, 40]
5  N = [100, 50, 25, 12, 6, 3, 1.5, 0.8]
6
7  k = log(2)/5.7
8
9  def modele(t):
10     return 100*exp(-k*t)
11
12 Tcalc = []
13 Ncalc = []
14 t = 0
15 while t < 40 :
16     Tcalc.append(t)
17     Ncalc.append(modele(t))
18     t = t + 0.1
19
20 plt.plot(Tcalc, Ncalc)
21 plt.plot(T, N, 'o')
22 plt.xlabel("Temps (1000 ans)")
23 plt.ylabel("m (carbone 14) (g)")
24 plt.title("Décroissance radioactive")
25 plt.show()
26
27 lnN =[]
28 for i in range(len(N)):
29     lnN.append(log(N[i]))
30
31 plt.plot(T,lnN,'o')
32 plt.xlabel("Temps (1000 ans)")
33 plt.ylabel("ln (m)")
34 plt.title("Linéarisation décroissance radioactive")
35 plt.show()
```

Les sorties graphiques obtenues sont les suivantes :





Qu'a-t-on fait dans ce programme ?

- On commence par importer les bibliothèques nécessaires (lignes 1 et 2).
- Lignes 4 et 5, on recopie dans les listes T et N les valeurs contenues dans le tableau.
- On aimerait dans un premier temps, comparer les points à un modèle mathématique. La valeur de la constante k est fixée ligne 7 et on définit une fonction *modele* pour modéliser la décroissance exponentielle $e^{-k\,t}$.
- Lignes 12 à 18, on crée et on complète les tableaux de valeurs $Tcalc$ et $Ncalc$ correspondants aux grandeurs calculées à l'aide du modèle.
- L'instruction `plt.plot(Tcalc, Ncalc)` ligne 20 permet d'obtenir la courbe modèle (les différents points sont reliés par des segments de droite) alors qu'on précise ligne 21 de ne représenter que les points « expérimentaux » que par un petit cercle.
- On précise ensuite l'étiquette sur l'axe des abscisses, des ordonnées ainsi que le titre du graphe.
- Comme on va tracer une nouvelle courbe ensuite, afin qu'elle ne se superpose pas, on utilise l'instruction `plt.show()` afin d'afficher le tracé du graphe.
- On cherche ensuite à linéariser la courbe expérimentale. On crée un tableau lnN ligne 27 que l'on complète par les différentes valeurs souhaitées.
- On trace ensuite le second graphe (lignes 31 à 35).
- Reste à faire une régression linéaire... nous le proposons en exercice dans le chapitre suivant et nous aborderons ce point dans le second document « Python : méthodes numériques pour le lycée ».

Chapitre 7

A vous de jouer...

Comme vous le dites à vos élèves... « jouez le jeu ! » ; nous vous donnons quelques pistes de résolution et la solution par la suite... essayez de résoudre sans aide !
L'importation de fonctions de la bibliothèque mathématique (cf Annexe) sera nécessaire pour certains exercices.

1 Quelques exercices

1.1 Chapitres 1, 2, 3 & 4

a Exemples « simples »

a.1 Écrire à l'aide d'une boucle `for` un programme qui calcule la somme de nombres pairs entre 1 et 100 (inclus) et affiche le résultat.

a.2 Écrire un programme qui demande un nombre x, calcule $\frac{x^2}{2}$ si $x < -2$ ou $x > 4$; $4 - \frac{x^2}{2}$ sinon, puis affiche le résultat.

a.3 Écrire un programme qui affiche les nombres réels compris entre 0 et 1 (inclus) espacés de 0,1 :

a) à l'aide d'une boucle `while` ;
b) à l'aide d'une boucle `for`

b Un petit peu plus difficile...

b.1 Écrire à l'aide de deux boucles `for` imbriquées un programme qui calcule la somme : $S = \sum_{i=1}^{n} \sum_{j=1}^{n} ij$. Que vaut cette somme pour $n = 10$?

b.2 Écrire un programme qui calcule la somme des entiers k positifs tels que $k + k^2 + k^3 \leq n$, n étant un nombre introduit dans le programme. Que vaut cette somme pour $n = 1000$?

b.3 Écrire un programme qui calcule la limite de la suite $S_n = \sum_{i=1}^{n} \frac{1}{i^2}$ sachant que l'on arrête l'exécution lorsque $|S_{n+1} - S_n| < \epsilon$ avec ϵ proche de zéro.

Vous vérifierez que la limite de la suite est une bonne approximation de $\frac{\pi^2}{6}$ et en déduirez une valeur approchée de π.

b.4 On introduit un entier positif n. Écrire un programme qui calcule la somme S_n des termes suivants :

$$\begin{array}{ccccccc} 1\times 1 & + & 1\times 2 & + & \cdots & + & 1\times n \\ & + & 2\times 2 & + & \cdots & + & 2\times n \\ & & & \ddots & & & \vdots \\ & & & & \ddots & & \vdots \\ & & & & & + & n\times n \end{array}$$

Que vaut S_8 ?

b.5 Écrire un programme qui demande les nombres a, b et c dans l'équation $ax^2+bx+c=0$ et qui donne en retour la(les) solution(s) de cette équation (ou affiche un message si elles n'existent pas).

1.2 Chapitre 5

a Exemples « simples »

a.1 A l'aide d'une boucle, remplir une liste :

a) qui contient tous les nombres entiers de 0 à 10 inclus ;
b) qui contient tous les nombres entiers pairs de 10 à -10 inclus ;
c) qui contient tous les nombres compris entre 0 et 10 (inclus) espacés de 0,1.
d) qui contient les 26 lettres minuscules de l'alphabet.

a.2 Intialiser les 3 tableaux suivants :

```
1 Eleves = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
2 Trim1 = [10, 12, 9, 14, 8, 15, 18, 13, 6, 10]
3 Trim2 = [11, 12, 8, 16, 12, 16, 16, 11, 8, 9]
```

Ecrire le code d'un programme permettant de :

a) déterminer le nombre d'élèves ayant progressé du premier au second trimestre ;
b) calculer la moyenne du premier trimestre ;
c) déterminer la meilleure note du premier trimestre ;
d) déterminer l'élève qui a le meilleur note au premier trimestre ;
e) déterminer la listes des élèves qui ont la meilleure note au second trimestre.

b Un petit peu plus difficile...

b.1 Nombres premiers Construire la table des nombres premiers inférieurs à $N = 1000$.

- première méthode : on construit la table, nombre premier par nombre premier en testant chaque nouveau nombre susceptible d'être premier.
- deuxième méthode : on utilise la méthode du crible d'Eratosthène (on élimine d'une table des entiers de 2 à N tous les multiples d'un entier premier).

1.3 Chapitre 6

a Exemples « simples »

a.1

- Écrire une fonction qui prend un nombre n en paramètre et qui renvoie le cube de ce nombre.
- Écrire une fonction `ValAbs` qui prend une nombre en paramètre, et renvoie sa valeur absolue. (Sans utiliser la fonction intégrée à Python)

a.2 Écrire une fonction qui accepte comme paramètre la longueur de l'arête d'un carré et retourne l'aire et le périmètre de ce carré. Coder l'impression du résultat sur un exemple de votre choix.

b Un petit peu plus difficile...

b.1 Courbe du Blanc-Mange

Dans cet exercice, on cherche à tracer une approximation de la courbe du Blanc-Mange (en référence à un dessert qui ressemble à cette courbe).
On définie les fonctions B_n, avec n un entier par :

$$B_n(x) = \sum_{k=0}^{n} \frac{1}{2^k} \left| 2^k x - E\left(2^k x + \frac{1}{2}\right) \right|$$

- La fonction E est la fonction partie entière. On la trouve en python sous le nom `floor`, dans la bibliothèque `math`. (`floor(12.95) = 12`)
- Les barres verticales désignent la valeur absolue.

Plus n est grand, plus la courbe de B_n ressemble à la "vraie" courbe du blanc-mange.

1. Créer une fonction python `B(n,x)` qui renvoie la valeur de $B_n(x)$.
2. Tracer la courbe de B_n sur $[0, 1]$. A tester pour $n = 10$, 100, 1000 par exemple.

1.4 Deux problèmes de synthèse !

On propose deux méthodes numériques permettant d'exploiter les données sur la décroissance radioactive (fin du chapitre précédent).

a Optimisation

L'idée est de chercher la « meilleure » valeur de k qui permet de modéliser la décroissance radioactive par une fonction $N_0 e^{-k\,t}$.

Tentons la version « brutale » : écrire un programme permettant de déterminer (à 0,01 près par exemple) la valeur de k pour laquelle l'écart entre valeurs tabulées et valeurs modélisées est minimal.

Quelques pistes sont suggérées dans la partie « Quelques pistes ».

b Régression linéaire

On considère un nuage de n points $M_i(x_i, y_i)$ que l'on désire ajuster au mieux par une courbe $y = a \times x + b$. Dans la méthode des moindres carrés, on cherche à minimiser la somme des distances entre les points M_i et la droite.
On note :

- $\overline{x}$ et $\overline{y}$ la moyenne des X et des Y ;
- $\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2}$ et $\sigma_y = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{y})^2}$ les écarts types correspondants ;
- $Cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x}) \times (y_i - \overline{y})$ la covariance (moyenne du produit des écarts à la moyenne)

Dans ces conditions :

- le coefficient de corrélation linéaire vaut $r(X,Y) = \frac{Cov(X,Y)}{\sigma_x \sigma_y}$;
- la pente de la droite de corrélation : $a = \frac{Cov(X,Y)}{(\sigma_x)^2}$
- l'ordonnée à l'origine de la droite de corrélation vaut $b = \overline{y} - a\overline{x}$

1. Écrire la routine $regrelin(X,Y)$. Les paramètres d'appel de cette fonction sont les deux tableaux X et Y (de même dimension) ; la valeur retournée : un tuple [1] contenant, dans l'ordre, a, b et le coefficient de corrélation r.
2. Tester à l'aide des tableaux de valeurs $X = [1,2,3,4,5]$ et $Y = [3.1, 4.9, 7, 8.9, 11.2]$. On trouve, sur Excel $a = 2,02$, $b = 0,96$ et $r^2 = 0,99843$.
3. Copier l'intégralité du code correspondant aux regressions linéaires dans un fichier nommé, par exemple utilitaire.py que vous placez dans votre dossier de travail. Ecrire from utilitaire import * permet alors d'utiliser ces routines dans n'importe quel programme. Essayez !

2 Quelques pistes...

2.1 Chapitre 1, 2, 3 & 4

a Exemples « simples »

a.1

- une boucle `for i in range` fera l'affaire (attention à la borne supérieure !) ;
- pour sommer, on initialise une variable que l'on incrémente !

a.2

- la gestion du `if/else` ne devrait pas poser de problème ;
- pensez à convertir la chaine de caractères (obtenue grâce à `input`) en entier !

1. 3 valeurs séparées par des virgules

a.3

- de classiques parcours de boucles... ;
- que l'on peut rater en oubliant l'incrémentation (boucle `while`) ou en se trompant dans les bornes (boucle `for`).

b Un petit peu plus difficile...

b.1

- si l'on sait faire une boucle pour faire une somme... ce n'est pas plus difficile d'en imbriquer deux !
- vérifiez que pour $n = 10$ la somme vaut 3025.

b.2

- on ne connait pas, *a priori*, la plus grande valeur de k telle que $k + k^2 + k^3 \leq n$; il faut donc utiliser une boucle `while` !
- il faudra alors gérer la variable k d'une part, la variable de test $k + k^2 + k^3$ à la fois dans l'initialisation et dans la boucle `while`.
- vérifiez que pour $n = 1000$ la somme vaut 45.

b.3

- cette fois encore, on ne sait pas immédiatement quand on va devoir s'arreter donc... on fait une boucle tant que l'incrément est plus grand qu'une limite fixée à l'avance.
- la fonction racine carrée est codée par `sqrt` ; il faut l'importer ou importer la bibliothèque `math` :

b.4

- la principale difficulté, ici, est la gestion des indices : on a un indice l pour la ligne qui doit aller de 1 à n inclus ; et un indice c pour la colonne qui doit aller de l (inclus) à n inclus [2].
- pour $n = 8$, je trouve 750.

b.5 Equation du second degré

- cette fois encore il faut convertir les chaines de caractère en réel ;
- la fonction racine carrée est codée par `sqrt` ; il faut l'importer ou importer la bibliothèque `math` (c'est la dernière fois que je le rappelle !).

2.2 Chapitre 5

a Exemples « simples »

a.1 A tout hasard... la table des codes ASCII permet de passer d'une lettre à un nombre et vice versa.

2. J'ai toujours eu du mal avec les indices i et j pour savoir qui était la ligne et qui était la colonne ; un jour, j'ai eu l'idée d'appeler l l'indice pour la ligne et c l'indice pour la colonne... c'est con mais c'est efficace !

a.2

a) fastoche !
b) un jeu d'enfant.
c) idem.
d) ne me dites pas que vous n'y arrivez pas !
e) un peu plus malan ! Essayez de récupérer la liste des meilleurs élèves à l'aide d'une seule boucle !

b Un petit peu plus difficile...

b.1 Nombres premiers

- On ajoute dans la liste des nombres premiers en cours de construction tout nombre qui n'admet pas de diviseur dans cette liste de nombre premiers. On pourra utiliser l'opérateur modulo (%) : a % b renvoie le reste de la division euclidienne de a par b. Ainsi : a%2 = 0 si a est pair, 1 si a est impair.
- L'idée est de partir d'une liste qui contient [0, 0, 2, 3, 4 ..., N]. On parcourt la table en mettant à 0 tous les multiples d'un nombre non nul. Et enfin, on récupère les nombres non nuls !

2.3 Chapitre 6

2.4 Problèmes

a Optimisation

On pourrait, par exemple :

1. définir une fonction $modele(N0, k, t)$ retournant la valeur $N_0 e^{-k\,t}$ en fonctions des paramètres.
2. définir une fonction $distance(k)$ qui calcule la somme des valeurs absolues des écarts, pour chaque valeur de t, entre la valeur de N et la valeur modèle.
3. pour k variant, par exemple, de 0,01 à 0,2 par pas de 0,01 on pourrait alors chercher la valeur minimale de k, soit en construisant une liste contenant toutes les distances correspondantes soit directement ;
4. resterait à tracer la courbe pour vérifier !

b Régression linéaire

1. En principe, vous n'avez que 3 fonctions (autre que regrelin) à programmer !
2. C'est l'heure de vérité pour votre fonction !
3. Pas de soucis pour peu que le fichier se trouve dans le répertoire de travail. Sinon, il faudra prévenir spyder ou pyzo pour lui dire où aller chercher les fichiers.

c Exemples « simples »

d Un petit peu plus difficile...

d.1 Courbe du Blanc-Mange

C'est surtout un exercice de gestion de parenthèses ! Si on a peur de se tromper dans les priorités, mieux vaut en mettre plus (sans abuser toutefois). Avec spyder, cliquer juste après une parenthèse permet de mettre en surbrillance la parenthèse ouvrante ou fermante correspondane.
Pour $n = 1000$, j'ai pris un pas de 0,001.

3 Solutions !

3.1 Chapitres 1, 2, 3 & 4

a Exemples « simples »

a.1

```
1 S = 0
2 for i in range(0,102,2):
3     S = S + i
4 print("Somme des nombres pairs de 1 à 100 : ", S)
```

Bien sûr, si la valeur 100 doit être incluse, il faut aller jusqu'à 102.

a.2

```
1 ch = input("Entrez un nombre !")
2 x = float(ch)
3 if x < -2 or x > 4 :
4     y = x*x/2
5 else :
6     y = 4 - x*x/2
7 print("résultat = ", y)
```

On pourrait aussi écrire `x = float(input("Entrez un nombre !"))`.

a.3

```
1 print("Avec boucle while")
2 x = 0
3 while x <= 1 :
4     print(x)
5     x = x + 0.1
6
7 print("Avec boucle for")
8 for i in range(0,11):
9     print(i/10)
```

Comme les matheux semblent préférer la boucle `for` à la boucle `while`, vous verrez bon nombre de vos élèves gérer des listes de réels de la sorte !

b Un petit peu plus difficile...

b.1

```
1 n = 10
2 S = 0
3 for i in range(1,n+1):
4     for j in range(1, n+1):
5         S = S + i*j
6 print(S)
```

il faut bien écrire `in range(1,n+1)` pour atteindre la valeur n ; bon OK je ne vous le redis plus !

b.2

```
n = 1000
S = 0
k = 1
test = 3
while test < n :
    S = S + k
    k = k + 1
    test = k + k**2 + k**3
print (S)
```

C'est l'écriture typique d'une boucle `while` sur un test logique, il faut prendre garde à bien initialiser le test avant d'entrer dans la boucle... et de le modifier ensuite.

b.3

```
epsilon = 1e-12
S = 0
i = 1
increment = 1
while increment > epsilon :
    S = S + increment
    i = i + 1
    increment = 1/(i*i)
print("Limite = ", S)
print("pi = ", sqrt(6*S))
```

La division est évaluée avant la multiplication : si on écrit $1/i*i$ il comprend $(1/i)*i$. Par contre la puissance est évaluée avant toute autre opération (sauf le moins unitaire) donc si on écrit 1/i**2 ; c'est bon. Bon, si on écrit $1/i\hat{}2$, autant dire que l'on a une erreur de syntaxe !

b.4

```
n = 8
S = 0
for l in range (1,n+1):
    for c in range(l, n+1):
        S = S + l*c
print("Somme = ", S)
```

Il me semble avoir dis que je ne vous disais plus de faire attention aux indices...

b.5 Equation du second degré

```
cha = input("a = ?")
chb = input("b = ?")
chc = input("c = ?")
a = float(cha)
b = float(chb)
c = float(chc)
if a == 0 :
    print("Solution unique : ", -c/b)
else :
    delta = b*b - 4*a*c
    if delta > 0 :
```

```
12          print("Deux racines : ", (-b + sqrt(delta))/(2*a), "
                et ", (-b - sqrt(delta))/(2*a))
13      elif delta == 0 :
14          print("Solution unique : ", - b/(2*a))
15      else :
16          print("Aucune racine")
```

3.2 Chapitre 5

a Exemples « simples »

a.1

```
1  L1 = []
2  for i in range(0,11):
3      L1.append(i)
4  # ou L1 = [i for i in range(0,11)], idem pour les suivantes
5
6  L2 = []
7  for i in range(10,-12,-2):
8      L2.append(i)
9
10 L3 = []
11 x = 0
12 while x <= 10 :
13     L3.append(x)
14     x = x + 0.1
15
16 # ou
17 #L3 = []
18 #for i in range(0,101):
19 #    L3.append(i/10)
20
21 L4 = []
22 for i in range(ord("a"), ord("a")+26):
23     L4.append(chr(i))
24 print(L4)
```

- au début, on construit les listes avec des `append` puis, avec l'habitude, on utilise les listes par compréhension !
- `ord` récupère le caractère ASCII de la lettre, on incrémente ce code et on récupère la caractère à l'aide de `chr`

a.2

```
1  #a)
2  n = 0
3  for i in range(len(Trim1)):
4      if Trim2[i] > Trim1[i]:
5          n = n + 1
6  print ("Nombre de progrès : ", n)
7
8  #b)
9  s = 0
```

```
10 for note in Trim1 :
11     s = s + note
12 moyenne =  s / len(Trim1)
13 print("Moyenne : ", moyenne)
14
15 #c)
16 maxi = Trim1[0]
17 for i in range(1, len(Trim1)):
18     if Trim1[i]> maxi :
19         maxi = Trim1[i]
20 print("Meilleure note : ", maxi)
21
22 #d)
23 maxi = Trim1[0]
24 indice = 0 # ou best = Eleves[0]
25 for i in range(1, len(Trim1)):
26     if Trim1[i]> maxi :
27         maxi = Trim1[i]
28         indice = i # ou best = Eleves[i]
29 print("Meilleure note : ", maxi, " pour ", Eleves[indice])
30 # ou print("Meilleure note : ", maxi, " pour ", best)
31
32 #e)
33 maxi = Trim2[0]
34 best = [Eleves[0]]
35 for i in range(1, len(Trim2)):
36     if Trim2[i] == maxi :
37         best.append(Eleves[i])
38     elif Trim2[i] > maxi :
39         maxi = Trim2[i]
40         best = [Eleves[i]]
41 print("Meilleure note : ", maxi, " pour ", best)
```

a) rien à dire puisque je ne vous parle plus des problèmes d'indices !

b) un classique

c) on initilise maxi, ici, avec la valeur du premier élément de la liste, on aurait tout aussi bien pu prendre -1 comme valeur.

d) Il faut, bien sûr, repérer en même temps que le maximum, l'indice correspondant (ou directement le nom dans la table Eleves).

e) Une solution « bourrin » consisterait à rechercher la meilleure note puis de parcourir la liste Eleves à la recherche des élèves qui ont cette note. En une seule boucle, on fait comme précédemmnet, simplement best n'est plus un nom (ou un indice) mais la liste des noms. Si on trouve une meilleure note, il faut réinitialiser cette liste (ligne 40).

b Un petit peu plus difficile...

b.1 Nombres premiers

```
1 N = 10000
2
3 start = time.time()
4 P = []
5 for i in range (2,N+1):
```

```
6       trouve = False
7       j = 0
8       while j < len(P) :
9           if i%P[j] == 0:
10              trouve = True
11          j = j + 1
12      if not trouve :
13          P.append(i)
14  interval = time.time() - start
15  print (P)
16  print (interval)
17
18  start = time.time()
19  E = [0,0]# 0 et 1 ne sont pas premiers !
20  for i in range(2, N+1):
21      E.append(i)
22  for i in range(2, int(sqrt(N))+1):
23      if E[i] != 0 : # c'est un nombre premier
24          for j in range(2*i, N+1, i):
25              E[j]= 0
26  Pr = []
27  for i in range(N+1):
28      if E[i]!= 0 :
29          Pr.append(i)
30  interval = time.time() - start
31  print (Pr)
32  print (interval)
```

Pour le fun, j'ai ajouté un décompte du temps nécessaire à l'éxécution d'un bloc de code. La méthode « bourrin » prend (pour $N = 1000$) 3,41 s alors que celle utilisant le crible d'Ertosthène ne prend que 5,4 ms (sur mon ordinateur).
Sinon, initialiser le tableau E avec 0 et 0 (pour les indices 0 et 1) permet d'avoir l'indice d'un nombre égal à ce nombre.

3.3 Chapitre 6

a Exemples « simples »

a.1

```
1  def cube(x) :
2      return x**3
3
4  def ValAbs(x) :
5      if x > 0 :
6          return x
7      else :
8          return -x
```

a.2 Une fonction avec deux valeurs de retour

```
1  def carre(a):
2      A = a**2
3      P = 4*a
```

```
4       return A, P
5
6   aire, per = carre(3)
7   print ("Carré de côté : ", a, " aire = ", aire, " périmètre =
        ", per)
```

Il ne faut surtout pas écrire deux instructions `return` de suite. L'éxécution du bloc de la fonction stoppe dès la première instruction `return`. Dans un code bien écrit, on ne devrait avoir qu'une seule instruction `return` à la fin de la routine

On aurait très bien pu terminer la fonction par return [A,p]. Lors de l'appel de la fonction, on stockerait le résultat dans une liste : L = carre(3) et on récupèrerait l'aire par L[0] et le périmètre par L[1].

b Un petit peu plus difficile...

b.1 Courbe de Blanc-Mange

```
1   def B(n,x):
2       b = 0
3       for k in range(0, n+1):
4           u = (2**k)*x
5           b = b + abs((u-floor(u+0.5))/(2**k))
6       return b
7
8   n = 1000
9   X =[]
10  Y = []
11  x = 0
12  while x <= 1 :
13      X.append(x)
14      Y.append(B(n,x))
15      x = x + 0.001
16
17  plt.plot(X,Y)
18  plt.show()
```

J'ai calculé $(2 * *k) * x$ à part pour me simplifier la vie avec les parenthèses mais ce n'est pas nécessaire.

3.4 Correction des problèmes

a Optimisation

```
1   from math import log, exp
2   import matplotlib.pyplot as plt
3
4   T = [0, 6, 11.5, 17, 22.5, 28.5, 34.5, 40]
5   N = [100, 50, 25, 12, 6, 3, 1.5, 0.8]
6
7   def modele(N0,k,t) :
8       return N0*exp(-k*t)
9
10  def distance(k):
```

```
    d = 0
    for i in range(len(T)) :
        d = d + abs(N[i] - modele(100,k,T[i]))
    return d

k = 0.01
minD = 10**6
bestK = -1
while k <= 0.20 :
    D = distance(k)
    print(D)
    if D < minD :
        minD = D
        bestK = k
    k = k + 0.01

print("Meilleur modèle : k = ", round(bestK,2))
print("Demi-vie  : ", round(log(2)/bestK, 1), "x 1000 ans")

Tcalc = []
Ncalc = []
t = 0
while t < 40 :
    Tcalc.append(t)
    Ncalc.append(modele(100, bestK,t))
    t = t + 0.1

plt.plot(Tcalc, Ncalc, 'b')
plt.plot(T, N, 'o')
plt.xlabel("Temps (1000 ans)")
plt.ylabel("m (carbone 14) (g)")
plt.title("Décroissance radioactive")
plt.show()
```

b Régression linéaire

```
def moyenne(X):
    m = 0
    for x in X :
        m = m + x
    return m/len(X)

def ecart_type(X):
    m = moyenne(X)
    u = 0
    for x in X :
        u = u + (x-m)**2
    return sqrt(u/len(X))

def cov(X,Y):
    mx = moyenne(X)
    my = moyenne(Y)
```

```
17      c = 0
18      for i in range(len(X)):
19          c = c + (X[i]-mx)*(Y[i]-my)
20      return c/len(X)
21
22  def regrelin(X,Y):
23      pente = cov(X,Y)/ecart_type(X)**2
24      ordonnee = moyenne(Y)-pente*moyenne(X)
25      coeff = cov(X,Y)/(ecart_type(X)*ecart_type(Y))
26      return pente, ordonnee, coeff
27
28  """ test """
29  X=[1,2,3,4,5]
30  Y=[3.1, 4.9,7,8.9,11.2]
31
32  a,b,r = regrelin(X,Y)
33  print(" a = ", a, " b = ", b, "r^2 = ", r**2)
```

Chapitre 8

Annexes

1 Éléments de base du langage python

1.1 Type et conversion

a Les type de données Python

Il existe différents **types de données**, dont voici les principaux :

Types de données	`Nom en python`
Nombre entier	`int`
Nombre « à virgule »	`float`
Variable booléenne : vrai ou faux (True/False)	`bool`
Liste (ordonnée)	`list`
Chaîne de caractères (du texte...)	`str`

Il en existe d'autres comme les ensembles non ordonnés (`set`) ou les dictionnaires (`dict`)...

Conversion de type

On peut demander à python de convertir le type d'une variable. Il suffit d'utiliser la fonction associé au nom du type. Attention, le résultat de la conversion doit être stocké dans une variable.

Attention : Le résultat de la fonction `input` est toujours une chaine de caractère. Si elle doit être interprétée comme un nombre, il faut la convertir à l'aide des fonctions `int` ou `float`

b Opérations booléennes

Une variable booléenne peut être :

- soit vraie : $True$
- soit fausse : $False$

On peut effectuer sur ces variables différentes opérations logiques :

- et : *and* ;
- ou : *or* ;
- ou exclusif : $\wedge$
- négation : *not*

1.2 Opérations arithmétiques de base

Un certain nombre d'opération de base sont disponible. Elles sont représentées, sans surprise, par les symboles : *+*, **-**, ***, /. On peut également utiliser des parenthèses.

Ajoutons **** pour une puissance (a**b revient à a^b) ou encore *%* pour un modulo, ***abs*** pour la valeur absolue.
Attention aux ordres de priorité des opérateurs. Exécuter le programme suivant et, s'il ne fait pas ce que vous souhaitez, ajoutez des parenthèses !

```
1 a = 3
2 b = 2
3 print(a + b/a - b)
4 print(a/b*b)
5 print(a/b**2)
```

Mais, la plupart des opérations usuelles (fonctions trigonométriques, π, logarithme ou exponentielle...) ne sont pas disponibles et il faudra quasiment systématiquement importer la bibliothèque mathématique.

1.3 Listes (ou tableau)

Nous présentons, ici, quelques manipulations de base sur les listes.

- `T=[]` permet d'initialiser un tableau vide ;
- `len(T)` permet de connaître le nombre d'éléments du tableau ;
- `T.append(valeur)` permet d'ajouter un élément au tableau

On peut parcourir un tableau :

- en accédant directement au contenu du tableau par l'instruction « for x in T » : ;
- par l'intermédiaire de l'indice d'un élément : $T[i]$ (instruction qui suit, par exemple, « for i in range(0, len(T)) »)

Une matrice doit être considérée comme une liste de listes.

2 Bibliothèques python

2.1 Importation de bibliothèques

Le langage python ne dispose pas, de base, de la fonction racine carrée, sinus, expoentielle... Pour pouvoir en bénéficier, il faut utiliser une bibliothèque de fonctions (que l'on appelle aussi module). Dans notre cas, nous allons importer la bibliothèque ***math***. Pour cela, il faut écrire, en tête du programme, une des lignes suivantes :

- « from math import * » ; on importe alors la totalité de la bibliothèque mathématique ;
- « from math import sqrt, sin, pi » ; on importe alors uniquement les deux fonctions souhaitées ;
- « import math as m » et l'on écrira alors [1] $m.sqrt(3)$ ou $m.sin(pi/4)$

Pour installer une bibliothèque non présente sur votre machine, il suffit de taper dans la console de Pyzo (Ici pour installer `numpy`) :

1. Cette méthode peut sembler un peu lourde de prime abord. Elle permet toutefois d'éviter des conflits si une même fonction est définie dans plusieurs bibliothèques. Elle permet également de disposer d'une aide contextuelle lors de la frappe ; il suffit de taper m. et ensuite un menu déroulant apparait avec toutes les fonctions disponibles

2.2 Bibliothèque « math »

On sera, par exemple, amené à importer :

- fonction exponentielle : *exp* et logarithme : *log* (logarithme népérien) ou *log*10 (logarithme décimal)
- des fonctions trigonométriques : *sin*, *cos*, *tan*. L'angle est en radian !
- racine carrée : *sqrt*
- *floor* retourne la partie entière d'un nombre réel

2.3 Bibliothèque « matplotlib.pyplot »

Il ne s'agit pas, ici, de résumer toutes les possibilités de la bibliothèque matplotlib. On pourra se reporter à la page : **http ://matplotlib.org** puis aux onglets « examples » ou « docs » pour exploiter au mieux cette bibliothèque.
Cette bibliothèque permet le tracé de courbes. Dans la version minimale, il suffit de remplir deux tableaux de valeurs (de même taille) et d'appeler le tracé.

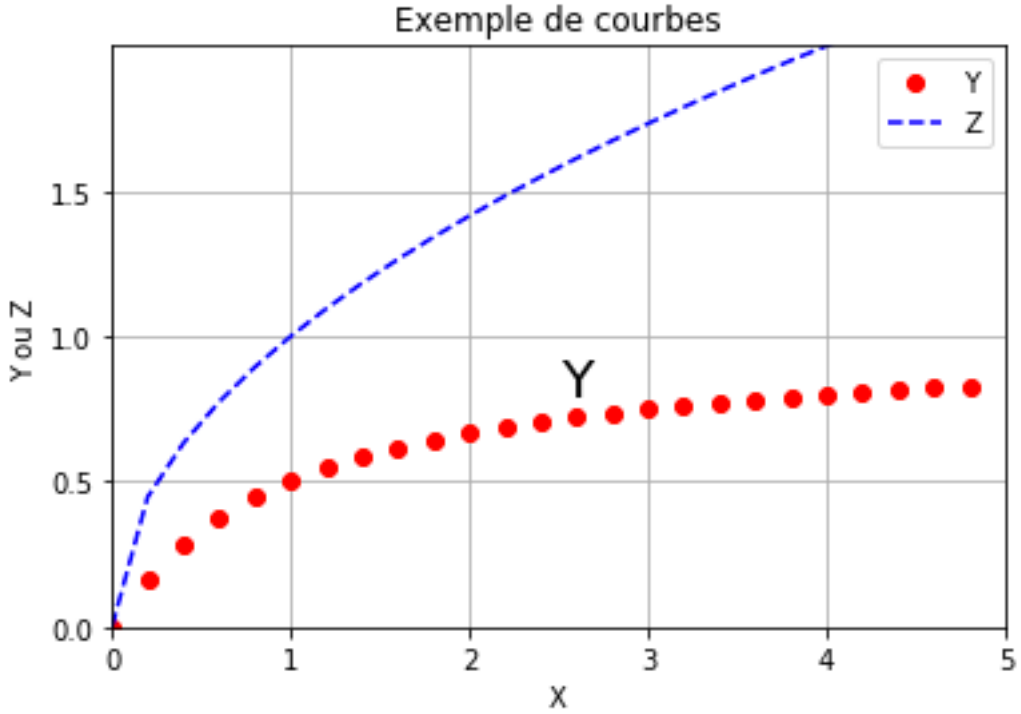
```
1  import matplotlib.pyplot as plt
2  X = []
3  Y = []
4  x = 0
5  while x <=5 :
6      X.append(x)
7      Y.append(x/(1+x))
8      x = x + 0.1
9
10 plt.plot(X, Y)
```

L'exemple ci-dessous permet d'enrichir le graphique :

- On ajoute une étiquette sur chaque axe, un titre et la légende des courbes.
- On précise, entre guillement, que pour la courbe Y en fonction de X, on ne représente que les points tabulés sous forme de ronds rouges et que la courbe Z est représentée en pointillé bleu.
- On a spécifié (lignes 18 et 19) les intervalles de tracé sur chacun des axes.
- On ajoute, ligne 20, du texte aux abscisses et ordonnées spécifiées en noir et avec une fonte de taille 20.
- La ligne 21 permet de préciser les étiquettes représentées sur l'axe y.
- On ajoute une grille (ligne 22).

```
1  import matplotlib.pyplot as plt
2  from math import sqrt
3  X = []
4  Y = []
5  Z = []
6  x = 0
7  while x <=5 :
8      X.append(x)
9      Y.append(x/(1+x))
10     Z.append(sqrt(x))
11     x = x + 0.2
12
```

```
13 plt.plot(X, Y, "ro", label="Y")
14 plt.plot(X, Z, "b--", label="Z")
15 plt.xlabel("X")
16 plt.ylabel("Y ou Z")
17 plt.title("Exemple de courbes")
18 plt.xlim(0,5)
19 plt.ylim(0,2)
20 plt.text(2.5, 0.8, "Y", {'color': 'k', 'fontsize': 20})
21 plt.yticks([i/2 for i in range(0,4)])
22 plt.grid()
23 plt.legend()
24 plt.show()
```



2.4 Bibliothèque « random »

Cette bibliothèque vous servira à la génération de nombres aléatoires.

```
1 import random as rd
2
3 print (rd.randint(1,6))      #entier aléatoire entre 1 et 6
    (inclus)
4 print (rd.random())      #nombre réel aléatoire entre 0 et 1
    (exclus)
```

2.5 Et pourquoi pas la tortue...

Turtle est un module python héritier d'une des plus vielles bibliothèques graphiques de l'informatique implémentées, à l'origine, en langage Logo.
Avec quelques instructions de base : reset() (pour effacer) ; forward(n) et backward(n) (pour avancer ou reculer de n pixels) ; left(angle) et right(angle) (pour tourner d'un angle donné en degré) ; up() (pour relever le crayon et donc se déplacer sans dessiner) ; down() (pour abaisser le crayon) ;

goto(x,y) (pour aller aux coordonées x,y) et color(couleur) (pour choisir la couleur red, blue...).

Que trace les petits programmes suivants ?

```
1 import turtle as t
2 t.reset()
3 t.forward (200)
4 t.left (45)
5 t.color("red")
6 t.forward(100)
```

```
1 from turtle import *
2 for i in range (5):
3     forward (200)
4     right (360 * 2/5)
```

3 Errare humanum est...

3.1 Erreurs lors de l'édition de code

Lors de l'écriture de programmes, on ne peut pas ne pas faire d'erreurs ou d'étourderies de syntaxe. Dans la console un message tente de nous expliquer l'origine mais, ce n'est pas toujours évident.

Au bout d'un certain temps, vous arriverez facilement à corriger vos erreurs, par contre corriger celles de vos élèves peut être un peu plus délicat !
Quelques messages sont très explicites :

Code	Message d'erreur
a = ln(2+3*(4-5))	NameError : name 'ln' is not defined
a = u + log(2+3*(8-5))	NameError : name 'u' is not defined
a = log(2+3*(4-5))	ValueError : math domain error
a = log(2+3*(8-5)))	SyntaxError : invalid syntax
print(T[3])#avec T=[1,2,3]	IndexError : list index out of range

d'autres un peu moins !

Code	Message d'erreur
a = log(2+3*(8-5)(2+8))	TypeError : 'int' object is not callable

'machin' object is not collable est une erreur très fréquente ; en gros on essaie de passer des paramètres à machin qui n'est pas une fonction... on cherche un peu et c'est la signature de l'oubli d'une opération entre 2 parenthèses !
Et enfin, une autre erreur très fréquente :

```
1 R =  8,31
2 RT = R*(25+273)
3 a = RT/1000
4 print(a)
```

TypeError : unsupported operand type(s) for / : 'tuple' and 'int'

Quand on a unsupported ... 'tuple' dans le même message d'erreurs, c'est que l'on a utilisé la virgule ou lieu du point comme séparateur décimal.
Dernière remarque enfin, le message d'erreur apparait lorsque l'interpréteur ne comprend plus rien ; il se peut que l'erreur provienne de la ligne précédente. C'est le cas, en particulier pour l'oubli de parenthèse fermante.

3.2 Gestion des erreurs lors de l'éxécution du programme

Après, le programme peut fonctionner mais que se passe-t-il si on n'introduit pas des données cohérentes ? On risque, par exemple, une division par zéro, des opérations sur des listes de tailles non adaptées...

La gestion de telles erreurs lors de l'exécution d'un programme est extrêmement complexe.

4 Gestion des variables

Ne sous-estimez pas, dans vos programmes, la gestion des variables !
Vous n'êtes pas obligés de tout comprendre ce qui suit (en particulier la partie 2.) mais il faut avoir conscience que l'on ne peut pas faire n'importe quoi avec les variables.

4.1 Variables globales, locales

Dans un programme, on peut très bien se trouver dans le cas de figure suivant :

- une fonction est définie par « def $maFonction(x)$: » ;
- quelque part on attribue à x une certaine valeur

Que se passe-t-il quand on appelle $maFonction$?

```
1 x = 2
2
3 def maFonction(x):
4     return x*x
5
6 y = maFonction(x)
7 z = maFonction(3)
8 print("x : ", x, " y : ", y, " z :", z)
```

Et bien... tout se passe bien car à l'intérieur de la fonction $maFonction$, x apparaît comme une variable **locale**. Cette variable a sa propre zone de stockage en mémoire, indépendamment de celle de la variable **globale** x définie dans le cœur du programme.
Pour éviter des conflits enter les différentes variables (on parle d'effet de bord) ; une fonction ne peut pas modifier de variable globale sauf si on spécifie explicitement que c'est une variable globale.
Que va-t-on avoir comme impression en exécutant le programme suivant ?

```
1  a = 3#variable globale
2
3  def fonction1(x):
4      y = a * x #on utilise la variable globale
5      print("Dans la fonction1 a = ", a)
6      return y
7  def fonction2(x):
8      a = 2 * x #a est redéfinie comme variable locale, x en est
           une aussi
9      print("Dans la fonction2 a = ", a)
10     return x*x
11 def fonction3(x):
12     global a# on précise que l'on veut utiliser la variable
           globale
```

```
13      a = 10 * x#ce qui permet de la modifier... à nos risques
            et périls
14      print("Dans la fonction3 a = ", a)
15      return x*x
16
17  print("Au départ : a = ", a)
18  y = fonction1(4)
19  print("Après fonction1 : a = ", a, " y = ", y)
20  y = fonction2(4)
21  print("Après fonction2 : a = ", a, " y = ", y)
22  y = fonction3(4)
23  print("Après fonction3 : a = ", a, " y = ", y)
```

La fonction *fonction3* est « dangereuse » car elle modifie la valeur d'une variable globale... en principe, seul le programme devrait être autorisé à le faire!!!

4.2 Variables mutables ou non mutables

Quesako ?

Un objet **mutable** est un objet que l'on peut modifier après sa création. Lorsqu'on modifie une liste, la liste est modifiée sans que sa place en mémoire change.

Un objet **non mutable** ne peut être modifié. Si on le modifie, on crée en fait une nouvelle instance de la variable à un nouvel emplacement mémoire. Les entiers, réels, chaînes de caractères sont non mutables.

Si on ne comprend pas tout, ce n'est pas grave mais il faut avoir, peut-être, conscience des conséquences.

```
1   a = 2
2   print("Avant : a=",a, " à l'adresse : ", id(a))
3   a = 4
4   print("Après : a=",a, " à l'adresse : ", id(a))
5
6   u = [4,3,2,1]
7   print("Avant : u=",u,"à l'adresse : ", id(u))
8   u.append(5)
9   print("Après : u=",u,"à l'adresse : ", id(u))
10
11  ch = "abcde"
12  print("Avant : ch=",ch,"à l'adresse : ", id(ch))
13  ch = ch + "f"
14  print("Après : ch=",ch,"à l'adresse : ", id(ch))
```

Lors de l'exécution du script précédent, on obtient les résultats suivants [2].
Avant : a= 2 à l'adresse : 4297327264
Après : a= 4 à l'adresse : 4297327328
Avant : u= [4, 3, 2, 1] à l'adresse : 4573137736
Après : u= [4, 3, 2, 1, 5] à l'adresse : 4573137736
Avant : ch= abcde à l'adresse : 4576922624
Après : ch= abcdef à l'adresse : 4567965624

2. L'instruction *id*() retourne l'adresse mémoire d'une variable donnée

L'adresse de a ou ch a changé, mais pas celle de u

a Conséquences sur l'affectation $x = y$

D'où piège lors de l'exécution du programme suivant :

```
1  a = 2
2  b = a
3  print("Avant : a=",a,"b=",b)
4  b = 3
5  print("Après : a=",a,"b=",b)
6
7  u = [4,3,2,1]
8  v = u
9  print("Avant : u=",u,"v=",v)
10 v[2] = 5
11 print("Après : u=",u,"v=",v)
12 v = [1,2,3,4]
13 print("Enfin : u=",u,"v=",v)
```

Aucun souci pour les types entier, réel, booléen, chaîne de caractère ; mais pour les listes l'instruction $v = u$ crée une nouvelle variable v qui pointe vers le même contenu mémoire que la variable u. Ces variables sont mutables, donc quand on modifie l'une d'elle, on n'en crée pas une nouvelle copie quelque part dans la zone mémoire mais on modifie le contenu de la zone mémoire commun aux deux variables.

Variable et zone mémoire					
variable non mutable			variable mutable		
booléen, entier, réel, chaîne de caractères			liste		
Instruction :	Variable(s) :	Zone mémoire :	Instruction :	Variable(s) :	Zone mémoire :
a = 2	a	2	u = [4,3,2,1]	u	4321
b = a	a, b	2	v = u	u, v	4321
b = 3	a b	2 3	v[2] = 5	u, v	4351

C'est grave, docteur ?
Un peu quand même ; pour y remédier il faut remplacer l'instruction $v = u$:

- soit par une copie dans le vecteur v de tous les éléments de u un par un
- soit, si on vous la donne, par l'utilisation de l'instruction *copy*() qui crée une nouvelle variable dans une nouvelle zone mémoire pour y stocker le contenu souhaité.

```
1  u = [4,3,2,1]
2  v = [x for x in u]
3  print("Avant : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
4  v[2] = 5
5  print("Après : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
6
```

```
7  u = [4,3,2,1]
8  v = u.copy()
9  print("Avant : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
10 v[2] = 5
11 print("Après : u=",u,"v=",v," aux adresses", id(u),"et",id(v))
```

A l'exécution u et v ne pointent pas vers la même zone mémoire et on peut donc modifier l'un sans modifier l'autre.
Avant : u= [4, 3, 2, 1] v= [4, 3, 2, 1] aux adresses 4585847560 et 4571253896
Après : u= [4, 3, 2, 1] v= [4, 3, 5, 1] aux adresses 4585847560 et 4571253896
Avant : u= [4, 3, 2, 1] v= [4, 3, 2, 1] aux adresses 4579894088 et 4585847560
Après : u= [4, 3, 2, 1] v= [4, 3, 5, 1] aux adresses 4579894088 et 4585847560

Remarque : dans le listing ci-dessus, on utilise une liste par compréhension, on pourrait tout aussi bien écrire une des solutions du code suivant :

```
1 v = [u[i] for i in range(len(u))]
2 # ou
3 v = []
4 for uu in u :
5     v.append(uu)
6 # ou
7 v = []
8 for i in range(len(u)):
9     v.append(u[i])
```

Nous ne reviendrons plus sur ces différentes alternatives par las suite.

b Conséquences sur les paramètres d'une fonction

Le fait d'accéder aux listes par leur adresse mémoire et non pas directement par leur valeur a également quelques conséquences (qui peuvent être bénéfiques !) lors du passage de paramètre à une fonction.
Qu'obtient-on à l'exécution du programme suivant ?

```
1  def permuteNombre(x,y):
2      print("Début de la fonction : x = ", x, "y = ", y)
3      temp = x
4      x = y
5      y = temp
6      print("Fin de la fonction : x = ", x, "y = ", y)
7
8  def permuteListe(T):
9      print("Début de la fonction : T = ", T)
10     temp = T[0]
11     T[0] = T[1]
12     T[1] = temp
13     print("Fin de la fonction : T = ", T)
14
15 a = 2
16 b = 3
17 print("Avant : a = ", a, "b = ", b)
18 permuteNombre(a,b)
19 print("Après : a = ", a, "b = ", b)
```

```
20
21 u = [2,3]
22 print("Avant : u = ", u)
23 permuteListe(u)
24 print("Après : u = ", u)
```

Avant : a = 2 b = 3
Début de la fonction : x = 2 y = 3
Fin de la fonction : x = 3 y = 2
Après : a = 2 b = 3
Avant : u = [2, 3]
Début de la fonction : T = [2, 3]
Fin de la fonction : T = [3, 2]
Après : u = [3, 2]

On constate :

- Que les valeurs des variables non mutables ne sont pas changées dans le corps du programme (même si on appelait a et b le nom des variables dans la fonction *permuteNombre* car ce sont des variables « muettes »). Pour se faire, il faudrait rajouter $returnx, y$ à la fin de la fonction *permuteNombre* et faire l'appel de la fonction sous la forme $a, b = permuteNombre(a, b)$.
- Que le tableau u est changé dans le corps du programme après l'appel de la fonction *permuteListe*. En fait, on a passé à la fonction non pas directement la liste mais l'adresse de la liste et, dans la fonction, on travaille sur l'adresse de cette liste.

Par conséquent, lorsqu'on travaille sur une liste ou un tableau :

- ce n'est pas nécessaire de faire un *return* pour récupérer le tableau modifié dans le corps du programme ;
- en contre-partie, il faut faire attention car si on modifie le tableau dans la fonction, le tableau est également modifié dans le corps du programme...