Chapitre 3

Introduction à l'algorithmique et l'informatique avec Python

Nous ferons tout de même des séance sur ordinateur car il est important, pour se familiariser avec l'informatique de tester les différentes commandes.

Les questions d'informatique dans les concours prennent de plus en plus d'importance au fil des années. Il ne faut surtout pas négliger cet aspect du programme.

La langage de programmation officiel du programme de Mathématiques Approfondies en ECG est **Python**. Il n'y a pas de séances de travaux pratiques devant ordinateur notées le jour des concours. Les questions d'informatiques sont directement insérées dans les sujets des écrits des concours. Elles consistent en écrire des programmes avec le langage Python, ou compléter des programmes à trous ou encore expliquer ce que fait un programme. Le but de l'informatique en filière ECG n'est donc pas de connaître avec précision Python (les codes ne seront jamais exécutés le jour des concours) qui sert juste de cadre de langage pour écrire des structures algorithmiques. Ce sont elles qui sont évaluées principalement.

L'objectif de ce chapitre introductif est double :

- présenter brièvement les types d'objets Python principaux et les opérations que l'on peut faire sur ces objets.
- présenter les structures de programmation élémentaires : concept de fonction, structures conditionnelles (if/then/else), structures itératives (boucles for, boucles while).

Par exemple, dans le chapitre 7, nous verrons comment calculer les termes successifs d'une suite. Dans le chapitre 14, nous verrons comment trouver une valeur approchée de la solution d'une équation du type $f(x) = 0$.

Nous développer les autres aspects du programme d'informatique dans les prochains chapitres de Mathématiques.

En annexe du polycopié d'exercices, il est expliqué comment installer Python sur votre ordinateur et comment le faire fonctionner.

I Les commandes de base en Python

1) Différents types d'objets

En Python (comme en Mathématiques) il y a différents types d'objets que l'on peut affecter à des variables avec la commande **nom=expression**. Le contenu de `expression` (une expression mathématique) est alors stocké dans une variable appelée `nom`.

Parmi les types d'objets, il y a notamment :

- Les entiers (type `int`).
- Les réels ou flottants (type `float`).
  En mathématiques les entiers sont des réels. En Python ce sont deux types différents : par exemple `1` est entier et `1.0` est un réel et Python ne considèrera pas le deuxième comme un entier. Il faut donc y penser quand on utilise une commande demandant un entier (comme `range`) et que l'on dispose d'un réel. La commande `int(a)` transforme un réel a en un entier (en enlevant les chiffres après la virgule).
- Les chaînes de caractère (type `str`). Pour définir une chaîne de caractères, on écrit les caractères entre guillemets (`"`) ou bien entre apostrophes (`'`). Si dans la chaîne, il y a déjà des guillemets ou des apostrophes, on les fait précéder de `\`.

  Par exemple `'Je m\'appelle Bond, James Bond.'`

  Une chaine de caractère contenant un mot ne doit pas être confondue avec le mot lui-même (sans `"` ou `'`, Python considèrera qu'il s'agit d'une variable).

Il y a plein de subtilités autour des entiers et des flottants pour Python qu'il n'est pas utile d'explorer en ECG.

Par exemple `"eps"` *affiche le mot* eps *alors que* eps *affiche le contenu de la variable* eps.

La commande `str(x)` transforme le contenu de la variable `x` en la chaîne de caractère correspondant au contenu de `x`.

Si `x=5`, alors la commande `'x'` renvoie `'x'`, alors que la commande `str(x)` renvoie `'5'`.

- Les booléens (type `bool`) qui ne prennent que deux valeurs : `True` (vrai) ou `False` (faux).
- Les listes (type `list`, cf. paragraphe 5).
- Les tableaux (type `ndarray`, cf. chapitre 19).
- Les fonctions (type `function`, paragraphe II.2 et chapitre 6).

La commande `type(A)` renvoie le type de l'objet `A`.

2) Opérations de base sur les entiers et sur les réels

- En Python, pour séparer la partie entière et la partie décimale d'un nombre, on utilise le point et non pas la virgule. La virgule sert de séparateur lorsque l'on veut construire une liste (cf. chapitre 4). Le point virgule est un séparateur permettant d'écrire plusieurs commandes sur une seule ligne.
- Soient x et y deux réels implémentés en Python dans les variables `x` et `y`. Les commandes **`x+y`**,**`x-y`**,**`x*y`** renvoient respectivement la somme de x et y, la soustraction de x par y et la multiplication de x par y. Si $y \neq 0$, **`x/y`** renvoie la division de x par y. Enfin, sous réserve d'existence, **`x**y`** renvoie x^y.

  Par exemple la commande `3**(1/2)` *renvoie une valeur approchée de* $\sqrt{3}$.

  La commande `x+=y` ajoute définitivement `y` à la variable `x` (c'est-à-dire elle remplace le contenu de `x` par le résultat de `x+y`). Elle fait la même chose que `x=x+y`.

  Les commandes `x-=y`, `x*=y`, `x/=y` et `x**=y` fonctionnent sur le même principe mais avec la soustraction, la multiplication, la division et l'élévation à la puissance.

  Si `x` et `y` sont de type entier, alors `x+y`,`x-y`,`x*y` sont encore de type entier. C'est aussi le cas pour `x**y` si `y` implémente un entier naturel. Cependant `x/y` renvoie un réel (flottant). Par exemple `(7/2)*2` renvoie `7.0` et non `7`.

  Les opérations ont un ordre de priorité (d'abord les **P**arenthèses, puis les **E**xposants, puis les **M**ultiplications/**D**ivisions de gauche à droite et enfin les **A**dditions/**S**oustractions de gauche à droite), comme en mathématiques... à la différence que l'on ne peut pas utiliser de grandes barres de factions horizontale ni de notation en exposant. Ainsi on doit utiliser beaucoup plus de parenthèses (mieux vaut trop que pas assez).
- Si x et y deux entiers implémentés en Python dans les variables `x` et `y`, alors la commande `x%y` renvoie le reste de la division euclidienne de x par y et la commande `x//y` le quotient.

La commande `x%y` est utile (pour savoir si x est un multiple de y notamment) mais elle n'apparaît pas dans le programme.

- On peut également évaluer des réels par des fonctions usuelles (cf. paragraphe 5) ou des fonctions que l'on définit soi-même (cf. paragraphe II).
- Python nous permet d'accéder à des approximations de π et de $e = \exp(1)$ via les commandes **`np.pi`** et **`np.e`** respectivement. Mais avant cela il faut importer la bibliothèque `numpy` en utilisant la commande `import numpy as np` (on en reparlera).

3) Opérations sur les booléens

Les booléens sont très utiles puisqu'ils apparaissent dans les conditions des structures conditionnelles et des boucles while (cf. paragraphe II).

- En général les booléens sont créés en faisant des comparaisons entre objets. Si `x` et `y` désignent deux objets Python (par forcément du même type), alors **`x==y`** renvoie `True` si ce sont les mêmes objets et `False` sinon. La commande **`x!=y`** renvoie `True` si ce sont deux objets distincts et `False` si ce sont les mêmes.

Ne pas confondre `==` qui teste l'égalité et `=` qui sert à l'affectation dans une variable. C'est une erreur classique et donc grave !

Ne pas confondre == qui teste l'égalité et = qui sert à l'affectation dans une variable. C'est une erreur classique et donc grave !

- Si x et y sont des réels implémentés en Python par `x` et `y`, alors les commandes **`x==y,x<y,x<=y,x>y,x>=y,x!=y`** renvoient `True` si x est respectivement égal à y, strictement inférieur à y, inférieur à y, strictement supérieur à y, supérieur à y et différent de y. Elles renvoient `False` sinon.
- On peut faire des opérations sur les booléens avec les commandes **`and,or,not`** qui implémentent respectivement les $\text{et}, \text{ou}, \text{non}$ logiques.

  Par exemple, `x>3 and x<=5` *renvoie* `True` *si et seulement si* $x \in]3\,;5]$. *La commande* `x>=8 or x<-2` *renvoie* `True` *si et seulement si* $x \in]-\infty\,;-2[\cup [8\,;+\infty[$.
- Si x est un réel implémentés en Python par `x`, alors la commande `int(x)==x` renvoie `True` si x est un entier, `False` sinon.

Ces trois commandes ne sont pas explicitement au programme mais sont très utiles.

- Si n et p sont des entiers (avec $p \neq 0$) implémentés en Python par `n` et `p`, alors :
  - `(n%2)==0` renvoie `True` si n est pair, `False` sinon.
  - `(n%2)==1` renvoie `True` si n est impair, `False` sinon.
  - `(n%p)==0` renvoie `True` si n est divisible par p, `False` sinon.

4) Opérations sur les listes

On peut construire une liste d'objets (pas forcément du même type) entre crochets et séparés par des virgules.

Par exemple `L=[1,2,0,8,-9,7,4,0]`.

- La commande `[]` crée une liste vide.
- Si `x` est une variable, alors la commande `x in L` renvoie `True` si la variable `x` est un élément de la liste `L` et `False` sinon.
- Si `L` est une liste, la commande **`len(L)`** renvoie le nombre d'éléments de la liste.

Dans ce cours, on différentiera indice (numérotation à partir de 0) et position (numérotation à partir de 1) dans la liste.

- ⚠ Python numérote les indices d'une liste à partir de 0 et non de 1. Ainsi le premier élément de la liste est, pour Python, celui d'indice 0. Le deuxième élément de la liste est, pour Python, celui d'indice 1, etc.
- `L[i]` renvoie la $(i+1)^{\text{ième}}$ coordonnée de `L`. Ainsi, pour accéder au $i^{\text{ième}}$ élément de `L`, la commande est `L[i-1]`.
- `L[-1]` renvoie le dernier coefficient de `L`.
- On peut modifier des éléments en utilisant la syntaxe usuelle pour l'affectation dans une variable.

  Par exemple `L[2]=5` *remplace le* $3^{ième}$ *(celui d'indice* 2*) élément de* L *par* 5.

Les commandes append, pop et len ne sont pas explicitement au programme mais sont très utiles.

- On peut ajouter un élément `a` à une liste `L` avec la commande `L.append(a)`.
  La commande `L.pop(i)` renvoie et enlève l'élément d'indice `i` (i.e. celui en position `i+1`) de la liste `L`.
- Si `L` et `M` sont deux listes, alors la commande `L+M` concatène les deux listes.
  Ainsi la commande `L=L+[a]` ajoute aussi l'élément `a` à la liste `L`.

Concaténer deux listes signifie les regrouper pour en créer une nouvelle.

- Si `L` est une liste et `a` un entier, alors `a*L` concatène `a` fois la liste `L`.

  Par exemple `10*[0]` *renvoie* `[0,0,0,0,0,0,0,0,0,0]`.
- La commande `L[:i]` renvoie la sous-liste de `L` constituée des `i` premiers éléments de `L` (i.e les éléments de l'indice 0 à l'indice `i-1`).
  La commande `L[i:]` renvoie la sous-liste de `L` constituée des éléments de `L` à partir de l'indice `i` (i.e de la position `i+1`).

  Par exemple `L[2:]` *renvoie* `[0,8,-9,7,4,0]`.

Plus généralement la commande `i:j` renvoie la sous-liste des éléments de `L` de l'indice `i` à l'indice `j-1` (i.e. de la position `i+1` à la position `j`).

Ainsi la commande `L=L[:i]+L[i+1,:]` supprime l'élément d'indice `i` (i.e. en position `i+1`). La commande `L.pop(i)` fait la même chose mais renvoie en plus l'élément supprimé (que l'on peut stocker dans une variable).

Enfin, la commande `L[:]` renvoie (une copie de) `L`.

Nous présentons l'intérêt des copies de listes dans le paragraphe suivant.

- Si n et m sont deux entiers tels que $0 \leqslant m < n$ implémentés en Python par `n` et `m`, alors **`range(n)`** renvoie la liste des entiers compris au sens large entre 0 et $n-1$. La commande **`range(m,n)`** renvoie la liste des entiers compris au sens large entre m et $n-1$. Et donc `range(m,n+1)` contient la liste des entiers compris au sens large entre m et n.

  ⚠ En fait, il faudrait plutôt écrire `[k for k in range(m,n)]` pour qu'il s'agisse vraiment d'une liste. A part si on veut lui ajouter ou enlever des éléments, il ne sera pas nécessaire en pratique d'ajouter `list`.

5) Échanger des variables

Si `x` et `y` sont deux variables Python, alors la commande **`x,y=y,x`** échange leurs contenus respectifs.

Par exemple, si on a stocké le réel `5` *dans la variable* `a` *et le réel* `9` *dans la variable* `b` *alors, en faisant* `a,b=b,a`, *c'est désormais la variable* `a` *qui contient* `9` *et la variable* `b` *qui contient* `5`.

⚠ Du moins cela fonctionne pour les flottants, les entiers, les booléens, les chaînes de caractères... mais pas les listes ou les tableaux. On touche ici à une subtilité de Python : si jamais on écrit `M=L`, alors toute modification de `M` entraînera la même modification de `L`. Mais, si on écrit `M=L[:]`, alors les listes `M` et `L` sont identiques mais elles sont désormais indépendantes.

C'est comme ça : il s'agit du même objet car il est stocké au même endroit dans la mémoire) et vice versa.

II Structures de bases de la programmation

Avant de commencer une commande très utile : **#**. Elle permet d'écrire un commentaire : tout ce qui suit cette commande (jusqu'au retour à la ligne) ne sera pas exécuté par Python et qui sert par exemple à expliquer ce que l'on fait, à s'y retrouver. Citons le programme : « les étudiants doivent savoir faire un usage judicieux des commentaires ».

1) Fonctions en Python

a) Définition et utilisation d'une fonction

Pour créer une fonction en Python, appelée `f`, qui prend en entrée des variables `x1,x2,...xn` et qui renvoie une variable `y`, voici la syntaxe :

```
def f(x1,x2,...xn):
    <bloc d'instructions>
    return y
```

On remplace `<bloc d'instructions>` par des commandes permettant de transformer les variables d'entrée en la variable de sortie. En exécutant la fonction dans la console Python puis la commande `f(x1,x2,...xn)`, on obtient l'image de `x1,x2,...xn` par `f`.

Exemple : *La fonction* $f : x \longmapsto x^2 + x - 1$ *est implémentée en Python ainsi :*

```

```

La commande `f(2)` *renvoie* `5` (on a bien $f(2) = 2^2 + 2 - 1 = 5$). *La commande* `f((1+5**(1/2))/2)` *renvoie* `0.0` *(on a bien* $f\left(\frac{-1+\sqrt{5}}{2}\right) = 0$*).*

Exemple : *Voici une fonction qui prend en argument un réel positif* x *et un entier* n *et qui renvoie* $\frac{\lfloor 10^n x \rfloor}{10^n}$.

```

```

La commande `tronc(np.pi,5)` *renvoie* `3.14159`. *Elle tronque l'écriture décimale de* π *à 5 chiffres.*

Remarquons les alinéas après la commande `def` (il y en a aussi après les commandes `if,elif,else`, `for,while`, cf. paragraphes suivants). Ce n'est pas facultatif : on appelle cela l'indentation. En Python, il n'y a pas de `begin` ou de `end`, ni de marqueurs du début ou de la fin d'un code. Les seuls délimiteurs sont les `:` et l'indentation (l'indentation démarre la structure et la « désindentation » la termine).

Remarques :

- Les variables d'entrée et de sortie sont muettes. La fonction f de l'exemple ci-dessus peut être implémentée ainsi :

  ```

  ```

- Les variables d'entrée, de sortie et celles introduites dans une fonction, n'existent que dans la fonction.

  Si on utilise `tronc(np.pi,5)`, *alors les variables* `x,y,n` *pourtant utilisées dans la fonction* `tronc` *ne contiennent rien (les appelées dans la console Python renvoie un message d'erreur). En revanche, si on écrit* `z=tronc(np.pi,5)`, *alors la variable* `z` *contient* `3.14159`.

- Les variables `x1,x2,...,xn,y` peuvent tout à fait être de types différents. Notamment `y` peut être une liste, un vecteur ou une matrice s'il y a plusieurs arguments de sortie.
- Il est possible pour une fonction de ne prendre aucun argument (cela arrive souvent en probabilités). Auquel cas, on ouvre les parenthèses en la construisant et en l'appelant mais on ne met rien dedans.
- Une fonction peut tout à fait ne rien retourner (si on ne met pas de `return`). Cela arrive par exemple lorsque le but d'une fonction est de tracer une courbe.
- Ne pas confondre `return` et `print` qui afficherait la variable de sortie mais sans la retourner (celle-ci serait inutilisable dans la suite car on ne pourrait pas récupérer ce qui est affiché dans une variable).
- S'il y a plusieurs `return` dans une fonction (cela arrive souvent lorsqu'il y a des structures conditionnelles), la fonction renvoie l'argument du premier d'entre eux.

b) Liste des images d'une liste par une fonction

Si `f` est une fonction qui prend en entrée une variable d'un type `truc` et qui revoie une variable du type `bidule`, il est classique de vouloir appliquer `f` à une liste ou un tableau `X` (contenant des variable du type `truc`) pour récupérer une liste ou un tableau `Y` contenant les images des éléments de `X` (donc contenant des variable du type `bidule`). Pour cela on utilise la syntaxe `Y=[f(x) for x in X]`.

Si `f` implémente une fonction de $\mathbb{R}$ dans $\mathbb{R}$, on peut aussi la « vectoriser » la fonction avec la commande `f=np.vectorize(f)` puis utiliser `Y=f(X)`. On en reparlera.

Exemple : *Si on veut créer la liste des 101 premiers carrés parfaits (c'est-à-dire* $0, 1, 4, 9, 16, 25, 36, \ldots, 10000$*) et la stocker dans la variable* `C` *on peut procéder ainsi :*

```

```

2) Structures conditionnelles

Si on veut que le bloc d'instructions `<bloc d'instructions>` soit exécuté si la condition `<condition>` est remplie, et que le bloc d'instructions `<bloc d'instructions sinon>` soit exécuté sinon, voici la syntaxe :

S'il n'y a pas d'instruction alternative, alors on ne met pas les deux dernières lignes.

```
if <condition>:
    <bloc d'instructions>
else:
    <bloc d'instructions sinon>
```

Si on veut que le bloc d'instructions `<bloc d'instructions 1>` soient exécuté si la condition `<condition 1>` est remplie, le bloc d'instructions `<bloc d'instructions 2>` si la condition `<condition 2>` est remplie, le bloc d'instructions `<bloc d'instructions 3>` si la condition `<condition 3>` est remplie... et enfin le bloc d'instructions `<bloc d'instructions sinon>` si aucune de ces dernières conditions n'est remplie, voici la syntaxe :

Vous aurez compris que `elif` est une contraction de `else if`.

```
if <condition 1>:
    <bloc d'instructions 1>
elif <condition 2>:
    <bloc d'instructions 2>
elif <condition 3>:
    <bloc d'instructions 3>
...
else:
    <bloc d'instructions sinon>
```

Exemple : *Voici une fonction qui prend en entrée trois réels a, b, c avec $a \neq 0$ et qui renvoie une liste contenant les solutions de l'équation $ax^2 + bx + c = 0$.*

```

```

3) Structures répétitives

Les structures répétitives (ou itératives) permettent d'effectuer plusieurs opérations à la suite. En général, la construction d'une telle structure au sein d'un programme se base sur quatre étapes :

- On identifie les variables d'entrées (qui proviennent du début du programme ou qui sont des arguments d'entrée d'une fonction par exemple).
- On définit des variables qui vont être modifiées lors de la boucle. C'est l'étape d'initialisation.
- On écrit la boucle en tant que telle. Elle utilise les variables d'entrée et, à chaque étape de la boucle, les variables définies à l'étape d'initialisation sont mises à jour.
- On renvoie les variables de sorties (en général des fonctions des quantités calculées pendant la boucle).

a) Boucles for

Souvent `T=range(a,b)` (la liste des entiers compris au sens large entre l'entier `a` et l'entier `b-1`).

On emploie les boucles for lorsqu'on veut exécuter un bloc d'instructions un certain nombre de fois, nombre que l'on connaît. Soit `T` une liste ou un tableau.

```
for x in T:
    <bloc d'instructions>
```

La variable `x` va prendre tour à tour et dans l'ordre les valeurs présentes dans `T` et, pour chacune d'entre elle, elle va exécuter le bloc d'instructions `<bloc d'instructions>`. Ce dernier peut dépendre ou non de `x`.

⚠ `x` est une variable muette dans la boucle (il faut la voir comme un indice de sommation par exemple) et ne doit pas être introduite.

Pour cela parcourt la liste avec une boucle `for`. Pour chaque élément, elle teste si il est strictement positif ou non et, si c'est le cas, elle incrémente un compteur de 1. Ici `n` est le compteur et sa valeur est mise à jour (1 de plus) si on rencontre un élément strictement positif.

Exemple : *La fonction suivante prend en argument une liste (ou un vecteur) et renvoie le nombre d'éléments qui sont strictement positifs.*

```

```

La commande `nb_st_positif([1,-8,5,9,-7,-4,7,3,2])` *renvoie* `6`.

Exemple : *Voici une fonction qui prend en entrée un entier naturel n et revoie* `True` *si n est premier (c'est-à-dire supérieur ou égal à 2 et n'admettant que 1 et lui-même pour diviseur) et* `False` *sinon.*

```

```

Les boucles `for` servent principalement à calculer des sommes, produits ou les termes successifs d'une suite. C'est l'objet des chapitres 4 et 7.

L'une des fonctionnalités les plus intéressantes de Python est la possibilité de définir des listes « par compréhension » : si `f` est une fonction et `T` une liste ou un tableau, alors la commande `[f(x) for x in T]` construit la liste des images des éléments de `T` par la fonction `f`. On a vu un exemple dans le paragraphe précédent.

b) Boucles while

On emploie les boucles while lorsqu'on veut exécuter un bloc d'instructions tant qu'une certaine condition est remplie (souvent lorsqu'on ne l'on connaît pas précisément le nombre d'itérations à exécuter).

Les variables intervenant dans la condition `<condition>` doivent être introduites préalablement contrairement à l'indice dans une boucle `for`.

```
while <condition>:
    <bloc d'instructions>
```

Tant que la condition `<condition>` est remplie, le bloc d'instructions `<bloc d'instructions>` sera exécuté. Il est donc essentiel que l'exécution de `<bloc d'instructions>` vienne modifier la condition `<condition>` afin que cette dernière finisse par devenir fausse (sinon la boucle va durer éternellement).

Pour trouver n_A, l'idée est de démarrer avec $n = 0$ et, tant que $n^3-7n+2021 < A$, on augment de 1 la valeur de n.

Exemple : *On sait que* $n^3 - 7n + 2021 \xrightarrow[n\to+\infty]{} +\infty$ *et donc, pour tout* $A > 0$, *il existe* $n_A \in \mathbb{N}$ *tel que, pour tout* $n \geqslant n_A$, $n^3 - 7n + 2021 \geqslant A$. *La fonction suivante prend* A *en argument et détermine* n_A.

```
```

La commande `rang_min(10000)` *renvoie* 21, *la commande* `rang_min(100000)` *renvoie* 47 *et la commande* `rang_min(1000000)` *renvoie* 100.

On verra d'autres exemples dans les chapitres 4 et 7.