Chapitre 1 - Rappels et compléments de programmation

Objectifs

- Conforter ou consolider les acquis de première année ;
- Introduire et comprendre des notions complémentaires de programmation, notamment les effets de bord et la mutabilité.
- Cerner les attentes des concours, et savoir où chercher des conseils supplémentaires.

I - L'ITC aux écrits de concours

- CCINP : pas d'épreuve d'ITC à part entière, mais :
  - environ 1/4 de l'épreuve d'option (SI ou info) porte sur l'ITC ;
  - quelques questions d'ITC possibles dans les épreuves de mathématiques et de physique. En général en lien avec l'épreuve, elles sont plutôt faciles.
- Mines-Ponts : une épreuve de 2h, comportant entre 20 et 25 questions, dont beaucoup sont abordables. Quelques stats : moyenne MP 2023 : 11, moyenne MPE 2023 : 14,3, moyenne nationale MP 2022 : 10,9.
- Centrale : probablement **pas d'épreuve en filière MP** (QCM de 2h uniquement en PC/PSI)... à confirmer.

À retenir

Des points faciles à prendre avec un travail régulier. Consulter les rapports de jurys des différents pour des conseils plus précis, notamment les erreurs à éviter.

II - Rappels méthodologiques de première année

a) Construction d'une boucle

Questions à se poser

- Vaut-il mieux privilégier une boucle `for` ou une boucle `while` ?
- Déterminer les valeurs à énumérer dans le cas d'une boucle `for`, ou la condition du `while`, qu'il ne faut pas confondre avec la condition d'arrêt ;
- Identifier les instructins à répéter dans la boucle.

$\Longrightarrow$ Plusieurs solutions sont souvent possibles.

Exemple 1. Écrire une fonction `somme(n)` qui calcule et renvoie la somme S_n ci-dessous, avec une complexité en $\mathcal{O}(n)$.

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

$\Rightarrow$ calcul de $k!$ et de la somme simultanément

```
def somme(n):
    S = 0
    f = 1
    for k in range(1, n+1):
        f = k * f
        # à ce stade, f contient k!
        S += (-1)**k / f
    return S
```

b) Parcours et construction de listes

Parcours d'une liste : 2 méthodes

- `for i in range(len(L))` ⟶ `i` désigne un indice / une position dans L
- `for x in L` ⟶ `x` désigne un élément dans L

Exemple 2. Écrire une fonction `estCroissante(L)` prenant en argument une liste de valeurs numériques, et renvoyant `True` si la liste contient des valeurs rangées dans l'ordre croissant (éventuellement une ou aucune), et renvoie `False` sinon.

```
def estCroissante(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Eq : return à l'extérieur dans tous les cas

Construction d'une liste : plusieurs approches

- Utilisation d'une boucle et de append ;
- Construction en compréhension dans les cas simples : `[f(x) for x in X]` ;
- Avec les opérateurs + (concaténation) et * (duplication).

Exemple 3. (CCINP 2020) Si $L = [\ell_1, \dots, \ell_n]$ est une suite finie d'entiers de $\{0;1;2\}$, on lui ajoute un élément égal à -1 si la somme $\ell_1 + \cdots + \ell_n$ est paire, et un élément égal à -2 sinon. Ce dernier élément permet alors d'essayer de détecter d'éventuelles erreurs de transmission. Écrire une fonction `ajout(L)` qui ajoute à la liste `L` un élément comme expliqué précédemment et renvoie la nouvelle liste. Puis, écrire une fonction `verif(L)` qui renvoie `True` si la valeur du dernier élément de `L` est correcte, et `False` sinon.

en exo (facile)

```
def ajout(L):
    S = 0
    for x in L:
        S = S + x
    if S%2 == 0:
        L.append(-1)
    else:
        L.append(-2)
    return L
```

remarque : "nouvelle liste" ambigü car L est directement modifiée...

III - Expression, instruction et effet de bord

a) Définitions

Expression ≠ instruction

Une *expression* est une combinaison de constantes, de variables, d'opérateurs, etc. **pouvant être évaluée en une valeur.** Une *instruction* est une unité de code **qui produit un effet,** ou *effet de bord,* typiquement :

une affectation, une modification de liste (append/del/pop), un affichage (print, plt.plot...), etc

b) Quelques exemples et points de vigilance

Exemple 4. Pour chacun des codes suivants, deviner le contenu de la liste L après exécution, vérifier vos conjectures avec Python et expliquer les résultats observés.

```
L = [1, 2]
for i in range(len(L)):
    L.append(i)
```

évalué une seule fois (avant l'appel à range)
⇒ cette boucle fait 2 tours
→ L devient [1, 2, 0, 1]

```
L = [1, 2]
i = 0
while i < len(L):
    L.append(i)
    i = i + 1
```

condition réévaluée à chaque tour, on a toujours len(L) = i+2
del L[1] ⇒ boucle ∞ (indices décalés)

```
L = [1, 2, 3, 4, 5, 6]
for i in range(len(L)//2):
    del L[i]
```

len(L)//2 = 3 → L = [~~1~~, 2, ~~3~~, 4, ~~5~~, 6] → [2, 4, 6]
del L[0] del L[2]

c) Instruction break

Dans une boucle, l'instruction **break** permet de sortir de celle-ci prématurément. À la différence d'un **return** (qui ne peut se trouver que dans une fonction), **break** ne provoque que la sortie de la boucle, et pas de l'ensemble de la fonction : on reprend donc le programme à la sortie de la boucle. Cette instruction permet de simplifier l'écriture de certaines boucles...

Exemple 5. Expliquer précisément ce que renvoie la fonction ci-dessous et les hypothèses éventuelles sur les entrées du programme. Proposer ensuite une version équivalente n'utilisant pas l'instruction **break** (et si possible sans **if**).

```
def mystere(L, x):
    i = 0
    while L[i] < x:
        i = i + 1
        if i == len(L):
            break
    return i
```

Cette boucle s'arrête au 1er indice i tq L[i] ⩾ x et le renvoie, ou si on parcourt toute la liste (c.à.d si ∀y∈L, y<x), auquel cas on renvoie i = len(L).

Hypothèses :
- L est indiçable (list, str, tuple...)
- les "éléments" de L sont comparables avec x.

Version équivalente : `while (i < len(L) and L[i] < x)`
1) i est un indice licite ⇒ accès L[i] ok
2) condition originelle

IV - Mutabilité et conséquences

Rq : `while (L[i] < x and i < len(L))` pose pb ds le cas où aucun élt de L n'est ⩾ x

Mutabilité

- Tout objet en Python est stocké dans la mémoire et est accessible en connaissant son *adresse* en mémoire.
- En Python, un objet est dit *mutable* lorsqu'on peut le modifier **sans changer son adresse en mémoire**. Il est dit *immuable* sinon.

Exemple 6. Classer les types usuels suivants dans la bonne catégorie : `int`, `float`, `bool`, `list`, `str` (chaînes de caractères), `tuple`, `dict` (dictionnaires).

- Types d'objets mutables : list, dict
- Types d'objets immuables : int, float, bool, str, tuple

> **Copie d'un objet mutable**
> Lorsque `var1` est une variable, l'instruction `var2 = var1` se contente de créer un nouveau référencement vers la même adresse en mémoire. Elle n'effectue donc pas une « vraie copie » dans le cas d'un objet mutable.

Exemple 7. Observer les codes suivants et deviner la valeur de b. On pourra s'aider de dessins (voir le corrigé).

```
# Code 1
a = 2
b = a
a = 1
b --> 2 (aucune surprise)
```

a est évalué
b → 2
a → 1 (immuable)

```
# Code 2
a = [2]
b = a
a = [1]
b --> [2]
```

b → [2] (mutable)
a → [1] (nouvel objet)

```
# Code 3
a = [2]
b = a
a.append(1)
b --> [2, 1]
```

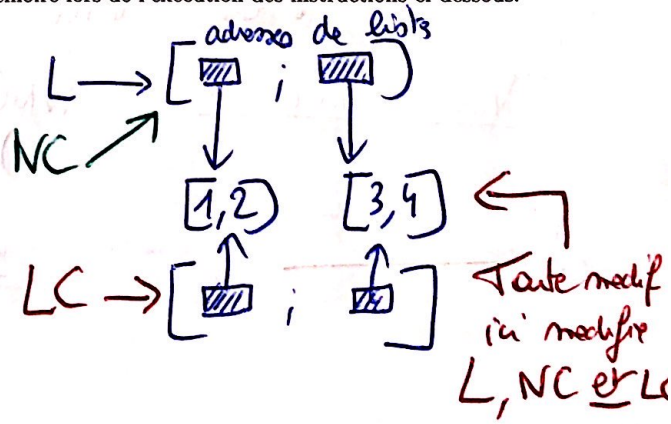
a → [2, 1] ← b
a et b ont la même adresse en mémoire

> **À retenir : syntaxes pour copier une liste L**
> L[:], L.copy(), [x for x in L], (boucle + append)...

Exemple 8. Représenter schématiquement l'état de la mémoire lors de l'exécution des instructions ci-dessous.

```
L = [[1, 2], [3, 4]]
NC = L # n'est pas une "vraie" copie, cf exemple 7
LC = L[:] # lui non plus... pourquoi ?
```

les adresses sont copiées, et non les sous-listes elles-même



Exercice 1. Écrire une fonction `deepCopy(L)` prenant en argument une liste de listes d'objets immuables (par exemple des entiers) et renvoie une « vraie » copie de L (on parle de *copie profonde*) : aucune modification de L après création de la copie ne doit altérer celle-ci.

(bien comprendre pourquoi...)

V - Rappels et compléments sur la récursivité

Une fonction récursive est une fonction qui s'appelle elle-même dans sa propre définition. Pour qu'elle renvoie toujours un résultat, une telle fonction doit comporter au moins un cas de base. Quand une fonction s'appelle elle-même, on dit qu'elle effectue un *appel récursif*. Les fonctions récursives sont souvent utilisées pour traduire des relations de récurrence.

À retenir : complexité d'une boucle / d'une fonction récursive

Pour une boucle : nombre de tours × complexité d'1 tour

Pour une fonction récursive : nb d'appels récursifs × cplxité des opérations HORS APPELS

Cette règle permet d'évaluer la complexité de fonctions récursives simples. Les cas plus techniques ne sont pas au programme d'ITC (mais le sont en option informatique), à l'exception d'une analyse de complexité déjà abordée : celle du parcours en profondeur d'un graphe... dont nous reparlerons en novembre.

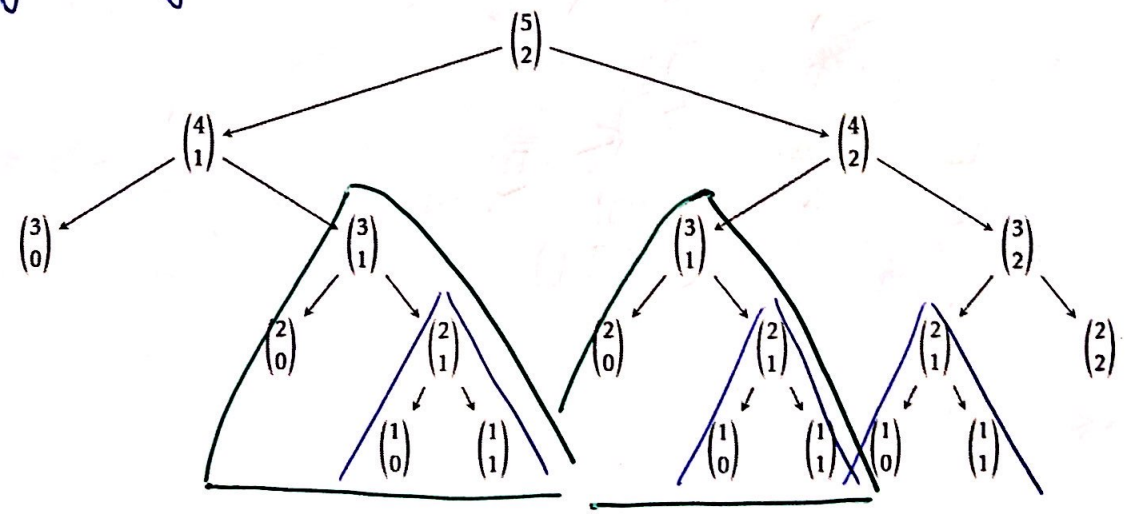
Exemple 9. Écrire une fonction récursive `binom(n, k)` qui prend en arguments deux entiers $n \in \mathbb{N}$ et $k \in \mathbb{N}$ tels que $0 \leq k \leq n$ (condition que n'aura pas à vérifier la fonction) et renvoie la valeur du coefficient binomial $\binom{n}{k}$ en utilisant uniquement la formule du triangle de Pascal :

On "recopie" en Python...

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k} + \binom{n-1}{k-1} & \text{si } 0 < k < n \\ 1 & \text{si } k = 0 \text{ ou } k = n \end{cases}$$

```
def binom(n,k):
    if k==0 or k==n:        # cas de base
        return 1
    return binom(n-1,k) + binom(n-1,k-1)
```

Inconvénient de cette fonction : beaucoup de coeffs sont évalués de nombreuses fois de façon indépendante...



Pour pallier à ce problème, on peut essayer de se souvenir des coefficients binomiaux $\binom{i}{j}$ déjà calculés...

Nous verrons au prochain cours comme traduire cette stratégie en conservant une approche récursive à l'aide de la *programmation dynamique*.

Remarque : Pour compter précisément le nombre d'appels récursifs effectués lors du calcul `binom(n, k)`, nombre que l'on note $A(n,k)$, on peut remarquer que $A(n,0) = A(n,n) = 0$ (calcul direct), et que si $0 < k < n$, on a $A(n,k) = A(n-1,k) + A(n-1,k-1) + 1$. On obtient alors que $A(n,k) = \binom{n}{k} - 1$ (car ils vérifient la même relation de récurrence), ce qui prouve notre conjecture concernant le nombre trop important de ces appels.

VI - Deux derniers exercices extraits de CCINP

Exercice 2. (CCINP 2021) La suite des nombres de Bernoulli notée $(b_n)_{n\in\mathbb{N}}$ est définie par :

$$b_0 = 0 \qquad \text{et} \qquad \forall n \geq 1 \quad b_n = -\frac{1}{n+1}\sum_{k=0}^{n-1}\binom{n+1}{k}b_k$$

Écrire une fonction non récursive `bernoulli(n)` qui renvoie une valeur approchée du nombre rationnel b_n. On pourra utiliser librement une fonction `binomial(n, p)` qui renvoie le coefficient binomial $\binom{n}{k}$.

Exercice 3. (CCINP 2020) Soit $x \in [0;1[$. On définit une suite $t(x) = (t_n(x))_{n\in\mathbb{N}^*}$ par

$$\forall n \in \mathbb{N}^* \qquad t_n(x) = \lfloor 3^n x \rfloor - 3\lfloor 3^{n-1}x \rfloor$$

On peut montrer que pour tout entier naturel n non nul, $t_n(x) \in \{0,1,2\}$, et que

$$x = \sum_{n=1}^{+\infty}\frac{t_n(x)}{3^n}$$

La suite $t(x) = (t_n(x))_{n\in\mathbb{N}^*}$ est appelée *développement ternaire propre* de x.

1) Écrire une fonction `flotVersTern(n, x)` qui prend en arguments un entier naturel `n` et un flottant `x` et renvoie sous forme d'une liste les n premiers chiffres $(t_1(x),\dots,t_n(x))$ du développement ternaire de x définis précédemment. Par exemple, `flotVersTern(4, 0.5)` doit renvoyer `[1, 1, 1, 1]`. On rappelle que la fonction `floor` du module `math` (que l'on importera au préalable) permet de calculer la partie entière d'un nombre.

2) Écrire une fonction `ternVersFlot(L)` prenant en argument une liste $L = [\ell_1,\dots,\ell_n]$ d'entiers et renvoyant le flottant $\sigma(L) = \sum_{k=1}^{n}\frac{\ell_k}{3^k}$. Par exemple, `ternVersFlot([1, 1, 1, 1])` doit renvoyer environ `0.493827`. Commenter ce résultat.