TP3 : Écriture de sommes finies en **Python**

▶ Dans votre dossier `Info_2a`, créer le dossier `TP_3`.

I. Avant propos

On considère dans ce TP les suites $(u_n)_{n\in\mathbb{N}}$ et (v_n) suivantes :

$$\forall n \in \mathbb{N},\ u_n = \frac{n^2}{3^n} \qquad \text{et} \qquad \begin{cases} v_0 = 1 \\ \forall n \in \mathbb{N},\ v_{n+1} = \dfrac{2\,v_n}{\mathrm{e}^{v_n} + \mathrm{e}^{-v_n}} \end{cases}$$

Objectif du TP : il s'agit d'explorer les différentes méthodes permettant le calcul des sommes partielles d'ordre n : $S_n = \sum_{k=0}^{n} u_k$ et $T_n = \sum_{k=0}^{n} v_k$.

II. Calcul des sommes partielles d'ordre n

II.1. Calcul de S_n

Il s'agit ici d'illustrer le cas où la suite (u_n) est donnée sous forme explicite.

II.1.a) Méthode itérative

▶ Écrire une fonction `calculSn` qui :
- × prend en paramètre un entier `n`,
- × renvoie une variable `S`,
- × à l'aide d'une structure itérative, calcule la valeur de $S_{\texttt{n}}$ et stocke le résultat dans `S`.

```
1  def calculSn(n) :
2      S = 0
3      for k in range(n+1) :
4          S = S + k**2 / 3**k
5      return S
```

▶ Que vaut S_0 ? S_5 ? S_{10} ? S_{100} ? S_{1000} ?

On trouve : $S_0 = 0$, $S_5 \simeq 1.4115$, $S_{10} \simeq 1.4989$, $S_{100} \simeq 1.5$ et $S_{1000} \simeq 1.5$.
(noter que l'affichage du résultat se fait avec un nombre fixé de chiffres après la virgule)

II.1.b) Utilisation des fonctionnalités Python

- Écrire une fonction `premSuiteU` qui :
  - × prend en paramètre un entier `n`,
  - × renvoie une variable `U`, vecteur contenant initialement `n` + 1 zéros,
  - × à l'aide d'une structure itérative, calcule les `n` + 1 premières valeurs de la suite (u_n) et stocke le résultat dans `U`.

```
def premSuiteU(n) :
    U = []
    for k in range(n+1) :
        U.append(k**2 / 3**k)
    return U
```

- Que réalisent les appels `V = [k for k in range(1,6)]`, puis `sum(V)` ?
  Détailler le rôle de la fonction `sum`.

La fonction `sum` prend en paramètre une liste et renvoie la somme de tous ses éléments. Ici, on considère la liste $[1, 2, 3, 4, 5]$ dont la somme vaut 15.

- En déduire un appel permettant de calculer S_{10}.

```
sum(premSuiteU(10))
```

- Une fonctionnalité de **Python** est de permettre la création de liste en compréhension. La syntaxe pour créer une liste en compréhension (ou parle aussi de compréhension de liste) est assez proche de celle utilisée en mathématique. Il s'agit de spécifier la propriété que doivent vérifier les éléments que l'on souhaite faire apparaître dans la liste. Typiquement, on peut demander de créer la liste contenant toutes les valeurs entières comprises entre `0` et `10` (exclu) ou encore la liste des cinq premiers carrés d'entiers. Pour ce faire, on procède généralement en demandant l'itération (à l'aide d'un `for`) des valeurs d'un intervalle (défini par un `range`).

```
Out [1] :   [k for k in range(10)]
In  [1] :   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Out [2] :   [k**2 for k in range(1, 6)]
In  [2] :   [1, 4, 9, 16, 25]
```

  Il est aussi possible d'ajouter des conditions à l'aide de `if`. Il est par exemple possible de créer la liste de tous les carrés multiples de 6 d'entiers plus petits que 20.

```
Out [3] :   [k**2 for k in range(20) if k**2 % 6 == 0]
In  [3] :   [0, 36, 144, 324]
```

- Comment peut-on, à l'aide d'une compréhension de liste, créer la viste `U` des 101 premiers éléments de la suite (u_n) ?

```
U = [k**2 / 3**k for k in range(n+1)]
```

▶ Comment obtient-on alors S_{100} à l'aide de `U` ? Et comment récupérer S_5 ?

- La valeur de S_{100} est donnée par l'appel `sum(U)`.
- S_5 est la somme des 6 premiers éléments de (u_n) (ne pas oublier 0). On récupère le vecteur contenant ces 6 éléments par l'appel `U[:6]` et donc S_5 par l'appel `sum(U[:6])`.

II.2. Calcul de T_n

Il s'agit ici d'illustrer le cas où la suite (u_n) est donnée sous forme récurrente.

II.2.a) Méthode itérative

▶ Écrire une fonction `calculTn` qui :

× prend en paramètre un entier `n`,

× renvoie une variable `T`,

× à l'aide d'une structure itérative, calcule la valeur de $S_{\texttt{n}}$ et stocke le résultat dans `T`.

On pourra utiliser une variable auxiliaire `v` afin de calculer les différents termes de (v_n).

```
1  import numpy as np
2  def calculTn(n) :
3      T = 0
4      v = 1
5      for k in range(n+1) :
6          T = T + v
7          v = 2*v / (np.exp(v) + np.exp(-v))
8      return T
```

▶ Que vaut T_0 ? T_{100} ? T_{10000} ?

On trouve : $T_0 = 1$, $T_{100} \simeq 18.18$, $T_{10000} \simeq 197.58$.

II.2.b) Utilisation des fonctionnalités Python

▶ Écrire une fonction `premSuiteV` qui :

× prend en paramètre un entier `n`,

× renvoie une variable `V`, vecteur contenant initialement `n` + 1 zéros,

× à l'aide d'une structure itérative, calcule les `n` + 1 premières valeurs de la suite (v_n) et stocke le résultat dans `V`.

```
1  def premSuiteV(n) :
2      V = [1]
3      for k in range(n) :
4          V.append(2*V[k] / (np.exp(V[k]) + np.exp(-V[k])))
5      return V
```

▶ En déduire un appel permettant de calculer T_{10}.

```
sum(premSuiteV(10))
```

III. Calcul des n premières sommes partielles

III.1. Calcul des n premières sommes partielles de $\sum u_n$

▶ Soit $n \in \mathbb{N}$. Quel lien y a-t-il entre S_n et S_{n+1} ?

$$S_{n+1} = S_n + \frac{(n+1)^2}{3^{n+1}}$$

▶ En tirant profit de l'égalité précédente, écrire une fonction `premSn` qui :

× prend en paramètre une variable `n`,

× renvoie une variable `tabS`, vecteur contenant initialement `n` + 1 zéros,

× à l'aide d'une structure itérative, stocke la valeur de S_i dans la $i^{\text{ème}}$ case de `tabS`.

On ne devra pas effectuer d'appel à `calculSn` mais on pourra s'inspirer de son code.

```
def premSn(n) :
    tabS = [0]
    for k in range(n) :
        tabS.append(tabS[k] + (k+1)**2 / 3**(k+1))
    return tabS
```

▶ Comme précédemment, on aurait aussi pu tirer parti des fonctionnalités de **Python**.
Que réalise les appels `V = [k for k in range(1,6)]` puis `np.cumsum(V)` ? Détailler le rôle de la fonction `cumsum`.

- La fonction `cumsum` (*Cumulative Sum*) prend en paramètre une liste ou un array `u` et renvoie un array `v` de même taille dont le $i^{\text{ème}}$ élément est la somme des i premiers éléments de `u`.
- Ainsi, `cumsum(V)` renvoie le tableau $[1, 3, 6, 10, 15]$.

▶ Quel appel, tirant parti des fonctionnalités **Python**, permet d'obtenir les 101 premiers éléments de la suite (S_n) ? On utilisera la fonction `cumsum`.

- Si on a codé la fonction `premSuiteU`, on peut réaliser l'appel : `np.cumsum(premSuiteU(100))`.
- Sinon on peut, commme précédemment, créer la liste qui contient les 101 premiers éléments de la suite (u_n) :

```
U = [k**2 / 3**k for k in range(n+1)]
np.cumsum(U)
```

III.2. Calcul des n premières sommes partielles de $\sum v_n$

▶ Soit $n \in \mathbb{N}$. Quel lien y a-t-il entre T_n et T_{n+1} ?

$T_{n+1} = T_n + v_{n+1}$

▶ En tirant profit de l'égalité précédente, écrire une fonction `premTn` qui :

× prend en paramètre une variable `n`,

× renvoie une variable `tabT`, vecteur contenant initialement `n` + 1 zéros,

× à l'aide d'une structure itérative, stocke la valeur de T_i dans la $i^{\text{ème}}$ case de `tabT`.

On ne devra pas effectuer d'appel à `calculTn` mais on pourra s'inspirer de son code. On pourra notamment utiliser une variable `v` calculant les valeurs successives des termes de la suite (v_n).

```
def premTn(n) :
    tabT = [1]
    v = 1
    for k in range(n+1) :
        v = 2*v / (np.exp(v) + np.exp(-v))
        tabT.append(tabT[k] + v)
    return tabT
```

À retenir : on a étudié deux méthodes en **Python** pour obtenir les éléments de (S_n).

1) La suite des sommes partielles étant une suite (grande nouvelle), on peut se servir des procédés vus en TP1.

2) On peut aussi préférer créer la liste des premiers éléments de la suite (u_n) et utiliser les instructions `sum` ou `cumsum` suivant ce que l'on cherche à obtenir. Encore une fois, il faut se reporter au TP1.

IV. Tracé des sommes partielles de $\sum u_n$

▶ Compléter le programme suivant afin qu'il permette d'effectuer le tracé des 51 premiers éléments de (S_n). On exécutera ce programme.

```
import matplotlib.pyplot as plt
n = 100
N = [k for k in range(n+1)]
U = [k**2 / 3**k for k in range(n+1)]
tabS = np.cumsum(U)
plt.plot(N, tabS, 'rx')
```

▶ Quelle conjecture peut-on émettre sur la nature de la série $\sum u_n$?

D'après la représentation graphique, on peut émettre l'hypothèse que $\sum u_n$ est une série convergente, de somme $\frac{3}{2}$.

V. Suite des sommes partielles aux concours

V.1. EML 2015

On considère l'application $f : \mathbb{R} \to \mathbb{R}$, $x \mapsto f(x) = x^3\, e^x$
et la suite réelle $(u_n)_{n\in\mathbb{N}}$ *définie par :* $u_0 = 1$ *et* $\forall n \in \mathbb{N}, u_{n+1} = f(u_n)$.
Il était demandé de démontrer que la série $\sum_{n\geqslant 1} \dfrac{1}{f(n)}$ converge (de somme S) et que :

$$\forall n \in \mathbb{N}^*,\ \left|S - \sum_{k=1}^{n} \frac{1}{f(k)}\right| \leqslant \frac{1}{(\mathrm{e}-1)\,\mathrm{e}^n}$$

▶ *En déduire une fonction* ***Python*** *qui calcule une valeur approchée de* S *à* 10^{-4} *près.*
(on pourra se reporter au TP2)

- Si on sait que : $\dfrac{1}{(\mathrm{e}-1)\,\mathrm{e}^n} \leqslant 10^{-4}$, on obtient par transitivité que : $|S - S_n| \leqslant 10^{-4}$.
  L'idée est donc de trouver le premier entier tel que : $\dfrac{1}{(\mathrm{e}-1)\,\mathrm{e}^n} \leqslant 10^{-4}$.
  On peut démontrer que cet entier vaut : $\lceil 4\ln(10) - \ln(\mathrm{e}-1)\rceil$ $(= 9)$ et ainsi, utiliser une boucle `for` pour calculer S_9 qui fournit l'approximation souhaitée.

```
def calcApprochS() :
    n = np.ceil(4 * np.log(10) - np.log(np.exp(1)-1))
    S = 0
    for k in range(1,n+1) :
        S = S + 1 / (k**3 * np.exp(k))
    return S
```

- Le deuxième choix est de calculer les valeurs successives de (S_n) tant que $\dfrac{1}{(\mathrm{e}-1)\,\mathrm{e}^n} \leqslant 10^{-4}$ **n'est pas** vérifiée *i.e.* tant que $\dfrac{1}{(\mathrm{e}-1)\,\mathrm{e}^n} > 10^{-4}$.

```
def calcApprochS() :
    n = 1
    S = 1 / np.exp(1)
    while 1 / ((np.exp(1)-1) * np.exp(n)) > 10**(-4) :
        n = n + 1
        S = S + 1 / (n**3 * np.exp(n))
    return S
```

- On pouvait également exploiter les fonctionnalités **Python**.

```
def calcApprochS() :
    n = np.ceil(4 * np.log(10) - np.log(np.exp(1)-1))
    S = [1 / (k**3 * np.exp(k)) for k in range(1,n+1)]
    S = sum(S)
    return S
```

V.2. ECRICOME 2018

Pour tout entier naturel n non nul, on pose : $u_n = \sum_{k=1}^{n} \frac{1}{k} - \ln(n)$.

▸ *Écrire une fonction d'en-tête* `def u(n) :` *qui prend en argument un entier naturel n non nul et qui renvoie la valeur de u_n.*

```
1  def u(n) :
2      S = 0
3      for k in range(1, n+1) :
4          S = S + 1/k
5      y = S - np.log(n)
6      return y
```

Détaillons les différents éléments de ce code :

× en ligne 2, on crée la variable `S` dont le but est de contenir, en fin de programme $\sum_{k=1}^{n} \frac{1}{k}$.
Cette variable `S` est donc initialisée à 0.

× de la ligne 3 à la ligne 5, on met à jour la variable `S` à l'aide d'une boucle.
Pour ce faire, on ajoute au $k^{\text{ème}}$ tour de boucle la quantité $\frac{1}{k}$.
Ainsi, `S` contient bien $\sum_{k=1}^{n} \frac{1}{k}$ en sortie de boucle.

× en ligne 6, on affecte à la variable `y` la valeur $u_n = \sum_{k=1}^{n} \frac{1}{k} - \ln(n)$.

Commentaire

Pour le calcul de la somme $\sum_{k=1}^{n} \frac{1}{k}$, on peut aussi tirer profit des fonctionnalités **Python** :

```
1  S = [1/k for k in range(1, n+1)]
2  S = sum(S)
```

Pour bien comprendre cette instruction, rappelons que :
× l'instruction `[1/k for k in range(1, n+1)]` permet de créer la liste $[\frac{1}{1}, \frac{1}{2}, \ldots, \frac{1}{n}]$.
× la fonction `sum` permet de sommer tous les éléments d'une liste.
On obtient donc bien la somme à calculer par cette méthode.

V.3. ESSEC-II 2019

On souhaite écrire une fonction en **Python** pour calculer l'entropie d'une variable aléatoire X dont le support de la loi est de la forme $A = \{0, 1, \dots, n\}$ où n est un entier naturel. On suppose que la liste `P` de **Python** est telle que pour tout k de A, `P[k+1]` $= \mathbb{P}([X = k])$. Compléter la fonction ci-dessous d'argument `P` qui renvoie l'entropie de X, c'est-à-dire $-\sum_{k=0}^{n} \mathbb{P}([X = k]) \log_2\big(\mathbb{P}([X = k])\big)$, où :

$$\begin{array}{rccc} \log_2 & : & \mathbb{R}_+^* & \to & \mathbb{R} \\ & & x & \mapsto & \dfrac{\ln(x)}{\ln(2)} \end{array}$$

```
def Entropie(P) :
    ...
    return h
```

Si nécessaire, on pourra utiliser l'instruction `len(P)` qui donne le nombre d'éléments de `P`.

Remarque : Ici, la liste `P` doit contenir les valeurs :

$$\Big[\ \mathbb{P}([X = 0]),\ \mathbb{P}([X = 1]),\ \dots,\ \mathbb{P}([X = n])\ \Big]$$

Ainsi :

× `P[0]` (premier élément de la liste `P`) contient $\mathbb{P}([X = 0])$.

× `P[1]` (deuxième élément de la liste `P`) contient $\mathbb{P}([X = 1])$.

× ...

× `P[n]` ($(n+1)^{\text{ème}}$ élément de la liste `P`) contient $\mathbb{P}([X = n])$.

Autrement dit, pour tout `k` de $[\![0, n]\!]$, `P[k]` contient $\mathbb{P}([X = k])$.

On propose la fonction suivante :

```
def Entropie(P) :
    n = len(P)
    S = 0
    for k in range(n) :
        S = S + P[k] * (np.log(P[k]) / np.log(2))
    h = -S
    return h
```

Détaillons les éléments de ce script.

- **Début de la fonction**
  L'énoncé commence par préciser la structure de la fonction :
  × cette fonction se nomme `Entropie`,
  × elle prend en paramètre d'entrée la liste `P`,
  × elle admet pour variable de sortie la variable `h`.

```
def Entropie(P) :
```

```
return h
```

La variable `n` contient le nombre d'éléments de la liste `P`.
La variable `S`, qui contiendra les valeurs successives de $\sum_{k=0}^{n} \mathbb{P}([X=k]) \log_2\big(\mathbb{P}([X=k])\big)$, est initialisée à 0 (choix naturel d'initialisation lorsqu'on souhaite coder une somme puisque 0 est l'élément neutre de l'opérateur de sommation).

```
2       n = len(P)
3       S = 0
```

- **Structure itérative**
  Les lignes 4 à 6 consistent à calculer les valeurs successives de $\sum_{k=0}^{n} \mathbb{P}([X=k]) \log_2\big(\mathbb{P}([X=k])\big)$. Pour cela, on utilise une structure itérative (boucle for) :

```
4       for k in range(n) :
5           S = S + P[k] * (np.log(P[k]) / np.log(2))
```

- **Fin du programme**
  À l'issue de cette boucle, la variable `S` contient la somme $\sum_{k=0}^{n} \mathbb{P}([X=k]) \log_2\big(\mathbb{P}([X=k])\big)$.
  On met alors à jour la variable `h` pour qu'elle contienne la valeur $H(X) = -\sum_{k=0}^{n} \mathbb{P}([X=k]) \log_2\big(\mathbb{P}([X=k])\big)$.

```
6       h = -S
```

Remarque
Il était bien sûr possible d'exploiter les fonctionnalités **Python**. On obtenait alors, par exemple, le script suivant :

```
1  def Entropie(P) :
2      n = len(P)
3      S = [P[k] * (np.log(P[k]) / np.log(2)) for k in range(n)]
4      h = -sum(S)
5      return h
```

V.4. ESSEC-I 2021

L'énoncé définissait la suite (z_n) suivante :

$$\begin{cases} z_0 = 1 \\ \forall n \in \mathbb{N},\ z_{n+1} = \dfrac{n+2}{n+1} \displaystyle\sum_{k=0}^{\min(n,d)} \alpha_k\ z_{n-k} \end{cases}$$

On note Δ la liste $[\alpha_0,\ \cdots,\ \alpha_d]$.
Écrire une fonction **Python** d'entête `def z(Delta,n)` qui calcule z_n si `Delta` représente la liste Δ et stocke le résultat dans une variable `r`.

- Voici le programme attendu.

```
1  def z(Delta, n) :
2      Z = [1]
3      d = len(Delta) - 1
4      for i in range(n) :
5          S = 0
6          for k in range(min(i,d) + 1) :
7              S = S + Delta[k] * Z[i-k]
8          Z.append( (i+2) / (i+1) * S )
9      r = Z[n]
10     return r
```

- **Début du programme**
  La suite (z_n) est une suite récurrente dont chaque terme dépend de tous les précédents. Pour calculer le terme d'indice n, il faut avoir accès aux termes d'indice $0, \ldots, n-1$ de la suite.
  D'un point de vue informatique, il est donc nécessaire de créer une liste `Z` permettant de stocker, au fur et à mesure du calcul, toutes ces valeurs. Il est à noter que, pour tout $i \in \mathbb{N}$, le terme z_i est le $(i+1)^{\text{ème}}$ terme de la suite (z_n) et sera donc stockée en $i)^{\text{ème}}$ position dans `Z`.
  On commence alors par stocker le terme z_0 comme premier élément de la liste `Z`.

```
2      Z = [1]
```

  On récupère aussi la valeur de d, et on la stocke dans une variable informatique `d` en exploitant le fait que la matrice Δ est de longueur $d+1$.

```
3      d = len(Delta) - 1
```

- **Structure itérative**
  Notons que, comme l'indexation des listes commence en 0, pour tout $i \in [\![0, n-1]\!]$, l'élément z_{i+1} sera stocké en case numéro $(i+1)$ de `Z`. Plus précisément, rappelons la formule donnant la valeur de z_i ainsi que les numéros des cases dans lesquelles chacun des éléments est stocké :

$$\begin{array}{lccccccc} \text{valeur :} & z_{i+1} & = & \dfrac{i+2}{i+1} & \displaystyle\sum_{k=0}^{\min(i,d)} & \alpha_k & \times & z_{i-k} \\ & \updownarrow & & & & \updownarrow & & \updownarrow \\ \text{stockée dans la variable :} & \texttt{Z[i+1]} & & & & \texttt{Delta[k]} & & \texttt{Z[i-k]} \end{array}$$

  Pour `i` variant de `0` à `n`, on va mettre à jour le contenu de `Z` en accord avec la formule ci-dessus. Pour ce faire, on met en place une boucle sur `i`.

```
4  for i in range(n) :
```

  Par ailleurs, notons que z_{i+1} requiert le calcul de la somme $\displaystyle\sum_{k=0}^{\min(i,d)} \alpha_k \; z_{i-k}$, ce qui nécessite de réaliser une boucle sur une variable `k` variant de `0` à `min(i, d)` qui permet de mettre à jour une variable `S` (comme somme), initialisée à 0.

```
5          S = 0
6          for k in range(min(i,d) + 1) :
7              S = S + Delta[k] * Z[i-k]
```

  En sortie de cette boucle, la variable `S` contient la somme $\displaystyle\sum_{k=0}^{\min(i,d)} \alpha_k \; z_{i-k}$.
  Il reste alors à concaténer à la liste `Z` la valeur de z_{i+1} comme annoncé plus haut. On complète alors le programme comme suit :

```
8          Z.append( (i+2) / (i+1) * S )
```

- **Fin de programme**
  Rappelons que le but du programme est de calculer z_n et que ce résultat, du fait de l'énoncé, doit être stocké dans la variable `r`. On conclut donc le programme comme suit :

```
10         r = Z[n]
```

Remarque
On peut s'interroger sur la pertinence des choix effectués par le concepteur concernant :

× le nom de la fonction `z`. Ce nom ressemble beaucoup plus au nom d'une variable qu'à un nom de fonction. On aurait pu par exemple la nommer `calculSuiteZ` comme cela est fait dans les énoncés du `TOP5`.

× le nom de la variable de sortie `r`. On aurait pu nommer cette variable `z` (si la fonction ne s'était pas appelée ainsi ...) ou tout simplement `res` comme « résultat » de la fonction.

Remarque
Afin de répondre à cette question, on pouvait aussi exploiter les fonctionnalités **Python**.
On obtient le programme suivant.

```
def z(Delta, n) :
    Z = [1]
    d = len(Delta) - 1
    for i in range(n) :
        S = [Delta[k] * Z[i-k] for k in range(min(i,d) + 1)]
        S = sum(S)
        Z.append( (i+2) / (i+1) * S )
    r = Z[n]
    return r
```