RÉVISIONS Python 1

# Correction

## I] Mise en bouche

### A. Pour commencer

**Q1.** Déterminer les entiers que l'on peut représenter sur sur 1 octet (ie sur 8 bits).

**Correction :** Chaque octet contient 8 bits qui peuvent valoir 0 ou 1, il y a donc $2^8$ possibilités, on peut donc coder en binaire les nombres entre 0 et 255.

**Q2.** Si on considère une image de 1024 par 768 pixels, chaque pixel est représenté par un triplet $(R, G, B)$ (chaque composante de couleur est un entier compris entre 0 et 255). De combien de mémoire (en octet) a-t-on besoin pour stocker une telle image.

**Correction :** Pour un pixel on a besoin de 3 octets, on a donc besoin de $3 \times 1024 \times 768 = 2\,359\,296$ octets.

**Q3.** Écrire une fonction `conv(t,n)` qui prend deux arguments, une température $t$ et un entier $n$, et qui retourne la conversion Celsius $\to$ Fahrenheit si $(n = 1)$, ou Fahrenheit $\to$ Celsius si $(n = 2)$. Rappel : $T_F = 32 + 1.8T_C$

**Correction :**
**Code Python :**

```
def conv(t,n):
    if n==1:
        return 32+1.8*t
    elif n==2:
        return (t-32)/1.8
```

**Q4.** Quand on veut parler la bouche grande ouverte (comme chez le dentiste), il ne reste que les voyelles, par exemple quand on veut dire "j'ai mal, stop", on dit "aiao". Écrire une fonction `dentiste(phrase)` qui prend en entrée une chaine de caractère `phrase` et qui retourne une chaine de caractère avec les voyelles de `phrase` dans le même ordre.

**Correction :**
**Code Python :**

```
def dentiste(phrase):
    VOYELLES="aeiouy"
    res=""
    for car in phrase:
        if car in VOYELLES:
            res=res+car
    return res
print(dentiste("j'ai mal, stop"))
```

**Q5.** Écrire une fonction `nbr_jours_mois(m,a)` qui donne le nombre de jours qu'il y a dans le moi `m` de l'année `a` il est bon de savoir si l'année est bissextile ou non, pour cela il sera plus claire de faire une fonction annexe `est_bissextile(a)` qui prend en entrée une année et renvoie le booléen `True` si elle est bissextile (`False` sinon). De plus, Wikipedia nous apprend que : « Depuis l'ajustement du calendrier grégorien, l'année sera bissextile (elle aura 366 jours) :

(a) si l'année est divisible par 4 et non divisible par 100, ou
(b) si l'année est divisible par 400.

Sinon, l'année n'est pas bissextile (elle a 365 jours).
Ainsi, 2022 n'est pas bissextile. L'an 2008 était bissextile suivant la première règle (divisible par 4 et non divisible par 100). L'an 1900 n'était pas bissextile car divisible par 4 mais aussi par 100 et non divisible par 400. L'an 2000 était bissextile car divisible par 400. »

**Correction :**
**Code Python :**

```
def bissextile(a):
    bis=(a % 4 == 0) and ((a % 100 <> 0) or (a % 400 == 0 ))
    return bis

def nbr_jours_moi(m,a):
    nbr=31
    if m==2:
        if bissextile(a):
            nbr=29
        else:
            nbr=28
    if m in [4,6,9,11]:
        nbr=30
    return nbr
```

## B. On poursuit avec des suites et des séries

**Q6.** Calcul du $n$-ième terme d'une suites.

1° Écrire une fonction qui prend en paramètre un entier $n$ et qui renvoie la valeur de $u_n$ où $(u_n)$ est la suite définie pour $n \in \mathbb{N}$ par $u_n = \cos(n)$.

**Correction :** Il n'y a pas de sous-exemple ...
**Code Python :**

```
def suiteu(n):
    return np.cos(n)
```

2° Écrire une fonction qui prend en paramètre un entier $n$ et qui renvoie la valeur de $v_n$ où $(v_n)$ est la suite définie par $v_0 = 1$ et pour $n \in \mathbb{N}$ par $v_{n+1} = \sqrt{2 + v_n}$. On écrira une version itérative et une version récursive.

**Correction :**
**Code Python :**

```
def suitev(n):
    v=1
    for k in range(n):
        v=np.sqrt(2+v)
    return v

def suitev_rec(n):
    if n==0:
        return 1
    else:
        return np.sqrt(2+suitev_rec(n-1))
```

3° Écrire une fonction qui prend en paramètre un entier $n$ et qui renvoie la valeur de $F_n$ où $(F_n)$ est la suite définie par $F_0 = 0$, $F_1 = 1$, et pour $n \in \mathbb{N}$ par $F_{n+2} = F_{n+1} + F_n$. On écrira une version itérative et une version récursive.

**Correction :**
Il faut faire attention ici, on a besoin de deux termes pour calculer le suivant, il ne faut donc pas perdre le terme précédent. On peut aussi remplacer les trois lignes dans la boucle par `u,v=v,u+v`. Attention toute-

fois, l'affectation multiple est spécifique à Python, de plus cette instruction n'est pas du tout équivalente au fait de faire `u=v` puis `v=u+v`.

Comme à la question précédente, en bonus, une version récursive. Attention toutefois avec la première version récursive la complexité vraiment pas terrible : elle est en $O(r^n)$ où $r = \frac{1+\sqrt{5}}{2}$. En effet si on note $c_n$ le nombre d'opération de `suiteF_rec(n)` on a $c_0 = c_1 = 1$ et pour $n \geq 2$ : $c_{n+2} = c_{n+1} + c_n$ (on néglige l'addition pour simplifier les calculs) qui est une suite récurrente linéaire d'ordre $n$, on sait donc exprimer $c_n$ en fonction de $n$.

Bonus pour ceux qui ont bien compris la récursivité : si on veut un algorithme récursif avec une complexité raisonnable il faut utiliser de la récursivité terminale, cf `suite_F_recterm`.

On verra une autre manière de régler ce problème de complexité avec de la mémoïsation.

**Code Python :**

```
def suiteF(n):
    u=0
    v=1
    if n==0
        return 0
    for k in range(n-1)
        w=u+v
        u=v
        v=w
    return v

def suiteF_rec(n):
    if n==0:
        return 0
    if n==1:
        return 1
    else:
        return suiteF_rec(n-1)+suiteF_rec(n-2)

def suite_F_recterm(n):
    return fib_aux(0, 1, n)

def fib_aux(a, b, n):
    if n == 0:
        return a
    else:
        return fib_aux(b, a+b, n-1)
```

4° On considère la suite définie par $u_0 = 2$ et pour $n \in \mathbb{N}$ par $u_{n+1} = u_n^2 + 1$. Écrire une fonction qui prend en paramètre un flottant $M$ et qui renvoie la plus petite valeur de $n$ telle que $u_n > M$

**Correction :**

**Code Python :**

```
def pluspetit(M):
    u=2
    n=0
    while U<=M:
        u=u**2+1
        n=n+1
    return n
```

**Remarque :**

Une question naturelle est : étant donnée une suite $(u_n)$ qui converge vers un certain $\ell \in \mathbb{R}$, écrire un algorithme qui détermine une valeur approchée de $\ell$ à une précision donnée (par exemple $10^{-3}$) près. Cependant, sans information supplémentaire : **Il n'est pas possible d'écrire un tel algorithme**.

En effet pour le faire il faut déterminer un $n$ tel que $|u_n - \ell| \leq 10^{-3}$, ce qui passe par l'étude de la suite (de sa vitesse de convergence pour être plus précis) ; dans le cas d'une suite récurrente l'inégalité des accroissements finis peut, par exemple, aider à trouver une majoration de $|u_n - \ell|$.

En particulier la condition $|u_{n+1} - u_n| \leq \varepsilon$ ne garantie en rien que $u_n$ est une valeur approchée de $\ell$ à $\varepsilon$ près, c'est

cependant la condition qu'on utilise cependant quand on ne peut (veut) pas faire mieux.

**Q7.** Calcul d'une somme.

1° Écrire une fonction qui prend en paramètre un entier $n$ et qui renvoie la valeur de $S_n$ où $S_n = \sum_{k=1}^{n} \frac{(-1)^{k+1}}{k}$.

**Correction :**
RAS, juste faire attention au `range`.
**Code Python :**

```
def series(n):
    S=0
    for k in range(1,n+1):
        S=S+(-1)**(k+1)/k
    return S
```

2° Écrire une fonction qui prend en paramètre un entier $n$ et qui renvoie la valeur de $T_n$ où $T_n = \sum_{k=1}^{n} v_k$ où $(v_n)$ est la suite définie en **Q6.2°**. On donnera la complexité de sa fonction (on considérera que calculer une racine carré est on $O(1)$), on essaiera d'avoir une complexité linéaire.

**Correction :**
Ici, il est préférable de ne pas ré-utiliser la fonction `suitev`, en effet si on le fait le nombre d'opération est $\sum_{k=1}^{n} c_k$ où $c_k$ est le nombre d'opération de `suitev(k)`, comme on a $c_k = 2k$, et on aurait une complexité en $O(n^2)$. Alors qu'avec le code ci dessous on est en $O(n)$.
**Code Python :**

```
def suitev(n):
    v=1
    S=1
    for k in range(n):
        v=np.sqrt(2+v)
        S=S+v
    return v
```

3° Le TSA [1] appliqué à la série de la question **Q7.1°** permet d'avoir la convergence de $(S_n)$ vers un certain [2] $S \in \mathbb{R}$, ainsi que la majoration $|S - S_n| \leq \frac{1}{n+1}$, en déduire un algorithme pour calculer une valeur approchée de $S$ à $10^{-3}$ près.

**Correction :** Pour $n \in \mathbb{N}^\star$ on a (attention on peut permuter la somme et l'intégrale ici car c'est une somme finie et une intégrale propre, si la somme était infinie ou si l'intégrale était généralisée on ne le pourrait pas, tout du moins sans utiliser un théorème que vous verrez plus tard) :

$$\sum_{k=1}^{n} \frac{(-1)^{k+1}}{k} = \sum_{k=1}^{n} (-1)^{k+1} \int_0^1 x^{k-1} \mathrm{d}x = \int_0^1 \sum_{k=1}^{n} (-1)^{k+1} x^{k-1} \mathrm{d}x = \int_0^1 \sum_{k=0}^{n-1} -(-1)^{k+1} x^k \mathrm{d}x =$$
$$\int_0^1 \sum_{k=0}^{n-1} (-x)^k \mathrm{d}x = \int_0^1 \frac{1-(-x)^n}{1-(-x)} \mathrm{d}x = \int_0^1 \frac{1}{1+x} \mathrm{d}x - (-1)^n \int_0^1 \frac{x^n}{1+x} \mathrm{d}x.$$

Or la seconde intégrale tend vers 0 quand $n$ tend vers $+\infty$, en effet $0 \leq \int_0^1 \frac{x^n}{1+x} \mathrm{d}x \leq \int_0^1 x^n \mathrm{d}x = \frac{1}{n+1}$, et la première intégrale vaut $\ln 2$. On en déduit donc que la série converge (ce qu'on savait déjà par le théorème spécial) et que sa somme vaut $\ln(2)$

Pour le code on utilise rien d'autre que $|S - S_n| = |R_n| \leq \frac{1}{n+1}$. On pourrait, en remarquant que $\frac{1}{n+1} \leq \varepsilon \iff n \geq \frac{1}{\varepsilon} - 1$, juste renvoyer `series(np.floor(1/eps))`.
**Code Python :**

```
def valap(eps):
```

1. Quelles sont ses hypothèses ?

2. Ce n'est pas le but ici, mais une astuce possible pour calculer sa valeur est de remarquer, pour tout $k \in \mathbb{N}^\star$, que $\frac{1}{k} = \int_0^1 t^{k-1} \mathrm{d}t$ et ainsi de faire une permutation dans $S_n$ entre une somme finie et une intégrale propre.

```
    n=1
    S=1
    while 1/(n+1)>eps:
        n=n+1
        S=S+(-1)**(n+1)/n
    return S
```

# II] Annotation, assertion et jeu de tests

Python détermine tout seul le type d'un objet (on appelle cela le typage dynamique), cependant pour éviter certaines erreurs il peut être intéressant d'imposer des types pour les arguments des fonctions (en effet une liste et un tableau numpy : ce n'est pas pareil ![3]), pour cela ont peut utiliser la syntaxe d'annotation des fonction (qui est décoratif).

Ainsi `def maFonction(n:int, x:float, l:[str]) -> (int, np.ndarray):` signifie que la fonction `maFonction` prend trois arguments, le premier est un entier, le deuxième un nombre à virgule flottante et le troisième une liste de chaines de caractères et qu'elle renvoie un couple dont le premier élément est un entier et le deuxième un tableau numpy.

Les types qu'on on utilisera en Python sont :

— `bool`, `int`, `float`, `str` pour les booléens, les entiers, les flottants et les chaînes de caractères ;
— `(type1,type2)` pour un tuple à deux éléments le premier de type `type1` et le second de type `type2` ;
— `[type]` pour les listes d'éléments de type `type` ;
— `callable` pour les fonctions
— `any` si le type n'est pas spécifié
— et pleins d'autres : `None`, `np.ndarray`, ...

De plus il est de bon ton de commencer sa fonction par une description, par exemple :

```
def racine_entiere(n:int) -> int:
        """ Fonction donnant le plus grand entier dont le carré est <= à n
        n doit être un entier positif ou nul
        """
        m = 0
        while (m+1)**2 <= n :
                m += 1
        return m
```

C'est cette description (et les annotations) qui sont affichées quand on fait `help(racine_entiere)`.

Cette précondition (`n` entier positif) n'est pas vérifiée, on peut demander (dans une optique de débogage, la charge de la vérifications des préconditions repose sur celui qui appel la fonction) de le vérifier en début de fonction, pour cela on utilise `assert test` qui interrompra le programme si le test n'est pas vérifié, ici on peut mettre au début de la fonction `assert isinstance(n,int) and n>=0`.
*Remarque* : Cette interruption de programme est en fait une exception nommée `AssertionError`, il s'avère qu'il y en a pleins d'autres (`ZeroDivisionError` pour une division par zéro, ...), et qu'on peut les provoquer (on dit lever une exception), il est possible de les intercepter et de les traiter en fonction, c'est le rôle de `try: bloc1 except ZeroDivisionError: bloc2 else: bloc3` où Python exécute le bloc d'instruction `bloc1`, s'il y a une division par zéro pendant cette exécution il exécute le bloc `bloc2` s'il y a une autre erreur il exécute le bloc `bloc3`. Cela peut être une manière de gérer des cas particulier. Toutes les notions de cette remarque sont hors programme.

Il est de bon ton de tester les fonctions qu'on écrit, pour cela on doit vérifier nos fonctions sur un ensemble de tests (ie de couples entrées/sorties) judicieusement choisis (on doit vérifier les différents cas possibles et les cas limites).
Par exemple pour notre fonction `racine_entiere` il me semble bon de vérifier un cas ou `n` est un carré parfait, un cas où `n` ne l'est pas et le cas limite qui est `0`, ainsi le jeu de tests pourrait être $(9,3)$, $(17,4)$ et $(0,0)$.

Il faut s'assurer de la terminaison (en particulier s'il y a une boucle `while` ou si on a une fonction récursive) de son algorithme et éventuellement démontrer qu'il fait bien ce qu'on veut.

3. La somme de deux listes est la concaténation alors que la somme de deux tableaux de même taille est la somme terme à terme

Enfin on termine par déterminer la complexité de son algorithme.