TP 2 : Boucles (I)

L'énoncé vous donne souvent une valeur numérique permettant de vous assurer du bon fonctionnement de vos programmes. Si ceux-ci ne fonctionnent pas, essayez de trouver l'erreur, mais n'hésitez pas à nous appeler en cas de besoin.

Exercice 1 Échauffement

F

Si `s` *est une chaîne de caractères (type string), et si* `n` *est un entier positif, alors* `s*n` *est une chaîne de caractères formée de* `n` *répétitions de* `s`. *Par exemple,* `'Toc '*3` *est la chaîne de caractères* `'Toc Toc Toc '`.

1. Écrire une fonction `rectangle` qui prend en paramètre deux entiers n et p et affiche un rectangle de n lignes et p colonnes, toutes formées du même caractère de votre choix ($*$ ou $-$ par exemple).
2. Sur le même principe, écrire une fonction `triangle`, qui prend en paramètre un seul entier n et affiche un triangle à n lignes, la première contenant un seul symbole, la seconde deux symboles, etc, la $n^{\text{ème}}$ ligne étant formée de n symboles.

Exercice 2 Échauffement (bis)

PD

1. Écrire une fonction qui prend comme paramètre un entier n et retourn la somme des carrés des entiers de 1 à n (c'est-à-dire $1^2 + 2^2 + \cdots + n^2$).
2. On a $1^3 = 1^2$, $1^3 + 2^3 = 9 = (1+2)^2$, $1^3 + 2^3 + 3^3 = 36 = (1+2+3)^2$.
   Mais est-il vrai que pour tout $n \in \mathbf{N}$, $1^3 + 2^3 + 3^3 + \cdots + n^3 = (1 + 2 + 3 + \cdots + n)^2$?
   Proposer un programme qui teste si cette égalité est vraie pour $n \leqslant 10\,000$, et qui affiche les valeurs de n pour lesquelles cette égalité n'est pas valable.
3. Écrire une fonction qui prend comme paramètre un entier n et retourne la somme des entiers impairs compris entre $2n$ et n^2.

Exercice 3 Des suites récurrentes

PD

1. On définit une suite (u_n) de la manière suivante : $u_0 = 3$ et $\forall n \in \mathbf{N}^*$, $u_{n+1} = \dfrac{3u_n + 2}{u_n + 4}$.
   Écrire une fonction `suite`, qui prend comme paramètre un entier n et retourne la valeur de u_n correspondante. *Auto-vérification : $u_5 \simeq 1.0123$.*
   Utiliser cette fonction pour émettre une conjecture quant à la convergence de (u_n).
2. On considère à présent la suite $(F_n)_{n\in\mathbf{N}}$ définie par $F_0 = F_1 = 1$ et pour tout $n \in \mathbf{N}$, $F_{n+2} = F_{n+1} + F_n$ (cette suite célèbre est appelée la suite de Fibonacci). Écrire une fonction `fibonacci(n)` qui retourne la valeur de F_n. *Auto-vérification : $F_{10} = 89$.*

Exercice 4 Encore des suites récurrentes

AD

1. Dans cette question, on considère une suite $(u_n)_{n\geqslant 1}$ vérifiant $u_1 \geqslant 0$ et pour tout $n \in \mathbf{N}^*$, $u_{n+1} = \sqrt{u_n} + \dfrac{1}{n}$.
   (a) Écrire une fonction `suite` qui prend comme paramètres un réel positif `u` (qui correspond à u_1) et un entier $n \geqslant 1$, et qui retourne la valeur de u_n. *Vérification : si $u_1 = 3$, alors $u_5 \simeq 1.591$.*
   (b) Tester cette fonction avec différentes valeurs de u et de n, et émettre une hypothèse quant à la limite de la suite (u_n).
2. Dans cette question, on considère la suite (u_n) définie par $u_0 = 1$ et pour tout $n \in \mathbf{N}$, $u_{n+1} = \dfrac{1}{u_n} + \cos(nu_n)$.
   Écrire une fonction qui prend comme paramètre un entier N et renvoie une valeur de $k \in [\![1, N]\!]$ telle que u_k soit le plus grand des nombres $u_0, u_1, \ldots, u_N$. *Vérification : pour $N = 100$, on trouve $k = 35$.*

Exercice 5 Moyenne arithmético-géométrique

PD

Soient a et b deux réels positifs. On définit deux suites (u_n) et (v_n) par $u_0 = a$, $v_0 = b$ et pour tout $n \in \mathbf{N}$,

$$\begin{cases} u_{n+1} = \sqrt{u_n v_n} \\ v_{n+1} = \dfrac{u_n + v_n}{2} \end{cases}$$

1. Écrire une fonction qui prend comme paramètres a, b et un entier n, et retourne le couple (u_n, v_n).
2. Vérifier sur des exemples que (u_n) et (v_n) semblent avoir une limite commune. *On admet (pour l'instant) que c'est bien le cas, on note $M(a,b)$ cette limite, que l'on appelle moyenne arithmético-géométrique de a et b. On admet également que pour tout $n \in \mathbf{N}$, $u_n \leqslant u_{n+1} \leqslant M(a,b) \leqslant v_{n+1} \leqslant v_n$.*
3. Écrire une fonction qui prend comme paramètres a, b et un réel $\varepsilon > 0$ et retourne une valeur approchée de $M(a,b)$ à ε près. *On vérifiera par exemple que $M(1, \sqrt{2}) \simeq 1,91814\ldots$*

► Pour aller plus loin

Exercice 6 Nombres parfaits

PD

Un entier n est dit parfait si la somme de ses diviseurs positifs vaut $2n$.
Par exemple, le plus petit nombre parfait est 6 car la somme de ses diviseurs vaut $1 + 2 + 3 + 6 = 12$.

1. Écrire une fonction qui teste si un entier n est parfait.
2. Écrire une fonction qui affiche la liste de tous les entiers parfaits inférieurs à 10 000.
3. On conjecture qu'il n'existe aucun entier parfait impair. Vérifier cette conjecture pour les entiers impairs inférieurs ou égaux à 50 000.

Pour les exercices qui suivent, on rappelle que :

- le nombre de chiffres de l'écriture d'un entier n en base 10 est $\lfloor \log(n) \rfloor + 1$ (où log est bien le logarithme de base 10)
- le chiffre des unités (en base 10) d'un entier n, s'obtient à l'aide de la commande `n%10`

Exercice 7 Palindromes

D

Un nombre est un palindrome s'il a la même valeur qu'on le lise de gauche à droite ou de droite à gauche, comme par exemple 121 ou 34643.

1. Écrire une fonction qui prend comme paramètre un entier n et retourne son «miroir». Par exemple, le miroir de 432 est 234 et celui de 1231 est 1321. En déduire une fonction qui teste si un entier est un palindrome. *On demande de n'utiliser que des opérations arithmétiques : `+,*,/,%,//` et pas d'opérations sur des objets de type* **`string`** *(pour ceux qui en connaîtraient déjà).*
2. Trouver le plus grand palindrome à 6 chiffres qui soit le produit de deux nombres de trois chiffres.
   (⋆) Adapter le programme précédent pour trouver le plus grand palindrome à $2n$ chiffres qui soit produit de deux nombres à n chiffres.

Exercice 8 La suite de Conway

TD

La suite de Conway (ou suite audioactive) a été introduite en 1986 par le mathématicien américain John Conway (décédé cette année du Covid-19), et popularisée en France par le roman *Le jour des fourmis* de Bernard Werber, paru en 1994.

Ses premiers termes sont : 1, 11, 21, 1211, 111221, 312211, 13112221.
Explication : chaque terme est ce que dirait un enfant qui ne sait pas compter en lisant à voix haute le précédent, en se contenant d'en décrire les chiffres qui le composent.
Par exemple, 111221 se lirait : «trois 1, deux 2, un 1», soit encore 312211. Et de même, ce dernier nombre se lirait : « un 3, un 1, deux 2, deux 1», soit encore 13112221.

1. Montrer que les chiffres des termes de la suite de Conway ne peuvent être que 1, 2 ou 3.
2. Écrire une fonction qui affiche les n premiers termes de la suite de Conway.

Exercice 9 Le nombre 145 a une bien curieuse propriété : $145 = 1! + 4! + 5!$

TD

Autrement dit, il est la somme des factorielles de ses chiffres.
Trouver tous les entiers qui ont également cette propriété.

Correction des exercices du TP 2

Solution de l'exercice 1

1. Il est facile d'obtenir une ligne formée de p symboles, avec `print('*'*p)`.
Et donc il suffit de répéter cela n fois :

```
def rectangle(n,p) :
        for i in range(n) :
                print('*'*p)
```

2. C'est le même principe, sauf que je souhaite que la première ligne contienne un symbole, la deuxième deux symboles, etc, la $i^{\text{ème}}$ contienne i symboles.

```
def triangle(n) :
        for i in range(1,n+1) :
                print('*'*i)
```

□

Solution de l'exercice 2

1. N'oublions pas que $0! = 1$.

```
def somme_carres(n) :
        S =0
        for i in range(n+1) :
                S+=i**2
        return S
```

2. Commençons par expliquer comment utiliser une boucle `for` pour calculer une telle somme : l'idée est d'utiliser une variable qui vaut initialement 0, puis de lui ajouter successivement $1, 2, 3, \ldots, n$.

```
s = 0
for i in range(1,n+1) :
        s += i
```

Il est donc possible d'utiliser ce principe pour chaque valeur de n pour calculer les deux sommes.

```
somme = 0
somme_cubes = 0
for i in range(1,10001) :
        somme +=i
        somme_cubes +=i**3
        if somme_cubes  != somme**2 :
                print(i)
```

Ce programme n'affiche rien, et donc la formule annoncée est valable au moins jusqu'à $n = 10\,000$.

3. Souvenons nous que la fonction `range` permet d'avancer de 2 en 2 :

```
def somme_impairs(n) :
        S = 0
        for i in range(1,2*n,2) :
                S +=i
        return S
```

□

Astuce

Une fois calculée $1^3 + \cdots + n^3$, il serait dommage de repartir de zéro pour calculer $1^3 + \cdots + (n+1)^3$: il suffit d'ajouter $(n+1)^3$ à la somme précédente !
Le temps de calcul s'en trouve considérablement raccourci.

Remarque

Cette formule est en réalité valable pour tout $n \in \mathbf{N}$, vous pouvez essayer de le prouver par récurrence sur n.

Remarque

La valeur de u est effacée à chaque passage dans la boucle, ce qui n'est pas un vrai problème si on ne veut au final que la valeur de u_n.

Solution de l'exercice 3

1. Pour calculer u_n, il suffit de partir de u_0 et d'appliquer n fois la «recette» : prendre la valeur précédente, la multiplier par 3, lui ajouter 2 et diviser par, etc.

```
def suite(n) :
        u=3
        for i in range(n) :
                u = (3*u+2)/(u+4)
        return(u)
```

2. Il faut être un peu plus subtil que dans la question précédente, puisqu'il faut à chaque étape garder deux valeurs de la suite, la dernière (F_n) et l'avant dernière (F_{n-1}), qui nous serons utiles pour calculer le terme suivant (F_{n+1}).
À cet effet, on peut utilise deux variables `a` et `b` qui à chaque instant stockent les deux derniers termes calculés de la suite.

```
def fibonacci(n) :
        a = 1
        b = 1
        for i in range(n-1) :
                a,b = b,a+b
        return(b)
```

□

Solution de l'exercice 4

1.a. Il faut un peu plus d'attention ici, car la transformation qu'on applique à u_n pour obtenir u_{n+1} dépend de n.
En particulier, on a $u_2 = \sqrt{u_1} + 1$ et $u_n = \sqrt{u_{n-1}} + \dfrac{1}{n-1}$.
Nous proposons donc le code suivant :

```
from math import sqrt
def suite2(u,n) :
        for i in range(1,n) :
                u = sqrt(u)+1/i
        return(u)
```

1.b. Quand n est grand, on constate qu'indépendamment de la valeur de u_1, u_n est proche de 1, et donc il semblerait[1] que $u_n \underset{n\to+\infty}{\longrightarrow} 1$.

2. Il faut calculer successivement les valeurs de u_n, sur le modèle de la question précédente, tout en gardant une trace de la plus grande valeur de u_n déjà rencontrée[2]
Il faut alors se méfier des éventuels décalages d'indices :si on choisit, comme ci-dessous d'utiliser `range(N)`, alors la boucle va commencer à $i = 0$.
Mais alors le terme de la suite calculé à ce moment sera u_1, et non u_0.

```
def max_suite(N) :
    maximum = 1
    k = 0
    u = 1
    for i in range(N) :
        u = 1/u+cos(i*u)
        if u>maximum :
            maximum = u
            k = i+1
        return(k)
```

⚠ J'ai constaté que la valeur retournée par ce programme dépend de l'ordinateur qu'on utilise en raison d'erreurs d'arrondi...
Les ordinateurs du lycée en particulier semblent retourner 343....

□

Bornes

Notons que pour calculer u_2, on n'effectue qu'une fois l'opération qui permet de passer d'un terme au suivant. Quand on calcule u_3, on l'effectue deux fois, ..., quand on calcule u_n, on l'effectue $n-1$ fois. D'où le `range(1,n)` et pas `range(1,n+1)`.

[1] Mais ce n'est qu'une hypothèse !

[2] C'est à cela que sert la variable `maximum` du programme qui suit.

Solution de l'exercice 5

1. Il est possible d'utiliser des variables temporaires pour stocker u_n et v_n, mais le plus simple est encore de changer les valeurs des deux variables à la fois :

```
def suites(a,b,n) :
        u,v = a,b
        for i in range(n) :
                u,v = (u*v)**.5,(u+v)/2
        return u,v
```

3. Puisque l'écart $v_n - u_n$ entre u_n et v_n est décroissant[3], il s'agit de trouver une valeur de n à partir de laquelle cet écart est inférieur à ε.
Puisque $M(a,b)$ est entre u_n et v_n, alors à la fois u_n et v_n seront une approximation de $M(a,b)$ à ε près.

[3] Cela découle de la croissance de (u_n) et de la décroissante de (v_n).

```
def moyenne(a,b,eps) :
    u,v = a,b
    while v-u>eps :
        u,v = (u*v)**.5,(u+v)/2
    return u
```

□

Solution de l'exercice 6

1. Remarquons qu'un entier n est parfait si et seulement si la somme de ses diviseurs strictement inférieurs à n vaut n.
Or un tel diviseur est nécessairement inférieur ou égal à $n/2$.

```
def parfait(n) :
        s = 0
        for i in range(1,n//2+1) :
                if n%i == 0 :
                        s +=i
        if n==s :
                return(True)
        else :
                return(False)
```

2. Il suffit de faire une boucle testant successivement tous les entiers à l'aide de la fonction précédente.

```
for n in range(1,10001) :
        if parfait(n) :
                print(n)
```

On trouve alors seulement 4 nombres parfaits : $6, 28, 496, 8128$.

3. Puisqu'il suffit de tester les impairs, utilisons un troisième paramètre dans la fonction `range`.

```
for i in range(1,50000,2) :
        if parfait(i) :
                print(i)
```

Pour la culture

On ignore s'il existe une infinité de nombres parfaits ou non.
À l'heure actuelle, on n'en connaît que 51, qui sont reliés à certains nombres premiers appelés nombres de Mersenne.

□

Solution de l'exercice 7

```
def miroir(n) :
        res = 0
        nb_ch = floor(log10(n))+1
        for i in range(nb_ch) :
                res =res*10 + n%10
                n = n//10
        return(res)

def palindrome(n) :
        if n==miroir(n) :
                return(True)
        else :
                return(False)
```

2. Une solution est de parcourir l'ensemble des palindromes à 6 chiffres, qui sont de la forme *abccba*, c'est-à-dire $a \times 100001 + b \times 10010 + c \times 1100$.
Et pour chacun de ces nombres, on teste s'il est ou non multiple de trois chiffres, ce qui peut se tester à l'aide d'une fonction auxiliaire.
Notons que si n possède deux diviseurs à trois chiffres, l'un d'entre eux, appelons le a, doit forcément être plus petit que $\sqrt{n}$ (et plus grand que 100 puisqu'il a trois chiffres).
Il faut alors tester :

- si $\frac{n}{a}$ est un entier
- si cet entier possède lui aussi trois chiffres, c'est-à-dire s'il est compris entre 100 et 999.

```
def prod_trois_chiffres(n) :
    for i in range(floor(sqrt(n)),99,-1) :
        a = n//i
        if n == a*i and a<1000 and a>99 :
            return True
    return False

def plus_grand_palindrome() :
    for a in range(9,0,-1) :
        for b in range(9,-1,-1) :
            for c in range(9,-1,-1) :
                n = a*100001 + b*10010 + c*1100
                if prod_trois_chiffres(n) :
                    return(n)
        print("Il n'existe aucun palindrome à 6 chiffres qui soit produit
 de deux nombres à 3 chiffres.")
```

Une fois encore, nous avons utilisé le fait que **return** interrompt l'exécution de la fonction. Donc la dernière ligne de la fonction `prod_trois_chiffres` (la ligne **return** `False`) n'est exécutée uniquement si on n'a pas réussi à écrire n comme produit de deux nombres à trois chiffres (car on aurait alors retourné **True**).
De même, la fonction `plus_grand_palindrome` ne retourne un résultat que s'il existe un palindrome de la forme cherchée, sinon elle ne retourne rien et affiche un message d'erreur.
Le plus grand palindrome cherché est alors $906\,609 = 913 \times 993$. Pour la généralisation, il ne sera pas très dur de généraliser la fonction `prod_trois_chiffres` en une fonction qui détermine si un entier est un produit de deux nombres de n chiffres, mais le problème sera de générer des palindromes à $2n$ chiffres. _ Notons que pour obtenir un tel palindrome, il suffit de choisir un nombre de n chiffres, qui sera la partie «gauche» de notre palindrome, et de prendre son «miroir».

```
def prod_k_chiffres(n,k) :
    for i in range(floor(sqrt(n)),10**(k-1)-1,-1) :
        a = n//i
        if n == a*i and a<10**k and a>10**(k-1)-1 :
            return True
    return False

def palindrome(gauche) :
        n = floor(log10(gauche))+1
        return(gauche*10**n + miroir(gauche))

def plus_grand_pal(n) :
        for i in range(10**n-1,10**(n-1),-1) :
                pal = palindrome(i)
                if prod_k_chiffres(pal,n) :
                        return(pal)
```

□

Solution de l'exercice 8

1. Pour que l'un des termes de la suite de Conway, appelons-le u_n contienne un 4, il faudrait que u_{n-1} contienne 4 fois d'affilée le même chiffre.
Par exemple 4 chiffres 1 consécutifs. Mais alors, en groupant les chiffres deux par deux[4], u_{n-1} contiendrait un bloc de la forme : $\ldots\ a1\,11\,1*\ \ldots$ ou $\ldots\ 11\,11\ \ldots$.
Le premier cas est impossible, puisque u_{n-2} contiendrait alors (au moins) $a+1$ chiffres 1 consécutifs, et donc u_{n-1} contiendrait les deux chiffres $(a+1)1$ au lieu de $a1\,11$.
De même, si u_{n-1} contient $11\,11$, alors u_{n-2} contient 2 chiffres 1 d'affilée, et donc u_{n-1} contient la suite 21 plutôt que $11\,11$.

[4] Aviez-vous remarqué que tous les termes de la suite de Conway excepté le premier ont un nombre pair de chiffres ?

On prouve de la même manière qu'il ne peut y avoir de 5, de 6, ou plus.

2. En français, la lecture se fait de gauche à droite. Mais comme nous l'avons fait plus tôt, il est plus facile de lire les chiffres d'un nombre de droite à gauche (en commençant par le chiffres de unités, puis des dizaines, etc).
Utilisons donc la fonction miroir de l'exercice sur les palindromes.
Puis il s'agit de parcourir les chiffres de u_n en vérifiant à chaque fois si ce chiffre fait partie d'une série de chiffres identiques déjà entamée, ou s'il est différent de celui qui précède, et donc entame une nouvelle série.

```
def lookandsay(n) :
    res = 0
    n = miroir(n)
    len = floor(log10(n))+1 // le nombre de chiffres de n
    actuel = n%10
    nb_actuel = 1
    n = n//10
    for i in range(len) : //on va parcourir un à un tous les chiffres
        if actuel == n%10 :
            n = n//10
            nb_actuel +=1
        else :
            res = res*100 + nb_actuel*10 + actuel
            actuel = n%10
            n = n//10
            nb_actuel = 1
    return(res)

def conway(n) :
    u = 1
        print(1)
    for i in range(n) :
        u = lookandsay(u)
        print(u)
    return(u)
```

□

Solution de l'exercice 9

Commençons par ramener le problème à l'étude d'un nombre fini de cas : si n possède k chiffres, alors $10^{k-1} \leqslant n < 10^k$.
Or, la somme des factorielles de ses chiffres est donc inférieure à $k \times 9!$.
Donc s'il vérifie la propriété requise, nécessairement,

$$10^{k-1} \leqslant k \cdot 9! \Leftrightarrow (k-1)\ln(10) \leqslant \ln(k) + \ln(9!)$$

Une rapide étude de la fonction $x \mapsto \ln(x) - (x-1)\ln(10)$ prouve qu'elle est croissante à partir de $\dfrac{1}{\ln(10)} \simeq 0.43$
En tâtonnant un peu, on arrive vite au fait que pour $k \geqslant 8$, $(k-1)\ln(10) > \ln(k) + \ln(9!)$
Donc les entiers que nous cherchons ont au plus 7 chiffres. Et donc valent au plus $7 \times 9!$.
Il suffit alors de tester un par un ces entiers.

```
for i in range(1,7*factorial(9)+1) :
        m=i
```

```
3         n = floor(log10(i))+1
4         s = 0
5         for j in range(n) :
6                 s += factorial(m%10)
7                 m = m//10
8         if s==i :
9                 print(i)
```

On obtient alors que, hormis 1 et 2 qui sont évidemment solutions, les deux seuls entiers égaux à la somme des factorielles de leurs chiffres sont 145 et 40585.

□