RECURSIVITE
Exercices - Corrigés

Exercice1 - Un calcul très classique

Ecrire une fonction Python qui calcule la somme des inverses des carrés des n premiers entiers naturels non nuls.

On pourra ensuite écrire un script plus complet qui, après le calcul précédent, évalue et affiche l'écart (en %) avec la limite de cette somme qui vaut $\frac{\pi^2}{6}$ (rappel : le nombre π ne fait pas partie intégrante du cœur du du langage Python. On importera donc `pi` via la bibliothèque `math` : `from math import pi`).

On cherche ici à calculer, pour tout entier naturel n non nul : $S_n = \sum_{i=1}^{n} \frac{1}{i^2}$.

On rappelle que l'on a : $\lim_{n\to+\infty} S_n = \lim_{n\to+\infty} \sum_{i=1}^{n} \frac{1}{i^2} = \frac{\pi^2}{6}$.

Le cas de base est celui où la somme ne comporte qu'un terme (1 en l'occurrence) : $S_1 = 1$.
C'est donc le cas où il n'y aura pas d'appel récursif.

On a alors :

```
def sum_sq_inv(n):
    if n == 1:
        return 1
    else:
        return 1/n**2 + sum_sq_inv(n-1)
```

Evidemment, lors de l'appel initial à cette fonction, on devra s'être assuré, d'une façon ou d'une autre, que l'argument `n` est bien un entier naturel non nul. C'est pourquoi, dans le programme (script) ci-après, le bloc d'instructions suivant a été ajouté :

```
n = 0
while n < 1:
    n = int(input('Nombre de termes ? '))
```

Si on souhaite cependant effectuer les tests dans la fonction elle-même (ce n'est pas une très bonne idée car cela génère de trop nombreux tests… inutiles !), on pourra utiliser le code suivant :

```
def sum_sq_inv(n):
      if type(n) != int:
            raise TypeError('Vous devez fournir un entier !')
      elif n <= 0:
            raise ValueError('Vous devez fournir un entier naturel non nul !')
      else:
            if n == 1:
                  return 1
            else:
                  return 1/n**2 + sum_sq_inv(n-1)
```

Voici un script possible :

```
# La fonction récursive pour le calcul de la somme
proprement dit.
def sum_sq_inv(n):
     if n == 1:
          return 1
     else:
          return 1/n**2 + sum_sq_inv(n-1)

# DEBUT DU SCRIPT
# ===============

# Importation de pi
from math import pi

# Nombre de termes de la somme.
n = 0
while type(n) != int or n < 1:
     n = int(input('Veuillez saisir le nombre de termes
     (entier naturel non nul) à sommer ? '))

# Calcul de la somme.
r = sum_sq_inv(n)

# Erreur relative commise (pourcentage).
error = 100 * (6 * r / pi**2 - 1)

# Affichage des résultats.
print('La somme des inverses des carrés des '+str(n)+
' premiers entiers naturels non nuls vaut : '+str(r))
print('L\'erreur commise vaut '+str(error)+' %.')

# FIN DU SCRIPT
# =============
```

Exercice 2 – Une fonction mystérieuse ?

```
def dk(L1,L2=[]):
if L1 == []:
     return L2
else:
     s = L1.pop(0)
     if s not in L2:
          L2.append(s)
     return dk(L1,L2)
```

Que renverra la fonction `dk` définie ci-dessus lorsqu'on l'appelle comme suit ?

`dk([34,2,3,11,11,2,34,7,1,7,7,11,3,11])`

Comme suggéré en séance, faire « tourner un algorithme à la main » est très formateur !
Que vous ayez agi de la sorte ou en écrivant rapidement la fonction ci-dessus sous IDLE (ou équivalent), vous avez rapidement réalisé que la fonction `dk` éliminait « simplement » les éléments redondants de la première liste passée en argument tout en construisant une deuxième liste correspondant à … `L1` privée des éléments redondants. Ainsi, la fonction `dk` supprime les doublons, … c'est une « doublons killer ».

L'appel

`dk([34,2,3,11,11,2,34,7,1,7,7,11,3,11])`

renverra donc la liste : `[34, 2, 3, 11, 7, 1]`.

Exercice 3 – Le compte est-il bon ?

Ecrire une fonction Python « `compte_a` » qui reçoit comme argument une chaîne de caractères (éventuellement vide) et renvoie le nombre de « `a` » qu'elle contient.

La chaîne passée en argument peut contenir des lettres majuscules. On pourra souhaiter transformer la chaîne en une chaîne ne contenant que des minuscules (à vous de trouver la méthode qui réalise très bien cette transformation). Pourquoi n'est-il pas pertinent d'utiliser cette méthode dans votre fonction ?

Vous pourrez tester votre fonction avec les chaînes suivantes : ‘ ‘, ‘sphinx’, ‘abracadabra’ et ‘automorphisme’.

Une chaîne de caractère non vide n'est rien d'autre qu'un premier caractère précédant … une autre chaîne de caractère (éventuellement vide) ! Et voilà notre principe de récurrence quasiment en place !

Le cas de base correspond à une chaîne de caractère vide.

Pour coder le principe précédent, il convient de rappeler que les chaîne de caractères sont indexables comme les listes (mais les méthodes classiques des listes ne sont pas disponibles).

On peut donc simplement accéder à un caractère quelconque d'une chaîne de caractères via son indice (comme pour une liste, le premier caractère est d'indice nul) et on peut également accéder à une « sous-chaîne » de caractères.

Ainsi, si on appelle `s` une chaîne de caractères non vide, on accèdera à son premier caractère grâce à `s[0]` et la chaîne s privée de son premier caractère s'obtient grâce à `s[-2 :]`.
On obtient alors la fonction :

```
def compte_a(s):
     if type(s) != str:
          raise TypeError('Vous devez fournir une chaîne de
          caractères !')
     elif len(s) == 0:
          return 0
     else:
          n = compte_a(s[1:])
          if s[0] == 'a':
               n += 1
          return n
```

Exercice 4 – Sens dessus dessous

Ecrire une fonction Python permettant de déterminer si une chaîne de caractères est ou non un palindrome (i.e. pouvant être lue indifféremment de la gauche vers la droite ou de la droite vers la gauche).

La démarche générale consiste à comparer les caractères situés aux extrémités de la chaîne.
En notant `s` la chaîne de caractères, on testera donc `s[0]` et `s[-1]`.
On poursuit cette comparaison sur la nouvelle chaîne obtenue en « amputant » la chaîne courante des deux caractères précédents tant que les deux caractères testés sont identiques.
Cette « amputation » consiste à conserver la sous-chaîne : `s[1:-1]`.

Pour ce qui est du cas de base, deux situations sont à prendre en compte : celle d'une chaîne de caractères vide et celle d'une chaîne ne comportant qu'un seul caractère.

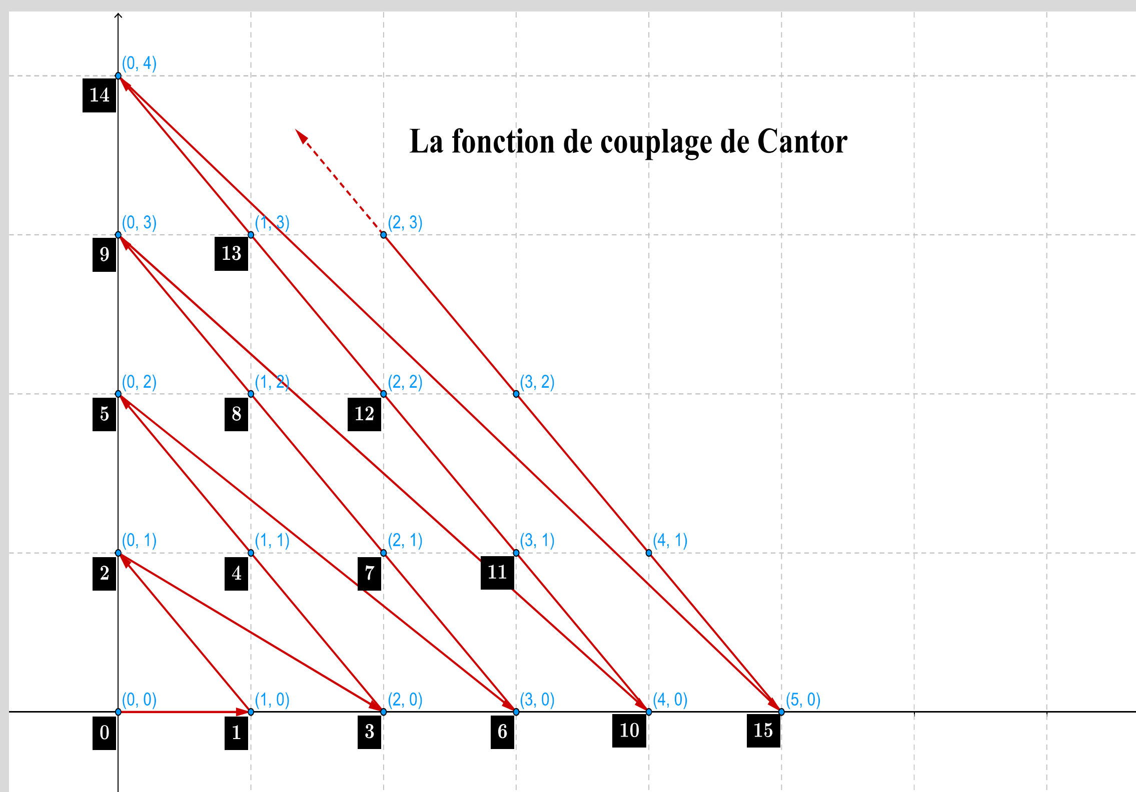
D'où la fonction `palindrome` qui, notons-le, renvoie un booléen.

```
def palindrome(s):
     if len(s) == 0 or len(s) == 1:
          return True
     else:
          if s[0] != s[-1]:
               return False
          else:
               return palindrome(s[1:-1])
```

Exercice 5 – Une bijection de $\mathbb{N}\times\mathbb{N}$ dans $\mathbb{N}$

Ci-après, une représentation de la fonction de couplage de Cantor qui établit une bijection de $\mathbb{N}\times\mathbb{N}$ dans $\mathbb{N}$ (les couples d'entiers naturels sont représentés en bleu et les entiers associés en blanc sur fond noir).

On a par exemple, en notant *f* la fonction : $f(1,1)=4$ et $f(1,3)=13$.



Ecrire une fonction récursive qui permet de calculer $f(x,y)$ pour tout couple (x,y) d'entiers naturels.

Graphiquement parlant, l'observation des segments rouges nous conduit à distinguer deux situations principales selon que *y* est nul ou pas :

- Si $y=0$ (x étant non nul), le couple qui précède le couple $(x,0)$ est le couple $(0,x-1)$. Dans ce cas, on aura donc : $f(x,0)=f(0,x-1)+1$.
- Si $y\neq 0$ (c'est le cas général), le couple qui précède le couple (x,y) est le couple $(x+1,y-1)$. Dans ce cas, on aura : $f(x,y)=f(x+1,y-1)+1$.

A partir du cas général, on a l'enchaînement :

$$(x,y)\to(x+1,y-1)\to(x+2,y-2)\to\ldots\to(x+y,0)\to(0,x+y-1)$$

Puis :

$$(0, x+y-1) \to (1, x+y-2) \to (2, x+y-3) \to \ldots \to (x+y-1, 0) \to (0, x+y-2)$$

En poursuivant de la sorte, on a les couples intermédiaires $(x+y-3, 0)$, $(x+y-4, 0)$, etc. On finit par aboutir au couple $(0, 0)$ qui correspond à notre cas de base, nécessaire pour programmer notre fonction récursive que nous baptisons simplement `num` :

```
def num(x,y):
     if x == 0 and y == 0:
          return 0
     elif y == 0:
          return 1 + num(0,x-1)
     else:
          return 1 + num(x+1,y-1)
```

A partir de la démarche ci-dessus, on peut donner une définition explicite de la fonction *f*.
En effet, l'enchaînement :

$$(x, y) \to (x+1, y-1) \to (x+2, y-2) \to \ldots \to (x+y, 0) \to (0, x+y-1)$$

nous donne : $f(x, y) = f(x+y, 0) + y$ et $f(x+y, 0) = f(0, x+y-1) + 1$ d'où :

$$f(x, y) = f(0, x+y-1) + y + 1$$

D'où :

$$\begin{aligned} f(x, y) &= f(0, x+y-1) + y + 1 \\ &= f(x+y-1, 0) + (x+y-1) + y + 1 \\ &= f(0, x+y-2) + (x+y) + y + 1 \\ &= f(x+y-2, 0) + (x+y-2) + (x+y) + y + 1 \\ &= f(0, x+y-3) + (x+y-1) + (x+y) + y + 1 \\ &= \ldots \\ &= \underbrace{f(0, 0)}_{=0} + 2 + 3 + \ldots + (x+y-1) + (x+y) + y + 1 \\ &= 1 + 2 + 3 + \ldots + (x+y) + y \\ &= \frac{(x+y)(x+y+1)}{2} + y \end{aligned}$$

Finalement :

$$\boxed{\forall (x, y) \in \mathbb{N}^2, f(x, y) = \frac{(x+y)(x+y+1)}{2} + y}$$

Il reste à s'assurer que la fonction *f* est bien une bijection de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$ … ☺

Exercice 6 – Les tours ? C'est carré !

Ecrire une fonction Python permettant de placer `n` tours sur un échiquier `nxn` de sorte qu'aucune ne soit en prise avec une autre (le programme devra déterminer toutes les configurations possibles et on affichera chacune des configurations. ATTENTION ! Il y en a beaucoup ! Et d'ailleurs, vous savez très bien combien … ☺).

On peut adopter diverses approches. Le cœur du programme que je vous fournis sur votre page est la fonction récursive `Place` qui reçoit en argument un entier correspondant au nombre de tours restant à placer. Le premier appel de `Place` se fait donc avec la taille `n` de l'échiquier comme argument. Un appel courant de `Place` est de la forme `Place(N)`. On cherche alors à placer une tour dans la colonne n-N. Pour cela, on balaie toutes les cases `(i,n-N)` avec `i` variant de `0` à `n-1` et dès que l'une d'elle n'est pas en prise on y place une tour, on met ensuite à jour l'échiquier car cette nouvelle tour placée engendre des cases en prise et, enfin, on appelle récursivement `Place` avec `N-1` comme argument (si N est différent de 1 bien sûr…).

Avec cette approche, on manipule de nombreux tableaux `numpy`, tous de même dimension.

Pour finir, il y a bien sûr $n!$ solutions. (le problème revenant à déterminer toutes les permutations de $[\![1\,;n]\!]$).