

## CORRECTION DU TD IPT<sup>2</sup> N° 1: RÉVISIONS 1/2

### RAPPELS ÉLÉMENTAIRES DE PROGRAMMATION: STRUCTURES CONDITIONNELLES ET ITÉRATIVES, FONCTIONS - EXTRAITS DE CONCOURS

#### Structures conditionnelles et itératives de base

##### EXERCICE N°1: Nombres parfaits

###### ❶ Fonction parfait(n)

Listing 1:

```
1 ### Nombres parfaits ###
2 import numpy as np
3 def parfait(n):
4     sommediviseurs=1
5     for diviseur in range(2,n//2+1):#on balaie les diviseurs
6         testés de 2 à n//2+1
7         if n%diviseur==0:
8             sommediviseurs=sommediviseurs+diviseur
9     if n==sommediviseurs:
10        print u"Le nombre",n,u"est parfait!"
11    else:
12        print u"Le nombre",n,u"n'est pas parfait!"
13 n=0.0
14 while type(n)<>int or n<1:
15     n=input("Entrer un entier naturel supérieur ou égal à 1:")
16 parfait(n)
```

- ❷ Pour déterminer tous les nombres parfaits inférieurs à  $10^5$ , on peut dans un premier temps modifier la fonction `parfait` afin qu'elle renvoie un booléen suivant que le nombre testé est parfait ou pas, puis rédiger une fonction `nbparfaits(nb)` qui va les compter à l'aide d'une boucle; cela donne pour la fonction `parfait` modifiée:

Listing 2:

```
1 ### Nombres parfaits ###
2 import numpy as np
3 def parfait_modifiee(n):
4     sommediviseurs=1
```

```
5     for diviseur in range(2,n//2+1):#on balaie les diviseurs
6         testés de 2 à n//2+1
7         if n%diviseur==0:
8             sommediviseurs=sommediviseurs+diviseur
9     return n==sommediviseurs
```

et la fonction de comptage:

Listing 3:

```
1 def nbparfaits(nb):
2     total=0
3     for n in range(nb+1):
4         if parfait_modifiee(n):
5             total+=1
6     return total
```

##### EXERCICE N°2: Suite ordonnée des nombres à petits diviseurs

Listing 4:

```
1 def suiteordo(N):
2     listefinale=[]
3     for nombre in range(1,N+1,1):
4         p,q,r=0,0,0
5         while nombre%2**p==0:
6             p=p+1
7         while nombre%3**q==0:
8             q=q+1
9         while nombre%5**r==0:
10            r=r+1
11        if nombre==2**(p-1)*3**(q-1)*5**(r-1):
12            listefinale.append([nombre,(p-1,q-1,r-1)])
13    print(u"La liste ordonnée est: "), listefinale
14
15 N=0.0
16 while (type(N)<>int) or (N<0):
```

```
17 N=input(" Entrer un nombre entier positif : ")
18 suiteordo(N)
```

### EXERCICE N°3: Bugs Bunny dans sa cage

- ❶ Fonction BordDroit(x,y) gérant le déplacement sur le bord droit:

Listing 5:

```
1 def BordDroit(x,y):
2     x=5
3     alea=randint(0,2)
4     if alea==0:
5         y=y+1
6     else:
7         y=y-1
8     N=N+1
```

- ❷ Simples changements de signe/valeur dans les relations d'affectations de x et y:

Listing 6:

```

1     y=5
2     if alea==0:
3         x=x-1
4     else:
5         x=x+1
6
```

• BordHaut(x,y) →

Listing 7:

```

1     x=1
2     if alea==0:
3         y=y-1
4     else:
5         y=y+1
6
```

• BordGauche(x,y) →

Listing 8:

```

1     y=1
2     if alea==0:
3         x=x-1
4     else:
5         x=x+1
6
```

• BordBas(x,y) →

- ❸ Procédure de gestion des déplacements intérieurs:

Listing 9:

```

1 def interieure(x,y):
2     alea=random(0,4)
3     if alea==0:
4         x=x+1
5         y=y+1
6     elif alea==1:
7         x=x+1
8         y=y-1
9     elif alea==2:
10        x=x-1
11        y=y-1
12    else:
13        x=x-1
14        y=y+1
15    N=N+1
```

- ❹ Programme principal:

Listing 10: Programme principal

```

1 import random
2
3 #Procédure d'atteinte d'une carotte
4 def test:
5     if (x==0 and y==0) or (x==6 and y==6) or (x==0 and y==6) or
6         (x==6 and y==0):
7         pasbougne=True
8
9 #Corps du programme principal
10 N=0
11 x,y=input(" entrer les coordonnées initiales du lapin sous forme
12            d'un tuple x0,y0: ")
13 pasbougne=False
14 while pasbougne!=True:
15     while (x!=0) or (x!=6) or (y!=0) or (y!=6):
16         interieure(x,y)
17     test
18     if pasbougne!=True:
19         if x==6:
20             BordDroit(x,y)
21         elif x==0:
```

```

22         BordGauche(x, y)
23     elif y==0:
24         BordBas(x, y)
25     else:
26         BordHaut(x, y)
27 print (u"Le nombre de déplacements effectués pour atteindre une
    des carottes est: ", N)

```

### ⑤ Modification pour un calcul de valeur moyenne de N:

On demande d'abord le nombre d'expériences nécessaires que l'on stocke dans la variable  $n$ , on initialise une variable  $somme$  à 0, puis on inclut la boucle `while` principale dans une boucle `for` assurant  $n$  itérations du programme complet. On renvoie enfin la valeur moyenne des  $n$  expériences menées:

Listing 11: Programme principal modifié

```

1 import random
2 N=0
3 x, y=3, 3
4 pasboug=False
5 def test:
6     if (x==0 and y==0) or (x==6 and y==6) or (x==0 and y==6) or
7       (x==6 and y==0):
8         pasboug=True
9 n=input (u"Combien d'expériences souhaitez-vous mener?: ", n)
10
11 #Corps du programme principal
12 somme=0
13 for k in range(0, n):
14     while pasboug != True:
15         while (x!=0) or (x!=6) or (y!=0) or (y!=6):
16             interieure(x, y)
17             test()
18             if pasboug != True:
19                 if x==6:
20                     BordDroit(x, y)
21                 elif x==0:
22                     BordGauche(x, y)
23                 elif y==0:
24                     BordBas(x, y)
25                 else:
26                     BordHaut(x, y)
27 somme=somme+N

```

```

28 print (u"la valeur moyenne du nombre de déplacements est: ", float
    (somme) / n)

```

## EXERCICE N°4: Résolution d'une énigme par force brute

- ① On propose 3 variantes pour la fonction `listecorrecte(L)`:

Listing 12:

```

1 def listecorrecte1(L):
2     if len(L)!=9:
3         return len(L)==9
4     else:
5         listeverif=[0]*9
6         for nb in L:
7             if nb>0 and nb
8               <10:
9                 listeverif[nb
10                  -1]=1
11         return not(0 in
12                    listeverif)

```

Listing 13:

```

1 import copy
2 def listecorrecte2(L):
3     L1=copy.deepcopy(L)
4     if len(L)!=9:
5         return len(L)==9
6     else:
7         L1.sort()
8         return L1
9     ==[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

Listing 14:

```

1 def listecorrecte3(L):
2     if len(L)!=9:
3         return len(L)==9
4     else:
5         listeconstr=[]
6         for nb in L:
7             if nb>0 and nb<10 and not(nb in listeconstr):
8                 listeconstr.append(nb)
9         return listeconstr==L

```

- ② On doit préalablement charger les modules `sympy` et `sympy.solvers`;

Il faut ensuite déclarer les noms de variables comme des symboles afin que python ne tente pas de renvoyer leur valeur. Enfin on lance la résolution pour obtenir le set d'équations définissant les expressions de  $A, B, C, D$ . Cela donne:

Listing 15:

```

1 from sympy import symbols
2 from sympy.solvers import solve
3 # création des noms des symboles

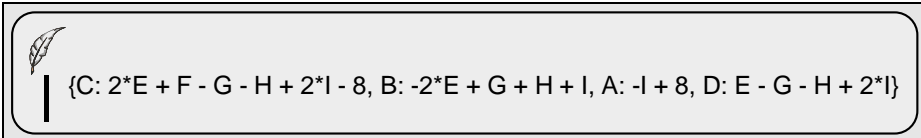
```

```

4 A,B,C,D,E,F,G,H,I=symbols ("A:I")
5
6 # Résolution du système d'équations
7 print solve ([A+B+C-2*I-F,D+E+F-A-C-I,G+H+I-2*E-B,A+I-8],A,B,C,D
  )

```

On obtient le résultat suivant:



{C: 2\*E + F - G - H + 2\*I - 8, B: -2\*E + G + H + I, A: -I + 8, D: E - G - H + 2\*I}

③ On propose le code complété suivant:

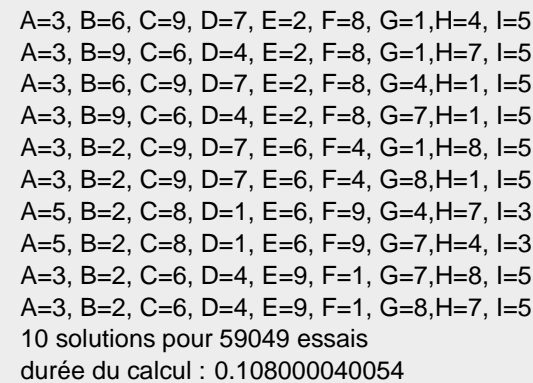
Listing 16:

```

1 import time as t
2 debut=t.time()
3 ##### Préparation de l'affichage des solutions #####
4 s="A={},B={},C={},D={},E={},F={},G={},H={},I="
5 nbessais = nbsol = 0
6 for E in range(1,10):
7     for F in range(1,10):
8         for G in range(1,10):
9             for H in range(1,10):
10                for I in range(1,10):
11                    nbessais+=1
12                    A=8-I
13                    B=-2*E+G+H+I
14                    C=2*E+F-G-H+2*I-8
15                    D=E-G-H+2*I
16                    if listecorrecte1 ([A,B,C,D,E,F,G,H,I]):
17                        nbsol+=1
18                        print(s.format(A,B,C,D,E,F,G,H,I))
19 print(' {} solutions pour {} essais'.format(nbsol,nbessais))
20 print(u"durée du calcul:", t.time()-debut

```

La sortie donne:



A=3, B=6, C=9, D=7, E=2, F=8, G=1, H=4, I=5  
 A=3, B=9, C=6, D=4, E=2, F=8, G=1, H=7, I=5  
 A=3, B=6, C=9, D=7, E=2, F=8, G=4, H=1, I=5  
 A=3, B=9, C=6, D=4, E=2, F=8, G=7, H=1, I=5  
 A=3, B=2, C=9, D=7, E=6, F=4, G=1, H=8, I=5  
 A=3, B=2, C=9, D=7, E=6, F=4, G=8, H=1, I=5  
 A=5, B=2, C=8, D=1, E=6, F=9, G=4, H=7, I=3  
 A=5, B=2, C=8, D=1, E=6, F=9, G=7, H=4, I=3  
 A=3, B=2, C=6, D=4, E=9, F=1, G=7, H=8, I=5  
 A=3, B=2, C=6, D=4, E=9, F=1, G=8, H=7, I=5  
 10 solutions pour 59049 essais  
 durée du calcul : 0.108000040054

## Manipulations de base sur les chaînes

### EXERCICE N°5: Recherche d'acides aminés dans une chaîne d'ADN

① Fonction valide(seq):

Listing 17:

```

1 def valide(seq):
2     ret=len(seq)!=0 #initialise le renvoi à True s'il y a une
3     séquence non nulle
4     for c in seq:
5         if not((c == 'a') or (c == 't') or (c == 'g') or (c ==
6         'c')):
7             ret=False
8     return ret

```

② Fonction de saisie valide:

Listing 18:

```

1 def saisie(chaine):
2     seq=""
3     while not valide(seq):
4         seq=input(u"Faire une saisie valide:")
5     return seq

```

③ Fonction proportion

Listing 19:

```
1 def proportion(chaine, sequence):
2     compteur=0
3     i=0
4     while i<len(chaine) and (i+len(sequence)<=len(chaine)): #
5         vérifie la longueur de chaine restante
6         if chaine[i:i+len(sequence)]==sequence:
7             compteur=compteur+1
8             i=i+len(sequence)
9         else:
10            i=i+1
11    return 100*compteur/(len(chaine)-len(chaine)%len(sequence)), compteur
```

Enfin le programme principal

Listing 20:

```
1 chaine=saisie(u"introduire la chaine d'ADN")
2 sequence=saisie(u"introduire la sequence")
3 print u"la proportion (en %) et le nombre d'occurrences de la
4     sequence dans la chaine d'ADN sont :", proportion(chaine,
5     sequence)
```

### EXERCICE N°6:

#### Recherche dans un texte

- ① On propose la fonction `caracmaj(c)` suivante, précédée d'une boucle `while` destinée à trouver l'indice de la première majuscule "A" dans le tableau ASCII (utile pour la suite):

Listing 21:

```
1 def caracmaj(c):
2     if ord(c) in range(ord("A"), ord("A")+26):
3         return c
4     else:
5         return chr(0)
```

La fonction `caracmaj(c)` vérifie simplement si le code ASCII du caractère `c` se trouve dans l'intervalle des codes correspondant aux majuscules de A à Z, et le cas échéant renvoie le caractère, ou bien ne renvoie rien du tout (ie le code ASCII 0 par la commande `chr(0)`) dans le cas contraire.

Autre proposition:

Listing 22:

```
1 def caracmaj(c):
2     inf, sup=ord("A"), ord("Z")
3     if (ord(c)>=inf) and (ord(c)<=sup):
4         return c
5     else:
6         return chr(0)
```

Listing 23:

```
2 1 def compte(s, c):
2     n=0
3     for car in s:
4         if car==c:
5             n+=1
6     return n
```

- ③ On peut en effet exploiter la fonction `compte(s, c)` dans la fonction `nb_lettres(s)` qui recense la fréquence de toutes les lettres majuscules de l'alphabet:

Listing 24:

```
1 def nb_lettres(s):
2     res=[]
3     for p in range(ord("A"), ord("A")+26):
4         res=res.append(compte(s, chr(p)))
5     return res
```

- ④ On constate que la chaîne `s` est parcourue à chaque appel de la fonction `compte(s, c)`, soit **26 fois au total**.
- ⑤ On propose la fonction optimisée `nb_lettres_opt(s)` suivante qui exploite avantageusement la fonction `caracmaj(c)` définie plus haut:

Listing 25:

```
1 def nb_lettres_opt(s):
2     res=[0 for p in range(26)]
3     for car in s:
4         if caracmaj(car) != chr(0):
5             res[ord(caracmaj(car))-ord("A")]+=1
6     return res
```

### EXERCICE N°7:

#### Découpage et recensement des mots dans un texte

- ❶ On propose la fonction suivante qui traite tous les cas possibles:

Listing 26: Sources\_Python/mot\_suivant.py

```
1 def mot_suivant(expression, i):
2     n=len(expression)
3     mot=""
4     if i>=n:
5         return "indice_i_incorrect"
6     else:
7         ind=i #initialisation indice courant
8         while ind!=n and expression[ind]!=' ': #tant que pas
9             bout de mot et pas en bout expression
10            mot=mot+expression[ind] # on ajoute le caractÃre
11            suivant au mot
12            ind+=1 #incrÃment de l'indice courant
13            if ind==n: #teste si seul le premier mot existe
14                return (mot,n) #alors on renvoie le mot et sa
15            longueur
16            else: #sinon
17                while ind<n and expression[ind]!=' ': #tant qu'on n
18                    'est pas en fin de liste et au prochain mot
19                    ind+=1 #on avance
20                return (mot,ind)
21 ch="La nuit est si belle"
22 print(mot_suivant(ch,9))
```

- ❷ mots exploite évidemment la fonction mot\_suivant. On notera la structure conditionnelle en ligne 7 qui permet d'éviter l'inclusion d'un mot vide dans l'hypothèse d'un espace en tête de expression:

Listing 27: Sources\_Python/mots.py

```
1 def mots(expression):
2     n=len(expression)
3     k=0
4     listemots=[]
5     while k<n:
6         (suivant, ind)=mot_suivant(expression, k)
7         if suivant!=' ':
8             listemots+=[suivant]
9         k=ind
10    return listemots, "Nombre de mots:", len(listemots)
```

- ❸ On inclut cette fois le caractère d'espacement et tous les signes de ponctuation dans une liste, et l'on teste si le caractère analysé est présent dans cette liste, plutôt que de se limiter comme ci-dessus à tester s'il s'agit d'un simple espace:

Listing 28: Sources\_Python/mots\_ponctuation.py

```
1 s=[" ", ",", ";", ".", "!", "?"]
2 def mot_suivant(expression, i, s):
3     n=len(expression)
4     mot=""
5     if i>=n:
6         return "indice_i_incorrect"
7     else:
8         ind=i #initialisation indice courant
9         while ind!=n and expression[ind] not in s: #tant que
10            pas bout de mot et pas en bout expression
11            mot=mot+expression[ind] # on ajoute le caractÃre
12            suivant au mot
13            ind+=1 #incrÃment de l'indice courant
14            if ind==n: #teste si seul le premier mot existe
15                return (mot,n) #alors on renvoie le mot et sa
16            longueur
17            else: #sinon
18                while ind<n and expression[ind] in s: #tant qu'on n
19                    'est pas en fin de liste et au prochain mot
20                    ind+=1 #on avance
21                return (mot,ind)
```

## Manipulations de base sur les listes

### EXERCICE N°8:

### Recherche de répétitions dans une liste

- ❶ On propose l'implémentation suivante en Python:

Listing 29:

```
1 def nombreZeros(t, i):
2     if t[i]==1:
3         return 0
4     else:
5         nb0=0
6         k=i
7         while k<len(t) and t[k]==0:
8             nb0+=1
9             k+=1
```

```
10 |         return nb0
```

On peut aussi proposer une version récursive (non terminale):

Listing 30:

```
1 | def nombreZerosRec(t, i):
2 |     if i > len(t)-1 or t[i] == 1:
3 |         return 0
4 |     else:
5 |         return 1 + nombreZerosRec(t, i+1)
```

## ② Méthode exploitant la fonction nombreZeros(t, i):

- on initialise un compteur comp à 0
- on initialise un indice  $i$  à 0
- on lance une boucle conditionnelle while qui se poursuit tant que  $i < \text{len}(t)-1$ , et qui vérifie si `nombreZeros(t, i)` renvoie une valeur supérieure à comp et remplace comp par cette valeur le cas échéant; enfin si il y a des 0 contigus, on incrémente  $i$  de la valeur retournée par `nombreZeros(t, i)` sinon on incrémente simplement de 1, puis la boucle se poursuit.

On propose l'implémentation suivante très naïve:

Listing 31:

```
1 | def nombreZerosMax(t):
2 |     comp=0
3 |     i=0
4 |     while i < len(t):
5 |         if nombreZeros(t, i) > comp:
6 |             comp=nombreZeros(t, i)
7 |         if nombreZeros(t, i) > 1:
8 |             i=i+nombreZeros(t, i)
9 |         else:
10 |             i+=1
11 |     return comp
```

et une version légèrement optimisée qui ne fait appel qu'une seule fois à `nombreZeros(t, i)`:

Listing 32:

```
1 | def nombreZerosMax(t):
2 |     comp=0
3 |     i=0
4 |     while i < len(t):
```

```
5 |         nombreZeroscontigus=nombreZeros(t, i)
6 |         if nombreZeroscontigus > comp:
7 |             comp=nombreZeroscontigus
8 |         if nombreZeroscontigus > 1:
9 |             i=i+nombreZeroscontigus
10 |        else:
11 |            i+=1
12 |        return comp
```

## Procédés aléatoires

### EXERCICE N°9:

### Méthodes de Monte-Carlo

- On entre une valeur de précision de calcul
  - On initialise les compteurs Nint et N
  - Tant que la précision n'est pas atteinte sur l'évaluation numérique de  $\pi$ , on itère la suite:
    - tirage des coordonnées  $x$  et  $y$  d'un point  $M$  comprise entre 0 et 1.
    - si le point  $M$  tombe dans un cercle de centre  $C(0.5, 0.5)$  et de rayon  $R = 0.5$  on incrémente le compteur intérieur  $N_{int}$ , ainsi que le compteur total  $N$ .
    - sinon (il tombe forcément dans le carré de côté 1 et de centre  $C$ ), on incrémente seulement le compteur total  $N$
    - A chaque itération, on évalue une valeur expérimentale de  $\pi$  stockée dans `resexp` en posant que le rapport  $N/N_{int}$  tend vers le rapport des surfaces  $S_{cercle}/S_{carré}$ ; la relation donnée correspond bien à l'évaluation de  $\pi$  (à faire à la main!)
- Version 3D de cet algorithme:

Listing 33:

```
1 | from random import *
2 | import numpy as np
3 | pi=np.pi
4 | erreur=0
5 | while not (erreur >= 1e-9):
6 |     erreur=input("Entrer la précision désirée pour
7 |     ce calcul (e>1E-9): ")
8 |     Nint=0
9 |     N=0
10 |    resexp=0
11 |    while abs(pi-resexp) > erreur:
12 |        x=random()
        y=random()
```

```

13         z=random()
14         if np.sqrt((x-0.5)**2+(y-0.5)**2+(z-0.5)**2)
15             <=0.5:
16                 Nint=Nint+1
17                 N=N+1
18                 resexp=3*Nint/(4*0.5**3*N)
19             else:
20                 N=N+1
21                 resexp=3*Nint/(4*0.5**3*N)
22                 print(resexp)
23
24         print(u"La valeur approchée recherchée est : ",
25               resexp)
26

```

### EXERCICE N°10: Le paradoxe des anniversaires

- ❶ On recherche la probabilité qu'au moins deux personnes parmi les  $N$  convives soient nées le même jour, soit le complément à 1 que toutes les personnes soient nées un jour différent dont la probabilité se calcule facilement:
- nombre total de possibilités de jour de naissance pour  $N$  convives: pour le premier: 365, pour le second: 365 etc... soit:  $\Omega = 365^N$
  - nombre de possibilités que les  $N$  convives soient tous nés un jour différent: pour le premier: 365, pour le second 364, pour le  $N^{ième}$  et dernier  $365 - N + 1$  soit  $\Omega_{\neq} = 365 \times 364 \times \dots \times (365 - N + 1) = \frac{365!}{(365 - N)!}$

La probabilité recherchée est donc:

$$P_{\text{même}} = 1 - P_{\neq} = 1 - \frac{\Omega_{\neq}}{\Omega} = 1 - \frac{365!}{(365 - N)! \times 365^N}$$

- ❷ Script Python du calcul de probabilité:  
Si l'on tente le calcul à partir de la relation obtenue ci-dessus, c'est à dire en écrivant:

Listing 34:

```

1 def fact(n):
2     res=1

```

```

3     while n>0:
4         res=res*n
5         n-=1
6     return res
7
8 def para_anniversaires(N):
9     return 100*(1-float(fact(365))/float(fact(365-N)*365**N))

```

Lorsque l'on exécute ce script par exemple pour  $N = 100$ , Python renvoie alors un message d'erreur signalant qu'il est incapable de convertir des entiers si longs en flottants:



Traceback (most recent call last):  
return 100\*(1-float(fact(365))/float(fact(365-N)\*365\*\*N))  
OverflowError: long int too large to convert to float

On peut alors procéder en formulant le calcul demandé par un procédé itératif évitant notamment de passer par le calcul d'une fraction à grands nombres:

$$P_{\neq} = \frac{(365 - N + 1)}{365^N} = \prod_{i=1}^{N-1} \frac{365 - i}{365}$$

Listing 35:

```

1 def para_anniversaires_bis(N):
2     p=float(1) #on peut aussi écrire p=1. pour la conversion en
3     flottant
4     for i in range(1,N):
5         p=p*(365-i)/365
6     return 1-p

```

Pour  $N = 100$  soit la centaine d'invités, ce script renvoie 0,999999692751, résultat plutôt différent de ce que dicte l'intuition, d'où l'appellation de *paradoxe des anniversaires*.

## Arithmétique

### EXERCICE N°11: Structure du code INSEE

On propose le code suivant qui manipule de bout en bout le code INSEE comme un nombre entier long:



Listing 36:

```
1 import math as m
2 INSEE=1.0
3 while not (type(INSEE)==long and int(1+m.floor(m.log10(INSEE)))==13
    and (int(m.floor(INSEE/1E12))==1 or int(m.floor(INSEE/1E12))
    ==2)):
4     INSEE=input(u"Entrer un numéro INSEE à 13 chiffres:")
5 ### Calcul de la clé ###
6 print(u"La clé du numéro INSEE est: "), int(97-INSEE%97)
```

On propose également la variante suivante, qui cette fois convertit le code INSEE en chaîne de caractère pour en analyser la structure:

Listing 37:

```
1 INSEE=1.0
2 while not (type(INSEE)==str and len(INSEE)==13 and (INSEE[0]=="1" or
    INSEE[0]=="2")):
3     INSEE=str(input(u"Entrer un numéro INSEE à 13 chiffres:"))
4
5 ### Calcul de la clé ###
6 INSEE=long(INSEE)
7 print(u"La clé du numéro INSEE est: "), int(97-INSEE%97)
```

**NB:** on peut facilement trouver la relation de calcul de la clé en remarquant qu'il existe un entier strictement positif  $n$  tel que  $A + K = n \times 97$ ; par conséquent, on a puisque la clé du code INSEE est strictement inférieure à 97:

$$K = (n \times 97 - A) \% 97$$

soit enfin:

$$K = 97 - A \% 97$$

### EXERCICE N°12:

### Vérification des codes barres

- 1 D'après la définition donnée, la clé qui est le chiffre  $a_{13}$  (le dernier!) du code barre correspond simplement au complément à 10 du nombre défini par:

$$3 \sum_{k=1}^6 a_{2k} + \sum_{k=0}^5 a_{2k+1}$$

ainsi:

$$cle = \left( 10 - \left( 3 \sum_{k=1}^6 a_{2k} + \sum_{k=0}^5 a_{2k+1} \right) \% 10 \right)$$

On propose un premier script manipulant le code barre en tant que nombre:

Listing 38:

```
1 import math as m
2 CODE=1.0
3 while not (type(CODE)==long and int(1+m.floor(m.log10(CODE)))
    ==12):
4     CODE=long(input(u"Entrer les douze chiffres 'produit' du
    code barre:"))
5 CODEp=CODE
6 i=1
7 cp=0
8 ci=0
9 while CODEp>0:
10     r=CODEp%10
11     CODEp=CODEp//10
12     if i%2==0:
13         cp=cp+r
14         print("somme paire au rang", i, ":", cp)
15     else:
16         ci=ci+r
17         print("somme impaire au rang", i, ":", ci)
18     i+=1
19 cle=(10-(3*cp+ci)%10)
20 print(u"La clé du code barre est: "), cle
21 print(u"Le code barre est: ", long(cle*1E12+CODE))
```

et un second manipulant les digits du code barre sous forme de chaîne de caractères:

Listing 39:

```
1 while not (type(CODE)==long and int(1+m.floor(m.log10(CODE)))
    ==12):
2     CODE=long(input(u"Entrer les douze chiffres 'produit' du
    code barre:"))
3 CODEp=str(CODE)
4 cp=0
5 ci=0
6 for i in range(11, -1, -1):
7     if i%2==0:
8         cp=cp+int(CODEp[i])
```

```

9     else :
10         ci=ci+int(CODEp[i])
11     cle=(10-(3*cp+ci)%10)
12     print(u"La clé du code barre est :", cle)
13     print(u"Le code barre est :", long(cle*1E12+CODE))

```

## ② Script de vérification d'un code barre:

Listing 40:

```

1 import math as m
2 CODE=1.0
3 while not(type(CODE)==long and int(1+m.floor(m.log10(CODE)))
4           ==13):
5     CODE=long(input(u"Entrer les treize chiffres du code barre :
6                  "))
7 CODEp=CODE
8 i=1
9 cp=0
10 ci=0
11 while CODEp>0:
12     r=CODEp%10
13     CODEp=CODEp//10
14     if i%2==0:
15         cp=cp+r
16         print("somme paire au rang", i, ":", cp)
17     else:
18         ci=ci+r
19         print("somme impaire au rang", i, ":", ci)
20     i+=1
21 if (3*cp+ci)%10==0:
22     print(u"Le code barre est valide")
23 else:
24     print(u"Le code barre est invalide")

```

## Extrait de concours

### EXERCICE N°13: Marche auto-évitante (d'après CCMP 2021)

- ① Cette question est simple: on implémente par exemple une liste contenant les positions des 4 points voisins de  $p$  sur la grille (attention: les déplacements en diagonale sont interdits puisque la distance entre deux points consécutifs est obligatoirement de 1), puis on vérifie pour chacun d'entre eux s'il n'est pas déjà dans la liste `atteints` avant de l'ajouter en queue de la liste `possibles`:

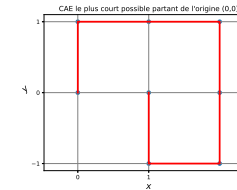
Listing 41: Sources\_Python/Positions\_possibles.py

```

1 def positions_possibles(p, atteints):
2     possibles=[]
3     P=[[p[0]+1,p[1]], [p[0]-1,p[1]], [p[0],p[1]+1], [p[0],p[1]-1]]
4     for pos in P:
5         if pos not in atteints:
6             possibles.append(pos)
7     return possibles

```

## ② Propositions de CAE le plus court possible:



Les autres CAE sont obtenus par rotation de  $\pi/2$  de ce chemin, puis symétrie par rapport à un plan (fixe) pour chacun d'entre-eux, soit 8 chemins au total.

## ③ On propose enfin le code suivant pour `genere_chemin_naif(n)`:

Listing 42: Sources\_Python/genere\_chemin\_naif.py

```

1 import random as rd
2 def genere_chemin_naif(n):
3     chemin=[[0,0]]
4     k=0
5     while k<n+1:
6         liste_points=positions_possibles(chemin[-1],chemin)
7         if liste_points==[]:
8             return None
9         else:
10            k+=1
11            chemin.append(rd.choice(liste_points))
12    return chemin

```