Corrigé

Introduction aux listes

Exercice 1. Parcours d'un tableau

On note $(a_0, a_1, \dots, a_{n-1})$ la liste `alt`.

a) La durée (en heures) de la randonnée est égale à la longueur de la liste, c'est-à-dire `len(alt)`.

b) Dans cette question on nous demande de calculer la valeur maximale de la liste `alt`.

```
def altmax(alt):
    m = 0
    for x in alt:
        if x > m:
            m = x
    return m
```

c) Nous devons maintenant calculer $\max\left\{a_k - a_{k-1} \;\middle|\; k \in [\![0, n-1]\!]\right\}$ avec la convention $a_{-1} = 0$.

```
def denivmax(alt):
    x, m = 0, 0
    for y in alt:
        if y - x > m:
            m = y - x
        x = y
    return m
```

Cette fonction utilise l'invariant : à l'entrée de la $(k+1)$-ième boucle, $x = a_{k-1}$ et $y = a_k$.

d) On procède de même pour déterminer la date à laquelle le dénivelé maximal a débuté, en utilisant l'instruction `enumerate` pour connaître l'indice correspondant au dénivelé maximal.

```
def denivmax2(alt):
    x, m = 0, 0
    for k, y in enumerate(alt):
        if y - x > m:
            m, t = y - x, k
        x = y
    return t
```

e) Dans cette question, il nous faut additionner les dénivelés positifs :

```
def denivtotal(alt):
    x, d = 0, 0
    for y in alt:
        if y - x > 0:
            d += y - x
        x = y
    return d
```

f) Enfin, pour obtenir la liste des sommets, nous allons partir d'une liste vide et ajouter à l'aide de la méthode `append()` les sommets au fur et à mesure de leur rencontre.

```
def sommets(alt):
    s = []
    x, y = 0, alt[0]
    for z in alt[2:]:
        if x < y and z < y:
            s.append(y)
        x, y = y, z
    return s
```

Cette fonction utilise l'invariant : à l'entrée de la $(k+1)$-ième boucle, $x = a_{k-1}$, $y = a_k$, $z = a_{k+1}$.

Exercice 2. Plus grand plateau

```
def plateau(tab):
    m = c = 0
    for x in tab:
        if x == 1:
            c = 0
        else:
            c += 1
        m = max(m, c)
    return m
```

À l'entrée de la $(k+1)$-ième boucle m est égal à la longueur du plus grand plateau de $[a_0,\dots,a_{k-1}]$ et c à la longueur du plus grand plateau se terminant par a_{k-1}.

Exercice 3. Nombre moyen d'éléments absents

Il est possible de définir le tableau demandé par compréhension :

```
from numpy.random import randint

tab = [randint(100) for k in range(100)]
```

mais si vous avez lu l'aide en ligne de la fonction `randint` vous avez peut-être remarqué que celle-ci possède un troisième paramètre optionnel indiquant la taille d'un tableau qui sera rempli aléatoirement [1]. Ceci nous permet de définir plus simplement :

```
tab =  randint(100, size=100)
```

On parcourt la liste des entiers de 0 à n en comptant ceux qui ne sont pas présents dans la liste :

```
def non_present(t):
    s = 0
    for k in range(100):
        if k not in t:
            s += 1
    return s
```

Il reste à définir une fonction pour réaliser un nombre x d'expériences :

```
def experience(x):
    s = 0
    for k in range(x):
        s += non_present(randint(100, size=100))
    return s / x
```

Exercice 4. In and Out shuffle

Si `lst` est une liste de longueur n, alors :

- `lst[:n//2]` décrit la première moitié de la liste ;
- `lst[n//2:]` décrit la seconde moitié de la liste ;
- `lst[::2]` décrit la liste des éléments de rangs pairs ;
- `lst[1::2]` décrit la liste des éléments de rangs impairs.

Une fois ceci rappelé, la définition des fonctions `out_shuffle` et `in_shuffle` devient évidente :

```
def out_shuffle(lst):
    n = len(lst)
    lst[::2], lst[1::2] = lst[:n//2], lst[n//2:]

def in_shuffle(lst):
    n = len(lst)
    lst[::2], lst[1::2] = lst[n//2:], lst[:n//2]
```

1. En réalité, le tableau créé n'est pas de type *list* mais de type *numpy.ndarray*, mais pour l'usage que nous allons en faire il n'y a pas de différence.

Notons que ces deux fonctions modifient la liste passée en paramètre et retournent la valeur `None`. Pour calculer la longueur du cycle il va falloir garder copie de la liste initiale :

```
def cycle(n):
    init = [i for i in range(n)]
    lst = init.copy()
    out_shuffle(lst)
    s = 1
    while lst != init:
        out_shuffle(lst)
        s += 1
    return s
```

```
>>> cycle(52)
8
```

Pour le *In Shuffle* il faut 52 mélanges pour retrouver la liste initiale.

Exercice 5. Nombres premiers

La première méthode demande de commencer par rédiger une fonction testant la primalité d'un nombre. On utilise le critère : *p est un nombre premier s'il n'est divisible par aucun entier compris entre 2 et* $\sqrt{p}$.

```
def premier(p):
    if p < 2:
        return False
    k = 2
    while k * k <= p:
        if p % k == 0:
            return False
    return True
```

On est maintenant en mesure de calculer le nombre premier qui suit un entier donné :

```
def next_prime(n):
    p = n + 1
    while not premier(p):
        p += 1
    return p
```

On génère ensuite la liste demandée à l'aide du script :

```
p, sol = 2, []
while p < 1000:
    sol.append(p)
    p = next_prime(p)
```

La seconde méthode utilise le crible d'Ératosthène. On commence par écrire une fonction qui supprime tous les multiples d'un élément donné de celle-ci :

```
def elimination(lst, k):
    p = lst[k]
    for n in lst[k+1:]:
        if n % p == 0:
            lst.remove(n)
```

On génère ensuite la liste des nombres premiers à l'aide du script :

```
lst = list(range(2, 1001))
k = 0
while k < len(lst):
    elimination(lst, k)
    k += 1
```

La troisième méthode utilise la définition par compréhension d'une liste :

```
l1 = [n for n in range(2, 1001)]
l2 = [(n, [d for d in range(1, n+1) if n % d == 0]) for n in l1]
l3 = [c[0] for c in l2 if len(c[1]) == 2]
```

Exercice 6. Amnistie à la prison de Sikinia

On choisit de représenter les cellules par une liste de booléens traduisant l'ouverture ou non de la porte. Au début de l'histoire toutes les portes sont fermées dont on définit :

```
cellules = [False for k in range(100)]
```

On suit les directives du président (on fera attention au fait que les éléments d'une liste Python sont indexées de 0 à 99 alors que dans l'énoncé les cellules sont indexées entre 1 et 100) :

```
for i in range(100):
    for j in range(i, 100, i+1):
        cellules[j] = not cellules[j]
```

Une fois le processus achevé, il reste à passer en revue les différentes cellules en notant celles qui sont ouvertes :

```
ouvertes = []
for i in range(100):
    if cellules[i]:
        ouvertes.append(i+1)
print(ouvertes)
```

On obtient la liste : `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]` qui laisse présager que dans le cas général ce sont les cellules dont le numéro est un carré parfait qui sont ouvertes.

Exercice 7. Permutations de Josephus

Une solution possible ; la méthode `pop(k)` retourne et supprime dans le même temps un élément décrit par son indice dans une liste.

```
def josephus(n, m):
    lst = list(range(1, n+1))
    sol = []
    k = 0
    for i in range(n):
        k = (k - 1 + m) % len(lst)
        sol.append(lst.pop(k))
    return sol
```

Exercice 8. Génération de permutations

Le calcul repose sur l'observation suivante : les 9! premières permutations débutent par le chiffre 0, les 9! suivantes par 1, etc. Autrement dit, la n^e permutation commence par le chiffre $k = \left\lfloor \frac{n}{9!} \right\rfloor$.

On procède ensuite récursivement en déterminant la permutation de rang $(n - k \cdot 9!)$ de l'ensemble $\{0, 1, \ldots, 9\} \setminus \{k\}$.

```
def fact(n):
    s = 1
    for k in range(1, n+1):
        s *= k
    return s

def permutation(n):
    sol = []
    lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    for i in range(9, 0, -1):
        k = n // fact(i)
        n -= k * fact(i)
        sol.append(lst.pop(k))
    return sol
```

```
>>> permutation(999999)
[2, 7, 8, 3, 9, 1, 5, 4, 6, 0]
```

Exercice 9. Nombres de Hamming

On observe que si n est un nombre de Hamming différent de 1 alors n est le produit par 2, par 3 ou par 5 d'un nombre de Hamming plus petit. Partant de cette remarque, on utilise trois listes égales initialement à `[1]`. À chaque étape on supprime l'élément minimal n de ces trois listes et on ajoute dans la première $2n$, dans la seconde $3n$ et dans la troisième $5n$.

```
def hamming(n):
    l2, l3, l5 = [1], [1], [1]
    for k in range(n):
        x = min(l2[0], l3[0], l5[0])
        if x == l2[0]:
            del l2[0]
        if x == l3[0]:
            del l3[0]
        if x == l5[0]:
            del l5[0]
        l2.append(2 * x)
        l3.append(3 * x)
        l5.append(5 * x)
    return x
```

```
>>> hamming(2000)
8062156800
```