Algorithmique

I. Compléments

I.1. Contrôle de flux dans une boucle

Syntaxe : Les instructions `break`, `continue` et `else` dans une boucle

Si l'on souhaite arrêter l'exécution d'une boucle, on utilise l'instruction `break`.
Si l'on souhaite passer à l'itération suivante d'une boucle sans exécuter le reste du code de l'itération en cours, on utilise l'instruction `continue`.
Si l'on souhaite exécuter un bloc de commandes uniquement dans le cas où une boucle a été entièrement exécutée (c'est-à-dire uniquement dans le cas où elle n'a pas été interrompue par un `break`), on utilise l'instruction `else`.
Exemple :

```
for p in range(2,11):
    for i in range(2,int(p**0.5)+1):
        if p%i==0:
            print(p," est divisible par ",i,".",sep='')
            break # on arrete la boucle, p a un diviseur, il n'est pas premier
    else: # n'est execute que si on n'a trouve aucun diviseur
        print('\n',p,'est premier.')
```

2 est premier.
3 est premier. 4 est divisible par 2.
5 est premier. 6 est divisible par 2.
7 est premier. 8 est divisible par 2. 9 est divisible par 3. 10 est divisible par 2.

I.2. Exemples

Ex. 7.1 Écrire une fonction `estPremier(n)` renvoyant `True` si n est premier et `False` sinon.

Ex. 7.2 Écrire une fonction `listePremiers(a,b)` renvoyant la liste de tous les nombres premiers dans l'intervalle $[\![a; b-1]\!]$.

Ex. 7.3 Soit n un entier supérieur à 2 et $n = p_1^{\alpha_1} p_2^{\alpha_2} ... p_k^{\alpha_k}$ sa décomposition en facteur premiers.
On note $f(n) = \alpha_1^{p_1} \alpha_2^{p_2} ... \alpha_k^{p_k}$.
Écrire une fonction f prenant en paramètre un entier $n \geqslant 2$ et renvoyant la valeur de $f(n)$ ainsi définie.

Ex. 7.4 Écrire une fonction demandant à l'utilisateur d'entrer un entier supérieur ou égal à 2 - et recommençant si ce n'est pas le cas - puis demandant à l'utilisateur d'entrer un second entier positif - et recommençant si ce n'est pas le cas - et renvoyant ces deux entiers.

Ex. 7.5 Soit n un entier supérieur ou égal à 2. On définit la suite u par $u_0 = n$ et $u_{i+1} = f(u_i)$ où f est la fonction définie plus haut.
Écrire une fonction demandant à l'utilisateur de rentrer deux entiers $n \geqslant 2$ et $i \in \mathbb{N}$ et renvoyant la valeur de u_i.
Tester votre fonction.

I.3. Différences (subtiles) entre certaines syntaxes

a) append et concaténation de listes

Examiner le code suivant :

```
import time as t
L1=[]
L2=[]
print("Adresse de L1 avant la boucle : ",id(L1))
print("\nAdresse de L2 avant la boucle : ",id(L2))
T=t.clock()
for i in range(100000):
    L1.append(i)
print("\nTemps mis pour remplir la liste L1 :",t.clock()-T,"seconde")
print("\nAdresse de la liste L1 après la boucle :",id(L1))
T=t.clock()
for i in range(100000):
    L2=L2+[i]
print("\nTemps mis pour remplir la liste L2 :",t.clock()-T,"secondes...")
print("\nAdresse de la liste L2 après la boucle:",id(L2))
```

Voici ce que ce code affiche :
Adresse de L1 avant la boucle : 2392049668552
Adresse de L2 avant la boucle : 2392055074248
Temps mis pour remplir la liste L1 : 0.0184528 seconde
Adresse de la liste L1 après la boucle : 2392049668552
Temps mis pour remplir la liste L2 : 13.287863100000001 secondes...
Adresse de la liste L2 après la boucle : 2392055175688
Que peut-on déduire de ce qui précède concernant les différences entre `liste.append(element)` ***et*** `liste=liste+[element]` ***?***

b) Listes et tableaux numpy

Examiner le code suivant :

```
import numpy as np
L1=[1,2,3,5]
L2=L1[:]
L1[-1]=4
print("Adresse de L1 : ",id(L1))
print("\nValeur de la liste L1 :",L1)
print("\nAdresse de L2 : ",id(L2))
print("\nValeur de la liste L2 :",L2)
T1=np.array(L1)
T2=T1[:]
T3=np.copy(T1)
T1[-1]=0
print("\nAdresse de T1 : ",id(T1))
print("\nValeur du tableau T1 :",T1)
print("\nAdresse de T2 : ",id(T2))
print("\nValeur du tableau T2 :",T2)
```

```
print("\nAdresse de T3 : ",id(T3))
print("\nValeur du tableau T3 :",T3)
```

Voici ce que ce code affiche :
Adresse de L1 : 2392074862920
Valeur de la liste L1 : [1, 2, 3, 4]
Adresse de L2 : 2392049668552
Valeur de la liste L2 : [1, 2, 3, 5]
Adresse de T1 : 2392055199344
Valeur du tableau T1 : [1 2 3 0]
Adresse de T2 : 2392055223040
Valeur du tableau T2 : [1 2 3 0]
Adresse de T3 : 2392068387744
Valeur du tableau T3 : [1 2 3 4]
Que peut-on déduire de ce qui précède concernant les différences entre listes et tableaux `numpy` ***?***

c) Listes et tableaux `numpy` : temps de calcul

Examiner le code suivant :

```
import numpy as np
import time as t
liste=list(range(100000))
tableau=np.array(liste)
T=t.clock()
val1=[np.sin(k) for k in liste]
print("Temps mis pour remplir la liste val1 :",t.clock()-T,"seconde")
T=t.clock()
val2=np.sin(tableau)
print("\nTemps mis pour remplir la liste val2 :",t.clock()-T,"seconde")
print("\nLes deux tableaux de valeur sont-ils égaux ?")
print(val1==val2.tolist())
```

Voici ce que ce code affiche :
Temps mis pour remplir la liste val1 : 0.07860180000000128 seconde
Temps mis pour remplir la liste val2 : 0.0010910000000006193 seconde
Les deux tableaux de valeur sont-ils égaux ? True
Que peut-on déduire de ce qui précède concernant les différences entre listes et tableaux `numpy` ***?***