06_bis - Exercices Supplémentaires

November 29, 2023

Listes Python - Introduction & Opérations élémentaires Pour aller plus loin - Exercices supplémentaires

Adapté de C. Poulmaire / AEIF

Vous constaterez que les exercices ci-dessous font usage de la fonction `assert` pour effectuer ce que l'on appelle des tests d'assertion. Je vous laisse en découvrir la syntaxe et le fonctionnement... (N'hésitez pas à faire une recherche Google si quelque chose n'est pas clair!)

1 Exercice A : jeu de rôle

Dans cet exercice, toutes les données sont des nombres entiers strictement positifs.

Dans un jeu, un joueur dispose d'un certain nombre de points de vie et d'une défense pour résister aux attaques qu'il va subir lors d'un certain nombre de combats.

Ces attaques prennent des valeurs aléatoires entre deux nombres entiers prédéfinis par le jeu.

Les points de vie du joueur évoluent selon la règle suivante : - si la valeur de l'attaque est strictement supérieure à celle de la défense, le joueur perd `attaque - defense` points de vie ; - si la valeur de l'attaque est inférieure ou égale à celle de la défense, le joueur gagne `attaque//2` points de vie.

Si, à la fin d'un combat, le nombre de points de vie du joueur est inférieur ou égal à 0, alors il ne participe pas aux combats suivants s'il y en a encore.

La fonction `evolutionPointsVie` ci-dessous prend pour paramètre : - `pointsVie` qui représente le nombre de points de vie initial du joueur ; - `defense` qui représente la défense du joueur ; - `valMinAttaque` qui représente la valeur minimale que peut prendre une attaque ; - `valMaxAttaque` qui représente la valeur maximale que peut prendre une attaque ; - `nbCombats` qui représente le nombre de combats auxquels participe le joueur.

Cette fonction renvoie, sous la forme d'une liste, l'évolution du nombre de points de vie du joueur aux cours des combats.

Compléter la fonction.

[ ]:
```
from ...

def evolutionPointsVie(pointsVie,defense,valMinAttaque,valMaxAttaque,nbCombats):
    assert isinstance(pointsVie,int) and pointsVie > 0
```

```
    assert isinstance(defense,int) and defense > 0
    assert isinstance(valMinAttaque,int) and valMinAttaque > 0
    assert isinstance(valMaxAttaque,int) and valMaxAttaque > 0
    assert isinstance(nbCombats,int) and nbCombats > 0
    listePointsVie = [pointsVie]
    for i in range(...):
        attaque = randint(... , ...)
        if attaque > defense:
            pointsVie = ...
        else:
            pointsVie = ...
        listePointsVie.append(...)
        if pointsVie <= ... :
            return ...
    return ...
```

2 Exercice B : suite de Syracuse

On considère la suite (u_n) d'entiers naturels définie de la façon suivante: u_0 est un entier strictement positif quelconque et, pour tout $n \geq 0$: Si n est pair: $u_{n+1} = \frac{u_n}{2}$; Si n est impair: $u_{n+1} = 3.u_n + 1$ Une conjecture, appelée *conjecture de Syracuse* et non encore démontrée à ce jour, affirme qu'il existe un entier $N \geq 0$ tel que $u_N = 1$.

Le plus petit entier N pour lequel $u_N = 1$ est appelé le *temps de vol de* u_0.

L'ensemble des valeurs prises par la suite (u_n) depuis u_0 jusqu'à $u_N = 1$ est appelé la *trajectoire de vol de* u_0.

Exemple : - si $u_0 = 1$, alors $N = 0$: le temps de vol de u_0 est 0 et sa trajectoire est $[1]$: - si $u_0 = 2$, alors $u = 1 = 1$; d'où $N = 1$: le temps de vol de u_1 est 1 et sa trajectoire est $[2, 1]$; - si $u_0 = 4$, alors $u_1 = 2$ et $u_2 = 1$; d'où $N = 2$: le temps de vol de u_2 est 2 et sa trajectoire est $[4, 2, 1]$.

La fonction `trajectoireVolSyracuse` ci-dessous prend en paramètre un entier `val` correspondant à la valeur initiale u_0 et renvoie la trajectoire de cet entier sous la forme d'une liste, ainsi que son temps de vol.

Compléter cette fonction.

[ ]:
```
def trajectoireVolSyracuse(val):
    assert ... # test d'assertion sur val
    trajectoireVol = [val]
    while ... :
        if val%2 == 0:
            val = ...
        else:
            val = ...
        trajectoireVol.append(...)
    return ... , ...
```

Les cinq cellules suivantes permettent de tester votre fonction. Elles doivent renvoyer `True` lors de leur exécution.

[ ]:
```
trajectoireVolSyracuse(1) == ([1],0)
```

[ ]:
```
trajectoireVolSyracuse(2) == ([2,1],1)
```

[ ]:
```
trajectoireVolSyracuse(4) == ([4,2,1],2)
```

[ ]:
```
trajectoireVolSyracuse(37) == ([37, 112, 56, 28, 14, 7, 22, 11, 34, 17, 52, 26,␣
↪13, 40, 20, 10, 5, 16, 8, 4, 2, 1],21)
```

[ ]:
```
trajectoireVolSyracuse(1024) == ([1024,512,256,128,64,32,16,8,4,2,1],10)
```

3 Exercice C : liste aléatoire à valeurs distinctes

La fonction ci-dessous renvoie une liste de `n` valeurs entières aléatoires distinctes comprises entre deux entiers `a` et `b` avec `b-a+1 >= n`.

Compléter cette fonction.

[ ]:
```
from random import randint

def listeAleatoireDistinct(n,a,b):
    assert ... # test d'assertion sur n
    assert ... # test d'assertion sur a
    assert ... # test d'assertion sur b
    assert b-a+1 >= n
    L = []
    while len(L) ... :
        val = ...
        if val ... :
            L.append(...)
    return L
```

4 Exercice D : tri d'une liste

Pour trier par ordre croissant une liste numérique non vide `L`, on procède de la manière suivante : - on crée une copie `liste` de la liste `L` et une liste vide `listeTriee` qui contiendra la liste `L` triée à la fin de l'algorithme ; - on recherche l'indice `indiceMin` de la première occurence du minimum dans la liste `liste`; - on supprime l'élément d'indice `indiceMin` dans la liste `liste` et on l'ajoute à la fin de la liste `listeTriee` ; - on recommence à l'étape 2 jusqu'à ce que la liste `liste` soit vide.

1. Compléter ci-dessous la fonction `rechercheIndiceMin` qui prend en paramètre une liste numérique non vide `L` et qui renvoie l'indice de la première occurence du minimum de `L`. *On ne fera pas de tests d'assertion.*

[ ]:
```
def rechercheIndiceMin(L):
    minimum = L[0]
    indiceMin = 0
    for i in ... :
        if L[i] < ...:
            minimum = ...
            indiceMin = ...
    return ...
```

Les trois cellules suivantes permettent de tester votre fonction. Elles doivent renvoyer `True` lors de leur exécution.

[ ]:
```
rechercheIndiceMin([1,4,5,3,2,4]) == 0
```

[ ]:
```
rechercheIndiceMin([5,4,7,5,9.4,2]) == 5
```

[ ]:
```
rechercheIndiceMin([4,2,4,3,5,7,8,2,2,2,4,5]) == 1
```

2. En utilisant cette fonction, compléter la fonction `triListe` ci-dessous qui prend en paramètre une liste numérique non vide `L` et renvoie une liste contenant les éléments de `L` triés par ordre croissant à l'aide de la méthode décrite au début de l'exercice. On fera des tests d'assertion pour vérifier que `L` satisfait bien aux conditions requises.

[ ]:
```
def triListe(L):
    assert isinstance(L,...) and L != ...
    for elem in L:
        assert ...
    liste = ... # création de la liste liste
    listeTriee = ... # création de la liste listeTriee
    while liste ... :
        indiceMin = ...
        valMin = ...
        listeTriee.append(...)
    return ...
```

Les trois cellules suivantes permettent de tester votre fonction. Elles doivent renvoyer `True` lors de leur exécution.

[ ]:
```
triListe([1,4,5,3,2,4]) == [1,2,3,4,4,5]
```

[ ]:
```
triListe([5,4,7,5,9.4,2]) == [2,4,5,5,7,9.4]
```

[ ]:
```
triListe([4,2,4,3,5,7,8,2,2,2,4,5]) == [2,2,2,2,3,4,4,4,5,5,7,8]
```