Le grand jeu de la vie

Dans « Jurassic Park », Jeff Goldblum disait que la vie trouve toujours un chemin pour se développer. Un jeu informatique de simulation serait bienvenu pour tester cette affirmation. Armez-vous d'un Python ;-) (jeu de mots à double détente : les puristes apprécieront) et faites confiance à FREELOG pour jouer au jeu de la vie.

La trame de l'aventure

Aujourd'hui, nous allons voir comment gérer des tableaux a 2 dimensions avec python. Mais, me direz-vous, que sont les tableaux à 2 dimensions ? En règle générale, on utilise, pour l'expliquer, l'analogie avec la bataille navale. Un tableur utilise aussi cette notion. Une cellule y est définie par 2 coordonnées. Ce principe sert aux jeux d'échecs, jeux de rôles sur une carte, aux jeux de labyrinthes et, mais oui, à certains jeux de simulation de cité urbaine... Pour faire mieux connaissance avec cette technique de base, nous allons voir un classique de la programmation qui s'y réfère : le jeu de la vie.

Tantalatan talalan

Le principe en est simple : on réalise un tableau de n par n cases, et on le remplit de façon aléatoire de 0 ou de 1. Le 1 désigne une cellule vivante, le 0, une cellule morte. Des règles d'évolution sont implémentées (c'est à dire incorporées dans le programme) : par exemple, si une cellule vivante est entourée par 2 cellules mortes, elle meurt. Si une cellule morte est entourée par 2 cellules vivantes, elle prend vie, etc... Ensuite, on fixe le nombre de générations successives à observer et on lance la simulation. A chaque étape vous verrez des cellules naître, vivre ou mourir suivant vos indications.

Case départ

Notre but est simple : nous allons nous fixer 4 règles. Celles-ci définiront le comportement de la machine face à la situation de départ générée de façon aléatoire. Nous ne respecterons pas ici les règles « officielles » du jeu de la vie standard, mais nous allons choisir les nôtres. Nous aurons donc un tableau de départ de 10*10 cases rempli de 0 et de 1. C'est la génération 0. A la génération 1, nous testerons chaque cellule du tableau pour voir si elle répond à certains critères que nous aurons définis (nous verrons que c'est le plus souvent dans une procédure spécifique qu'ils sont fixés). En fonction, nous déterminerons si ladite case doit comporter 0 ou 1, et quand toutes les cases auront reçu leur traitement, nous afficherons le résultat. Nous procéderons de même pour le nombre de générations que nous aurons fixé.

Double six

L'articulation sera classique : Tout d'abord les imports, ensuite les définitions, enfin le corps du programme. Le seul import que nous ayons à faire concerne whrandom qui assurera les tirages aléatoires. Nouveauté : cette fois, nous allons utiliser un dictionnaire. Il se déclare ainsi :
population ={}
De façon générale, il sert à stocker des informations (nous en reparlerons dans un prochain article). Python est un langage à typage dynamique. Autrement dit, nous pouvons stocker n'importe quoi dans notre dictionnaire, il n'y a pas besoin de déclaration préalable du type de la variable (entier, flottant, chaîne de caractères...). Dans ce cas précis, nous allons y stocker notre tableau. Apprécions aussi la souplesse du langage.

Passez un tour

Maintenant, intéressons-nous à notre fonction. Nous l'avons arbitrairement appelée voisinage. Nous savons qu'elle reçoit 2 paramètres, x et y qui définissent la cellule traitée. Dans le corps de notre procédure, nous regardons ce que contiennent les cellules x-1 et x+1 qui encadrent notre cellule x. Ensuite, nous examinons les 4 cas possibles (voir plus bas les cas détaillés). Ce qui nous donne la valeur finale de x que nous retournerons à l'appelant. Voici le code in extenso.

```
def voisinage(x,y):
  if(x>9):
     x=9
  if(x<2):
     x=2
  resultat=0
  transitionx=population[(x+1,y)]
  transitioninvx =population[(x-1,y)]
```

Juste un aparté à ce niveau pour vous signaler que c'est ici, que l'on peut fixer les règles de vie de notre population avec des if. Voici les 4 règles que nous avions mentionné plus haut. Nous considérons les 4 cas ((0,0);(0,1);(1,0);(1,1)) d'où l'emploi de l'opérateur logique and (ET).

```
  if(transitionx==0 and transitioninvx==0):
     resultat=0
  if(transitionx==1 and transitioninvx==1):
     resultat=1
  if(transitionx==0 and transitioninvx==1):
     resultat=0
  if(transitionx==1 and transitioninvx==0):
     resultat=1
  return resultat
```

On retrouve la structure, classique maintenant, du si, alors. Dans cette procédure, une astuce est à mentionner qui figure dans les 4 premières lignes. Nous avons dit que notre tableau faisait 10*10 cases. Si nous transmettons comme paramètre, la case (10,3); nous allons chercher la valeur de la case x+1, soit 10+1 (donc 11 ;-))) qui est à l'extérieur du tableau déclaré. Par conséquent, nous aurons un message d'erreur. Nous limitons par cette astuce la valeur de x à 9, pour que la valeur maximale testée soit 10. Notez la syntaxe de définition des cellules à l'intérieur du dictionnaire population. Nous y reviendrons plus tard, mais il est important de remarquer ici que population ne fait l'objet d'aucune déclaration et qu'il est pourtant reconnu. Ceci vient du fait, qu'ayant été déclaré dans l'en-tête, il est une variable globale dont la portée est reconnue en tout point du programme. Notre fonction peut y faire référence sans problème.

Case départ : encaissez 20 000f

Nous allons créer le tableau initial et le remplir de 0. Nous utilisons les sempiternelles variables i et j que nous faisons varier de 1 à 11 (hé oui, toujours limite_sup + 1). Et voilà la syntaxe de création de population, le tableau à deux dimensions, objet de notre étude. Pour chaque case, nous mettons la valeur à 0 pour initialiser le tableau.

```
for i in range(1,11):
  for j in range(1,11):
     population[(i,j)]=0
```

Tirez une carte chance

Bien entendu, maintenant, nous allons mettre dans ce tableau entièrement à 0, des valeurs 1 avec des coordonnées x et y déterminées de façon aléatoire. Nous avons choisi de mettre la moitié des cellules à 1. Nous devons donc générer 50 valeurs. Comme cous pouvez le voir dans ce qui suit, il n'y a aucune difficulté à assimiler le code. Nous multiplions random() par 10 pour obtenir une valeur finale comprise entre 1 et 10, et dans le tableau population, nous affectons les valeurs ainsi générées à x et y pour indiquer que la cellule correspondante est égale à 1.

```
for depart in range(1,51):
  x=int(random()*10)
  y=int(random()*10)
  population[(x,y)]=1
```

Nous affichons le tableau initial pour suivre l'évolution de nos petits 1.

```
for i in range(1,11):
  print
  for j in range(1,11):
     print population[(i,j)],
print '\n voici le résultat initial'
```

Construisez pour 10 générations

Notre programme est terminé. Grace à la fonction voisinage, nous allons faire évoluer la population initiale que nous venons de générer. Nous avons fixé un seuil de 10 générations. Nous créons donc un grande boucle de 10 unités (libre à vous d'en faire 5 seulement) avec la variable boucle, dans laquelle nous allons afficher chaque tableau (avec les variables k et l).

```
for boucle in range(1,11):
  for k in range(1,11):
     print
     for l in range(1,11):
```

Ici, on teste l'environnement de la position (k,l) avec la procédure voisinage pour déterminer teste_pos qui prend la valeur retournée par voisinage, et on l'affecte à la cellule (k,l) de population. Puis, on affiche.

```
        teste_pos=voisinage(k,l)
        population[(k,l)]=teste_pos
        print population[(k,l)],
  print '\n voici le résultat à la ', boucle, ' ieme generation'
```

En d'autres termes, nous avons créé une variable teste_pos qui reçoit la valeur de retour de voisinage(x,y) lorsqu'on lui passe le variables de position k et l.

Arrêt buffet

Voilà : nous venons de réaliser un programme de simulation (Messieurs, restez sérieux ;-)). Cette fois, nous n'avons pas respecté les vraies règles du jeu de la vie telles qu'elles ont été définies à l'origine. Mais nous comptons sur vous pour le faire. Ce programme en apparence anodin est en fait un « basique » : imaginez que vous choisissiez d'autres types de conditions et vous pourrez réaliser le jeu qui consiste à aligner 4 pions. Bien sûr, la qualité graphique brille par son absence, et vous devrez gérer 3 types de cases : les cases vides, celles occupées par vos pions, celles occupées par l'adversaire. Pour ceux qui sont plus tentés par la cryptographie, les tables de Vigenières sont des tableaux à deux dimensions comportant 26 alphabets. Envoyez nous vos réalisations, elles figureront en bonne place sur le CD.