Sujet 0 Capes option informatique

Problème 1

PARTIE A : représentation des L-systèmes

Le module turtle

```
# Attention, le module 'turtle' ne fonctionne pas bien avec Pyside :
# utiliser un shell qui utilise TK!!
from turtle import *
from matplotlib.pyplot import *
from math import *
```

Un alphabet pour coder les figures

1)

```
def maison(L=100):
    for angle in (90, 30, 120, 30):
        forward(L)
        left(angle)
    forward(L)
```

2) On peut décrire la maison ainsi : F(100, 30, "F+++F+F++++F+F").

3)

```
def dessiner(unite, angle, motif):
    for ch in motif:
        if ch == "F":
            forward(unite)
        elif ch == "+":
            left(angle)
        elif ch == "-":
            right(angle)
        else:
            return 1
    return 0
```

Pour dessiner la maison, il suffit d'exécuter la commande : `dessiner(100,30,"F+++F+F++++F+F")`.

Les L-systèmes

4)

```
def suivant(motif, regle):
    rep = ""
    for ch in motif:
        if ch == "F":
            rep += regle
        else:
            rep += ch
    return rep
```

On peut aussi le faire en 1 ligne à l'aide des fonctions `split` et `join` :

```
def suivant(motif, regle):
    return regle.join(motif.split("F"))
```

ou bien en trichant un peu, à l'aide de la fonction `replace` :

```
def suivant(motif, regle):
    return motif.replace("F", regle)
```

5)

```
def evolution(axiome, regle, etape):
    rep = axiome
    for _ in range(etape):
        rep = suivant(rep, regle)
    return rep
```

6) Si F_n est le nombre de 'F' à l'étape n alors $F_0 = 3$ et pour tout $n \in \mathbb{N}$, $F_{n+1} = 4F_n$. Par récurrence immédiate, $\forall n \in \mathbb{N}, F_n = 3 \times 4^n$. De même, si R_n est le nombre de caractères de rotation '+' ou '-' à l'étape n alors $R_0 = 4$ et pour tout $n \in \mathbb{N}$, $R_{n+1} = R_n + 4F_n$. Par récurrence immédiate, $\forall n \in \mathbb{N}, R_n = 4^{n+1}$.

7) Avec un segment de longueur 2, on parcourt une longueur égale à $2 \times 3 \times 4^4$ pixels à 1000 pixels par seconde et on tourne au total de 60×4^5 degrés dans un sens ou dans l'autre à la vitesse de 800 degrés par seconde. Cela prend au total : $\frac{6\times256}{1000} + \frac{60\times1024}{800} \approx 78,3s$.

Nouveau mode de représentation

8)

```
def maison2():
    X = [0, 100, 100, 50, 0, 0]
    Y = [0, 0, 100, 100+50*sqrt(3), 100, 0]
    plot(X, Y, "k-")
    show()
```

9) Il n'y a pas de question 9 !!

10) Si on rencontre 'F' : $(x, y, d) \mapsto (x + \ell \times \cos(d), y + \ell \times \sin(d), d)$.
Si on rencontre '+' : $(x, y, d) \mapsto (x, y, d + a)$.
Si on rencontre '-' : $(x, y, d) \mapsto (x, y, d - a)$.

11)

```
def dessine(unite, angle, motif):
    x, y, d = 0, 0, 0
    listX, listY = [X], [Y]
    for ch in motif:
        if ch == 'F':
            x += unite*cos(d*pi/180)
            y += unite*sin(d*pi/180)
            listX.append(x)
            listY.append(y)
        elif ch == "+":
            d += angle
        elif ch == "-":
            d -= angle
        else:
            return 1
```

```
    plot(listX, listY)
    show()
    return 0
```

Gestion des ramifications

12) Cela donnerait : $F \mapsto F+F++++++F++++++F-F-----F++++++F$ (en s'arrêtant au 3ème $-$ si l'on tronque à 20 caractères comme il est demandé).

13) On peut revenir à une position enregistrée sans refaire tout le parcours à l'envers. Cela simplifie le codage et cela raccourcit les temps de tracé.

14) On obtient un H couché, de 4 centimètres de large et 2 centimètres de haut.

15) Avec le module `turtle`, on peut utiliser la fonction `goto(x,y)` pour déplacer le curseur à la position (x, y) et la fonction `seth(d)` pour tourner le curseur dans la direction donnée par l'azimut d en degrés. Il faut bien sûr lever le stylo `up()`, puis le rebaisser `down()`.

```
def dessiner(unite, angle, motif, azimut=0):
    pile = []
    x, y, d = 0, 0, azimut
    for ch in motif:
        if ch == 'F':
            x += unite*cos(d*pi/180)
            y += unite*sin(d*pi/180)
            forward(unite)
        elif ch == "+":
            d += angle
        elif ch == "-":
            d -= angle
        elif ch == "[":
            pile.append((x, y, d))
        elif ch == "]":
            (x, y, d) = pile.pop()
            up()
            goto(x, y)
            seth(d)
            down()
        else:
            return 1
    return 0
```

Avec `matplotlib.pyplot`, on peut réécrire la fonction `dessine` ainsi :

```
def dessine(unite, angle, motif, azimut=0):
    pile = []
    x, y, d = 0, 0, azimut
    lx, ly = [x], [y]
    for ch in motif:
        if ch == 'F':
            x += unite*cos(d*pi/180)
            y += unite*sin(d*pi/180)
            lx.append(x)
            ly.append(y)
        elif ch == "+":
            d += angle
        elif ch == "-":
            d -= angle
        elif ch == "[":
```

```
            pile.append((x, y, d))
        elif ch == "]":
            (x, y, d) = pile.pop()
            plot(lx, ly, "k-")
            lx, ly = [x], [y]
        else:
            return 1
    plot(lx, ly, "k-")
    show()
    return 0
```

On peut alors tester avec les commandes suivantes :

```
dessiner(20, 90, "F[-F[+F]-F]F")
dessiner(5, 20, evolution("F", "F[+F]F[-F]F", 4), 90)
dessine(3, 20, evolution("F", "F[+F]F[-F]F", 5), 90)
```

PARTIE B : Génération automatique de L-systèmes

Génération de la population d'origine

```
from random import *
```

1)

```
def verifie(regle):
    t = 0
    for ch in regle:
        if ch == "[":
            t += 1
        elif ch == "]":
            t -= 1
            if t < 0:
                return False
    return t == 0
```

```
1  def simplifie(regle) :
2      #
3      # Fonction qui reçoit une règle sous forme d'une chaîne de caractères et
4      # retourne une chaîne de caractères représentant la règle simplifiée.
5      #
6      i, reponse = 0, ""
7      while i < len(regle):
8          double = regle[i] + regle[i+1]
9          if double == "+-" or double == "-+":
10             i = i + 1
11         elif double != "-]" or double != "+]":
12             reponse = reponse + regle[i]
13         i = i + 1
14     reponse = reponse + regle[-1]
15     if len(reponse) != len(regle):
16         reponse = simplifie(reponse)
17     return reponse
```

2) a) Les lignes 15 et 16 du code précédent servent à réitérer le procédé tant qu'il y a des simplifications possibles.

b) Voici une liste des erreurs :

- ligne 7 : il faut écrire `while i < len(regle)-1` car sinon `regle[i+1]` provoquera une erreur "Index out of range".
- ligne 11 : il faut écrire `and` à la place de `or`. Il faut en effet que les deux conditions aient lieu en même temps pour que l'on conserve le i-ème caractère dans la réponse.
- ligne 14 : il ne faut rajouter le caractère de fin que lorsque l'on s'est arrêté avant de l'avoir traité.

La fonction corrigée doit donc s'écrire (avec quelques simplifications syntaxiques et le rajout de la simplification de `[]`) :

```
def simplifie(regle) :
    i, reponse = 0, ""
    while i < len(regle)-1:
        double = regle[i: i+2]
        if double == "+-" or double == "-+" or double == "[]":
            i += 1
        elif double != "-]" and double != "+]":
            reponse += regle[i]
        i += 1
    if i == len(regle)-1:
        reponse += regle[-1]
    if len(reponse) != len(regle):
        reponse = simplifie(reponse)
    return reponse
```

3)

```
def genereRegle():
    alphabet = "F+-[]"
    regle = ""
    for _ in range(randint(15, 30)):
        regle += choice(alphabet)
    return regle

def compte(regle):
    ouvrants = rot = 0
    for ch in regle:
        if ch == "[":
            ouvrants += 1
        elif ch in "+-":
            rot += 1
    return ouvrants >= 3 and rot >= 2

def population(n):
    liste = []
    total = valides = 0
    while len(liste) < n:
        regle = genereRegle()
        total += 1
        if verifie(regle):
            valides += 1
            regle = simplifie(regle)
            if compte(regle) and regle not in liste:
                liste.append(regle)
    print("total =", total)
    print("valides =", valides)
    print("longueurs des règles :", set(len(regle) for regle in liste))
    return liste
```

Mutation

4)

```
def mutation(regle):
    pos, r = choice([(i, ch) for i,ch in enumerate(regle) if ch in "+-"\
     ])
    return regle[:pos] + ("-" if r == "+" else "+") + regle[pos+1:]

def mutationPopulation(L, p):
    for i in range(len(L)):
        if random() < p:
            L[i] = mutation(L[i])
    return L
```

5) a) Le test `random()< p` renvoie `True` avec une probabilité p puisque c'est la probabilité de l'évènement $X \in]0, p[$ pour la loi uniforme sur $]0, 1[$.

b) En moyenne, on modifiera donc $100p$ règles sur un total de 100.

Croisement

6)

```
def extraitBranche(regle):
    deb = choice([i for i,ch in enumerate(regle) if ch == "["])
    t = 1
    fin = deb+1
    while t > 0:
        ch = regle[fin]
        if ch == "[":
            t += 1
        elif ch == "]":
            t -= 1
        fin += 1
    return (regle[:deb], regle[deb: fin], regle[fin:])
```

7)

```
def croise(r1, r2):
    ex1 = extraitBranche(r1)
    ex2 = extraitBranche(r2)
    return ex1[0] + ex2[1] + ex1[2]
```