Projet Python : L-System
Compte Rendu

Brisset Joachim Kemgne Jules-Antoine Hopsore Theo

16 janvier 2021

Table des matières

I Présentation

Ce projet est l'évaluation finale du module python. Ce projet aura pour but de réalisé un programme capable de, a partir d'un fichier d'entrée normée représentant un L-System, générer un code permettant de le dessiner.

II Caractéristique de Projet

Langage de programmation Le programme finale devra être rédiger en langage Python étant donné que ce projet est l'évaluation finale de ce module.

Entrée le fichier d'entrée devra être sous la forme suivante :

```
axiome=   <chaine de caractère>
regles=   <symbole> = <chaine de caractère>
                    ⋮
          <symbole> = <chaine de caractère>
taille=   <nombre>
angle=    <nombre>
niveau=   <nombre>
```

L'ordre de règles du fichier n'est pas considéré important. Cependant la présence et l'unicité des règles excepté "regles" est primordial. Il en vas de même si dans la règle "regles", un même symbole apparait plusieurs fois.

Sortie Le fichier de sortie doit être en langage Python pour pouvoir être exécuter immédiatement après.

III Organisation

Git Pour avoir un travail collaboratif efficace utilisons git, un logiciel qui permet de facilité le travail collaboratif sur les projets de programmation. Il permet a chacun de modifier le code sans trop se soucier des problèmes de modification simultanée, de garder une trace de toute nos modification du code et ainsi de revenir à un code antérieur si nécessaire, de travailler à part sur de nouvelle fonctionnalité en toute simplicité et revenir travailler sur le projet principal sans soucie.

Le projet sera hébergé sur GitHub en version public.

Afin de facilité la vie a certain qui n'ont pas le courage d'apprendre le bash de git, nous utilisons gitKraken, une application permettant une utilisation intuitive de git a l'aide d'une interface graphique.

Processus de développement Pour s'assurer que le programme fonctionnera convenablement nous nous sommes concentré tout d'abord sur le fonctionnement du programme sous sa forme la plus basique. C'est-à-dire que nous ne nous attardions pas sur les vérifications, interactivité du programme et tout autre ajout. A ce stade le programme ne peut que lire un fichier spécifique considéré valide et créer le code dans un autre fichier donné. Puis nous étendrons le programme en ajouter les vérification, l'interactivité et toutes autre fonctionnalités.

Planning de travail Pour être efficace dans le réalisation du projet nous avons commencé par réfléchir ensemble à la conception du programme pour ne pas trop revenir sur nos morceau de code précédent. Puis chaque semaine nous faisions un point sur l'avancement du projet aidions certain à finir le travail de la semaine précédente et enfin établissions le travail de chaque membre pour la semaine

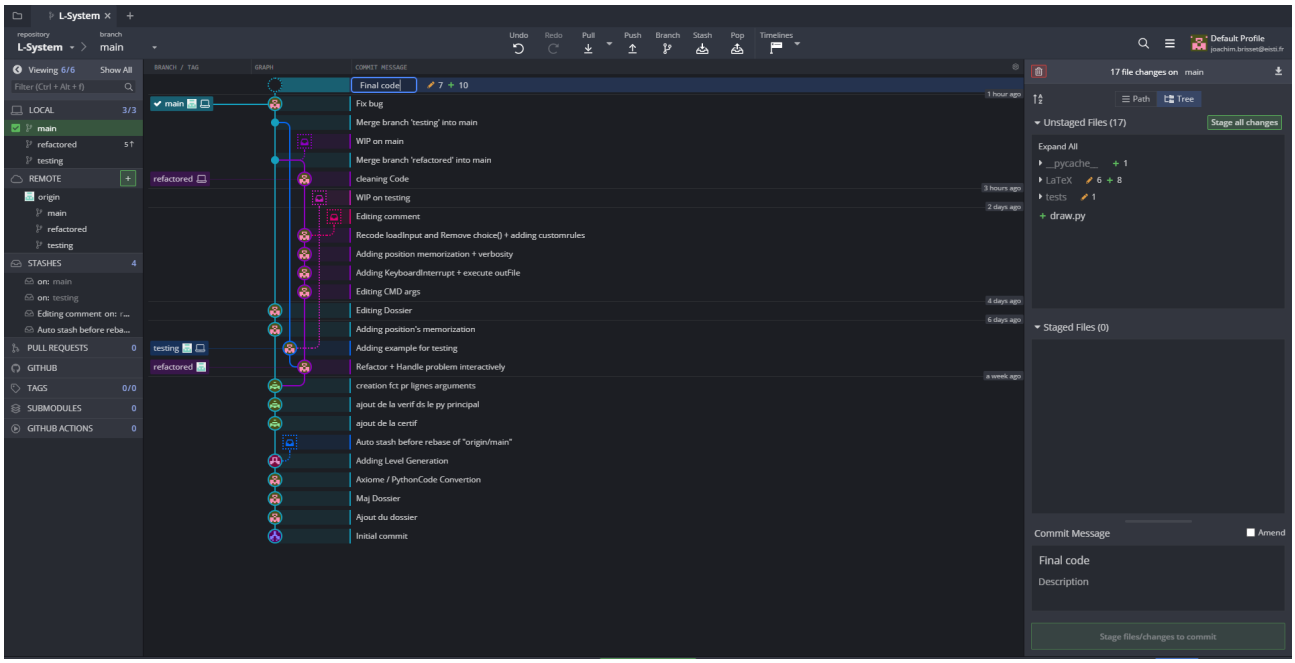

FIGURE 1 – screenshot de l'application GitKraken

IV Conception

Programme basique Voici ci-dessous le processus du programme basique :

Chacune de ces étapes fera office d'un fonction a part entière

Modèle du L-System Le l-System est représenter par un dictionnaire avec les index suivant : axiome, regles, taille, angle, niveau. Généralement en programmation nous ecrivons en anglais, cependant pour facilité le programme et la compréhension nous préférerons utiliser les mots français qui seront les même que dans le fichier d'entrée.

```
lsystem = {"axiome":None, "regles":{}, "angle":None, "taille":None, "niveau":None}  # default L-System
```

FIGURE 2 – modélisation en code Python du L-System

Structure Le projet n'étant pas très complexe nous avons décidé de rassembler l'intégralité programme dans un seul fichier, nommé *l-system.py*, et de le décomposé en plusieurs fonction effectuant chacune une tâche particulière.

V Développement

V.1 Programme basique

Dans sa version basique, le programme n'est pas très compliqué, on n'y distingue que trois fonctionnalités importantes :

- En premier lieu, le chargement du L-System par la lecture d'un fichier :

Fonction de tri (load input) Cette fonction va lire ligne par ligne de lsysteme afin de pouvoir créer des variables compréhensibles par le programme , elle va aussi verfier que toutes les instructions énoncées éxistes bien , qu'il n'y a pas plusieurs occurences pour une même règle.

- Puis ensuite la croissance du L-System jusqu'au niveau souhaité : Cette tâche est assuré par la fonction *generate_L_System_by_Level*. Le rôle de cette fonction est de générer le nouvel axiome à partir des règles de construction jusqu'au niveau de croissance voulu. Lap première boucle sert a repeter la tâche jusqu'aux niveau souhaité et la seconde boucle de parcourir tout l'axiome caractère par caractère et de remplacer les caractère remplacable.

```python
def generate_L_System_by_Level(dictionnaire,axiome,niveau):
    for loop in range(niveau):
        t=''
        for i in axiome:
            t += dictionnaire[i] if i in dictionnaire else i
        axiome = t
    return(t)
```

FIGURE 3 – fonction generate_L_System_by_Level

- Et enfin la transcription du L-System en un code python capable de le dessiner. Cette fonctionnalité se fait au travers de l'appel de la function *LSystemToPythonCode()* qui prend en paramètre l'axiome a dessiner (on aurait pu prendre le L-System en entier) et le nom du fichier de sortie.

Les instructions associées a leur symbole sont stocké dans un dictionnaire :

```python
actions = {
    'a': ["pd()", "fd({taille})"],
    'b': ["pu()", "fd({taille})"],
    '+': "right({angle})",
    '-': "left({angle})",
    '*': "right(180)",
    '[': "#TODO",
    ']': "#TODO"
}
```

FIGURE 4 – pas de titre

La notation *{taille}* et *{angle}* sont appelé placeholder, ce sont des variable qui seront remplacer par autre chose plus tard et notamment par leur valeur dans ce programme.

Par la suite la fonction na plus qu'a écrire les ligne obligatoire tel que *from turtle import ** et pour chaque symbol qu'elle rencontre dans l'axiome elle cherche les instruction associé et les écrit dans le fichier et finit son exécution

Memorisation de la position Une autre fonctionnalité certe bien moins importante mais pas des moidres, la memorisation de la position, est assuré par les ligne de codes suivantes inseré au debut du fichier de sortie :

```
write(openedFile, "savedPos = [] \n")
write(openedFile, "def savePos():\n\tsavedPos.append( (pos(), heading()) )\n")
write(openedFile, "def getLastPos():\n\ttemp = savedPos.pop()\n\tgoto(temp[0])\n\tsetheading(temp[1])\n")
```

FIGURE 5 – memorisation de la position

V.2 Vérification et Interactivité

Fonction verification Cette fonction permet de verifier la validité des donnée entrées : elle vérifie que la type de variable associé à chaque instruction est correcte et vérifie aussi que leur validité (par exemple ne pas dépasser une valeur précise)

Ce programme permet à l'utilisateur de compiler avec un nouveau fichier et a vérifier si le fichier est valide et excite.

```
def inputFile(message="file's path : ", defaultFile = None, shouldExist = None):
    ''' handle user's file input validity '''

    filename = os.path.abspath(defaultFile or input(message))  # ask user for file path if no default file path p

    if shouldExist and not os.path.isfile(filename):                    # if file should exist check it
        print("Sorry but the path you provide for the file does not exist")
        return inputFile(message=message, shouldExist=True)

    if filename.find("<>:*?|") != -1:                                   # check naming normes
        print("Sorry but you provided a non valid filename ('<>:*?|')")
        return inputFile(message=message, shouldExist=shouldExist)

    if shouldExist == None and os.path.isfile(filename):                # handle if file alredy exist
        print("The file you provided already exist should we continue ? (yes/no)")
        response = input()
        while not response in ("yes","no","Y","N"):
            print("The file you provided already exist should we continue ?")
            response = input()

        if response in ("no","N"): return inputFile(message=message, shouldExist=shouldExist)

    return filename
```

FIGURE 6 – interaction

V.3 Argument de commande

Fonction ligne de commande Pour implementer cette fonctionnalité nous avons separer le code principale en 2 partie :

- la fonction *app* qui est l'exécution de la fonctionnalité première du programme.
- la fonction *main* qui vas gérer les argument et l'arret du programme et qui execute la fonction *app*

La lecture des arguments est réalisé à l'aide du module emphgetopt largement utilisé par la communauté et permet au programme d'interpreter des arguments de commande :

- le -i est associé au fichier d'entré
- le -o à celui de sortie.
- le –nodraw pour ne pas dessiner le L-System.

Si il n'y a pas de fichier d'entré spécifié , ce dernier sera demandé au cours du programme , de meme , pour celui de sortie. Si la syntax de la commande est est invalide l'utilisateur s'en voit informer par une aide.

V.4 Module

Le fichier a été organisé de sorte à ce que'on puisse l'importer dans un autre projet si on le souhaite.

```
l-system.py est un programme pour generer des programme dessinant le L-System
SYNTAX : python l-system.py [options]

OPTIONS :
        x -i <fichier> ou --inFile=<fichier>  : permet de definir le fichier d'entree
        x -o <fichier> ou --outFile=<fichier> : permet de definit le fichier de sortie
        x --nodraw : permet de ne pas dessiner a la fin du programme
```

FIGURE 7 – exemple de l'aide du programme

Le code ne s'execute que si il est le fichier principal du projet.

```
if __name__ == "__main__":      # call main() if this file is the primary file
    main()
```

FIGURE 8 – Execution du programme en fichier principal

De plus les fonctions ont été créer de sorte à ce que celle qui utilisé exclusivement par le programme pour sont bon fonctionnement soit privé et non accessible par l'import. Tandis que celle qui peuvent être utilile lors d'un import sont accessible et sont normalement independant des variables exterieurs ce qui les rend utilisable par d'autre developpeur et maniable par les paramètres.

```
if __name__ == "__main__":      # call main() if this file is the primary file
    actions = {                         # switcher by symbols···
    }

    def app(inFile = '', outFile = '', shouldDraw=True):···

    def showHelp():···
    def main():···

    main()
```

FIGURE 9 – Exemples fonctions privées

V.5 En plus

Des actions customisables Il est possible de rajouter et modifier des action associé au symbole dynamiquement a l'aide de la règles *customrules*.

```
customrules=
"c=color("blue")"
"#=width(5)"
```

Il est même possible de mettre des scripts Python.

VI Acquis

with syntax ???

```
import sys
import getopt
import os

> def inputFile(message="file's path : ", defaultFile = None, shouldExist = None):···
> def loadInput(filename, lsystem):···
> def checkLSystem(lsystem):···
> def formatActions(lsystem, actions):···
> def generate_L_System_by_Level(rules, axiome, level):···
> def LSystemToPythonCode(axiome, action, f, verbose=False):···

> if __name__ == "__main__":      # call main() if this file is the primary file···
```

FIGURE 10 – Fonctions publiques

dict switcher Durant le projet nous avons voulu utilisé un block switch pour pourvoir déterminer quelle instruction sont associé a un symbole néanmoins cette syntax n'existe pas en python (bien qu'on aurait put utilise 'if – elif'). Ce problème nous a amener a entrevoir une nouvelle utilisation des dictionnaire comme switcher grâce a « switcher['case'] ». De plus utilisé un dictionnaire de cette manière permet d'avoir un switch dynamique (qui n'est pas fixe dans le programme) et ainsi de pouvoir le modifier pendant l'execution.

type of input On a appris a bien gérer le type d'entrée voulus a l'aide de la fonction input et du block try : except ValueError :