Analyse et amélioration d'un petit jeu de labyrinthe

Olivier Elophe

Objectifs

1. Proposer une analyse des paradigmes de programmation mis en jeu dans le projet.
2. Proposer des versions alternatives de certaines parties du projet en changeant de paradigme. Par exemple, proposer des versions récursive et itérative de certaines fonctions, introduire des objets ou se contenter de tuples ou tuples nommés. Proposer des versions alternatives de certaines fonctions avec/sans map/reduce/schéma de compréhension. Proposer des variantes du projet utilisant des structures de données mutables ou pas.
3. Réfléchir à la pertinence du découpage fonctionnel et éventuellement proposer des variantes.
4. Réfléchir au caractère générique ou pas de l'implantation et proposer des améliorations.
5. Réfléchir à la méthodologie de test de chacune des parties.
6. Éventuellement, quand c'est pertinent, analyser la performance des solutions mise en œuvre.
7. Si le projet ne met pas en œuvre les paradigmes impératifs/fonctionnel/objet/événementiel proposer une extension qui incitera à mettre en œuvre ces paradigmes.

Sources du projet d'origine

Code source du code d'origine : version originale

Code source avec quelques améliorations : version raisonnable

Code source en utilisant pleinement pygame : version complètement différente

Il s'agit d'un petit jeu de labyrinthe utilisant la bibliothèque Pygame.
Contraintes :
Le personnage peu se déplacer dans le labyrinthe et ramasser trois objets disposés aléatoirement. Une fois sur la case d'arrivée, si les trois objets ont été ramassé la partie est gagnée, sinon la partie est perdue.

1 Analyse

Le projet est fonctionnel et c'est déjà une bonne nouvelle.
Du côté des paradigmes de programmation c'est essentiellement procédural(impératif) avec une tentative d'orienté objet et nécessairement de l'événementiel avec pygame.
Le code n'est quasiment pas commenté et pas du tout documenté, aucune docstring.
La modularité est très limitée. On peut remarquer que le choix de stocker la structure du labyrinthe dans un fichier externe est un élément positif pour la modularité.
Aucun test et aucune gestion des erreurs.

2 Pistes d'amélioration

***

Sur le plan algorithmique je remarque un passage intéressant de le code d'origine :

```
continuerob = True
while continuerob:
        x = randint(0,14)
        y = randint(0,14)
        if self.structure[y][x] == '0':
                continuerob = False
                self.structure[y][x] = 'i'
        else:
                continuerob = True
```

Cette portion code consiste à tirer des coordonnées aléatoires tant qu'elles tombent sur des murs. En théorie rien ne prouve que cette boucle se termine. Une alternative plus raisonnable serait de faire un random.choice sur une liste ne contenant que les positions possibles.

Pour arranger ce problème, au moment de générer la structure à partir du fichier je récupère les emplacements libres.

```
free_path = set()
```

Que j'utilise ensuite ainsi :

```
free = sample(niveau.free_path, 1)
niveau.free_path.remove(free[0])
y = free[0][0]
x = free[0][1]
self.niveau.structure[y][x] = 'i'
```

J'ai hésité entre un set et une liste. S'il ne s'agit que de cette partie de code les listes seraient adaptées. Mais, dans une perspective d'amélioration on peut imaginer que dans les déplacements il faille faire des tests d'appartenance l'ensemble des positions libres. Dans ce cas les ensembles sont bien plus performants ! (table de hachage)

***

Je remarque la ligne suivante :

```
image_icone = "images/mg_droite.png"
```

Le nom de l'image me laisse penser qu'il voulait une image différente en fonction de la direction du personnage. Il ne semble pas avoir réussi.
Ici, plusieurs possibilités. On peut utiliser une image contenant les 4 images et extraire celle qui convient ou alors charger les 4 images dans une liste ou une autre structure de données.
Je vais opter pour la structure de données. Quelle structure de donnée retenir ? Les set ou les frozenset ne conviennent pas ici parce que j'ai besoin que les éléments soient dans un certains ordre. L'avantage du dictionnaire est la clarté du code grâce aux clés (Toujours pas d'ordre mais je pourrais appeler la bonne image avec la clé). Mais cela ne me semble pas être la structure la plus adaptée parce que si on veut, en plus, animer le personnage il peut être utile de faire des opérations sur les index. Entre liste et tuple, plutôt tuple, on ne devrait pas avoir à rajouter une direction supplémentaire... Je vais donc proposer la structure suivante :

```
IMAGES_PERSONNAGE = ("images/droite.png",
                     "images/bas.png",
                     "images/gauche.png",
                     "images/haut.png")
```

Que je transforme plutôt en :

```
DIRECTIONS = ("droite", "bas", "gauche", "haut")
IMAGES_PERSONNAGE = [i+".png" for i in DIRECTIONS]
```

Maintenant je décide de rajouter trois positions par direction et de rendre le tout plus modulaire :

```
# Images
PATH = "images/"
FORMAT = ".png"
DIRECTIONS = ("droite", "bas", "gauche", "haut")
IMAGES_PERSONNAGE = [[PATH+i+str(j)+FORMAT for j in range(1, 4)] for i in DIRECTIONS]
```

De cette façon je peux appeler les images ainsi

```
self.image = pygame.image.load(
                IMAGES_PERSONNAGE[d][self.pas%3]).convert_alpha()
```

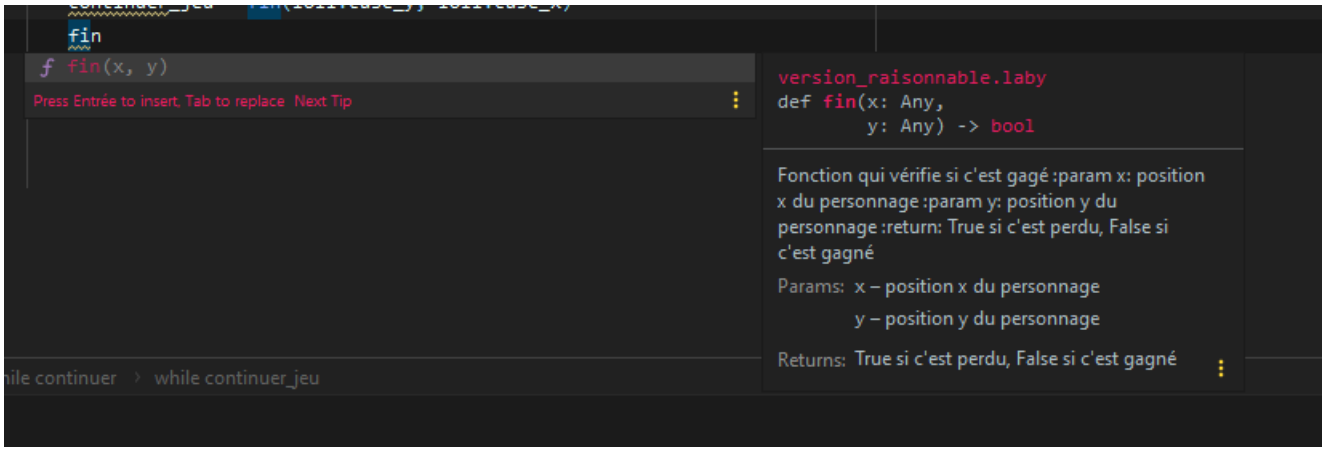
Conclusion : Le personne possède 9 images, 3 pour chaque direction, et en même temps la méthode deplacement de la classe Perso s'en retrouve factorisée.

***

Pour tester la victoire j'ai crée une fonction, elle va me servir de prétexte pour insérer une docstring.

```
def fin(x, y):
        """
        Fonction qui vérifie si c'est gagé
        :param x: position x du personnage
        :param y: position y du personnage
        :return: True si c'est perdu, False si c'est gagné
        """
```

ce qui donne, à la demande d'aide :



Évidemment il faudrait ajouter les spécifications de chaque fonction et méthode.

***

Du côté du paradigme fonctionnel, en plus des listes en compréhension utilisées précédemment, je pourrais proposer de remplacer cette partie de code :

```
with open('n1', "r") as fichier:
        structure_niveau = []
        for ligne in fichier:
                ligne_niveau = []
                for sprite in ligne:
                        if sprite != '\n':
                                ligne_niveau.append(sprite)
                structure_niveau.append(ligne_niveau)
        structure = structure_niveau
```

Par le code suivant :

```
with open('n1', "r") as fichier:
        structure_niveau = list(map(lambda liste: list(filter(lambda x: x != '\n', liste)),
                                                               list(map(list, fichier))))
```

Cette portion de code peut être testée via ce fichier : paradigme_fonctionnel.py

Occasion de placer un **assert** et d'ajouter un mot sur les tests.

***

Un dernier mot sur les tests. Les tests unitaires sont difficile à mettre en place sur un tel code. Cela montre la pertinence du **Test-Driven Development**. En TDD les fonction ou méthode auraient été pensées autrement. Sur d'autres projets je lance mes tests à l'aide de **pytest** et je vérifie ma couverture avec **coverage**.