Cléa Flatres & Ewen Le Bihan

Hyperis est un jeu vidéo d’aventure textuel totalement contenu dans un terminal écrit en Python.
Sans framework, sans librairie.
Sans tutoriel.

Une entrée utilisateur en langage naturel.
Une histoire dans laquelle vous êtes plongé, tel un roman.
Des personnages, chacun ayant un rôle important.
Ils se rappelleront de vos actions, et vous mènerons aux diverses fins de jeu possible.

01/
le code

Nous avons, dans l'optique d'un jeu tourné autour d'une histoire principalement de dialogues, et de conditions basées sur les réponses du joueur, l'utilisation de classes nous a semblé logique.
Le code est structuré en divers fichiers ayant tous des fonctions bien distinctes:

```
story/
     # Fichiers controllant le déroulement de l'histoire

characters.py # Défini la classe Character, ainsi que tout les objets
player.py # Défini la class Player, et instancie `player`, représentant le joueur


ui.py # Fonctions d'abstraction, ex. poser des questions
```

LE CODE/01
characters.py

La classe `Character` ne contient que peu de propriétés, mais permet la création de nombreux objets, représentant les différents personnage du jeu. De loin, la méthode `say(text)` est la plus importante.
Elle prend en compte le nom, la couleur si applicable, et d'autres facteurs pour afficher à l'écran le dialogue prononcé par le personnage représenté par son objet.

Cela permet de garder une grande lisibilité dans les fichiers "story", et évite la répétition quand au choix des couleurs de texte.

LE CODE/02
story/*.py

À l'inverse des fichiers contenant des fonctions d'abstractions qui exécutent de la logique complexe, les fichiers décrivant l'histoire devrait se limiter à l'appel de fonctions basiques ou définies dans d'autres fichiers, et à des conditionnements, avec les mot-clés `if`, `elif` et `else`.
En effet, l'histoire est contée par la "voix off", les dialogues des personnages et les réponses données par le joueur. Dans ces fichiers, tout les objets représentant les personnages non-joueurs, ainsi que l'objet

player lui-même, sont importés pour que l'on puisse faire parler les personnages, et faire évoluer les différentes "stats" du joueur, que ce soit sa vie, sa force, sa réputation, son camp ou d'autres encore.

Pour faciliter l'écriture de l'histoire, il est prévu dans le futur de mettre en place une syntaxe simplifiée qui serait traduite dans le fichier python correspondant, augmentant ainsi encore la lisibilité du code.

Le texte suivant ...

```
C'est dans ce contexte que vous devez choisir un camp, tout neutre
étant considéré pour un traître aux deux côtés.

Vos choix pourront peut-être changer l'histoire ...

>> reine, royaume
     [Lydia] Bon choix. Bienvenue parmi nous.
>> résistance
     [Artur] Merci d'avoir rejoint nos rangs, #name. Bienvenue
     dans la résistance.
>?
     La neutralité n'est pas une option à Hyperis. Choisissez
     judicieusement, mais faites votre choix.
```

Serait traduit en python comme ceci:

```
from ui import typewriter
from player import player
from characters import *

typewriter("C'est dans ce contexte que vous devez choisir un camp,
tout neutre étant considéré pour un traître aux deux côtés.\n\nVos
choix pourront peut-être changer l'histoire ... ")

_ans = player.ask(['reine', ['résistance', 'royaume']], "La
neutralité n'est pas une option à Hyperis. Choisissez judicieusement,
mais faites votre choix.")

if _ans == 'reine':
     Lydia.say("Bon choix. Bienvenue parmi nous.")
elif _ans == 'résistance':
     Artur.say("Merci d'avoir rejoint nos rangs, %s. Bienvenue dans
la résistance." % player.name)
```

Mais les fonctions comme typewriter(text, speed, method) doivent bien être définies quelque part: nous en venons aux fonctions d'abstraction, où est concentrée la plupart de la logique complexe, définies dans ui.py

LE CODE/03

ui.py

Ce fichier définit quelques fonctions fréquemment utilisées dans les fichiers story, ou dans la classe `Character`. Deux fonctions y sont particulièrement importante, `ask(choices, error_msg)` et `typewriter(text)`. Ces fonctions permettent-respectivement-de poser une question à l'utilisateur, et d'écrire un texte progressivement, caractère par caractère, à l'écran.

UI.PY/01

ask(choices, error_msg)

Pour parvenir à interpréter une réponse de l'utilisateur en langage naturel, un simple `input()` n'est pas suffisant. L'utilisateur doit faire choix parmis plusieurs, et ces différents choix sont spécifés par l'argument `choices`, contenant une liste de de listes de chaines de caractères représentant des synonymes de chaque choix valide

De plus, si l'utilisateur ne répond pas par un choix valide, il faut lui reposer la question, préférablement avec un message réagissant à sa réponse incorrecte: l'argument `error_msg`. Il faut ensuite reposer la question, et, autant de fois qu'il le faut, répéter ces actions tant que le joueur ne répond correctement. Pour se faire, nous faisons appel à la récursion, en appellant la fonction dans sa définition. Nous pourrions aussi faire appel à une boucle `while`, mais il faudrait répéter du code.
Voici la définition de la fonction:

```
def ask(choices: list, error_msg: str):
    # Récupérer la réponse du joueur
    ans = input('> ')
    # Pour chaque choix ...
    for choice in choices:
        if type(choice) is not list:
            choice = [choice]
        # Pour chaque synonyme dans chaque choix ...
        for synonym in choice:
            # On ignore les accents et autres diacrites
            synonym, ans = unidecode(synonym), unidecode(ans)
            # Si le synonyme est dans la réponse ...
            if ans in synonym:
                # Renvoyer le premier choix
                return choice[0]

    # Afficher le message
    typewriter(error_msg)
    # Retourner la valeur renvoyée par un nouvelle appel à la
    # fonction, re-demandant une réponse à l'utilisateur (récursion)
    return ask(choices, error_msg)
```

1

2

notes

1 Si le choix n'a pas de synonyme, et que ce n'est donc qu'une chaîne de caractères (if type(choice) is not list), on la transforme en une liste contenant un seul élément (avec choice = [choice]), pour que la boucle for n'itère pas sur chaque caractère

2 Comme return stopppe l'exécution de la fonction et renvoie une valeur, ces lignes seront exécutées seulement si aucune valeur n'a été renvoyée, c'est à dire si la réponse ne contenait aucun des choix valides.

UI.PY/02

typewriter(text, speed = 30, method = 'char')

Afin de rendre l'expérience de jeu plus vivante, on affiche le texte peu à peu, en "l'écrivant" à l'écran, au lieu de simplement l'afficher instantanément. Cela aide aussi à donner à notre jeu un air de jeu d'arcade.

Pour ce faire, on passe sur chaque caractère du texte avec une boucle for (ou chaque ligne, suivant la valeur de l'argument method), et on écrit chaque caractère sans retour à la ligne, en changeant l'argument end de print(), le passant de '\n' (nouvelle ligne) à '' (rien). On appelle ensuite la fonction sleep() du module time, permettant d'introduire un délai en secondes entre l'écriture de chaque caractère, délai correspondant à l'inverse de la vitesse d'écriture, spécifiée par l'argument speed.

```
def typewriter(text: str, speed: int = 30, method = 'char'):
    # On divise le texte en fragments, selon la méthode utilisée
    if method == 'char':
        # Division caractère par caractère
        fragments = list(text)
    elif method == 'line':
        # Division ligne par ligne
        fragments = text.split('\n')
    else:
        # Si jamais la méthode utilisée n'est pas reconnue, on lève
        # une erreur.
        raise ValueError('Unknown typewriter method ' + method)
    # On calcule la valeur du délai à appliquer entre l'écriture de
    # chaque fragment
    delay = 1/speed
    # Pour chaque fragment ...
    for fragment in fragments:
        # On affiche le fragment, sans retour à la ligne
        print(fragment, end='')
        # On "flush" la sortie (module sys)
        sys.stdout.flush()
        # On attend le délai avant d'écrire le prochain fragment
        sleep(delay)
    # On met un retour à la ligne final
    print()
```

LE CODE/04

player.py

Ce fichier décrit la classe Player, et instancie un unique objet player, qui représente le joueur, et les diverses "stats" qui lui sont associées. Même si, avec l'évolution de l'histoire, l'ajout de nouvelles "stats" est certain, voici une liste des "stats" présentes sur le personnage:

propriété	*nom dans l'interface*	*valeur d'initialisation*
	Dans R^+	
hp	points de vie	100
speed	vitesse	100
	Dans [-1;1]	
reputation	réputation	0
	Dans [0; 1]	
food	niveau d'alimentation	0.5
strength	force	0
smart	intelligence	0.5

En plus de ces "stats", la classe Player définit d'autres attributs, comme son nom.

Cependant, changer une "stat" du joueur sans qu'il le sache n'est pas très judicieux. De ce fait, effectuer un simple ...

```
player.speed = player.speed + 20
```

n'est pas suffisant. Il faudrait que l'on puisse en informer le joueur: c'est là qu'intervient la méthode `change(stat, value)`. L'exemple précédent se transforme en ...

```
player.change('speed', +20)
```

Et, à l'exécution de cette ligne, non seulement la valeur de speed est modifiée, mais un message sera affiché à l'utilisateur:

```
Vitesse: +20 -> 120
```

Indiquant la "stat" modifiée (Vitesse), le gain/la perte (+20) et la nouvelle valeur (120).

Bien évidemment, le message est en rouge lors d'une perte et en vert lors d'un gain.