Projet : S'entraîner sur les boucles

-
-
-

1. Introduction

Oh ouiii du RRRRubyyy ! Dans ce projet tu vas faire plein de programmes passionnants dans lesquels tu vas jouer avec des boucles, t'éclater avec des méthodes, et te sentir comme Champollion en Égypte quand il a enfin pigé ce que voulaient dire les hiéroglyphes. Par ailleurs, (tu vas dire qu'on insiste…) mais évidemment, toutes tes lignes de code devront être imbriqués dans des méthodes.

2. Le projet

Pour ce projet, crée un repository GitHub (un par équipe de pair-programmers). Pour chaque section du projet, tu devras créer un fichier qui contient les méthodes que l'on te demandera de faire. Afin d'éviter d'avoir un dossier un peu bordélique, range tous les programmes dans un dossier `lib` (c'est la convention de nommage pour les dossiers Ruby). À la fin, ton dossier devrait ressembler à ça :

```
ton_dossier
├── lib
│   ├── 00_hello.rb
│   ├── 01_pyramids.rb
│   ├── 02_password.rb
│   └── 03_stairway.rb
└── README.md
```

2.1. Programme 0 : Hello !

Ton programme sera contenu dans un `lib/00_hello.rb`.

Vald a besoin de toi pour l'écriture de sa prochaine chanson à succès : il lui faudrait un programme qui dit bonjour.

2.1.1 Dis bonjour

Code une méthode `say_hello` qui va dire bonjour quand on l'exécute. Ça devrait ressembler à quelque chose du genre :

```
def say_hello
  # mon code ici
end
```

2.1.2 Coucou toi

Modifie la méthode pour qu'elle prenne désormais en entrée une variable `first_name` et que ton script affiche "Bonjour, `first_name` !"

2.1.3 Le combo

Maintenant rajoute une méthode `ask_first_name` qui demande à l'utilisateur son prénom et retourne le résultat avec un `return`. Combine-la avec ta `say_hello(first_name)` pour avoir un programme qui demande à l'utilisateur son prénom et lui dit bonjour. Quelque chose comme :

```
# METHODS ARE ABOVE
first_name = ask_first_name
say_hello(first_name)
```

🚀 **ALERTE BONNE ASTUCE**

Nous profitons de cette double manipulation de `puts` et `return` dans tes méthodes pour te demander de garder tête baissée et essayer de comprendre la différence entre les deux. N'hésite pas à expliquer à tes co-moussaillons, à demander de l'aide, ou chercher sur internet les subtilités.

2.2. Programme 1 : vacances en Égypte

Ton programme sera contenu dans un fichier `lib/01_pyramids.rb`.

Ramsès II a été impressionné par ton savoir-faire architectural de précédemment et voudrait te débaucher pour construire des pyramides dont on parlera encore dans 4000 ans. Pas besoin de bipper Panoramix et sa bande pour le taff : Ruby sera ta potion magique.

2.2.1 Moitié de pyramide

Reprends ton deuxième exercice de la pyramide en l'encapsulant dans une méthode `half_pyramid`.

2.2.2 Pyramide de Gizeh

Catastrophe, cette pyramide ne tient pas debout, tout s'effondre sur le côté. Ramsès II a le sum, mais grâce à une superbe présentation PowerPoint bien pipeautée, tu as pu lui vendre la version premium de la pyramide : celle avec deux côtés. Champion 🔺

Crée une méthode `full_pyramid` qui va construire plusieurs étages avec ce rendu dans le terminal :

```
Salut, bienvenue dans ma super pyramide ! Combien d'étages veux-tu ?
> 5
Voici la pyramide :
    #
   ###
  #####
 #######
#########
```

2.2.3. Alexandrie, Alexandra

Ramsou (ça y est vous êtes potos) vient d'avoir une idée de génie : il voudrait une pyramide en losange. Les normes de sécurité étant assez flex à l'époque, tu fonces sur l'occasion pour implémenter cette idée qui va (sûrement) rendre le monde meilleur.

Crée une méthode `wtf_pyramid` qui va générer plusieurs étages de cette manière :

```
Salut, bienvenue dans ma super pyramide ! Combien d'étages veux-tu ? (choisis un nombre impair)
> 9
Voici la pyramide :
    #
   ###
  #####
 #######
#########
 #######
  #####
   ###
    #
```

Là tu touches à un concept qui va changer le monde de l'architecture !

Bonus : envoie bouler l'utilisateur s'il saisit un nombre pair.

2.3. Programme 2 : Mon petit mot de passe

Ton programme sera contenu dans un fichier `lib/02_password.rb`.

Ramsès II t'a recommandé auprès d'Edward Snowden pour coder un programme de cybersécurité. Le but est de faire un écran de sécurité qui demande à l'utilisateur de définir un mot de passe, et un autre qui lui demande de le saisir.

- Tant que l'utilisateur ne donne pas le bon mot de passe, le programme le lui redemande.
- Si l'utilisateur rentre le bon mot de passe, il accède à un espace secret.

2.3.1. Décomposition du programme

Le programme va se décomposer en trois parties :

- une partie `signup`, demandant à l'utilisateur de définir un mot de passe
- une partie `login`, demandant à l'utilisateur de rentrer son mot de passe jusqu'à ce qu'il corresponde à celui défini précédemment
- une partie `welcome_screen`, affichant un écran de bienvenue avec des informations top secrètes de la NSA

2.3.2. Méthode signup

La partie signup sera assez simple, le programme va demander à l'utilisateur de définir un mot de passe et le garder en mémoire (en le stockant dans une variable).

2.3.2. Méthode login

La partie login va demander à l'utilisateur son mot de passe. Tant que l'utilisateur n'a pas rentré le même mot de passe que renseigné au moment du signup, le programme va lui dire qu'il s'est trompé et qu'il doit recommencer. Si l'utilisateur trouve le bon mot de passe, le programme va le rediriger vers l'écran d'accueil.

2.3.3. Méthode welcome_screen

L'écran d'accueil va dire à l'utilisateur qu'il est bienvenu dans sa zone secrète, et lui dire quelques secrets (par exemple le contenu – supposé – des textos et messages WhatsApp du téléphone d'un des membres de ton groupe).

2.3.4. Méthode perform

Maintenant englobe l'exécution de chaque méthode dans une méthode `perform` qui va appeler chaque morceau de code dans le bon ordre.

2.4. Programme 3 : 6ème sans ascenseur

Ton programme sera contenu dans un fichier `lib/03_stairway.rb`.

2.4.1. Jeu vidéo

Tu vas maintenant coder un super jeu qui déchire. Mario n'a qu'à bien se tenir avec sa pyramide ! Nous allons imaginer un programme dans lequel une gentille personne va devoir monter 10 marches en fonction d'un jet de dé. Une version informatique du jeu de l'oie en quelque sorte !

Voici comment cela va se dérouler : à l'exécution, le programme lance une partie. Le joueur est tout en bas d'un escalier à 10 marches, et à chaque tour il va lancer un dé :

- S'il fait `5` ou `6`, il avance d'une marche et le programme le lui dit (ainsi que la marche où il est à présent)
- S'il fait `1`, il descend d'une marche et le programme le lui dit (ainsi que la marche où il est à présent)
- S'il fait `2`, `3`, ou `4` rien ne se passe, et le programme le lui dit (ainsi que la marche où il est resté)

Quand le joueur atteint la 10ème marche, le programme l'en informe avec un message super enthousiaste, et le jeu se termine.

2.4.2. Statistiques

L'une des grandes forces de l'informatique est de pouvoir faire un nombre conséquent d'itérations et de pouvoir sortir des moyennes et médianes très facilement.

Fais donc une méthode `average_finish_time` qui va simuler au moins 100 parties, et te retourner le nombre de tours moyen pour arriver à la 10ème marche.

3. Rendu attendu

Un joli repo GitHub (un par groupe de pair-programming) avec les fichiers `.rb` correspondant aux exercices ci-dessus. Pour rappel, voici la structure attendue :

```
ton_repo
├── lib
│   ├── 00_hello.rb
│   ├── 01_pyramids.rb
│   ├── 02_password.rb
│   └── 03_stairway.rb
└── README.md
```

Petit conseil : afin d'aider tes futurs correcteurs, nous allons te demander de laisser les méthodes `perform` dans les programmes, ce qui permettra de les lancer et les tester facilement.