Compte-rendu du projet python en Concepts avancés de programmation

GROSJACQUES Marwane et SANTINI Maya

Plan

I. Code interne
II. Interface

I. Code interne

Le code interne correspond au fichier *interne.py* créant le système de carburant d'un avion et permettant sa gestion à l'aide de différentes classes. Le fichier est aussi commenté pour améliorer la compréhension du code. De plus, le nom des variables, fonctions, classes, attributs et méthodes ont été choisis afin d'être les plus simples et explicites possibles.

Il comporte une constante **NOM_VALVES** qui est un tuple des différents noms des valves ("VT12", "VT23", "V12", "V13", "V23").

Le code possède de l'encapsulation pour garder l'intégrité des classes et restreindre l'accès aux attributs et méthodes des classes depuis l'extérieur. Certains attributs n'ont qu'un getter (**@property** dans le code), on a uniquement besoin de connaître leur contenu et non pas de les modifier en dehors de la classe. Certains attributs ont un setter en plus d'un getter (**@nom_attribut.setter** dans le code). Ce dernier permet de modifier l'attribut en dehors de la classe.

Le code comporte également de la hiérarchie, qui sera expliquée plus loin dans ce rapport.

Moteur : Cette classe permet de définir le fonctionnement d'un moteur d'avion.

Pompe : Cette classe simule le fonctionnement d'une pompe d'un réservoir.

Réservoir : Cette troisième classe permet de gérer le fonctionnement des réservoirs.

Ces classes sont détaillées dans le fichier *main.py*.

A) Classe Composant

Toutes ces classes (**Moteur**, **Pompe**, **Reservoir**) ont l'attribut **numero** en commun. Ce dernier permet d'identifier le composant. Ainsi on peut créer une classe parent **Composant** qui initiera cet attribut. Ces dernières ont également toutes une méthode **reset**. Celle-ci permet d'initialiser (si appelée pour la première fois) ou de réinitialiser les attributs propres à ces classes.

Par conséquent, notre nouvelle classe **Composant** initiera un attribut **numero** dont la valeur sera donnée en argument et fera appel à la méthode **reset** de la classe fille dans sa méthode **__init__**. La méthode **__init__** est appelée lorsque l'on fait appel aux classes filles de **Composant** grâce à la hiérarchie.

La hiérarchie permet de transmettre les attributs et les méthodes de la classe Parent (ici **Composant**) aux classes filles (ici **Moteur**, **Pompe**, **Reservoir**).

Ainsi les classes filles n'ont pas besoin de méthode **__init__**.

B) Classe Avion

Cette classe est la classe principale : elle utilise toutes les classes vues précédemment. C'est également la plus complexe. Elle modélise le fonctionnement d'un système de carburant d'un avion.

Elle possède une méthode **__init__**. Cette méthode permet la création des trois réservoirs, des trois moteurs et des cinq valves dans un dictionnaire dont le nom de la valve est la clé. A chaque clé est attribuée le booléen False car les valves sont initialement fermées.

Cette méthode permet aussi d'initier l'attribut **nbr_erreurs** à 0, l'attribut **note** à la note maximale possible soit 10 et l'attribut **historique** à une liste vide.

Cette classe comporte également des méthodes :

Les méthodes publiques dans la classe **Avion** sont utiles pour le ficher "main.py" (l'interface), on peut les citer ainsi que les détailler :

- **reset** : elle permet de réinitialiser les attributs de l'avion à leur valeur initiale

- **update** : cette méthode est celle de la mise à jour de l'avion. Celle-ci prend en argument un booléen compte initié par défaut à True indiquant si la mise à jour compte dans la note ou non. C'est-à-dire que cela permet de différencier les évènements comme les pannes aléatoires qui ne feront pas baisser la note, et les erreurs de l'utilisateur qui, en revanche, lui feront perdre un point.
- **switch_valve** : la méthode prend en argument le nom d'une valve et change son état. SI la valve est ouverte, elle sera fermée ; si elle est fermée, alors elle passe à l'état ouvert.
  La variable choix correspond à 'x' ou 'v' que retourne la méthode **update**. On met ensuite l'historique à jour en ajoutant à la liste le choix ainsi que l'état de la valve et son nom.
- **switch_pompe_secours** : la méthode est similaire à **switch_valve**, elle prend en argument le numéro du réservoir dont la pompe de secours doit être activée ou désactivée. Ainsi, cette méthode échange l'état de la pompe de secours selon si elle est active ou inactive. La variable choix et l'historique sont mis à jour de la même manière que dans la méthode précédente.
- **vidange_reservoir** : cette méthode possède comme argument le numéro d'un réservoir et permet de vider ce réservoir en appelant la méthode vidange de ce dernier. On ajoute à l'historique cet évènement en détaillant quel réservoir ne possède plus de carburant. On fait ensuite appel à la méthode **update**, tout en précisant que l'argument compte est False (ainsi la note de l'utilisateur n'est pas affectée). Cette méthode sert à la génération d'une panne.

- **panne_pompe** : même principe que pour **vidange_reservoir**, la méthode prend en argument le numéro d'une pompe et génère une panne sur celle-ci. On récupère le numéro du réservoir à partir du numéro de la pompe (le chiffre des dizaines identifie le réservoir) puis, à l'aide du chiffre des unités, on identifie si la pompe est principale ou de secours. On génère ensuite la panne sur la pompe que l'on vient d'identifier grâce à la méthode **genere_panne** sur cette dernière.
  On ajoute ensuite cette panne à l'historique si elle n'y est pas déjà car une pompe ne peut théoriquement tomber en panne qu'une seule fois. On précise le numéro de la pompe et on met ensuite à jour avec la méthode **update** (encore une fois avec le compte à False pour ne pas que l'utilisateur ne perde de point).

- **panne_aleatoire** : la méthode génère une panne aléatoire : soit un réservoir est vidé, soit une pompe (principale ou de secours) tombe en panne. Pour cela, on utilise la fonction **choice** du module **random**. Cette fonction tire au hasard un élément d'une liste ou d'un tuple. Ainsi, chaque réservoir et chaque pompe peut subir une panne. Cette fonction sera très utile pour la fonctionnalité 'exercice' du pilote.

Les méthodes privées sont utiles uniquement à l'intérieur de la classe et n'ont pas besoin d'être appelées en dehors de celle-ci.

- **reequilibrage_reservoirs** : permet de remplir un réservoir vide si la valve entre ce réservoir et un réservoir plein est ouverte. Pour les deux valves (VT12 et VT23) entre les réservoirs, on vérifie si elles sont ouvertes pour un rééquilibrage ou si leur ouverture est inutile. Dans ce cas-là, on augmente de 1 le nombre d'erreurs (préalablement initié localement à 0). On retourne à la fin de la méthode le nombre total d'erreurs.

- **verification_moteurs** : le nombre d'erreurs est initié localement à 0.
  On regarde pour chaque moteur, pour lequel on crée une **liste_flux** vide, et pour chaque réservoir (sachant que l'on commence par le réservoir qui a le même numéro que le moteur concerné) si ce dernier alimente le moteur.
  Pour plus de clarté, on détaille pour le moteur 2 :
  - on commence par regarder le réservoir 2 (car il a le même numéro). On vérifie que le réservoir est fonctionnel, et que le flux peut passer entre le moteur et le réservoir à l'aide de la méthode privée **flux_ouvert** (expliquée après). Ici, il n'existe pas de valve entre le moteur et le réservoir, le flux passe. On ajoute le numéro de ce réservoir dans la **liste_flux**.
    On vérifie ensuite le réservoir 3, on regarde également s'il est fonctionnel et si le flux est ouvert (s'il existe bien une valve entre le moteur 2 et le réservoir 3 et que celle-ci est ouverte). SI c'est le cas, on ajoute le numéro du réservoir dans la **liste_flux**. On fait la même chose pour le réservoir 1.

S'il existe au moins un flux alimentant le moteur, on enlève le premier flux trouvé de la **liste_flux** pour l'affecter à l'attribut **source** du moteur. Le nombre de flux restants dans la liste est ajouté au nombre d'erreurs. S'il n'existe pas de flux, on ajoute une erreur à **nbr_erreurs** et on supprime la source du moteur à l'aide de son deleter.

Ensuite, on vérifie que la pompe de secours n'est pas activée pour rien (donc activée alors que la pompe principale n'est pas en panne) et ce pour chaque réservoir. Si elle est activée inutilement, on ajoute 1 au nombre d'erreurs.

Enfin, si aucun des moteurs n'est fonctionnel, la note est mise à 0.

Pour finir, on retourne le nombre d'erreurs cumulées au cours de la fonction.

- **flux_ouvert** : cette méthode prend en argument deux composants (les deux composants sont soit un réservoir, soit un moteur). Si il existe une valve entre ces derniers (ce que l'on détermine grâce à la fonction **nom_valve** détaillée en dessous) alors on retourne l'état de la valve (True (ouverte) ou False (fermée)). Sinon on retourne True, le flux est ouvert puisqu'il n'existe pas de valve. entre les deux composants considérés.

La fonction **nom_valves** est indépendante de toutes les classes. Elle retourne le nom de la valve si situant entre deux composants. Les deux composants en argument peuvent être deux réservoirs ou un réservoir et un moteur. On récupère le numéro de chaque composant avec le getter **numero**.

On vérifie que les deux composants n'ont pas le même numéro. Ainsi, on est certains de ne pas avoir le même réservoir ou un moteur et un réservoir ayant le même numéro puisqu'il n'existe pas de valve entre les deux. SI le numéro des deux composants est identique, on retourne None.

SI le numéro n'est pas le même :

- dans le premier cas (deux réservoirs), on retourne une chaîne de caractère "VT" suivie du numéro le plus petit puis de l'autre numéro (supérieur).
- dans le second cas, on retourne la même chose mais la chaîne de caractère initiale n'est que "V".

II. Interface

Pour l'interface,soit le fichier *main.py,* permet de mettre en relation les fonctions du fichier *interne.py* et de présenter le tour sous forme d'application. Comme pour le code interne, ce fichier est commenté pour améliorer la compréhension du code.

Nous avons importé le module **tkinter** pour l'interface, notre fichier **interne** pour utiliser son contenu, le module **datetime** pour noter la dans l'historique, ainsi que **os** et **webbrowser** pour ouvrir les fichiers historique générés.

A. Les constantes

Plusieurs constantes ont été établies pour clarifier le code.

- **TIMER_EXERCICE** décide du temps accordé à l'utilisateur dans la partie Exercice.
- **COULEUR_FOND, COULEUR_CONTOUR, COULEURS_TANK, COULEUR_CONTOUR_TANK, COULEUR_MOTEUR, COULEUR_TEXTE, TAILLE_BORDURE, fonte** permettent de décider et de modifier plus facilement les couleurs composant les interfaces, les textes, les tailles de bordures ainsi que la police d'écriture et sa taille.
- **WIDTH** et **HEIGHT** servent à définir la taille de la fenêtre de l'application et par conséquent la taille des widgets la composant.

- **pseudo** et **mdp** sont des chaînes de caractères vides pour pouvoir récupérer les valeurs entrées par l'utilisateur dans la partie Exercice.
- **profils** est le chemin pour stocker et réutiliser le fichier contenant le pseudo, le mot de passe et le score de chaque utilisateur

B. Fonctions pour dessiner le système

- **reset_interface** appelle la fonction reset dans la classe Avion du code interne, et update_interface pour mettre les données à condition initiales.
- **update_interface** met à jour les données de l'interface pour la Simulation et l'Exercice en les récupérant dans le code interne.

C. Interaction utilisateur avec le système et générations de panne

- **switch_valve, switch_pompe_secours, vide_reservoir** et **panne_pompe** et change les données de l'élément correspondant dans la classe Avion (valve ouverte/fermée, pompe secours activée/désactivée, réservoir vide/rempli, pompe en panne/fonctionnelle) dans le code interne selon le clic utilisateur et appelle **update_interface**

D. Fonctions pour dessiner les formes des composants de l'avion

- **round_rectangle** permet d'arrondir les angles des rectangles des interfaces Simuler et Exercice à l'aide d'une formule.
- **create_valve**, **create_reservoir** et **create_moteur** permettent la création du visuel des valves, réservoirs et moteurs ainsi que leur noms pour les identifier.
- **etat_systeme** créer l'interface affichant l'état visuel des réservoirs, valves, pompes et moteurs. Les réservoirs seront pleins (couleur du réservoir) ou vides (couleur transparente), les valves ouvertes (trait horizontal) ou fermées (trait vertical), les pompes fonctionnelles (vertes), en panne (rouges) ou désactivées (transparentes), et les moteurs fonctionnels (gris) ou dysfonctionnels (rouge). On voit également les flux alimentant les moteurs, et de quel réservoir ils proviennent selon la couleur.
- **tableau de bord** créer l'interface des boutons interactifs contrôlant les pompes et les valves. On utilise une fenêtre Toplevel pour le faire apparaître.

E. Authentification

- **verifcreer** et **verifconnecter**, selon l'option choisie entre se connecter ou créer son profil permet de vérifier si l'utilisateur remplit les conditions nécessaires pour s'entraîner : avoir un compte, le bon mot de passe, un mot de passe et un pseudo différent, pas d'espace ni dans le mot de passe ni dans le pseudo. Selon les cas, on appelle la fonction **exo** ou **errorcreer** et **errorconnecter**. Une variable locale *compt* permet de repérer l'erreur.
- **errorcreer** et **errorconnecter** sont appelées par **verifcreer** et **verifconnecter**, et permettent d'afficher le bon message d'erreur selon les différents cas possibles qu'on vérifie dans les fonctions précédentes.

F. Partie exercice

- **timer** fait appel à elle même toutes les secondes afin de simuler un timer.
  Le timer prend fin si le temps tombe à 0 ou que la note tombe à 0.
  Elle fait appel à la mise à jour de l'avion toutes les deux secondes pour vérifier les erreurs et génère une panne toutes les six secondes.
- La fonction **exo** réinitialise l'interface, ouvre la fenêtre de l'exercice puis lance le timer décrit au-dessus.

G. Création et changements des fenêtres

- **simuler, menu, exercer, authentification, creer, update_score, score, fin_exercice**

Ces fonctions créent dès l'exécution du code les différentes fenêtres de l'interface, ainsi elles pourront être affichées ou non grâce aux fonctions ci-dessous

Pour créer la fenêtre principale de l'interface, on utilise la classe **Tk** du module tkinter, auquelle on ajoute des méthodes personnalisées pour changer de fenêtres plus simplement.
Pour se faire on a créé une classe **Root** avec comme parent la classe **Tk**, ainsi elle hérite de tous les attributs et méthodes de cette dernière.

Ici nous avons les méthodes permettant de changer de fenêtres :

- **ferme_fenetres, ouvre_menu, ouvre_simuler, ouvre_exercice, ouvre_exercer, ouvre_authentification, ouvre_creation, ouvre_score, ouvre_identification, ouvre_fin**

Ces fonctions permettent chacune d'ouvrir la fenêtre désirée :

- Pour se faire on ferme d'abord toutes les fenêtres
- Puis on ouvre celle en question
- On applique aussi quelques modifications sur cette fenêtre pour donner du confort à l'utilisateur.

La classe **Root** est aussi munie du méthode **quit** permettant de quitter entièrement l'application.

- **sauvegarde_historique** permet la création des fichiers dans le dossier Historique, où on peut observer nos erreurs selon la partie choisie : nous avons accès à l'heure à laquelle a été effectué l'exercice, le pseudo, la note ainsi que les erreurs pour l'historique.
- **update_score** permet de mettre à jour le score dans le fichier *profils.txt* pour y inscrire le score le plus haut de l'utilisateur.
- **update_fenetre_score** affiche une fenêtre à la fin de l'exercice, où est inscrit le score obtenu, le pseudo de l'utilisateur, un bouton permettant de refaire l'exercice et un bouton pour un retour au menu.

H. Main

- **main** créer l'interface de base. C'est la première fonction appelée, on créer la racine où toute l'interface sera créée par la suite.

Après cette fonction, on l'appelle et on va initialiser la variable avion qui appelle la classe Avion du code interne, pour avoir accès à ses fonctionnalités et ainsi rendre le projet fonctionnel.