Feuille d'exercices : orienté objet

1 - Jeu de carte

1.1 : Dans un fichier `carte.py`, créer une classe `Carte`. Une carte dispose d'une `valeur` (1 à 10 puis J, Q et K) et d'une `couleur` (coeur, pique, careau, trèfle). Par exemple, on pourra créer des cartes en invoquant `Carte(3, 'coeur')` et `Carte('K', 'pique')`. Le constructeur doit valider que les données fournies sont valides.

1.2 : Implémenter la méthode `points` pour la classe `Carte`, qui retourne un nombre entre 1 et 13 en fonction de la valeur de la carte. Valider ce comportement depuis un fichier `main.py` qui importe la classe Carte.

1.3 : Implémenter la méthode `__repr__` pour la classe `Carte`, de sorte à ce que `print(Carte(3, "coeur"))` affiche `<Carte 3 de coeur>`.

```
c = Carte("Q", "pique")

print(c.couleur)
# Affiche pique

print(c.points)
# Affiche 12

print(c)
# Affiche <Carte Q de pique>
```

1.4 : Dans un nouveau fichier `paquet.py`, créer une classe `Paquet` correspondant à un paquet de 52 cartes. Le constructeur devra créer toute les cartes du jeu et les stocker dans une liste ordonnée. Vous aurez probablement besoin d'importer la classe `Carte`. Testez le comportement de cette classe en l'important et en l'utilisant dans `main.py`.

1.5 : Implémenter la méthode `melanger` pour la classe `Paquet` qui mélange l'ordre des cartes.

1.6 : Implémenter la méthode `couper` qui prends un nombre aléatoire du dessus du paquet et les place en dessous.

1.7 : Implémenter la méthode `piocher` qui retourne la `Carte` du dessus du paquet (et l'enlève du paquet)

1.8 : Implémenter la méthode `distribuer` qui prends en argument un nombre de carte et un nombre de joueur (e.g. `p.distribuer(joueurs=4, cartes=5)`), pioche des cartes pour chacun des joueurs à tour de rôle, et retourne les mains correspondantes.

```
p = Paquet()
p.melanger()

main_alice, main_bob = p.distribuer(joueurs=2, cartes=3)

print(main_alice)
# affiche par exemple [<Carte 3 de pique>, <Carte J de carreau>, <Carte 1 de trefle>]

print(p.pioche())
# affiche <Carte 9 de carreau>

print(main_alice[1].points())
# affiche 11
```

2 - Cercles et cylindres

2.1 : Implémenter une classe `Cercle` avec comme attributs un rayon `r` et les coordonnées `x` et `y` de son centre. Par exemple on pourra instancier un cercle avec `c = Cercle(5, (3,1))`

2.2 : Dans la classe `Cercle`, implémenter une propriété `aire` dépendante du rayon.

2.3 : Implémenter une classe `Cylindre`, fille de `Cercle`, qui est caractérisée par un rayon `r`, une hauteur `h` et des coordonnées `x`, `y` et `z`. On écrira le constructeur de `Cylindre` en appelant le constructeur de `Cercle`.

2.4 : Dans la classe `Cercle`, implémenter une méthode `intersect` qui retourne `True` ou `False` suivant si deux cercles se touchent. Exemple d'utilisation : `c1.intersect(c2)`

2.5 : Surcharger la méthode `intersect` pour la classe `Cylindre`, en se basant sur le résultat de la méthode de la classe mère.

3. Design patterns 'Observateur' appliquée aux chaînes Youtube

Les design patterns sont des patrons de conception qui permettent de gérer de manière des problèmes génériques qui peuvent survenir dans une grande variété de contextes. L'une d'entre elle est la design pattern "observateur". Il définit deux types d'entités "observables" et "observateur". Une observable peut être surveillée par plusieurs observateurs. Lorsque l'état de l'observable change, elle notifie alors tous les observateurs liés qui propage alors le changements.

Concrètement, ceci peut correspondre à des éléments d'interface graphique, des capteurs de surveillances (informatique ou physique), des systemes de logs, ou encore des comptes sur des médias sociaux lorsqu'ils postent de nouveaux messages.

(Reference plus complète : https://design-patterns.fr/observateur)

Nous proposons d'appliquer ce patron de conception pour créer un système avec des journaux / chaines youtube (observables, qui publient des articles / videos) auxquels peuvent souscrire des personnes.

3.1 : Créer deux classes Channel (chaîne youtube) et User (suceptibles de s'abonner)

- Chaque Channel et User a un nom.
- La classe Channel implémente des méthodes `subscribe` et `unsubscribe` qui ajoutent/enlèvent un compte observateur donné en argument. On introduira également un attribut dans User qui liste les chaînes auxquelles un compte est abonné et qui est modifié par les appel de `subscribe` et `unsubscribe` .
- La classe Channel implémente aussi une méthode `notifySubscribers` qui appelle `compte.actualiser()` pour chaque compte abonné de la chaîne. Pour le moment, la méthode `actualiser` de la classe User ne fait rien (`pass`)

3.2 : Ajoutons une méthode `publish` à la classe `Channel` qui permet d'ajouter une vidéo à la liste de vidéo de la chaíne. Chaque vidéo correspondra uniquement à un titre et une date de publication (gérée avec la librairie datetime). Lorsque la méthode publish est appellée, elle déclenche aussi `notifySubscribers` .

3.3 : La méthode `actualiser` de la classe `User` s'occupe de parcourir toutes les chaines auxquelles le compte est abonné, et de récupérer le titre des 3 vidéos les plus récentes parmis toutes ses chaines. Ces 3 titres (et le nom du channel associé!) sont ensuite écris dans `latest_videos_for_{username}.txt` . Par exemple:

```
C'est pas sorcier - Les chateaux forts
C'est pas sorcier - Le génie des fourmis
ARTE - La grenouille, un animal extraordinaire
```

3.4 : Tester l'ensemble du fonctionnement avec un programme tel que:

```
arte = Channel("ARTE")
cestpassorcier = Channel("C'est pas sorcier")
videodechat = Channel("Video de chat")

alice = User("alice")
bob = User("bob")
charlie = User("charlie")
```

```
arte.subscribe(alice)
cestpassorcier.subscribe(alice)
cestpassorcier.subscribe(bob)
videodechat.subscribe(bob)
videodechat.subscribe(charlie)

cestpassorcier.publish("Le système solaire")
arte.publish("La grenouille, un animal extraordinaire")
cestpassorcier.publish("Le génie des fourmis")
videodechat.publish("Video de chat qui fait miaou")
cestpassorcier.publish("Les chateaux forts")
```

4. Introduction aux ORM avec ActiveAlchemy

On se propose de reprendre le jeu de données des apps Yunohost (fichier `app.yunohost.org/default/v2/apps.json`) et d'importer ces données dans une base SQL (plus précisémment SQLite)

4.0 - Installer `active_alchemy` à l'aide de `pip3` : `sudo pip3 install active_alchemy`

4.1 - Créer un fichier `mydb.py` qui se contente de créer une base `db` (instance de ActiveAlchemy) de type sqlite. Dans la suite, on importera l'objet `db` depuis `mydb.py` dans les autres fichiers si besoin.

4.2 - Créer un fichier `models.py` et créer dedans une classe (aussi appellé modèle) `App`. On se limitera aux attributs (aussi appellés champs / colonnes) suivants :

- un **nom** qui est une chaîne de caractère *unique* parmis toutes les `App` ;
- un **niveau** qui est un entier (ou vide) ;
- une **adresse** qui est une chaîne de caractère *unique* parmis toutes les `App` ;

4.3 - Créer un fichier `nuke_and_reinit.py` dont le rôle est de détruire et réinitialiser les tables, puis de les remplir avec les données du fichier json. On utilisera pour ce faire `db.drop_all()` et `db.create_all()`. Puis, itérer sur les données du fichier json pour créer les objets `App` correspondant. Commiter les changements à l'aide de `db.session.add` et `commit`.

4.4 - Créer un fichier `analyze.py` qui cherche et affiche le nom de toutes les `App` connue avec un niveau supérieur ou égal à `n`. En utilisant l'utilitaire bash `time` (ou bien avec `time.time()` en python), comparer les performances de `analyze.py` avec un script python équivalent mais qui travaille à partir du fichier `community.json` directement (en local, pas via `requests.get`)