Fichiers à joindre : obj_ihm_texte.py / vehicules.json

Les notions abordées

Au cours de ce TP vous allez aborder les notions de base de la POO comme :

- Le codage de classes et de son constructeur.
- La création d'instances.
- La manipulation d'attributs d'instances.
- La création de méthodes.
- Mettre en œuvre des attributs « ***protégés*** » et « ***pseudo privés*** ».

Partie 1 : voitures.py : le début du commencement…

On considère le code suivant :

Code de la classe…

```
class Voitures():
    # Constructeur avec 3 arguments...
    def __init__(self, marque, modele, annee) :
        # Trois attributs d'instance...
        self.marque = marque
        self.annee = annee
        self.modele = modele
```

Code du programme principal…

```
car = Voitures("Renault", "Clio", 2018)     # Création d'une instance.
caisse = car.modele     # Lecture d'un attribut d'instance.
print(f"J'ai une {car.marque} {caisse} de {car.annee} !") # Affichage.
car.annee = 2020   # Ecriture (modification) d'un attribut d'instance.
print(f"J'ai une {car.marque} {caisse} de {car.annee} !")
```

Codez le script « `voitures.py` » en suivant les étapes suivantes :

1. Récupérez le code précédent avec deux « *Ctrl-C* et *Ctrl-V* » et **indentez correctement le code collé**.
2. Ajoutez le code qui permet d'affichez l'instance « **`car`** ».

QCM

Pour simplifier l'affichage de toutes les instances qui vont être créées dans la suite de cet exercice, on pourrait « ***redéfinir*** » la méthode « ***spéciale*** » ou « ***magique*** » nommée « **`__str__()`** » comme suit :

```
    def __str__(self) :
        # Redéfinition pour le print(instance)...
        return f"Valeurs des attributs de l'instance : {self.marque} {self.modele} {self.annee}"
```

Cela permet d'utiliser :

```
print(car)
```

Et d'obtenir directement :

```
# Avant la redéfinition…
<__main__.Voitures object at 0x102811fd0>
# Après la redéfinition…
Valeurs des attributs de l'instance : Renault Clio 2020
```

3. Ajoutez la « ***redéfinition*** » de la méthode « **`__str()__`** » comme décrit ci-dessus. **Ce code doit se situer dans la classe !!! Il ne doit donc pas être placé n'importe où…**
4. Testez votre code.

QCM

5. Créez dans le programme principal une seconde instance et affichez ses attributs d'instances.
6. Ajoutez une valeur par défaut à l'argument « **annee** » du constructeur.

Exemple :

```
    def __init__(self, marque, modele, annee = 2010):
```

7. Créez une 3ème instance de l'objet « **Voitures** » sans précisez l'année… Affichez ses attributs !

Exemple :

```
    ma_voiture = Voitures("Bugatti", "Veyron")
```

QCM

8. Ajoutez à présent et toujours dans les arguments du constructeur, une valeur par défaut à la marque (« *Ferrari* » par exemple) et au modèle (« *SF90_spider* » par exemple).
9. Créez une 4ème instance sans précisez ni la marque, ni le modèle, ni l'année et affichez ses attributs.
10. Créez une 5ème instance en précisant **un seul** argument (« *F40* » par exemple) et affichez ses attributs.

***Note** : Si on ne précise qu'un seul argument lors de la création d'une nouvelle instance, c'est forcément le 1er argument du constructeur…*

11. Créez une 6ème instance utilisant la notation suivante :

```
voiture6 = Voitures(modele = "296_GTS")
print(voiture6)
```

QCM

12. Affichez « **type(voiture6)** » ou son équivalent : « **voiture6.__class__** ».
13. Affichez « **vars(voiture6)** » ou son équivalent : « **voiture6.__dict__** ».
14. Affichez « **dir(voiture6)** »…

QCM

Partie 2 : obj_voitures.py → module…

15. Créez un nouveau script Python : « `obj_voitures.py` ».
16. Copiez le code de la classe du script « `voitures.py` » dans ce nouveau fichier. Il ne doit donc pas contenir la ligne « **car = Voitures("Renault", "Clio", 2018)** » et toutes les suivantes…
17. Créez un second nouveau script Python : « `garage.py` » qui va être le programme principal de l'exercice 2.

Pour utiliser l'objet (la classe) « **Voitures** » dans le script « `garage.py` », il faut importer le module « `obj_voitures.py` ».

18. Importez le module « `obj_voitures.py` » dans le script « `garage.py` » en ajoutant tout au début de ce dernier, la ligne :

```
from obj_voitures import Voitures
```

Le nom du module. Le nom de la classe qu'il contient.

19. Créez une instance de l'objet « **Voitures** » dans « `garage.py` » et affichez ses attributs.

Vous allez maintenant enrichir l'objet « **Voitures** » avec d'autres attributs et des méthodes…
Pour ajouter un attribut qui va exister dans toutes les instances, il faut le faire dans le constructeur et donc modifier son code…

20. Ajoutez les attributs « **prix** » (int), « **couleur** » et « **conso** » (float) à l'objet « **Voitures** » avec respectivement comme valeur par défaut : « **None** », « **Blanc** » et « **6.0** » (litres / 100km).
21. Corriger la création de l'instance de la question 19 et créez une nouvelle instance pour avoir deux instances de voitures nommées « **clio** » et « **captur** » et les caractéristiques suivantes :

Renault : Captur_TCE_90ch Conso : 7,2 l/100km Gris foncé Prix : 20 000 €	Renault : Clio_TCE_100ch Conso : 5,5 l/100km Bleu nuit Prix : 17 000 €

Remarque : Ces deux véhicules fonctionnent à l'essence…

22. Modifiez la méthode « **__str()__** » pour que soit affiché les trois nouveaux attributs et pour avoir le résultat suivant :

```
Voiture : Renault Clio - 2018 - Bleu nuit - 5.5 l/100km - 17000 €
```

23. Dans « `garage.py` », appelez cette méthode pour chaque instance de l'objet « **Voitures** ».
24. Codez une nouvelle méthode : « **calcul_consommation(self, …)** » qui calcule le nombre de litre de carburant utilisé pour faire une distance (en km) fournie en argument.
25. Testez cette méthode pour un trajet Colmar-Biarritz (1060 km) avec les DEUX véhicules.

On souhaite avoir une méthode qui calcule le prix du carburant pour un voyage donné. A un instant « t » donné, le prix du carburant est le même pour tous les véhicules. *On négligera les variations de tarifs en fonction des régions…*
Pour le prix au litre du carburant, vous allez ajouter un **attribut de classe**. Pour mémoire, un tel attribut est commun et identique pour toutes les instances.
Par usage, on place les attributs de classe au début du code juste avant le constructeur comme l'illustre le code suivant :

```
class Voitures():
    prix_litre = 1.70
    # Constructeur avec 3 arguments...
    def __init__(self, ...) :
```

Notez qu'il n'est pas précédé d'un « **self** » qui caractérise l'instance…

26. Codez une nouvelle méthode « **calcul_prix(self, …)** » qui calcul le prix du carburant sur une distance donnée en argument. Il serait élégant que cette nouvelle méthode appelle la méthode « **calcul_consommation(self, …)** » pour obtenir le nombre de litre de carburant… Testez votre code avec le trajet Colmar-Biarritz pour les DEUX instances de « **Voitures** ».

QCM

Il convient de pouvoir mettre à jour occasionnellement le prix au litre du carburant qui est rappelons-le, est toujours un **attribut de classe**.

27. Codez une troisième méthode « **modif_prix_litre(…)** » qui modifie la valeur de « **prix_litre** ». La nouvelle valeur est passée en argument à cette méthode.

Note : On peut pour cela faire appel à une méthode d'instance voire une méthode de classe… Au choix !

Pour tester et bien comprendre le fonctionnement des **attributs de classe**, effectuez les étapes suivantes :

28. Affichez l'attribut « **prix_litre** » pour les DEUX instances de l'objet « **Voitures** ».
29. Appliquez un changement de « **prix_litre** » avec la méthode « **modif_prix_litre(…)** » pour une des deux instances.
30. Affichez à nouveau l'attribut « **prix_litre** » pour les DEUX instances de l'objet « **Voitures** ».

QCM

Pour finir cet exercice on souhaite disposer d'une méthode « **calcul_co2(…)** » qui donne le nombre de kg de CO_2 généré pour parcourir une distance donnée.

Donnée : la combustion d'un litre d'essence produit 2,3 kg de CO_2 ! C'est une constante invariable…

31. Codez cette nouvelle méthode et testez-là sur le trajet Colmar-Biarritz pour les deux instances de « **Voitures** ».

Note *: Vous pouvez comparer ces résultats avec un voyage en TGV qui génère sur cette même distance environ 3,9 kg de CO_2 par passager !!!*

Partie 3 : attributs protégés et pseudo-privés…

On veut ajouter deux nouveaux attributs :

« **_id_serie** » qui stocke le numéro de série du véhicule. C'est une « *string* » de 12 caractères (chiffres et lettres majuscules). Cet attribut est « ***protégé*** » par la présence d'UN « **_** » au début de son nom. Sa valeur par défaut est « **A123 B456 C789** » (*string*).

« **__audio_code** » qui conserve le code de déverrouillage du système audio qui est intégré au tableau de bord. Ce dernier code est une « *string* » de 4 chiffres qui doit être saisie lorsque le système audio est démonté ou lors d'un changement de batterie. Pour des raisons évidentes, cet attribut doit être « ***pseudo-privé*** » d'où les DEUX « **_** » au début de son nom. Sa valeur par défaut est « **0000** » (*string*)…

En plus, on impose un « ***chiffrage*** » de cette donnée lorsqu'elle est affectée à son attribut.

32. Ajoutez ces nouveaux attributs au constructeur.

*Remarque : Comme des valeurs par défaut sont définies dans le constructeur, il n'est pas nécessaire de modifier le code de création des deux instances dans « **garage.py** » …*

Indications *: Pour chiffrer simplement en Python et de manière réversible (possibilité de déchiffrer), on peut importer le module « **base64** » et utiliser ses méthodes « **b64encode(string_a_chiffre)** » et « **b64decode(string_chiffre)** ».*

Exemple :

```
import base64

    texte = "Hello"
    texte_chiffre = base64.b64encode(texte.encode()).decode()
        # donne : b'SGVsbG8='
    texte_dechiffre = base64.b64decode(texte_chiffre).decode()
        # donne : Hello
```

33. Afficher dans le code « `garage.py` » pour une des instances, ces deux attributs.

QCM

34. Codez une méthode « **`affiche_prot_priv()`** » qui affiche ces deux attributs puis appelez cette méthode dans « `garage.py` ».

QCM

35. Codez un « ***getter*** » et un « ***setter*** » pour chacun de ces attributs en utilisant les noms suivants : « **`get_id_serie()`** », « **`set_id_serie(num_serie)`** », « **`get_audio_code()`** » et **`set_audio_code(code)`** »… Testez ces quatre nouvelles méthodes.

Note *: Un contrôle des formats des valeurs fournies en argument des « **setters** » est souhaitable. La mise en œuvre d'un motif REGEX permet de valider cela simplement… Les « **setters** » doivent retourner un booléen pour valider ou l'opération de mise à jour.*

Remarque : Les motifs REGEX qui vont bien pour cela sont :
`r"[A-Z0-9]{4} [A-Z0-9]{4} [A-Z0-9]{4}"` *et* `r"[0-9]{4}"`

Indication *: Pour le code audio, un « **déchiffrage** » doit être intégré dans le « **getter** » et un « **chiffrage** » doit être présent dans le « **setter** »…*

36. Pour finaliser votre code ajouter un « ***docstring*** » à la classe et à chacune de ses méthodes…

Activité complémentaire

Partie 4 : export JSON

On souhaite stocker l'ensemble des instances de « **`Voitures`** » dans un fichier JSON. Le souci est que chaque instance a été créée indépendamment des autres et avec des noms propres (« **`clio`** » et « **`captur`** »).
On aurait pu créer une liste d'instances « **`vehicules`** » dans le « `garage.py` » un peu comme ceci :

```
vehicules = []
clio = Voitures("Renault", " Clio_TCE_100ch", 2020, "Bleu nuit", ...)
vehicules.append(clio)
vehicules.append(Voitures("Renault", " Captur_TCE_90ch", 2021, ...))
```

Cette approche marche très bien mais ne vous apporte rien de nouveau en matière de POO, si ce n'est que l'on peut créer une liste d'instances d'un objet…

Une autre approche consiste à utiliser une liste similaire mais comme **attribut de classe** !!! Pour cela :

37. Ajouter à la classe « **`Voitures`** », l'**attribut de classe** : « **`vehicules = []`** ».
38. Ajoutez à la fin du constructeur le code qui ajoute à la liste « **`vehicules`** » l'instance elle-même.

Indication *: Cela peut se faire avec un « **`Voitures.vehicules.append(self)`** »…*

Pour récupérer le contenu de la liste des instances cela peut se faire par le biais d'une méthode classique (méthode d'instance). En revanche, il est plus élégant et plus intéressant pédagogiquement de mettre en œuvre une **méthode de classe** ! Reprenez la partie du cours qui traite de ce sujet. Vous allez utiliser le décorateur « `@classmethod` » et le mot clé « `cls` »…

39. Ajoutez une **méthode de classe** « `retourne_liste_instances()` » qui retourne l'**attribut de classe** « `vehicules` » autrement dit une liste des instances de l'objet « `Voitures` »…

40. Affichez dans « `garage.py` » la liste des instances en invoquant cette **méthode de classe** et en utilisant la syntaxe : « *`nom_de_classe.methode_de_classe`* ».

Résultat *: Vous obtenez une liste d'objets pas très parlante.*

41. Itérez (avec un « `for` ») cette liste d'instances et affichez chaque instance.
42. Affichez également dans cette itération l'attribut « ***spécial*** » ou « ***magique*** » : « `__dict__` ».

Résultat *: S'affichent deux choses pour chaque véhicule de la liste des instances :*
– *Le contenu de l'instance.*
– *Un dictionnaire contenant tous les éléments de l'instance.*
Ce dictionnaire est directement utilisable pour l'exportation dans le fichier JSON.

43. Créez une fonction (dans « `garage.py` ») nommée « **`export_json(`***`nom_json, liste`***`)`** » qui va :
    – Générer dans un premier temps et à partir de la liste des instances d'objet passée en argument, une liste de dictionnaires issus de l'attribut « ***spécial*** » : « `__dict__` ».
    – Enregistrer dans un second temps, cette liste de dictionnaires dans un fichier JSON dont le nom est aussi passé en arguments. Le format JSON souhaité est donné ci-contre.
44. Créez une autre fonction (aussi dans « `garage.py` ») nommée « **`import_json(`***`nom_json`***`)`** » qui va :
    – Récupérer pour commencer une liste de dictionnaires dans le fichier JSON dont le nom est aussi passé en arguments.
    – Créer ensuite au sein d'une itération une nouvelle instance quelconque (utilisant les valeurs par défaut du constructeur) et d'importer dans les attributs de cette instance, les valeurs du dictionnaire.


```
[
    {
        "marque": "Renault",
        "annee": 2021,
        "modele": "Captur_TCE_90ch",
        "couleur": "Gris foncé",
        "conso": 7.2,
        "prix": 20000,
        "_id_serie": "A123 B456 C789",
        "_Voitures__audio_code": "MDAwMA=="
    },
    {
        "marque": "Renault",
        "annee": 2020,
        "modele": "Clio_TCE_100ch",
        "couleur": "Bleu nuit",
        "conso": 5.5,
        "prix": 17000,
        "_id_serie": "A111 B222 C333",
        "_Voitures__audio_code": "MTIzNA=="
    }
]
```


Indication *: Dans cette fonction d'importation, vous pouvez utiliser le code suivant pour chaque dictionnaire « `dico` » de la liste :*

```
instance_temp = Voitures()
instance_temp.__dict__.update(dico)
```

45. Testez les fonctions « **`export_json(…)`** » et « **`import_json(…)`** », vous devez au final avoir QUATRE instances de « **`Voitures`** » qui sont deux à deux identiques.
46. Affichez dans « `garage.py` » la liste des instances en invoquant cette **méthode de classe**.

Remarque : Le résultat de l'importation est toujours accessible en utilisant la syntaxe :
« *`nom_de_classe.methode_de_classe`* »

Partie 5 : IHM textuelle…

L'idée est d'avoir une application de gestion du parc de véhicules d'un concessionnaire RENAULT. Pour éviter de sensiblement alourdir le code avec une interface graphique et garder une très grande portabilité du code, il est proposé de mettre en œuvre une interface « *texte* » comme cela existait il y a 30 ans…

47. Créez un nouveau script principal « `concession.py` » qui reprend le code de « `garage.py` » tout en purgeant tout le programme principal (après les deux fonctions d'importation et d'exportation JSON).
48. Récupérez sur Moodle le module « `obj_ihm_texte.py` ».
49. Analysez en détails ce code (constructeur, et ses méthodes).
50. Importez l'objet « **`Ihm`** » du module « **`obj_ihm_texte`** ».
51. Récupérez le code suivant dans le programme principal :

```
import json
.....
FICHIER_JSON = "vehicules.json"

parc_vehicule = Voitures.retourne_liste_instances()
ihm = Ihm(parc_vehicule)
import_json(FICHIER_JSON)

while(True):
    if ihm.choix == 0:
        ihm.menu_accueil()
    elif ihm.choix == 1:
        ihm.menu_lister()
        ......
    elif ihm.choix == 8:
        export_json(FICHIER_JSON, parc_vehicule)
        break
```

A placer au début du code…

Récupération de la liste des instances de « **`Voitures`** » (attribut de classe).

Création de l'instance « **`ihm`** ».

Importation dans l'attribut de classe « **`vehicule`** » (liste) des données des JSON…

Gestion des menus.
L'attribut « **`choix`** » contient le numéro du menu courant…

52. Complétez le script « `garage.py` » pour les différentes valeurs de l'attribut « choix » (de 2 à 7).

***Indication** : Référez-vous au contenu de la page d'accueil !!!*

53. Complétez la méthode « **`menu_ajouter()`** » de classe « **`Ihm`** ».

Indication *: Le traitement des valeurs saisies est :*

- **modele** : *une « `ValueError` » doit être levée si c'est une « string » vide.*
- **annee** & **prix** : *doivent être convertis en entier.*
- **conso** : *doit être convertie en flottant.*
- **id_serie** : *une « `ValueError` » doit être levée s'il n'est pas au format "XXXX XXXX XXXX".*

54. Complétez la méthode « **`menu_supprimer()`** » de classe « **`Ihm`** ».

Indication *: Inspirez-vous de la méthode « `menu_ajouter()` » pour le choix du véhicule à supprimer. Une confirmation avec saisie de « OUI » est exigée avant la suppression…*

55. Codez les méthodes « **`menu_prix()`** » et « **`menu_carbone()`** » de classe « **`Ihm`** ».

Indication *: Inspirez-vous de la méthode « `menu_conso()` » pour cela.*

56. Testez l'ensemble du script. Vous pouvez utiliser le fichier « `vehicules.json` » qui se trouve sur Moodle !