Exercices – Programmation orientée objet (POO)

Exercice 1 – *(À faire sur papier)*

On considère le code suivant, dans lequel on définit la classe `Identite` :

```
class Identite:
    """ Identite(str, str, int) -> Identite
    Classe représentant l'identité d'une personne
    """
    def __init__(self, nom, prenom, annee):
        self.nom = nom
        self.prenom = prenom
        self.naissance = annee

    def age(self, annee):
        """ age(int) -> int
        Calcule l'âge de la personne 'self' au cours de l'année 'annee'.
        """
        return annee - self.naissance

# Exécution
hugo = Identite('Bonneau', 'Jean', 1970)
print(hugo.age(2020))
```

1. Quels sont les attributs de la classe `Identite` ?

2. Quels sont les membres de la classe `Identite` ?

3. À quelle ligne le constructeur `__init__()` est-il appelé ?

4. Quel est le résultat affiché par le programme à la fin de l'exécution ?

Exercice 2 – *(TP - Compte bancaire)*

On souhaite définir une classe `CompteBancaire` qui représente un compte bancaire ayant pour attributs :

- numero_compte, un entier
- nom, une chaîne de caractères
- prenom, une chaîne de caractères
- solde, un entier

1. Dans un fichier Python, écrire la définition de la classe et le constructeur de cette classe. Tester que le programme fonctionne en créant un compte bancaire quelconque.

2. Écrire une méthode `__str__` permettant d'afficher les caractéristiques du compte à l'aide de la fonction `print`. On cherchera à obtenir l'affichage ci-dessous :

   `N° Compte : xxxxxx | Prénom : xxxxx | Nom : xxxxxx | Solde : xxxxx`

   Tester que cet affichage fonctionne.

3. Créer deux instances de la classe `CompteBancaire` aux noms de Inès Si et Jordi Nateur qui ont respectivement sur leurs comptes 10000 euros et 5000 euros. Leurs comptes sont par ailleurs respectivement numérotés 123 et 321.

4. (a) Créer les méthodes `depot(self, somme)` et `retrait(self, somme)` qui gèrent respectivement les versements (ajout d'argent sur le solde) et les retraits (retirer de l'argent du solde). Ces deux méthodes prennent en paramètre un entier `somme` qui est le montant versé ou retiré du compte bancaire. Ces deux méthodes renvoient le nouveau solde du compte.

   (b) Inès fait un virement de 100 euros à Jordi. Quelles instructions utiliser pour modéliser cet échange d'argent ?

Exercice 3 – *(TP - Do you want a date?)*

On souhaite définir une classe `Date` pour représenter une date, elle aura trois attributs :

- le jour : un entier compris entre 1 et 31 ;
- le mois : un entier compris entre 1 et 12 ;
- l'année : un entier relatif.

1. Écrire la définition et le constructeur de cette classe.

   Le constructeur doit lever une `ValueError` si un des paramètres fournis n'est pas valide.

   Aide : Une exception (par exemple `ValueError`) peut être levée lorsqu'un paramètre inadapté est donné à une fonction. C'est ce qu'on peut lire dans la console Python en cas d'erreur d'exécution. Il y en a de nombreuses sortes.

   Voici la syntaxe :

   ```
   if condition :
       raise ValueError("Texte expliquant l'erreur")
   ```

2. (a) Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme « 14 juillet 1789 ». On pourra se servir d'un tableau (type `list`) ou d'un dictionnaire (type `dict`) pour faire le lien entre le numéro du mois et son nom.

   (b) Tester en construisant des objets de la classe `Date` puis en les affichant avec `print`.

   (c) Afficher l'année correspondant à une de ces dates. De quelle nature sont les attributs de la classe `Date` ?

3. (a) Ajouter une méthode `__lt__` qui permet de déterminer si une date `d1` est inférieure strictement à une date `d2`.

   (b) Tester cette méthode.

4. Si ce n'est déjà fait, documenter les méthodes créées.

5. On souhaite maintenant protéger les attributs de cette classe.

   (a) Rendre les attributs privés et vérifier qu'on ne peut plus les modifier directement depuis une instance.

   (b) Écrire les accesseurs (`get_<nom_attribut>()`) et mutateurs (`set_<nom_attribut>(valeur)`) pour chaque attribut et vérifier qu'ils permettent de lire et de modifier ces attributs.

6. Qu'affiche l'instruction `help(Date)` ?

Exercice 4 – *(TP)*

Une agence immobilière développe un programme pour gérer les biens immobiliers qu'elle propose à la vente. Dans ce programme, pour modéliser les données de biens immobiliers, on définit une classe `Bim` ayant comme attributs :

- `nature`, une chaîne de caractères qui représente la nature du bien (appartement, maison, bureau, commerces, etc.) ;
- `surface`, un entier représentant la surface du bien ;
- `prix_moyen`, un nombre décimal représentant le prix moyen du bien au m².

La classe `Bim` possède trois méthodes :

- `estime_prix` qui renvoie une estimation du prix du bien ;
- `modifie_prix_moyen` qui modifie le prix moyen du bien ;
- `est_plus_cher` qui prend en paramètre un objet de type `Bim` et qui renvoie un booléen `True` si le prix estimé du bien courant est supérieur à celui du bien passé en paramètre, `False` sinon.

Le code (incomplet) de la classe Bim est donné ci-dessous :

```
class Bim:
    """ Bim(str, int, float) -> Bim
    Classe représentant un bien immobilier
    """

    def __init__(self, nature, surface, prix_moyen):
        # à compléter

    def estime_prix(self):
        return self.surface * self.prix_moyen

    def modifie_prix_moyen(self, prix):
        """ modifie_prix_moyen(float) -> None
        Modifie le prix au métre carré
        """
        self.prix_moy = prix

    def est_moins_cher(self, autre_bien):
        """ est_moins_cher(Bim) -> bool
        param: autre_bien est un objet, instance la classe Bim
        Renvoie True si le prix estimé de self est strictement inférieur à celui de autre_bien et False sino
        """
        return # à compléter
```

1. Recopier ce programme et compléter le code du constructeur de la classe Bim. Tester.
2. On exécute l'instruction suivante :

   ```
   bien1 = Bim('maison', 70, 2000.0)
   ```

   Que renvoie l'instruction `bien1.estime_prix()` ? Préciser le type de la valeur renvoyée.
3. On souhaite affiner l'estimation du prix d'un bien en prenant en compte sa nature :
   - pour un bien dont l'attribut `nature` est « maison », la nouvelle estimation du prix est le produit de sa surface par le prix moyen multiplié par $1,1$;
   - pour un bien dont l'attribut `nature` est « bureau », la nouvelle estimation du prix est le produit de sa surface par le prix moyen multiplié par $0,8$;
   - pour les biens d'autres natures, l'estimation du prix ne change pas.

   Modifier le code de la méthode `estime_prix` afin de prendre en compte ce changement de calcul.
4. La méthode `modifie_prix_moyen` présente une coquille.
   (a) Expliquer ce qui se produit après avoir exécuté l'instruction `bien1.modifie_prix_moyen(2500)`.
   (b) Modifier le script de la méthode `modifie_prix_moyen` afin qu'elle fonctionne comme attendu.
5. (a) Compléter la méthode `est_moins_cher`.
   (b) Proposer un exemple d'exécution de la méthode `est_moins_cher` entre deux biens immobiliers `bien1` et `bien2` de type `Bim`.
6. Écrire le code Python d'une fonction `compte_maison(lst_biens)` qui prend en argument une liste Python de biens immobiliers de type `Bim` et qui renvoie le nombre d'objets de nature 'maison' contenus dans la liste `lst_biens`.

Exercice 5 – *(TP - Les irréductibles fractions)*

On souhaite définir une classe `Fraction` pour représenter un nombre rationnel. Cette classe possède deux deux attributs entiers, `numerateur` et `denominateur`. Il faut que le dénominateur soit un entier strictement positif.

1. Écrire la définition et le constructeur de cette classe. Le constructeur doit lever une `ValueError` si le dénominateur n'est pas strictement positif.
2. Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme $12/35$, ou simplement 12 si le dénominateur vaut 1.
3. Ajouter des méthodes `__eq__` et `__lt__` qui reçoivent une deuxième fraction en argument et renvoient `True` si la première fraction est égale (respectivement strictement inférieure) à la seconde.
4. Ajouter des méthodes `__add__` et `__mul__` qui reçoivent une deuxième fraction en argument et renvoient une fraction (instance de la classe `Fraction`) représentant la somme (respectivement le produit) de ces deux fractions.
5. Tester et documenter ces méthodes si ce n'est pas déjà fait.
6. *BONUS : écrire une méthode `rendre_irreductible` qui modifie la fraction courante pour la rendre irréductible.*

Exercice 6 – *(TP - Tableaux)*

Dans certains langages de programmation, comme Pascal et Ada, les tableaux peuvent être indexés sur des plages d'indices choisis par le programmeur. Par exemple, un tableau pourra être indexé de -10 à 9.

Il s'agit ici de construire une classe `Tableau` pour réaliser de tels tableaux. Un objet de cette classe aura quatre attributs :

- `i_min` un entier représentant le premier indice ;
- `i_max` un entier représentant le dernier indice ;
- `val` la valeur qui initialise tous les éléments du tableau (autrement dit une valeur par défaut).
- `tab` un tableau (`list`) Python contenant les éléments (ce tableau est un vrai tableau/liste Python, indexé à partir de 0).

1. Écrire la définition et le constructeur de cette classe.

   Par exemple, `Tableau(-10,9,42)` instanciera un tableau de vingt cases indexées de -10 à 9 et toutes initialisées à la valeur 42.
2. Écrire une méthode `__len__(self)` qui renvoie la taille du tableau.
3. Écrire une méthode `__getitem__(self, i)` qui renvoie l'élément `i` du tableau `self`.

   De même écrire une méthode `__setitem__(self, i, v)` qui modifie l'élément `i` du tableau `self`.

   Ces deux méthodes doivent vérifier que l'indice `i` est bien valide et dans le cas contraire, lever l'exception `IndexError`.
4. Enfin, écrire une méthode `__str__(self)` qui renvoie une chaîne de caractères décrivant le contenu du tableau.