Informatique 2: Travaux Pratiques

Programmation Orientée Objet 3/3 – Héritage

Le but de cette séance est de consolider les connaissances sur les classes et les objets en général et de se familiariser avec la notion d'héritage
Procéder par étape et tester votre code (dans une bloc main) après chaque étape.
Temps mentionné (🕓) à titre strictement indicatif.

Exercices de base

Question 1: (🕓 *30 minutes*) Cet exercice complète la classe **Produit** de la semaine passée (si vous n'avez pas terminé la partie "avancée" de l'exercice ou pour éviter d'accumuler d'éventuelles erreurs, vous pouvez repartir des solutions fournies la semaine passée). On veut définir des catégories de produits.

1. Définir une classe **ProduitFrais** qui hérite de la classe **Produit**.
2. Cette classe contient un attribut supplémentaire : la date d'emballage (au même format que la date, c'est-à-dire un `int`). Modifier le constructeur de la classe **ProduitFrais** en conséquence.
3. Créer une instance *p* de **Produit** et tester les deux appels suivants :
   `isinstance(p, Produit)`, `isinstance(p, ProduitFrais)`
   Créer une instance *p* de **ProduitFrais** et faire de même.
4. Dans la version du corrigé, la méthode `__str__` de la classe **Produit** génère une chaîne de caractère de la forme suivante : `'[Produit] ...'`. Que se passe-il quand on affiche un produit frais ?

   Exemple

   ```
   saumon = ProduitFrais('Saumon', 10.0, 6, 2)
   print(saumon)
   ```

   Modifier la méthode `__str__` dans la classe **Produit** pour afficher `'[ProduitFrais] ...'` lorsque l'objet en question est un produit frais.

   Attention

   Ne pas redéfinir la méthode `__str__` dans la classe **ProduitFrais** pour l'instant !

5. Redéfinir la méthode `__str__` pour la classe **ProduitFrais** pour qu'elle ajoute la date d'emballage à la fin de la chaîne générée par la méthode `__str__` de la classe **Produit**.

   Exemple

   ```
   "[ProduitFrais] Saumon (10.0 CHF), date d'emballage : 5"
   ```

6. La classe **Produit** contient un attribut de classe (`_delai_expiration_reduction`) permettant de fixer combien de jours avant la date de péremption une réduction devrait être appliqué. Comment faire pour que la classe **ProduitFrais** utilise une valeur différente (3 par exemple) ? Mettre en oeuvre la solution proposée.
7. Remplacer `self.__class__._delai_expiration_reduction` par `Produit._delai_expiration_reduction` dans la méthode **expire_bientot** et re-tester votre solution. Que se passe-t-il ? Pourquoi ?
8. Tester l'exemple complet en ajoutant des objets **Produit** standards et des objets **ProduitFrais**.

```
if __name__ == '__main__':
    rayon = Supermarche()
    rayon.ajouter(ProduitFrais("Saumon", 10.0, 6, 2))  # la réduction pour le saumon devrait apparaître dès le jour 3
    rayon.ajouter(ProduitFrais("Pain", 3.0, 2, 1))
    rayon.ajouter(Produit("Sauce", 6.0, 10))
    print(rayon)
    print()
```

```
8     for date in range(12):
9         print("Day " + str(date))
10        if date == 4:
11            Produit.set_reduction_generale(20)
12        if date == 7:
13            Produit.set_reduction_generale(0)
14        rayon.mettre_a_jour(date)
15        print(rayon)
```

En exécutant l'exemple complet vous devriez voir apparaître ceci :

```
'[Product] Sauce (4.800000000000001 CHF, valait 6.0 CHF)'
```

Question 2: (*40 minutes*) On veut reconstruire la hiérarchie des figures géométriques vue en cours et y ajouter une classe **Ellipse**.

1. On veut que la classe **Figure** soit abstraite, c'est à dire qu'elle ne puisse pas être instanciée. Pour ce faire, rendre le constructeur abstrait en levant l'exception appropriée. Essayer d'instancier la classe **Figure**.

   Attention

   Ne pas utiliser `super().__init__` dans les classes enfants !

2. On veut que chaque classe fille de **Figure** implémente les méthodes **get_aire**()->**float** et **get_perimetre**()->**float**. Comment imposer ceci ? Mettre en oeuvre la solution proposée.
3. On veut pouvoir calculer directement le rapport *aire/périmètre* pour une instance de **Figure**. Quel est le meilleur endroit pour implémenter cette méthode ? Implémenter la méthode.
4. Ecrire les classes **Rectangle**, **Carré**, **Ellipse** et **Cercle**. Approximer le périmètre d'une ellipse par :

   $$\pi * \sqrt{2 * (a^2 + b^2)}$$

   où a et b sont les demi longueurs du grand et du petit axe.
5. Ecrire la méthode **__str__** dans chacune d'elle.

   >_Exemple

   print(**rect**) affiche **'Rectangle (3x2)'** pour un rectangle de côté *3* et *2*.

6. On veut implémenter une méthode **est_cercle**()->**bool** qui renvoie un booléen indiquant si la figure considérée est un cercle. A quels endroits faut-il implémenter cette méthode ? L'implémenter.

   >_Exemple

   rect.est_cercle() renvoie **False**.

7. Même question pour **est_polygone**.

Question 3: (*10 minutes*) Le but de cet exercice est de se familiariser avec l'outil *Debug*.

1. Ouvrir dans `PyCharm` la version de la classe **Fraction** disponible sur Moodle (dans la section de la semaine 5).
2. Placer un point d'arrêt en cliquant à côté du numéro de la ligne  . Ici, on veut explorer le contenu de la liste **__list_instances**. Il faut donc placer un point d'arrêt **après** l'exécution de la Ligne 138.
3. Lancer l'exécution du programme avec l'outil *Debug* en cliquant sur le bouton (représentant un bug) à côté de la flèche run  .

4. Regarder les valeurs des différentes variables dans la fenêtre qui vient de s'ouvrir. Le type et la valeur de chaque variable sont affichés. Parcourir le contenu de la variable *l* (ligne 173) en faisant attention aux différents types des variables que contient cette liste (liste, Fraction, entier, etc).

Question 4: (*20 minutes*) projet La classe (abstraite) **LocationProvider** (fournisseur de données de localisation) décrit des objets permettant de produire une liste d'objets **LocationSample**. Elle spécifie l'existence d'une méthode **get_location_samples** (abstraite) qui renvoie une liste d'objets **LocationSample** triés par ordre chronologique (qui n'est pas implémentée, c'est aux classes filles de l'implémenter).

1. Définir la classe abstraite **LocationProvider** et son constructeur (abstrait).

> **Attention**
>
> Aucune forme d'héritage ne doit pour l'instant être implémentée.

2. Spécifier l'existence de **get_location_samples** (méthode abstraite).
3. Implémenter la méthode **print_location_samples**, qui renvoie une chaîne de caractères décrivant les objets **LocationSample** produit par un objet **LocationProvider** (c-à-d renvoyés par la méthode **get_location_samples**). Par exemple (avec de "vrais" timestamps et non juste des entiers) :
   'LocationSample [timestamp: 1615806676, location: Location [latitude: 48.85479, longitude: 2.34756]−>LocationSample [timestamp: 1615536472, location: Location [latitude: 46.51774, longitude: 6.63223]−>LocationSample [timestamp: 1616990937, location: Location [latitude: 46.54654, longitude: 6.64453]'

> **Attention**
>
> A ce stade vous ne pourrez pas encore complètement tester votre code car une classe abstraite ne peut pas être instanciée. Les points suivants sont nécessaires à ce que l'implémentation du code soit complète.

4. Implémenter la classe **ListLocationProvider**. Celle-ci hérite de la classe **LocationProvider**. Attention : ne pas appeler le constructeur de la classe mère avec **super** sous peine de lever une **NotImplementedError** dans la classe fille. Le constructeur de la classe **ListLocationProvider** aura un paramètre qui prendra en argument une liste contenant des objets **LocationSample**.
5. Créer un attribut privé samples et lui affecter une liste contenant des objets **LocationSample**. De plus, cette liste doit être triée au plus tard lors de l'affectation (pour rappel, l'opérateur d'affectation est =). Aussi, s'assurer d'avoir renvoyé une liste avec une copie des éléments qu'elle contient (i.e., une copie profonde, cf. diapositive 25 de la semaine 4). Ce point est crucial afin que le programme que l'on va créer avec le projet semestriel fonctionne correctement et ne modifie pas de manière non souhaitée les différents objets **LocationSample** existant ailleurs dans le programme.
6. Implémenter la méthode **get_location_samples** dans **ListLocationProvider** en s'assurant, encore une fois, de bien renvoyer la copie des éléments contenus dans la liste.
7. Pour vérifier que vous avez bel et bien stocké une copie (profonde) des éléments de la liste contenant des objets **LocationSample**, vous pouvez exécuter le code suivant dans un bloc main.

```
1  # creer des objets Location
2  geneve = Location(46.2044, 6.1432)
3  lausanne = Location(46.517738, 6.632233)
4  # creer des objets LocationSample differents
5  ls1 = LocationSample(lausanne, date=1615806676)
6  ls2 = LocationSample(geneve, date=1605804670)
7  # creer une instance de la classe ListLocationProvider
8  lst_locations = ListLocationProvider([ls1, ls2])
9  # creer une liste avec les identifiants des objets LocationSample contenus
10 # dans l'objet de la classe ListLocationProvider
11 lst_loc_ids = [id(loc) for loc in lst_locations.get_location_samples()]
12 memes_ids = []  # instantier une liste vide pour collecter les valeurs booleennes ci−apres
13 for ls in ls1, ls2:  # iterer sur des objets LocationSample
14     vraie_id = id(ls)  # invoquer l'identifiant du LocationSample et on le stocke dans vraie_id
15     meme_id = vraie_id in lst_loc_ids  # verifier que l'identifiant n'est pas repete
16     memes_ids.append(meme_id)
17 if sum(memes_ids) == 0:  # s'il y n'y a pas d'identifiants qui se repetent
18     # NB: ici on utilise une astuce pour verifier qu'il n'y ait pas d'identifiants qui se repetent:
```

```
19      # sum([False, False]) vaudrait 0, sum([True, False]) vaudrait 1 et sum([True, True]) vaudrait 2.
20      # En effet, True == 1 et False == 0.
21      print("On travaille en securite: tous les identifiants sont differents.")
22  else:  # s'il y a au moins un identifiant qui se repete
23      print("Le programme n'est pas sur.")
```