TP : Programmation Orientée Objet avec Python

****

A. Introduction :

Lors de ce TP, nous allons définir complètement une classe ***Intervalle*** permettant de modéliser des intervalles mathématiques formés de réels strictement positifs et de les manipuler. À la fin du TP, nous pourrons écrire des commandes du type :

```
>>> i1 = Intervalle(10.0, 12.0)
>>> i2 = Intervalle(9.0, 11.0)
>>> print(i1 & i2) # intersection
>>> print(i1 + i2) # addition
```

Exercice 1 : Définition de la classe Intervalle

Créer une classe ***Intervalle*** possédant les attributs et les méthodes suivantes :

1. Deux données « ***attributs*** » : une borne inférieure et une borne supérieure.
2. La méthode ***__init__*** qui permet d'initialiser les objets de cette classe aura donc trois paramètres : le premier étant l'objet que l'on initialise (traditionnellement nommé ***self***) et deux paramètres qui permettent d'initialiser les bornes d'un intervalle.

Exercice 2 : Méthodes pour l'affichage de l'objet « Intervalle »

Soit **`a = Intervalle(2,3).`**

Redéfinir les deux méthodes spéciales ***__str__*** et ***__repr__*** pour que :

```
>>> print(a) # affiche  <2, 3>
>>> a  # affiche Intervalle(2, 3)
```

Indications :

- La méthode spéciale de nom « ***__str__*** » est utilisée par la fonction « ***str*** » (et par ***print***) pour donner une représentation utilisateur des objets sous forme de chaîne de caractères.
  Comment s'exécute la fonction « ***str*** » de Python ?
  Lorsque on invoque la fonction « ***str*** » sur un objet « ***str(a)*** », Python recherche l'objet de nom « a » et regarde dans sa classe si une méthode « ***__str__(self)*** » est définie. Si oui, il l'appelle et retourne son résultat. Sinon, il regarde dans sa classe parente, et remonte jusqu'à la classe mère de toutes les classes : la classe prédéfinie ***object***.
- Une autre méthode spéciale de nom « ***__repr__*** » est utilisée par « ***repr*** » et par Python pour une **représentation "canonique"** des objets sous forme de chaîne.

B. Sécurisation minimale :

Les quatre exercices suivants ont pour objectif de sécuriser la ***classe Intervalle***. Il s'agit de ne pas avoir de donnée invalide dans un programme utilisant cette classe. Pour cela, nous mettrons en œuvre la règle minimale : si l'on détecte une donnée fallacieuse, on arrête le programme.

Exercice 3 — Quels problèmes de sécurité ?

Avec la classe Intervalle définie comme ci-dessus, entrez les lignes suivantes dans le ***« shell »*** lancé :

```
>>> print(Intervalle(0.25, 0.3))
>>> print(Intervalle(0.35, 0.1))
>>> print(Intervalle("abc", [1,2,3]))
```

Que donneront les affichages ?

Exercice 4 — Mettre les bornes dans l'ordre

Le second intervalle pose un problème : les bornes ne sont pas dans l'ordre. Cela peut être corrigé par programme à l'initialisation (méthode ***__init__***) : il suffit de stocker dans ***borne_inf*** le plus petit de a et b et dans ***borne_sup*** le plus grand (utilisez ***min*** et ***max***). Modifiez la méthode ***__init__*** de la ***classe Intervalle*** pour qu'elle crée automatiquement un intervalle dans le bon ordre.

Indications :

La création d'un intervalle avec une initialisation non numérique pose un problème plus compliqué : on peut le corriger automatiquement s'il est possible de les traduire en « ***float*** », mais sinon on ne peut rien faire.

La sécurisation minimale consiste à arrêter le programme avec un message d'erreur. Python permet d'essayer « ***try*** » des instructions et en cas d'erreur d'exécuter un gestionnaire d'erreurs. Ici, nous allons simplement arrêter le programme avec un message signalant l'erreur. C'est la sécurité minimale.

Il ne faudrait jamais qu'un programme continue son exécution avec des données erronées (frelatées).

Exercice 5 — Forcer des float convenables dans les bornes

Modifiez le programme pour qu'il s'arrête également si l'une des données est négative ou nulle. Vous pouvez lancer une exception avec l'instruction ***raise Exception()*** qui interrompra le programme.

Indications :

L'accès direct à des données d'une instance de classe peut être dangereux. En effet, avec la modification dans l'exercice précédent de la méthode ***__init__***, on ne peut plus créer d'intervalle mal formé. Toutefois, il est toujours possible à un programmeur d'écrire directement ***a.borne_sup = -2***, ce qui mettra « −2 » dans la borne supérieure de l'intervalle a.

On peut remédier à cet inconvénient en ne permettant d'accéder aux données que par des méthodes qui peuvent inclure des tests de protection.

Ainsi, dans la méthode ***__init__***, on peut cacher les noms des attributs ***borne_inf*** et ***borne_sup*** en les préfixant par deux tirets bas : ***__borne_inf*** et ***__borne_sup***.

Pourquoi deux tirets bas devant un nom d'attribut ?

Préfixer par deux tirets bas un nom d'attribut permet de diminuer le risque que, par mégarde, un programmeur écrive une instruction qui modifie une donnée sans qu'elle ne soit vérifiée. Deux tirets bas indiqueront à Python que la donnée est sensible : dans la définition de classe, Python transformera son nom en lui préfixant le nom de la classe, empêchant ainsi d'y accéder trop facilement.

Les données ***__borne_inf*** et ***__borne_sup*** ne seront visibles que depuis les méthodes de la classe, mais pas de l'extérieur.

Exemple :

```
1.  a = Intervalle(2.0, 3.0)
2.  print(a.__borne_inf) # inaccessible de l'extérieur
```

Exercice 6 — Méthode sécurisée

Pour modifier une valeur de l'intervalle, écrire maintenant dans la classe Intervalle une méthode sécurisée ***modif_sup(self, x)*** qui permettra de protéger la borne supérieure en ne pouvant y écrire que des nombres supérieurs à la borne inférieure.

Ajoutez une méthode ***modif_inf*** à la classe Intervalle. Faites attention à ce qu'une valeur négative ne puisse pas être enregistrée.

Exercice 7 — Méthode de lecture des données sécurisées

Écrivez deux méthodes d'accès ***lire_inf(self)*** et un ***lire_sup(self)*** qui retourneront les valeurs des bornes.

C. Opérations sur les intervalles :

Exercice 8 — Appartenance : x in I

La méthode magique ***__contains__*** permettant de tester si un objet en contient un autre.

Dans la classe Intervalle, écrivez la méthode ***__contains__(self, x)*** qui retournera ***True*** si x est dans l'intervalle, et ***False*** sinon.

```
1.  >>> "sa" in "salut"
2.  True
```

Exercice 9 — Intersection : I1 & I2

La méthode magique ***__and__*** permet d'utiliser l'opérateur ***« & »*** entre deux objets. Le programmeur malin décide de programmer l'intersection de deux intervalles en utilisant cet opérateur et donc en écrivant cette méthode. Il remarque (à tort !) que la borne inférieure de l'intersection est la plus grande des deux bornes inférieures et que la borne supérieure de l'intersection est la plus petite des deux bornes supérieures. Dans la classe Intervalle :

1. Ecrivez la méthode ***__and__(self,autre)*** qui retournera l'intersection de deux intervalles self et autre.
2. Donnez un exemple pour lequel cela ne marche pas. Modifiez la méthode pour qu'elle retourne ***« None »*** dans ce cas.

Exercice 10 — Union : I1 | I2

La méthode magique ***__or__*** permet d'utiliser l'opérateur ***« | »*** entre deux objets. Sur le même modèle que ci-dessus, écrivez la méthode ***__or__(self,autre)*** qui retourne l'intervalle union des deux intervalles et ***« None »*** si leur intersection est vide.

Exercice 11 — Autres opérations sur le même principe, écrivez les méthodes

1. **__add__(self, autre)** associée à « + » qui retourne un nouvel Intervalle somme des deux intervalles. **Exemple :** <2 , 5 > + < 3 , 4 > → < 5 , 9 >
2. **__mul__(self, autre)** associée à « * » qui retourne un nouvel Intervalle produit des deux intervalles. **Exemple :** < 2 , 5 > * < 3 , 4 > → <6 , 20>