Université de Montpellier
Faculté des sciences

Rapport des projets

Métaprogrammation et réflexivité

Étudiant :
Thomas Di Giovanni

Encadrant :
Christophe Dony

Année universitaire : 2019-2020

1 Étape 2 : orientation "langages"

Pour cette partie, j'ai décidé de recréer les exercices de TP en Python. Pour cela j'ai utilisé Visual Studio Code (récupérable à l'adresse https://code.visualstudio.com/#alt-downloads), avec l'extension Python : pour la télécharger, rechercher simplement "Python" dans l'onglet "Extensions".
L'exercice a été réalisé sous Windows 10, cependant il reste faisable sous d'autres systèmes d'exploitation et avec d'autres éditeurs de texte que VS Code.

Reprenons les exercices à partir du point 1.4 du sujet de TP : classes, instances, méthodes d'instance.
Premièrement, redéfinition de la classe Pile, avec quelques test unitaires de ses méthodes. Rien d'inhabituel ici, simple transformation du code réalisé précédemment en Python, avec l'utilisation du module "unittest" pour réaliser les tests.

Ensuite, création d'un simple inspecteur (point 2.3) avec le module "inspect" qui affiche les attributs d'un objet et ses valeurs : la classe Inspecter possède une méthode statique qui itère __dict__ (qui est la liste des attributs).

Passons ensuite à la programmation de méta-classes. Avant toute chose, nous allons clarifier le modèle utilisé dans ce langage :
Python utilise un modèle de méta-classes similaire à ObvjVlisp : toute classe est sous-classe d'une super-classe "object", et instance d'une méta-classe "type" (qui est aussi instance d'elle-même). Cependant, similairement au modèle Smalltalk, la métaclasse est propagée aux classes enfants : toutes les sous-classes d'une instance de la méta-classe M seront aussi instances de cette méta-classe M.

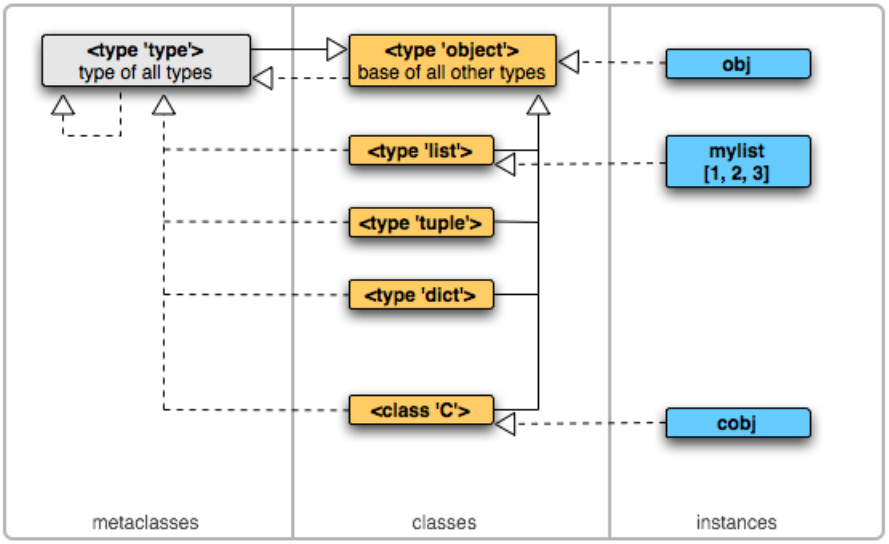

FIGURE 1 – Modèle utilisé dans Python

Passons au point 3.2 du sujet de TP : faire de Pile une classe une MemoClass, autrement dit une classe qui se souvient de ses instances.
Afin de créer une nouvelle méta-classe, on crée une nouvelle classe héritant de "type". On redéfinit ensuite la méthode __call__, qui est la méthode appelée à chaque fois qu'une instance de notre méta-classe (donc une classe) est appelée après avoir été créée.
Dans ce cas-là, on veut simplement ajouter la nouvelle instance dans notre liste.
On retourne ensuite à la définition de Pile pour rajouter "metaclass = MemoClass" dans sa signature.

Les blocs de codes suivants correspondent au point 3.3 : création d'une classe représentant une salle universitaire. On souhaite ici limiter le nombre de salles et donc empêcher la création de nouvelles instances.
On crée donc une nouvelle méta-classe nommée "CapacityClass", qui a un nombre limité d'instances. A chaque nouvelle création, on incrémente ce nombre et prévient l'utilisateur par un simple message dans la console.
Lorsque ce nombre est atteint, l'instanciation est annulée et une exception est levée.

Enfin, exercice 3.4 : création d'une classe abstraite Animal, et de deux classes concrètes Chien et Chat héritant d'Animal. Considérant le modèle de méta-classes de Python, nous nous retrouvons face au même dilemme qu'en Smalltalk : Animal étant une classe abstraite, Chien et Chat seront aussi abstraits.
La solution reste la même : nous devons spécifier dans la méthode d'instanciation qu'on ne lève l'exception que dans le cas d'Animal.

Le code se termine avec une simple fonction main permettant de tester toutes les fonctionnalités définies précédemment.